Local Observability and Controllability Analysis of Test Scenarios for Time-constrained Distributed Systems: VDM++ Specifications

<Anonymized Authors>

January 2019

Contents

1. Introduction

This report presents the complete specification in VDM++ [1] of the local observability and local controllability checking procedures and associated test cases introduced in out paper [2].

The specification follows a combination of the functional and imperative styles supported by VDM++. Classes are used simply as modules. The imperative style is used in some cases for performance reasons.

In this document, we kept the ASCII notation of VDM++ supported by Overture (instead of the equivalent mathematical notation [1]).

The test cases can be executed with the Overture interpreter [1]. It is only needed to copy the VDM++ code to one source file per class and execute the operation *testAll()* in class *TestCases*.

[1] http://overturetool.org/

2. Class DifferenceConstraints (Manipulation of Difference Constraints)

```
/**
 * Manipulation of difference constraints (DC) and boolean combinations in disjunctive
 * normal form (DNF).
 */

class DifferenceConstraints

types

public VariableId = nat;   -- time variable identifier
public Duration   = int;   -- difference between time values
public TimeValue  = nat;   -- time value in the desired scale (sec, mili, etc.)

-- Difference constraint, meaning vi - vj <= d
public DC ::
            i: VariableId
            j: VariableId
            d: Duration;

-- Expressions in Disjunctive-Normal-Form (DNF)
public AndExp :: args: set of DC;
public OrExp  :: args: set of (AndExp | DC);
public DCExp = DC | AndExp | OrExp;

values

public FalseExp: DCExp = mk_OrExp({}); -- existential quantifier on empty set
public TrueExp: DCExp = mk_AndExp({}); -- unversal quantifier on empty set

functions

-- Given an expression in DNF, returns the negation in DNF (partially simplified).
public mkNotExp: DCExp -> DCExp
mkNotExp(exp) ==
  if is_AndExp(exp) then mkOrExp2({mkNotExp(arg) | arg in set exp.args})
  else if is_OrExp(exp) then mkAndExp({mkNotExp(arg) | arg in set exp.args})
  else mk_DC(exp.j, exp.i, -(exp.d + 1));

operations
-- Given a set of expressions in DNF, returns the conjunction in DNF (partially
-- simplified).
public static pure mkAndExp: set of DCExp ==> DCExp
mkAndExp(args) == (
  dcl left : DCExp;

  -- special case
  if FalseExp in set args then
    return FalseExp;

  -- and's one argument at a time (starting with neutral element, i.e., TrueExp)
  left := TrueExp;
  for all right in set args do
    if right = TrueExp then
      skip
    else if left = TrueExp then
      if is_DC(right) then
```

```
        left := mk_AndExp({right})
      else
        left := right
    else if is_OrExp(left) or is_OrExp(right) then (
      -- applies distributive property
      left := mkOrExp2({mkAndExp2(e1, e2) |
                          e1 in set (if is_OrExp(left) then left.args else {left}),
                          e2 in set (if is_OrExp(right) then right.args else {right})}
                      \ {FalseExp});
      -- aborts with absorbing element, i.e., FalseExp
      if left = FalseExp then
        return FalseExp
    )
    else (
      left := mkAndExp2(left, right);
      -- aborts with absorbing element, i.e., FalseExp
      if left = FalseExp then
        return FalseExp
    );

    return left
 );

functions

-- Creates a DNF expression (partially simplified) for the conjunction of two
-- expressions of type AndExp or DC.
public mkAndExp2: (AndExp | DC) * (AndExp | DC) -> DCExp
mkAndExp2(exp1, exp2) ==
  let args1 = if is_AndExp(exp1) then exp1.args else {exp1},
      args2 = if is_AndExp(exp2) then exp2.args else {exp2}
  in
     -- check for contradictory constraints
     if exists mk_DC(i, j, d1) in set args1 &
            exists mk_DC((j), (i), d2) in set args2 & d1 + d2 < 0
     then FalseExp
     -- merge the constraints except for redundant ones
     else mk_AndExp({mk_DC(i, j, d1) | mk_DC(i, j, d1) in set args1 &
                      not exists mk_DC((i), (j), d2) in set args2 & d2 < d1}
                      union
                     {mk_DC(i, j, d2) | mk_DC(i, j, d2) in set args2 &
                      not exists mk_DC((i), (j), d1) in set args1 & d1 < d2});

-- Simplifes a set of difference constraints by removing redundant constraints.
private simplify: set of DC -> set of DC
simplify(C) ==
  {mk_DC(i, j, d) | mk_DC(i, j, d) in set C &
       not (i = j and d >= 0)
       and not (exists mk_DC((i), (j), d2) in set C & d2 < d)};

-- Creates a DNF expression (partially simplified) for the disjunction
-- of a set of expressions in DNF.
public mkOrExp: set of DCExp -> DCExp
mkOrExp(args) ==
  mkOrExp2(dunion {if is_OrExp(arg) then arg.args else {arg} | arg in set args});

-- Creates a DNF expression (partially simplified) for the disjunction of a set of
-- expressions of type AndExp or DC.
public mkOrExp2: set of (AndExp | DC) -> DCExp
```

```
mkOrExp2(args) ==
  -- remove redundant terms (that imply others)
  let args2 = {a | a in set args & not exists b in set args & a <> b and implies(a, b)}
  -- in case of a single term, doesn't need or'ing
  in  if card args2 = 1 then (let arg in set args2 in arg)
  -- normal case
      else mk_OrExp(args2);

-- Checks if an expression in DNF implies, for sure, another expression in DNF.
public implies: DCExp * DCExp -> bool
implies(exp1, exp2) ==
  if is_OrExp(exp1) or is_OrExp(exp2) then
    exists e1 in set (if is_OrExp(exp1) then exp1.args else {exp1}),
           e2 in set (if is_OrExp(exp2) then exp2.args else {exp2}) &
              implies(e1, e2)
  else
     let args1 = if is_AndExp(exp1) then exp1.args else {exp1},
         args2 = if is_AndExp(exp2) then exp2.args else {exp2}
     in forall mk_DC(i, j, d2) in set args2 &
           exists mk_DC((i), (j), d1) in set args1 & d1 <= d2;

operations

-- Checks the satisfiability of an expression in DNF.
-- Assumes implicit ordering constraints between (numbered) variables, so the problem
-- is to check if the there is an assignment of non-decreasing values to the variables
-- that satisfy the given expression.
public static pure sat: DCExp  ==> bool
sat(exp) == (
  dcl weights : map VariableId * VariableId to Duration; -- distance
  dcl dist : map VariableId to Duration; -- distance do first vertex/variable
  dcl vertices: seq of VariableId;
  dcl changed : bool;

  -- special cases
  if is_OrExp(exp) then
    return exists arg in set exp.args & sat(arg);
  if is_DC(exp) then
    return exp.i <> exp.j or exp.d >= 0;
  if exp.args = {} then
    return true;

  -- ordered vertex set
  vertices := [i | i in set {c.i | c in set exp.args} union {c.j | c in set exp.args}];

  -- edge weights (implicit ordering constraints plus explicit constraints)
  weights := {mk_(vertices(i),vertices(i+1)) |-> 0 | i in set {1,...,len vertices - 1}};
  for all mk_DC(i, j, d) in set exp.args do
    if (mk_(i, j) not in set dom weights or weights(mk_(i, j)) > d) then
        weights := weights ++ {mk_(i, j) |-> d};

  -- Bellman-Ford algorithm to find shortest paths from a source vertex (first) to all
  -- vertices in the presence of edges of negative weight
  dist := {v |-> 0 | v in seq vertices}; -- start with 0 because of implicit ordering
  for i = 1 to len vertices-1 do (
    changed := false;
    for all mk_(u, v) in set dom weights do
      if dist(v) > dist(u) + weights(mk_(u, v)) then (
        dist(v) := dist(u) + weights(mk_(u, v));
        changed := true;
      );
      if not changed then
        return true  -- optimization
  );

  -- If didn't converge, there are loops of negative size, so exp is not satisfiable
  return not exists mk_(u, v) in set dom weights & dist(v) > dist(u) + weights(mk_(u,v))
);

functions

-- Reduces (simplifies) an expression, by eliminating non-satisfiable terms.
public red: DCExp  -> [DCExp]
red(exp) == (
  if is_OrExp(exp) then
    let feasible = {arg | arg in set exp.args & sat(arg)}
    in if card feasible = 1 then let arg in set feasible in arg
     else mk_OrExp(feasible)
  else if sat(exp) then exp else FalseExp
);

operations
-- Eliminates variables after a given one in a given expression (i.e., projects the
-- expression onto the variables that remain).
-- Assumes implicit ordering constraints between consecutively numbered vertices.
public static pure elimVarsAfter:  VariableId * DCExp ==> DCExp
elimVarsAfter(maxV, c) ==
  if is_OrExp(c) then mkOrExp({elimVarsAfter(maxV, arg) | arg in set c.args})
  else if is_AndExp(c) then mkAndExp(elimVarsAfter(maxV, c.args))
  else mkAndExp(elimVarsAfter(maxV, {c}));

-- Eliminates variables after a given one (v) in a set of difference constraints (C1).
-- Assumes implicit ordering constraints between consecutively numbered vertices.
public static pure elimVarsAfter: VariableId * (set of DC) ==> set of DC
elimVarsAfter(v, C1) ==
(
   dcl C: set of DC;
   dcl vertices : seq of VariableId;
   dcl idx : nat;
   dcl e: VariableId;

   -- special cases
   if v = 0 or C1 = {} then return {};

   -- relevant vertices in constraint graph (referenced vertices plus v!), sorted
   vertices := [i | i in set {v} union dunion {{i,j} | mk_DC(i, j, -) in set C1}];

    -- adds implicit ordering constraints, and then simplifies
    C := simplify(C1 union
    {mk_DC(vertices(i), vertices(i+1), 0) | i in set {1,..., len vertices-1}});

   -- removes one variable/vertex at a time (shortcircuiting constraints/edges)
   idx := len vertices;
   while idx > 0 and vertices(idx) > v do (
     e := vertices(idx);
     C := {mk_DC(i1, j2, d1 + d2) | mk_DC(i1, (e), d1),
                     mk_DC((e), j2, d2) in set C & i1 <> e and j2 <> e}
           union {mk_DC(i, j, d) | mk_DC(i, j, d) in set C &
```

```
                    i <> e and j <> e};
     idx := idx-1
   );

   -- simplifies and then removes implicit ordering constraints
   return {mk_DC(i, j, d) | mk_DC(i, j, d) in set simplify(C) &
                                 not (j > i and d >= 0)}
);

 -- Projects an expression onto a set of variables (eliminating other variables),
 -- and renumbers the variables to sequential numbers starting in 1.
public static pure projectToVars: DCExp * (set of VariableId) ==> DCExp
projectToVars(c, V) ==
(
   dcl C: set of DC;
   dcl Vs : seq of VariableId;
   dcl vars : seq of VariableId;

   if is_OrExp(c) then
       return mkOrExp({projectToVars(arg, V) | arg in set c.args});

   -- set of difference constraints
   C := if is_AndExp(c) then c.args else {c};

   -- special cases
   if V = {} or C = {} then
      return TrueExp;

   -- sorted list of relevant variables (mentioned in constraints and range)
   vars := [i | i in set V union dunion {{i,j} | mk_DC(i, j, -) in set C}];

   -- add implicit ordering constraints and then simplify
   C := simplify(C union
                {mk_DC(vars(i), vars(i+1), 0) | i in set {1,..., len vars-1}});

   -- remove unwanted variables (shortcircuiting constraints/edges)
  for all e in set elems vars \ V do
     C := {mk_DC(i1, j2, d1 + d2) |
              mk_DC(i1, (e), d1), mk_DC((e), j2, d2) in set C &
                i1 <> e and j2 <> e}
           union {mk_DC(i, j, d) |
              mk_DC(i, j, d) in set C & i <> e and j <> e};

     -- simplify and then remove implicit constraints
     C := {mk_DC(i, j, d) | mk_DC(i, j, d) in set simplify(C) &
             not (j > i and d >= 0)};

    -- renumber (pack) variables sequentially, mapping old to new numbers
    Vs := [v | v in set V]; -- sort
    return mk_AndExp(renumVars({Vs(i) |-> i | i in set inds Vs}, C));
 );

functions

-- Renumbers the variables in a set C of difference constraints, based a given
-- map or sequence from old ids to new ids.
public renumVars: (map VariableId to VariableId | seq of VariableId) * (set of DC) ->
set of DC
renumVars(renum, C) ==
  {mk_DC(renum(i), renum(j), d)| mk_DC(i, j, d) in set C};

-- Checks if a binding of variables to values satisfies a set of constraints.
public satisfies: (map VariableId to TimeValue | seq of TimeValue) * (set of DC) -> bool
satisfies(binding, C) ==
  forall mk_DC(i, j, d) in set C & binding(i) - binding(j) <= d;

end DifferenceConstraints
```

3. Class SequenceDiagrams (Specification of Sequence Diagrams)

```
/**
* Specification of UML Sequence Diagrams (UML Interactions) used for describing
* integration test scenarios of distributed systems, conditions for local observability
* and local controlability, primitives for conformance checking and test input
* selection, and examples.
**/

class SequenceDiagrams is subclass of DifferenceConstraints

/** Configuration parameters **/

instance variables
public static MaxClockSkew : TimeValue := 10;  -- e.g., 10 ms

types

/** Values, Value Specifications, Bindings and Timing info (based on UML meta-model) **/

public Value = nat | bool | real | String;
public String = seq of char;

public ValueSpecification = Value | Variable | Expression | <Unknown>;
public Variable :: name: String;
public Expression :: symbol: ExpSymbol
                     operands: seq of [ValueSpecification];
public ExpSymbol = <Neg> | <Eq> | <Plus> | <Minus> | <Lt> | <Lte> | <Gt> | <Gte> | <And>
| <Or>;

public Bindings = map Variable to Value;

public TimeInterval = [TimeValue] * [TimeValue];
public DurationInterval = [Duration] * [Duration];

/** UML Interactions (base on UML meta-model) **/

public Interaction ::
  lifelines         : set of Lifeline
  messages          : set of Message
  combinedFragments : set of CombinedFragment
  timeConstraints   : set of TimeConstraint
inv i ==
      -- message ids and send and receive locations are unique
  (forall m1, m2 in set i.messages & m1 <> m2 =>
       m1.id <> m2.id
       and m1.sendEvent <> m2.sendEvent
       and m1.receiveEvent <> m2.receiveEvent)
      and
      -- lifeline names are unique
  (forall l1, l2 in set i.lifelines & l1 <> l2 => l1.name <> l2.name)
      and
      -- referenced lifelines exist
  (forall m in set i.messages & {m.sendEvent.#1, m.receiveEvent.#1} subset i.lifelines)
  and
  (forall c in set i.combinedFragments & c.lifelines subset i.lifelines)
  and
  -- time variables are unique
  (forall m1, m2 in set i.messages & m1 <> m2 =>
             let l = [m1.sendTimestamp, m1.recvTimestamp, m2.sendTimestamp,
m2.recvTimestamp]
             in not exists i, j in set inds l &
                  i <> j and l(i) <> nil and l(j) <> nil and l(i) = l(j))
      and
       (forall m in set i.messages & m.sendTimestamp <> nil and m.recvTimestamp <> nil
=>
                           m.sendTimestamp <> m.recvTimestamp);

public Lifeline :: name : String;

public MessageType = <Synch> | <Asynch>;

public Message ::
      id            : MessageId
  sendEvent     : LifelineLocation
  receiveEvent  : LifelineLocation
  signature     : MessageSignature
  sendTimestamp : [Variable]
  recvTimestamp : [Variable]
  type          : MessageType
  guard         : [TimeConstraint]
inv m == m.sendEvent <> m.receiveEvent;

public MessageSignature = String;
public MessageId = nat;
public Location = nat;
public LifelineLocation = Lifeline * Location;

public CombinedFragment ::
  interactionOperator : InteractionOperatorKind
  operands            : seq1 of InteractionOperand
  lifelines           : set of Lifeline
inv f ==
  cases f.interactionOperator:
    <loop>, <opt> -> len f.operands = 1,
    <alt>, <par>, <strict>, <seq> ->
        len f.operands > 1 and forall op in seq f.operands & op.guard = nil
  end
  and (forall o in seq f.operands &
         {lf | mk_(lf, -) in set o.startLocations} = f.lifelines
         and {lf | mk_(lf, -) in set o.finishLocations} = f.lifelines)
  and (forall i in set {1, ..., len f.operands - 1} &
          f.operands(i+1).startLocations = f.operands(i).finishLocations);

public InteractionOperatorKind = <seq> | <alt> | <opt> | <par> | <strict> | <loop>;

public InteractionOperand ::
  guard            : [InteractionConstraint]
  startLocations   : set of LifelineLocation
  finishLocations  : set of LifelineLocation;

public InteractionConstraint ::
  minint       : [ValueSpecification]  -- loop
  maxint       : [ValueSpecification]  -- loop
  specification: [ValueSpecification] | <else>;

public TimeConstraint ::
  firstEvent : Variable
  secondEvent: Variable
  min        : [Duration]
  max        : [Duration]
inv tc == tc.min <> nil or tc.max <> nil;

/** Traces **/

public Trace = seq of Event;

public TCTrace = Trace * DCExp; -- Time constrained trace

public Event ::
                   type        : EventType
                   signature   : MessageSignature
                   lifeline    : Lifeline
                   timestamp   : [Variable | TimeValue]; --Var. in event; Value in event
occurrence

public EventType = <Send> | <Receive> | <Stop>;

protected TraceExt = seq of EventExt;

protected EventExt ::
                   type        : EventType
                   signature   : MessageSignature
                   lifeline    : Lifeline
                   timestamp   : [ValueSpecification]
                   location    : Location
                   messageId   : nat
                   itercounter: seq of nat;

functions

/** Auxiliary functions for creating things **/

public mkInteraction : (set of Lifeline) * (set of Message) * (set of CombinedFragment)
*
                       (set of TimeConstraint) -> Interaction
mkInteraction(lifelines, messages, combinedFragments, timeConstraints) ==
      mk_Interaction(lifelines, messages, combinedFragments, timeConstraints);

public mkInteraction : (set of Lifeline) * (set of Message) * (set of CombinedFragment)
                       -> Interaction
mkInteraction(lifelines, messages, combinedFragments) ==
      mkInteraction(lifelines, messages, combinedFragments, {});

protected mkMessage: MessageId * LifelineLocation * LifelineLocation *
                     MessageSignature -> Message
mkMessage(id, sendEvent, receiveEvent, signature) ==
  mk_Message(id, sendEvent, receiveEvent, signature, nil, nil, <Asynch>, nil);


protected mkMessageTimed: MessageId * LifelineLocation * LifelineLocation *
                          MessageSignature -> Message
mkMessageTimed(id, sendEvent, receiveEvent, signature) ==
  mk_Message(id, sendEvent, receiveEvent, signature,
      mk_Variable("s_" ^ signature ^ VDMUtil`val2seq_of_char[MessageId](id)),
      mk_Variable("r_" ^ signature ^ VDMUtil`val2seq_of_char[MessageId](id)),
<Asynch>,nil);

protected mkMessageTimedGuarded: MessageId * LifelineLocation * LifelineLocation *
                          MessageSignature * TimeConstraint -> Message
mkMessageTimedGuarded(id, sendEvent, receiveEvent, signature, guard) ==
  mk_Message(id, sendEvent, receiveEvent, signature,
    mk_Variable("s_" ^ signature ^ VDMUtil`val2seq_of_char[MessageId](id)),
    mk_Variable("r_" ^ signature ^ VDMUtil`val2seq_of_char[MessageId](id)),
<Asynch>,guard);


protected mkMessageTimedSynch: MessageId * LifelineLocation * LifelineLocation *
                               MessageSignature -> Message
mkMessageTimedSynch(id, sendEvent, receiveEvent, signature) ==
  mk_Message(id, sendEvent, receiveEvent, signature,
             mk_Variable("s_" ^ signature ^ VDMUtil`val2seq_of_char[MessageId](id)),
             mk_Variable("r_" ^ signature ^ VDMUtil`val2seq_of_char[MessageId](id)),
<Synch>, nil);

protected mkEvent: EventType * MessageSignature * Lifeline -> Event
mkEvent(type, signature, lifeline) == mk_Event(type, signature, lifeline, nil);

protected mkEvent: EventType * MessageSignature * Lifeline *
                   [ValueSpecification] -> Event
mkEvent(type, signature, lifeline, timestamp) ==
  mk_Event(type, signature, lifeline, timestamp);

public mkStopEvent: Lifeline -> Event
mkStopEvent(l) == mk_Event(<Stop>, [], l, nil);

/** Containment checking functions **/

protected contains: CombinedFragment * CombinedFragment -> bool
contains(f1, f2) ==
  contains(f1.operands(1).startLocations, f1.operands(len f1.operands).finishLocations,
           f2.operands(1).startLocations, f2.operands(len f2.operands).finishLocations);

protected contains: InteractionOperand * CombinedFragment -> bool
contains(o, c) ==
  contains(o.startLocations, o.finishLocations,
          c.operands(1).startLocations, c.operands(len c.operands).finishLocations);

protected contains: InteractionOperand * LifelineLocation -> bool
contains(o, lfloc) == contains(o.startLocations, o.finishLocations, lfloc);

protected contains: CombinedFragment * LifelineLocation -> bool
contains(f, lfloc) == contains(f.operands(1).startLocations,
                               f.operands(len f.operands).finishLocations, lfloc);

protected contains: (set of LifelineLocation) * (set of LifelineLocation) *
                    LifelineLocation -> bool
contains(startLocs, endLocs, mk_(lf,loc)) ==
  (exists mk_(lf1, loc1) in set startLocs & lf1 = lf and loc1 < loc)
  and (exists mk_(lf2, loc2) in set endLocs & lf2 = lf and loc2 > loc);

protected contains: (set of LifelineLocation) * (set of LifelineLocation) *
                    (set of LifelineLocation) * (set of LifelineLocation) -> bool
contains(startLocs1, endLocs1, startLocs2, endLocs2) ==
  (forall mk_(lf, loc2) in set startLocs2 &
     exists mk_(lf1, loc1) in set startLocs1 & lf1 = lf and loc1 < loc2)
  and (forall mk_(lf, loc2) in set endLocs2 &
         exists mk_(lf1, loc1) in set endLocs1 & lf1 = lf and loc1 > loc2);

/** Auxilairy query functions **/

-- Get timestamp of an event.
protected t: Event -> [Variable | TimeValue]
t(e) == e.timestamp;

-- Get 'send' event of a message.
protected s: Message -> Event
s(m) == mk_Event(<Send>, m.signature, m.sendEvent.#1, m.sendTimestamp);

-- Get 'receive' event of a message.
protected r: Message -> Event
r(m) == mk_Event(<Receive>, m.signature, m.receiveEvent.#1, m.recvTimestamp);

-- Gets the message corresponding to an event.
protected msg: Interaction * Event -> Message
msg(sd, e) ==
  if e.type = <Send> then
     let m in set sd.messages be st e = s(m) in m
  else
     let m in set sd.messages be st e = r(m) in m;

-- Checks if a difference constraint is a maximum duration constraint.
protected isMaxDuration: DC -> bool
isMaxDuration(mk_DC(i, j, d)) == i > j /*and d >= 0*/;

-- Get time constraints related with a lifeline.
protected getTimeConstraints: Interaction * Lifeline -> set of TimeConstraint
getTimeConstraints(sd, l) ==
      {c | c in set sd.timeConstraints & exists m in set sd.messages &
                   (m.sendTimestamp = c.secondEvent and m.sendEvent.#1 = l)
                   or (m.recvTimestamp = c.secondEvent and m.receiveEvent.#1 = l)};

 -- Counts the number of occurrences of event 'e' in trace 't'.
protected count: Event * Trace -> nat
count(e,t) == if t = [] then 0 else (if e = hd t then 1 else 0) + count(e, tl t);

 -- Checks if an event is of type Send.
public isSend: Event -> bool
isSend(e) == e.type = <Send>;

-- Checks if an event is of type Receive.
public isReceive: Event -> bool
isReceive(e) == e.type = <Receive>;

end SequenceDiagrams
```

4. Class ValidTraces (Calculation of Valid Traces)

```
/**
 * Computation of valid traces defined by an Interaction.
 */

class ValidTraces is subclass of SequenceDiagrams

functions

-- Determine the valid formal traces defined by an Interaction (sd).
public static validTraces: Interaction -> set of Trace
validTraces(sd) == {t | mk_(t,-) in set validTimedTraces(sd)};

public validTracesUntimed: Interaction -> set of Trace
validTracesUntimed(sd) ==  removeExtraTraceInfo(validTracesExt(sd));

-- Determine the set of valid timed traces defined by an Interaction (sd).
public static validTimedTraces: Interaction -> set of TCTrace
validTimedTraces(sd) ==
  let cand =  {mk_(removeExtraInfo(t), constraintExp(t, sd)) | t in set
validTracesExt(sd)}
  in {mk_(t,c) | mk_(t,c) in set cand & checkSyncMessagesOrdering(mk_(t, c)) and
sat(c)};


/**
 * Computation of valid traces timed.
 */

-- Checks if the send and receive events of sync messages (with 0 max delay) are
contiguous
protected checkSyncMessagesOrdering: TCTrace -> bool
checkSyncMessagesOrdering(mk_(t,C)) ==
   is_AndExp(C) =>
      forall mk_DC(i,j,d) in set C.args &
             i > j and d = 0 and t(i).type = <Receive> and t(j).type = <Send> and
t(i).signature = t(j).signature
             => i = j + 1;


-- Given a trace t and a set of time constraints C, returns the tuples (i, j, c)
-- where i and j are indices of events in t that are subject to a constraint c in C
protected static getConstrainedPairs: Trace * (set of TimeConstraint) -> set of (nat *
nat * TimeConstraint)
getConstrainedPairs(t, C) ==
   {mk_(i,j,c) | i in set inds t, j in set inds t, c in set C &
        i < j and t(i).timestamp = c.firstEvent and t(j).timestamp = c.secondEvent
        and if t(i).lifeline = t(j).lifeline then
                  not exists k in set {i+1, ..., j-1} &
                   t(k).timestamp = c.firstEvent or t(k).timestamp = c.secondEvent
            else
               card{k|k in set {1,...,i} & t(k).timestamp = c.firstEvent}
               = card{k|k in set {1,...,j} & t(k).timestamp = c.secondEvent}
   };

private constraintExp: TraceExt * Interaction -> DCExp
constraintExp(t, sd) ==
  mkAndExp(dunion {ev2ocConstr(i,j,c) |
            i in set inds t , j in set inds t, c in set sd.timeConstraints  &
              i < j and t(i).timestamp = c.firstEvent and t(j).timestamp = c.secondEvent
              and t(i).itercounter = t(j).itercounter}
           union guardConstraints(t, sd));

operations
private static pure guardConstraints:  TraceExt * Interaction ==> set of DC
guardConstraints(t, sd) ==
(
   dcl C : set of DC := {};
   for all m in set sd.messages do
     if m.guard <> nil then
        let c = m.guard in
           for all k in set inds t do
              if t(k).type = <Send> and t(k).messageId = m.id then
                C := C union dunion {ev2ocConstr(i,j,c) |
                      i in set {1,...,k-1}, j in set {1,...,k-1} &
                      i < j and t(i).timestamp = c.firstEvent and t(j).timestamp =
c.secondEvent
                      and t(i).itercounter = t(j).itercounter and t(i).itercounter =
t(k).itercounter};
  return C
);

functions

-- Applies a constraint to a pair of events
protected ev2ocConstr: nat * nat * TimeConstraint ->  set of DC
ev2ocConstr(i, j, c) ==
   if c.max = nil then
        if c.min = nil then {}
            else {mk_DC(i, j, -c.min)}
   else
        if c.min = nil then {mk_DC(j, i, c.max)}
            else {mk_DC(i, j, -c.min), mk_DC(j, i, c.max)};

/**
 * Computation of valid traces untimed.
 */

private removeExtraInfo: EventExt -> Event
removeExtraInfo(e) == mkEvent(e.type, e.signature, e.lifeline, e.timestamp);

private removeExtraInfo: TraceExt -> Trace
removeExtraInfo(t) == [removeExtraInfo(e) | e in seq t];

private removeExtraTraceInfo: set of TraceExt -> set of Trace
removeExtraTraceInfo(s) == {removeExtraInfo(t) | t in set s};


private validTracesExt: Interaction -> set of TraceExt
validTracesExt(sd) ==
      freeComb({{s} | s in set topLevelEvents(sd) & s <> []}
                        union {expandCombinedFragment(sd, c) | c in set
topLevelCombFrag(sd)});

private topLevelEvents: Interaction -> set of seq of EventExt
topLevelEvents(sd) ==
      {(if not exists c in set sd.combinedFragments & contains(c, m.sendEvent) then
        [mk_EventExt(<Send>, m.signature, m.sendEvent.#1, m.sendTimestamp,
m.sendEvent.#2, m.id, [])]
        else [])
       ^
       (if not exists c in set sd.combinedFragments & contains(c, m.receiveEvent) then
          [mk_EventExt(<Receive>, m.signature, m.receiveEvent.#1, m.recvTimestamp,
m.receiveEvent.#2, m.id, [])]
        else [])
      | m in set sd.messages};

private topLevelCombFrag: Interaction -> set of CombinedFragment
topLevelCombFrag(sd) ==
      {c | c in set sd.combinedFragments &
                         not exists c2 in set sd.combinedFragments & contains(c2, c)};

private freeComb: set of set of TraceExt -> set of TraceExt
freeComb(s) ==
      if s = {} then {[]}
      else let s1 in set s
           in dunion {freeComb2(t1, t2) | t1 in set s1, t2 in set freeComb(s \ {s1})};

private freeComb2: TraceExt * TraceExt -> set of TraceExt
freeComb2(t1, t2) ==
  if t1 = [] or t2 = [] then {t1 ^ t2}
  else (if exists e in seq t2 & precedes(e, hd t1) then {}
              else {[hd t1] ^ r | r in set freeComb2(tl t1, t2)})
       union
       (if exists e in seq t1 & precedes(e, hd t2) then {}
              else {[hd t2] ^ r | r in set freeComb2(t1, tl t2)});

private precedes: EventExt * EventExt -> bool
precedes(e1, e2) ==
      (e1.messageId = e2.messageId and e1.itercounter = e2.itercounter and e1.type =
<Send> and e2.type = <Receive>)
      or (e1.lifeline = e2.lifeline
         and (e1.location < e2.location
              or e1.location = e2.location and precedesIter(e1.itercounter,
e2.itercounter)));

private precedesIter: (seq of nat) * (seq of nat) -> bool
precedesIter(s1, s2) ==
      s1 <> [] and s2 <> [] and
      (hd s1 < hd s2 or hd s1 = hd s2 and precedesIter(tl s1, tl s2))
pre len s1 = len s2;

/** Valid (formal) traces defined by combined fragments **/

private expandCombinedFragment: Interaction * CombinedFragment -> set of TraceExt

expandCombinedFragment(sd, c) ==
  cases c.interactionOperator:
    <seq>     -> expandNary(sd, c.operands, seqComb),
    <strict>  -> expandNary(sd, c.operands, strictComb),
    <par>     -> expandNary(sd, c.operands, parComb),
    <alt>     -> expandAlt(sd, c.operands),
    <opt>     -> expandOpt(sd, c.operands(1)),
    <loop>    -> expandLoop(sd, c.operands(1))
  end;
```

```
private expandNary: Interaction * (seq of InteractionOperand) * (TraceExt * TraceExt ->
set of TraceExt) -> set of TraceExt
expandNary(sd, args, comb) ==
  if args = []  then {[]}
  else dunion {comb(t1, t2) | t1 in set expandOperand(sd, hd args),
                              t2 in set expandNary(sd, tl args, comb)};

-- Weak sequencing combination of two traces, given by the interleavings
-- that preserve the order of events per trace and lifeline.
private seqComb: TraceExt * TraceExt -> set of TraceExt
seqComb(t1, t2)==
  if t1 = [] or t2 = [] then {t1 ^ t2}
  else {[hd t1] ^ r | r in set seqComb(tl t1, t2)}
       union if exists e in seq t1 & (hd t2).lifeline = e.lifeline then {}
             else {[hd t2] ^ r | r in set seqComb(t1, tl t2)};

-- Strict sequencing of two traces, given by their concatenation.
private strictComb: TraceExt * TraceExt -> set of TraceExt
strictComb(t1, t2) == {t1 ^ t2};

-- Parallel combination of two traces, given by the interleavings
-- that preserve the order of events per trace.
private parComb: TraceExt * TraceExt -> set of TraceExt
parComb(t1, t2) ==
  if t1 = [] or t2 = [] then {t1 ^ t2}
  else {[hd t1] ^ r | r in set parComb(tl t1, t2)}
       union {[hd t2] ^ r | r in set parComb(t1, tl t2)};

private expandAlt: Interaction * seq of InteractionOperand -> set of TraceExt
expandAlt(sd, args) == dunion {expandOperand(sd, arg) | arg in seq args};

private expandOpt: Interaction * InteractionOperand -> set of TraceExt
expandOpt(i, arg) == expandOperand(i, arg) union {[]};

private expandLoop: Interaction * InteractionOperand -> set of TraceExt
expandLoop(sd, arg) ==
  let argExpansions = expandOperand(sd, arg)
  in if arg.guard <> nil and arg.guard.maxint <> nil
     then let nums = {(if arg.guard.minint = nil then 0 else arg.guard.minint), ...,
arg.guard.maxint}
                   in dunion { iterate(argExpansions, n) | n in set nums}
     else dunion {iterate(argExpansions, n) | n: nat & arg.guard = nil or n >=
arg.guard.minint};

private iterate: (set of TraceExt) * nat -> set of TraceExt
iterate(s, numIter) ==
  if numIter = 0 then {[]}
  else dunion {seqComb(t1, addIterNumber(t2, numIter)) | t1 in set iterate(s, numIter-
1), t2 in set s};

private addIterNumber: TraceExt * nat -> TraceExt
addIterNumber(t, iter) == [mu(e, itercounter |-> [iter] ^ e.itercounter) |    e in seq
t];

private expandOperand: Interaction * InteractionOperand -> set of TraceExt
expandOperand(i, o) ==
      freeComb({{s} | s in set nestedEvents(i, o) & s <> []}
                        union {expandCombinedFragment(i, c) | c in set nestedCombFrag(i,
o)});

private nestedEvents: Interaction * InteractionOperand -> set of seq of EventExt
nestedEvents(sd, o) ==
  let cf = {c | c in set sd.combinedFragments & contains(o, c)}
  in
     {(if contains(o, m.sendEvent) and not exists c in set cf & contains(c, m.sendEvent)
       then [mk_EventExt(<Send>, m.signature, m.sendEvent.#1, m.sendTimestamp,
m.sendEvent.#2, m.id, [])]
       else [])
      ^
      (if contains(o, m.receiveEvent) and not exists c in set cf & contains(c,
m.receiveEvent)
       then [mk_EventExt(<Receive>, m.signature, m.receiveEvent.#1, m.recvTimestamp,
m.receiveEvent.#2, m.id, [])]
       else [])
    | m in set sd.messages};

private nestedCombFrag: Interaction * InteractionOperand -> set of CombinedFragment
nestedCombFrag(sd, o) ==
  let cf = {c | c in set sd.combinedFragments & contains(o, c)}
  in {c | c in set cf & not exists c2 in set cf & c2 <> c and contains(c2, c)};

end ValidTraces
```

5. Class Observability (Local Observability Checking)

```
/**
 * Analysis of local observability.
 */

class Observability is subclass of ValidTraces

values

-- Semantic variation point: FIFO channel between each pair of lifelines
public static FIFO_CHANNELS = false;

functions
-- Determines if conformance checking can be performed locally.
public isLocallyObservable: Interaction -> bool
isLocallyObservable(sd) == uncheckableLocallyTimed(sd) = {};

-- Gives global (symbolic) traces that are locally but not globally valid.
public uncheckableLocally: Interaction -> set of Trace
uncheckableLocally(sd) ==
  {t | mk_(t,-) in set uncheckableLocallyTimed(sd)};

-- Gives global (symbolic) traces that are locally but not globally valid.
public uncheckableLocallyTimed: Interaction -> set of TCTrace
uncheckableLocallyTimed(sd) ==
  let V = validTimedTraces(sd),
      P = projectTCTraces(V, sd.lifelines),
      J = joinTimedTraces(P)
  in subtractTimedTraces(J, V);

-- Gives global (formal) traces that are locally but not globally valid.
public uncheckableLocallyUntimed: Interaction -> set of Trace
uncheckableLocallyUntimed(sd) ==
  let V = validTraces(sd),
      P = projectTraces(V, sd.lifelines)
  in joinTraces([], P) \ V;

/** Basic operations required on traces - Project ***/

-- Projects a set of traces (T) onto a set of lifelines (L).
public projectTraces: (set of Trace) * (set of Lifeline) -> map Lifeline to set of Trace
projectTraces(T, L) == {l |-> projectTraces(T, l) | l in set L};

-- Projects a set of traces (T) onto a lifeline (l).
public projectTraces: (set of Trace) * Lifeline -> set of Trace
projectTraces(T, l) == {projectTrace(t, l) | t in set T} ;

-- Projects a trace (t) onto a lifeline (l).
public projectTrace: Trace * Lifeline -> Trace
projectTrace(t, l) == [e | e in seq t & e.lifeline = l];

-- Projects a set T of time constrained traces onto a set L of lifelines.
public projectTCTraces: (set of TCTrace) * (set of Lifeline) -> map Lifeline to set of
TCTrace
projectTCTraces(T, L) == {l |-> projectTCTraces(T, l) | l in set L};
```

```
-- Projects a set T of time constrained traces onto a lifeline (l).
public projectTCTraces: (set of TCTrace) * Lifeline -> set of TCTrace
projectTCTraces(T, l) ==
  let P = {projectTCTrace(t, l) | t in set T}
  in {mk_(t,c) | mk_(t,c) in set P &
          not exists mk_((t), c2) in set P & c2 <> c and implies(c, c2)};

-- Projects a time constrained trace (t, c) onto a lifeline (l).
public projectTCTrace: TCTrace * Lifeline -> TCTrace
projectTCTrace(mk_(t,c), l) ==
   let I = lifelineInds(l, t)
   in mk_([t(i) | i in seq I], projectToVars(c, elems I));

-- Check if two traces are equal, ignoring timestamps
public eqIgnTimestamps: Trace * Trace -> bool
eqIgnTimestamps(t1, t2) ==
  len t1 = len t2 and forall i in set inds t1 &
     mu(t1(i), timestamp |-> 0) = mu(t2(i), timestamp |-> 0);

-- Check if two events are equal, ignoring timestamps
public eqIgnTimestamps: Event * Event -> bool
eqIgnTimestamps(e1, e2) ==
  mu(e1, timestamp |-> 0) = mu(e2, timestamp |-> 0);

-- Subtracts two sets of time constrained traces (S1 - S2).
public subtractTimedTraces: (set of TCTrace) * (set of TCTrace) -> set of TCTrace
subtractTimedTraces(S1, S2) ==
  dunion {let c2 = mkOrExp({c2 | mk_(t2, c2) in set S2 & eqIgnTimestamps(t1, t2)}),
              c3  = red(mkAndExp({c1, mkNotExp(c2)}))
          in if c3 <> FalseExp then {mk_(t1, c3)} else {}
         | mk_(t1,c1) in set S1}};


-- Joins time constrained traces. Given sets of time constrained traces per lifeline,
-- obtains all the possible combinations of time constrained traces from different
-- lifelines, preserving the order of events per lifeline and message (send before
-- receive), and such that the joined time constraints are satisfiable.
protected joinTimedTraces:  map Lifeline to set of TCTrace -> set of TCTrace
joinTimedTraces(M) ==
  {mk_(t,c) | mk_(t,c) in set joinTimedTraces(mk_([], TrueExp), M) &
      checkSyncMessagesOrdering(mk_(t,c))};

protected joinTimedTraces: TCTrace * map Lifeline to set of TCTrace -> set of TCTrace
joinTimedTraces(mk_(t,c), m) ==
  (if forall l in set dom m & exists mk_(t1,-) in set m(l) & t1 = [] then {mk_(t,c)}
   else {})
  union
  dunion {
      dunion {
        let newT = t ^ [e],
            r = lifelineInds(l, newT),
            C2 = elimVarsAfter(len r, lc.args),
            newC = mkAndExp({c} union renumVars(r, C2)),
            newM = m ++ {l |-> {mk_(rt, lc)}} -- restricts to this trace in l
        in if sat(newC) then joinTimedTraces(mk_(newT, newC), newM) else {}
     | mk_([e] ^ rt, lc) in set m(l) & isFeasibleAddition(t, e)}
  | l  in set dom m};
```

```
-- Obtains the sequence of indices of events in a trace 't' that occur at a lifeline
'l'.
public lifelineInds: Lifeline * Trace -> seq of nat
lifelineInds(l, t) == [i | i in set inds t & t(i).lifeline = l];

-- Gives the feasible joins of traces from different lifelines,
-- respecting the order of events per trace and message.
-- The first argument is an accumulator for already processed events.
protected joinTraces: Trace * map Lifeline to set of Trace -> set of Trace
joinTraces(left, m) ==
  if m = {|->} then {left}
  else dunion { dunion {
          if t = [] then joinTraces(left, {l} <-: m)
          else joinTraces(left ^ [hd t], m ++ {l |-> {tl t}})
        | t in set m(l) & t = [] or isFeasibleAddition(left, hd t)}  | l  in set dom m};

-- Gives the feasible joins of traces from different lifelines,
-- respecting the order of events per trace and message. The
-- first argument is an accumulator for already processed events.
protected joinActualTraces: Trace * map Lifeline to Trace -> set of Trace
joinActualTraces(t, m) ==
  if forall l in set dom m & m(l) = [] then {t}
  else dunion {joinActualTraces(t ^[hd m(l)], m ++ {l |-> tl m(l)})
         | l in set dom m & m(l) <> [] and isFeasibleAddition(t,  hd m(l))};

-- smilar, with one trace per lifeline
protected joinTraces: map Lifeline to Trace -> set of Trace
joinTraces(localTraces) ==
  joinTraces([], {l |-> {localTraces(l)} | l in set dom localTraces});

-- Checks if an event occurrence is a feasible addition to a
-- trace, i.e., respects the fact that messages can only
-- be received after being sent, and respects timestamp ordering.
protected isFeasibleAddition: Trace * Event -> bool
isFeasibleAddition(t, e) ==
  (e.type = <Receive> =>
     len [ 0 | mk_Event(<Send>, sig, -, -) in seq t & sig = e.signature] >
     len [ 0 | mk_Event(<Receive>, sig, -, -) in seq t & sig = e.signature])
  and
  (e.timestamp <> nil and not is_Variable(e.timestamp) =>
      forall f in seq t &
        f.timestamp <> nil =>
          if f.lifeline = e.lifeline then f.timestamp <= e.timestamp
          else f.timestamp <= e.timestamp + MaxClockSkew
  );

end Observability
```

6. Class ConformanceChecking (Decentralized Conformance Checking)

```
/**
 * Incremental and global conformance checking primitives, and local input selection
primitives.
 */

class ConformanceChecking is subclass of Observability

types
public Verdict = <Pass> | <Fail> | <Inconclusive>;

functions

-- Checks if the next observed event in a lifeline is valid,
-- given a (valid) sequence of previously observed events in the
-- lifeline, and the set of valid traces for the lifeline.
public checkNextEvent: Trace * Event * (set of Trace) -> bool
checkNextEvent(prevEvents, event, validLocalTraces) ==
--  exists p ^ [e] ^ - in set validLocalTraces & e = event and p = prevEvents;
    exists t in set validLocalTraces & len t > len prevEvents and t(1,..., len
prevEvents + 1) = prevEvents ^ [event];

-- Checks if the next observed event occurrence (e) in a lifeline
-- is valid, given a valid sequence of previously observed
-- event occurrences in the lifeline (p), the set of valid local
-- traces (V) and the set of local time constraints (C).
public timedCheckNextEvent: Trace * Event * (set of Trace) * (set of TimeConstraint) ->
bool
timedCheckNextEvent(p, e, V, C) ==
  exists t in set V &  len t > len p
    and matches(p ^ [e],  t(1, ..., len p + 1), C) = <Pass>;

-- Final conformance checking, given the observed local traces.
public finalConformanceChecking: Interaction * (map Lifeline to Trace) -> Verdict
finalConformanceChecking(sd, localTraces) ==
    let V = validTraces(sd),
        J = joinTraces(localTraces)
    in if J inter V = {} then <Fail>
       else if J subset V then <Pass>
       else <Inconclusive>;


-- Similar, with timing information.
public timedFinalConformanceChecking: Interaction * (map Lifeline to Trace) -> Verdict
timedFinalConformanceChecking(sd, localTraces) ==
    let V = validTraces(sd),
        J = joinTraces(localTraces),
        C = sd.timeConstraints
    in if forall j in set J & forall v in set V & matches(j, v, C) = <Fail> then <Fail>
       else if forall j in set J & exists v in set V & matches(j, v, C) = <Pass> then
<Pass>
       else <Inconclusive>;

-- Checks if an actual trace (a) matches a formal trace (f),
-- given a set of time constraints (C).
protected matches: Trace * Trace * (set of TimeConstraint) -> Verdict
matches(a, f, C)  ==
  if not matchesUntimed(a, f) then <Fail>
  else let verdicts = {checkConstraint(a(i), a(j), c) | mk_(i, j, c) in set
getConstrainedPairs(f, C)}
       in if <Fail> in set verdicts then <Fail>
          else if <Inconclusive> in set verdicts then <Inconclusive>
          else <Pass>;

-- Checks if an actual trace (a) matches a formal trace (f),
-- without taking time constraints into consideration.
public matchesUntimed: Trace * Trace -> bool
matchesUntimed(a, f)  ==
  len a = len f
  and forall i in set inds a & mu(a(i), timestamp |-> nil) = mu(f(i), timestamp |-> nil)
;

-- Checks a time constraint (c) between two events (e1 before e2).
-- trace (a) that is being matched against a formal trace (f).
protected static checkConstraint: Event * Event * TimeConstraint -> Verdict
checkConstraint(e1, e2, c) ==
  let d = e2.timestamp - e1.timestamp,
      s = (if e1.lifeline = e2.lifeline then 0 else MaxClockSkew),
      ds = mk_(if d-s < 0 then 0 else d-s, d+s),
      it = intersect({mk_(c.min, c.max), ds})
  in if it = ds then <Pass>
     else if it = nil then <Fail>
     else <Inconclusive>;


public static prefixes: set of Trace -> set of Trace
prefixes(T) == {[]} union dunion{{t(1, ..., i) | i in set inds t} | t in set T};


/**** Primitives for local test selection *****/

public nextSendEvents: Trace * (set of Trace) -> set of Event
nextSendEvents(prevEvents, validLocalTraces) ==
--  {e | (prevEvents) ^ [e] ^ - in set validLocalTraces & e.type = <Send>};
  {t(len prevEvents + 1) | t in set validLocalTraces &
        len t > len prevEvents
        and t(1,..., len prevEvents) = prevEvents
        and t(len prevEvents + 1).type = <Send>};

-- Gives the next events that can be sent by a lifeline, and
-- the time interval for sending each event, given the actual
-- trace observed locally so far (a), the formal traces valid
-- locally (V) and the local time constraints (C).
public nextSendEventsTimed: Trace * (set of Trace) * (set of TimeConstraint) -> set of
(Event * TimeInterval)
nextSendEventsTimed(a, V, C) ==
  {mk_(f(len a +1), eventInterval(a, f, len a +1, C)) | f in set V &
    len f > len a and f(len a +1).type = <Send> and matches(a, f(1, ..., len a), C) =
<Pass>
    and eventInterval(a, f, len a +1, C) <> nil};


-- Determines the TimeInterval for occurring the i-th event of a
-- formal trace (f), given the previous actual trace (a)
-- and time constraints (C). Returns nil if impossible.
```

```
protected static eventInterval: Trace * Trace * nat * set of TimeConstraint ->
[TimeInterval]
eventInterval(a, f, i, C) ==
  intersect({mk_(if c.min = nil then nil else a(k).timestamp + c.min,
                 if c.max = nil then nil else a(k).timestamp + c.max)
             | mk_(k, n, c) in set getConstrainedPairs(f, C) & n = i});

protected static intersect: set of TimeInterval -> [TimeInterval]
intersect(s) ==
   if s = {} then mk_(nil, nil)
   else let mk_(min1, max1) in set s in
           let r = intersect(s \ {mk_(min1, max1)}) in
              if r = nil then nil
              else let mk_(min2, max2) = r,
                       min3 = if min1 = nil then min2 else if min2 = nil then min1
                              else if min1 > min2 then min1 else min2,
                       max3 = if max1 = nil then max2 else if max2 = nil then max1
                              else if max1 < max2 then max1 else max2
                    in if min3 <> nil and max3<> nil and min3 > max3 then nil
                    else mk_(min3, max3);

protected static contains: DurationInterval * DurationInterval -> bool
contains(i, j) == intersect({i, j}) = j;

end ConformanceChecking
```

7. Class Controllability (Local Controllability Checking)

```
/**
 * Analysis of local controllability.
 */

class Controllability is subclass of ConformanceChecking

types

-- Transmission channel for each pair of lifelines (FIFO_CHANNELS = true)
-- or pair of lifelines and message signature (FIFO_CHANNELS = false).
private Channel = (Lifeline * Lifeline) | (Lifeline * Lifeline * MessageSignature);

-- Each extension of a tc-trace is a pair of an added event and added time constraints.
private Extension = Event * (set of DC);

functions

-- Determines if an Interaction (Sequence Diagram) is locally controllable, i.e.,
-- no invalid traces are generated and all valid traces are generated when lifelines
-- behave using local knowledge only (traces observed locally and traces valid locally),
-- without exchanging coordination messages between them, and transmission channels
-- behave correctly.
public isLocallyControllable: Interaction -> bool
isLocallyControllable(sd) == unintendedTraces(sd) = {} and missingTraces(sd) = {};

-- Determines the invalid time traces that can be generated when lifelines
-- behave using local knowledge only (traces observed locally and traces valid locally).
-- The invalid traces are truncated up to the first invalid event.
public unintendedTraces: Interaction -> set of Trace
unintendedTraces(sd) ==
  if sd.timeConstraints = {} then unintendedTracesUntimed(sd)
  else unintendedTracesTimed(sd);

-- Determines the valid timed traces that are not generated when lifelines
-- behave using local knowledge only (traces observed locally and traces valid locally).
public missingTraces: Interaction -> set of Trace
missingTraces(sd) ==
  let V = validTimedTraces(sd),
      P = projectTCTraces(V, sd.lifelines),
      S = simulExec(sd, P, mk_([], TrueExp), {|->})
  in {t | mk_(t, -) in set subtractTimedTraces(V, S)};

-- Determines the invalid traces that can be generated when lifelines
-- behave using local knowledge only (sets of traces valid locally),
-- in the presence of time constraints.
public unintendedTracesTimed: Interaction -> set of Trace
unintendedTracesTimed(sd) ==
  let V = validTimedTraces(sd),
      P = projectTCTraces(V, sd.lifelines),
      S = simulExec(sd, P, mk_([], TrueExp), {|->}),
      U = subtractTimedTraces(S, V)
  in {truncateOnError(V, tc) | tc in set U};

-- Similar, but tc-traces, not truncated
```

```
public unintendedTracesTimedRaw: Interaction -> set of TCTrace
unintendedTracesTimedRaw(sd) ==
  let V = validTimedTraces(sd),
      P = projectTCTraces(V, sd.lifelines),
      S = simulExec(sd, P, mk_([], TrueExp), {|->}),
      U = subtractTimedTraces(S, V)
  in U;

/** Auxiliary private features - for untimed SDs ***/

-- Determines the invalid traces that can be generated (truncated on error) when
-- lifelines behave using  local knowledge only (traces observed locally and traces
-- valid locally), in the absence of time constraints.
-- Gives (formal) subtraces that can be generated according to causality rules,
-- but end in an unintended send (us), receive (ur) or termination (ut).
private unintendedTracesUntimed: Interaction -> set of Trace
unintendedTracesUntimed(sd) ==
  let V = validTracesUntimed(sd),
      T = prefixes(V),
      L = sd.lifelines,
      P = projectTraces(V, L),
   us = {q ^ [t2(len t2)] | q in set T, t2 /*p ^ [e]*/ in set T &
        len t2 > 0 and let p = t2(1,...,len t2-1), e = t2(len t2) in
          e.type = <Send>
          and projectTrace(q, e.lifeline) = projectTrace(p, e.lifeline)} \ T  ,
   ur =   dunion {{q  ^ [t2(len t2)] | q in set prefixes({t2(1,...,len t2-1)}) &
                                         isFeasibleAddition(q, t2(len t2))}
                | t2 in set T & len t2 > 0 and t2(len t2).type = <Receive>} \ T,
   ut =  {p | p in set T & allMsgsReceived(p) and
               mayRemainQuiescentUntimed(sd, p, P)} \ V
  in us union ur union ut;

-- Checks (in a simplified way) if all message have been received in a trace (t).
private allMsgsReceived: Trace -> bool
allMsgsReceived(t) ==
  card {i | i in set inds t & isSend(t(i))}
  = card {i | i in set inds t & isReceive(t(i))};

-- Determines if a lifeline may remain quiescent after a valid global trace (t).
private
mayRemainQuiescentUntimed: Interaction * Trace * (map Lifeline to set of Trace) -> bool
mayRemainQuiescentUntimed(sd, t, P) ==
  forall l in set sd.lifelines &
    let p = projectTrace(t, l)
    in p in set P(l)
       or (not exists (p) ^ [e] ^ - in set P(l) & e.type = <Send>)
       or (exists (p) ^ [e] ^ - in set P(l) & e.type = <Receive>);

/** Auxiliary private features - for timed SDs ***/

public static simulExec: Interaction -> set of TCTrace
simulExec(sd) ==
  let V = validTimedTraces(sd),
      P = projectTCTraces(V, sd.lifelines)
  in simulExec(sd, P, mk_([], TrueExp), {|->});

-- Recursively computes the time constrained traces that can be generated by the
-- execution of an interaction (sequence diagram), if each lifeline behaves according
-- to local knowledge only (traces observed locally and traces valid locally) and the
-- transmission chanel respects transmission constraints.
-- Parameters:
--  sd - interaction (sequence diagram)
--  P - valid local time constrained traces per lifeline
--  (t, c) - time constrained trace generated so far (initially empty)
--  m - map from channel identifier to queue of messages in transit
private simulExec: Interaction * (map Lifeline to set of TCTrace) * TCTrace *
           (map Channel to seq of (nat * Event)) -> set of TCTrace
simulExec(sd, P, mk_(t, c), m) ==  (
 -- Handle different cases in disjunction separately
 if is_OrExp(c) then
   dunion {simulExec(sd, P, mk_(t, arg), m) | arg in set c.args}

 -- Handle normal cases
 else
   let -- Compute possible trace extensions, from the perspective of each lifeline,
       -- as well as quiescence condition and emission deadline for each lifeline.
       E = {l |-> traceExtLf(mk_(t, c), l, P(l)) | l in set sd.lifelines},

       -- Select emission candidates at lifelines and respective constraints
       S = {mk_(e,C) | mk_(e,C) in set dunion {E(l).#1 | l in set sd.lifelines} &
isSend(e)},

       -- updated status of projections per lifeline
       newP = {l |-> E(l).#2 | l in  set sd.lifelines},

       -- Compute reception candidates, based on messages in transit and transmission
constraints
        R = candFromChannels(sd, mk_(t,c), m),

        -- Compute constraint for system emission deadline
        cS = {E(l).#4 | l in set sd.lifelines},

       -- Compute constraint for system reception deadline
        cR = dunion {{ct | ct in set C & isMaxDuration(ct)} | mk_(-, -, C) in set R}
    in
     -- Reception
     dunion {let newC = red(mkAndExp({c} union C union cR union cS))
             in if newC = FalseExp then {}
                else simulExec(sd, consumeEvent(e, newP), mk_(t ^ [e], newC),
                               updChannelsRecv(e, t(i), m))
             | mk_(i, e, C) in set R}

     -- Emission
     union
     dunion {let newC = red(mkAndExp({c} union C union cR union cS))
             in if newC = FalseExp then {}
                else if msg(sd, e).type = <Synch> then
                       let r = r(msg(sd, e)),
                           c2 = mk_DC(len t + 2, len t +1, 0)
                       in
                         simulExec(sd, consumeEvent(r, consumeEvent(e, newP)),
                                   mk_(t ^ [e, r], mkAndExp({newC, c2})), m)
                     else
                        simulExec(sd, consumeEvent(e, newP), mk_(t ^ [e], newC),
                                  updChannelsSend(sd, len t + 1, e, m))
             | mk_(e, C) in set S}

     -- Termination (quiescense)
     union (if R = {} then
                let cQ = red(mkAndExp({c} union {E(l).#3 | l in set sd.lifelines}))
                in if cQ = FalseExp then {} else {mk_(t, cQ)}
            else {})
);

-- Update status (m) of transmission channels of an interaction (sd) after an emission
-- event (s) in position 'i' of a trace
private updChannelsSend: Interaction * nat * Event * (map Channel to seq of (nat *
Event))
                         -> (map Channel to seq of (nat * Event))
updChannelsSend(sd, i, s, m) ==
   let r = r(msg(sd, s)),
       channel = if FIFO_CHANNELS then mk_(s.lifeline, r.lifeline)
                 else mk_(s.lifeline, r.lifeline, s.signature)
   in if channel in set dom m then m ++ {channel |-> m(channel) ^ [mk_(i, r)]}
      else m munion {channel |-> [mk_(i, r)]};

-- Update status (m) of transmission channels after reception/delivery event (r).
private updChannelsRecv: Event * Event * (map Channel to seq of (nat * Event))
                         -> (map Channel to seq of (nat * Event))
updChannelsRecv(r, s, m) ==
  let channel = if FIFO_CHANNELS then mk_(s.lifeline, r.lifeline)
                else mk_(s.lifeline, r.lifeline, s.signature)
  in if len m(channel) = 1 then  {channel} <-: m
     else m ++ {channel |-> tl m(channel)};

-- Determine candidate events from transmission channels
private candFromChannels: Interaction * TCTrace * (map Channel to seq of (nat * Event))
                          -> set of (nat * Event * (set of DC))
candFromChannels(sd, mk_(t, -), m) ==
  {let mk_(i, r) = hd m(channel),
       C = dunion {ev2ocConstr(i, len t + 1, c2) | c2 in set sd.timeConstraints &
                      c2.firstEvent = t(i).timestamp and c2.secondEvent = r.timestamp}
   in mk_(i, r, C)
   | channel in set dom m};

-- Update status (P) of lifelines after an event (e)
private consumeEvent: Event * (map Lifeline to set of TCTrace)
                      -> (map Lifeline to set of TCTrace)
consumeEvent(e, P) ==
   P ++ {e.lifeline |-> {mk_(tl t, c) | mk_(t, c) in set P(e.lifeline) &
                            t <> [] and eqIgnTimestamps(hd t, e)}};

operations
-- Obtains the possible extensions of a global time constrained trace (t,c), from the
-- perspective of a lifeline l with a set V of locally valid time constrained traces.
-- Each extension is a pair of an added event and added time constraints.
-- Return a tuple with:
--   set of extensions
--   update V (restricting to satisfiable tc-traces).
--   quiescence condition after the given tc-trace
--   emission deadline condition after the given tc-trace.
private static pure traceExtLf: TCTrace * Lifeline * (set of TCTrace)
            ==> (set of Extension) * (set of TCTrace) * DCExp * DCExp
traceExtLf(mk_(t, c), l, V) == (
  dcl E : set of Extension := {};
  dcl newV : set of TCTrace := {};
  dcl newE : Event;
  dcl newC : set of DC;
  dcl r1 : seq of nat := lifelineInds(l, t);
  dcl r2 : seq of nat := r1 ^  [len t + 1];
  dcl hasSend : bool := false;
  dcl hasUnrestrictedStop : bool := false;
  dcl hasUnrestrictedRecv : bool := false;

  for all mk_(lt, lc) in set V do (
    if lt = [] then (
      newC := renumVars(r1, lc.args);
      newE := mkStopEvent(l)
    )
    else (
      newC := renumVars(r2, elimVarsAfter(len r2, lc.args));
      newE := hd lt
    );

    if sat(mkAndExp({c} union newC)) then (
      E := E union{mk_(newE, newC)};
      newV := newV union {mk_(lt, lc)};
      cases newE.type:
         <Send>    -> hasSend := true,
         <Stop>    -> if newC = {} then hasUnrestrictedStop := true,
         <Receive> -> if newC = {} then hasUnrestrictedRecv := true
      end
    )
  );

  -- cases in which may remain quiescent for sure
  if not hasSend or hasUnrestrictedStop or hasUnrestrictedRecv then
       return mk_(E, newV, TrueExp, TrueExp);

  -- other cases
  let n = len t + 1,
      preE = {C |-> elimVarsAfter(len t, C) | mk_(e, C) in set E & e.type <> <Stop>},
      maxE = {C |-> {mk_DC(n, j, d) | mk_DC((n), j, d) in set C & j < n}
               | mk_(e, C) in set E & e.type <> <Stop>},

      A = mkOrExp({mk_AndExp(C) | mk_(e, C) in set E & e.type = <Stop>}),

      -- for all emission candidates 's', if 's' is enabled, then there is at least
      -- on reception event 'r' such that 'r' is enabled and deadline(r) <= deadline(s)
      B = mkAndExp({
            mkOrExp({
              mkNotExp(mk_AndExp(preE(Cs))),
              mkOrExp({
                mkAndExp({
                  mk_AndExp(preE(Cr)),
                  mkAndExp({mkOrExp2({mk_DC(js,jr,dr-ds)| mk_DC(-,js,ds) in set
maxE(Cs)})
                           | mk_DC(-, jr, dr) in set maxE(Cr)})
                })
              | mk_(r, Cr) in set E & r.type = <Receive>})
            })
          | mk_(s, Cs) in set E & s.type = <Send>}),

     -- for at least one emission event, it may be enabled and deadline is met
     C = mkOrExp2({mk_AndExp(preE(Cs) union maxE(Cs))
                 | mk_(s, Cs) in set E & s.type = <Send>})
  in
    return mk_(E, newV, mkOrExp({A,B}), mkOrExp({A,B,C}));
);

-- Truncates an invalid tc-trace (t,c) to the shortest invalid sub-trace,
-- to facilitate error diagnosis, given the V of valid tc-traces.
operations
private static pure truncateOnError: (set of TCTrace) * TCTrace ==> Trace
truncateOnError(V, mk_(t, c)) == (
  dcl t1 : Trace := t;
  dcl c1 : DCExp := c;
  dcl res : Trace := t;
  while t1 <> [] do (
     -- truncate removing last event
     t1 := t1(1,..., len t1 - 1);
     c1 := elimVarsAfter(len t1, c1);

     -- if this is a valid subtrace, at least partially, then stop
     if exists mk_(vt, vc) in set V &
           len t1 <= len vt and eqIgnTimestamps(t1, vt(1,...,len t1))
           and sat(mkAndExp({c1} union elimVarsAfter(len t1, vc.args)))
     then
        return res;
     res := t1
  );
  return res;
);


end Controllability
```

8. Class TestCases (Test Cases)

```
/**
 * Test cases.
 */

class TestCases is subclass of Controllability

operations

-- Simulates assertion checking by reducing it to pre-condition checking.
-- If 'arg' does not hold, a post-condition violation will be signaled.
protected assertTrue: bool ==> ()
assertTrue(arg) ==
      return
post arg;

-- Simulates assertion checking by reducing it to post-condition checking.
-- If values are not equal, prints a message in the console and generates
-- a post-conditions violation.
protected assertEqual: ? * ? ==> ()
assertEqual(expected, actual) ==
      if expected <> actual then (
            IO`print("Actual value (");
            IO`print(actual);
            IO`print(") different from expected (");
            IO`print(expected);
            IO`println(")\n")
      )
post expected = actual;

-- Simple scenario.
public testSimple() ==
(
  let l1 = mk_Lifeline("L1"),
      l2 = mk_Lifeline("L2"),
      m1 = mkMessage(1, mk_(l1, 1), mk_(l2, 1), "m1"),
      m2 = mkMessage(2, mk_(l2, 2), mk_(l1, 2), "m2"),
      sd1 = mkInteraction({l1, l2}, {m1, m2}, {})
  in
  (
    assertEqual({[s(m1), r(m1), s(m2), r(m2)]}, validTraces(sd1));
    assertEqual({}, uncheckableLocally(sd1));
      assertEqual({}, unintendedTraces(sd1));
      assertEqual(<Pass>, finalConformanceChecking(sd1,
                           {l1 |-> [s(m1), r(m2)], l2 |-> [r(m1), s(m2)]}));
    assertEqual({}, missingTraces(sd1)) ;
  )
);

public testIndepMessages() ==
(
  let l1 = mk_Lifeline("L1"),
      l2 = mk_Lifeline("L2"),
      l3 = mk_Lifeline("L3"),
      l4 = mk_Lifeline("L4"),
      m1 = mkMessage(1, mk_(l1, 1), mk_(l2, 1), "m1"),
      m2 = mkMessage(2, mk_(l3, 1), mk_(l4, 1), "m2"),
      sd1 = mkInteraction({l1, l2, l3, l4}, {m1, m2}, {}),
      e1 = s(m1),
      e2 = r(m1),
      e3 = s(m2),
      e4 = r(m2)
  in
  (
    assertEqual({[e1, e2, e3, e4], [e1, e3, e2, e4], [e1, e3, e4, e2], [e3, e1, e2, e4],
                                             [e3, e1, e4, e2], [e3, e4, e1, e2]},
validTraces(sd1));
    assertTrue(isLocallyObservable(sd1));
      assertEqual({}, unintendedTraces(sd1));
    assertEqual({}, missingTraces(sd1)) ;
  )
);

public testOpt() ==
(
  let l1 = mk_Lifeline("L1"),
      l2 = mk_Lifeline("L2"),
      m1 = mkMessage(1, mk_(l1, 2), mk_(l2, 2), "m1"),
      o1 = mk_InteractionOperand(nil, {mk_(l1, 1), mk_(l2, 1)}, {mk_(l1, 3), mk_(l2,
3)}),
      f1 = mk_CombinedFragment(<opt>, [o1], {l1, l2}),
      sd1 = mkInteraction({l1, l2}, {m1}, {f1}),
      e1 = s(m1),
      e2 = r(m1)
  in
  (
    assertEqual({[e1, e2], []}, validTraces(sd1));
    assertEqual({[e1]}, uncheckableLocally(sd1));
    assertEqual({}, unintendedTraces(sd1));
    assertEqual(<Pass>, finalConformanceChecking(sd1, {l1 |-> [], l2 |-> []}));
    assertEqual(<Pass>, finalConformanceChecking(sd1, {l1 |-> [e1], l2 |-> [e2]}));
    assertEqual(<Fail>, finalConformanceChecking(sd1, {l1 |-> [e1], l2 |-> []}));
    assertEqual({}, missingTraces(sd1)) ;
  )
);

public testAlt() ==
(
  let l1 = mk_Lifeline("L1"),
      l2 = mk_Lifeline("L2"),
      m1 = mkMessage(1, mk_(l1, 2), mk_(l2, 2), "m1"),
      m2 = mkMessage(2, mk_(l1, 4), mk_(l2, 4), "m2"),
      o1 = mk_InteractionOperand(nil, {mk_(l1, 1), mk_(l2, 1)}, {mk_(l1, 3), mk_(l2,
3)}),
      o2 = mk_InteractionOperand(nil, {mk_(l1, 3), mk_(l2, 3)}, {mk_(l1, 5), mk_(l2,
5)}),
      f1 = mk_CombinedFragment(<alt>, [o1, o2], {l1, l2}),
      sd1 = mkInteraction({l1, l2}, {m1, m2}, {f1}),
      e1 = s(m1),
      e2 = r(m1),
      e3 = s(m2),
      e4 = r(m2)
  in
  (
    assertEqual({[e1, e2], [e3, e4]}, validTraces(sd1));
    assertEqual({}, uncheckableLocally(sd1));
```

```
      assertTrue(isLocallyControllable(sd1));
      assertEqual(<Pass>, finalConformanceChecking(sd1, {l1 |-> [e1], l2 |-> [e2]}));
    assertEqual({}, missingTraces(sd1));
  )
);

public testStrict() ==
(
  let l1 = mk_Lifeline("L1"),
      l2 = mk_Lifeline("L2"),
      l3 = mk_Lifeline("L3"),
      m1 = mkMessage(1, mk_(l1, 2), mk_(l2, 2), "m1"),
      m2 = mkMessage(2, mk_(l3, 4), mk_(l2, 4), "m2"),
      o1 = mk_InteractionOperand(nil, {mk_(l1, 1), mk_(l2, 1), mk_(l3, 1)}, {mk_(l1, 3),
mk_(l2, 3), mk_(l3, 3)}),
      o2 = mk_InteractionOperand(nil, {mk_(l1, 3), mk_(l2, 3), mk_(l3, 3)}, {mk_(l1, 5),
mk_(l2, 5), mk_(l3, 5)}),
      f1 = mk_CombinedFragment(<strict>, [o1, o2], {l1, l2, l3}),
      sd1 = mkInteraction({l1, l2, l3}, {m1, m2}, {f1}),
      e1 = s(m1),
      e2 = r(m1),
      e3 = s(m2),
      e4 = r(m2)
  in
  (
    assertEqual({[e1, e2, e3, e4]}, validTraces(sd1));
    assertEqual({[e1, e3, e2, e4], [e3, e1, e2, e4]}, uncheckableLocally(sd1));
      assertEqual({[e1, e3], [e3]}, unintendedTraces(sd1));
      assertEqual(<Inconclusive>, finalConformanceChecking(sd1,
                     {l1 |-> [e1], l2 |-> [e2, e4], l3 |-> [e3]}));
    assertEqual({}, missingTraces(sd1)) ;
  )
);

public testLoop() ==
(
  let l1 = mk_Lifeline("L1"),
      l2 = mk_Lifeline("L2"),
      m1 = mkMessage(1, mk_(l1, 2), mk_(l2, 2), "m1"),
      o1 = mk_InteractionOperand(mk_InteractionConstraint(1, 2, nil),
             {mk_(l1, 1), mk_(l2, 1)}, {mk_(l1, 3), mk_(l2, 3)}),
      f1 = mk_CombinedFragment(<loop>, [o1], {l1, l2}),
      sd1 = mkInteraction({l1, l2}, {m1}, {f1}),
      e1 = s(m1),
      e2 = r(m1)
  in
  (
    assertEqual({[e1, e2], [e1, e2, e1, e2], [e1, e1, e2, e2]}, validTraces(sd1));
    assertEqual({[e1, e1, e2], [e1, e2, e1]}, uncheckableLocally(sd1));
    assertEqual({}, unintendedTraces(sd1));
    assertEqual({}, missingTraces(sd1)) ;
  )
);

public testAltNested() ==
(
  let l1 = mk_Lifeline("User"),
      l2 = mk_Lifeline("Watch"),
      l3 = mk_Lifeline("Smartphone"),
      l4 = mk_Lifeline("WebServer"),
      o11 = mk_InteractionOperand(nil, {mk_(l1, 2), mk_(l2, 2), mk_(l3, 1), mk_(l4, 1)},
                                       {mk_(l1, 4), mk_(l2, 4), mk_(l3, 2), mk_(l4,
2)}),
      o12 = mk_InteractionOperand(nil, {mk_(l1, 4), mk_(l2, 4), mk_(l3, 2), mk_(l4, 2)},
                                       {mk_(l1, 6), mk_(l2, 12), mk_(l3, 11), mk_(l4,
8)}),
      f1 = mk_CombinedFragment(<alt>, [o11, o12], {l1, l2, l3, l4}),
      o21 = mk_InteractionOperand(nil, {mk_(l2, 6), mk_(l3, 4), mk_(l4, 3)},
                                       {mk_(l2, 8), mk_(l3, 6), mk_(l4, 4)}),
      o22 = mk_InteractionOperand(nil, {mk_(l2, 8), mk_(l3, 6), mk_(l4, 4)},
                                       {mk_(l2, 10), mk_(l3, 10), mk_(l4, 7)}),
      f2 = mk_CombinedFragment(<alt>, [o21, o22], {l2, l3, l4}),
      m1 = mkMessage(1, mk_(l1, 1), mk_(l2, 1), "m1"),
      m2 = mkMessage(2, mk_(l2, 3), mk_(l1, 3), "m2"),
      m3 = mkMessage(3, mk_(l2, 5), mk_(l3, 3), "m3"),
      m4 = mkMessage(4, mk_(l3, 5), mk_(l2, 7), "m4"),
      m5 = mkMessage(5, mk_(l3, 7), mk_(l4, 5), "m5"),
      m6 = mkMessage(6, mk_(l4, 6), mk_(l3, 8), "m6"),
      m7 = mkMessage(7, mk_(l3, 9), mk_(l2, 9), "m7"),
      m8 = mkMessage(8, mk_(l2, 11), mk_(l1, 5), "m8"),
      sd1 = mkInteraction({l1, l2, l3, l4}, {m1, m2, m3, m4, m5, m6, m7, m8}, {f1, f2}),

      e1 = s(m1),
      e2 = r(m1),
      e3 = s(m2),
      e4 = r(m2),
      e5 = s(m3),
      e6 = r(m3),
      e7 = s(m4),
      e8 = r(m4),
      e9 = s(m5),
      e10 = r(m5),
      e11 = s(m6),
      e12 = r(m6),
      e13 = s(m7),
      e14 = r(m7),
      e15 = s(m8),
      e16 = r(m8)
  in
  (
    assertEqual({[e1, e2, e3, e4], [e1, e2, e5, e6, e7, e8, e15, e16],
                 [e1, e2, e5, e6, e9, e10, e11, e12, e13, e14, e15, e16]},
             validTraces(sd1));
    assertTrue(isLocallyObservable(sd1));
    assertTrue(isLocallyControllable(sd1));
    assertEqual({}, missingTraces(sd1)) ;
  )
);

public testRace() ==
(
  let l1 = mk_Lifeline("L1"),
      l2 = mk_Lifeline("L2"),
      l3 = mk_Lifeline("L3"),
      m1 = mkMessage(1, mk_(l1, 1), mk_(l2, 1), "m1"),
      m2 = mkMessage(2, mk_(l3, 2), mk_(l2, 2), "m2"),
      sd1 = mkInteraction({l1, l2, l3}, {m1, m2}, {}),
      e1 = s(m1),
      e2 = r(m1),
      e3 = s(m2),
      e4 = r(m2)
  in
  (
    assertEqual({[e1, e2, e3, e4], [e1, e3, e2, e4], [e3, e1, e2, e4]},
validTraces(sd1));
    assertTrue(isLocallyObservable(sd1));
    assertEqual({[e1, e3, e4], [e3, e1, e4], [e3, e4]}, unintendedTraces(sd1));
    assertEqual({}, missingTraces(sd1)) ;
  )
);

public testRaceByMsgOvertaking() ==
(
  let l1 = mk_Lifeline("L1"),
      l2 = mk_Lifeline("L2"),
      m1 = mkMessageTimed(1, mk_(l1, 1), mk_(l2, 1), "m1"),
      m2 = mkMessageTimed(3, mk_(l1, 3), mk_(l2, 3), "m2"),
      sd1 = mkInteraction({l1, l2}, {m1, m2}, {}),
      sd2 = mkInteraction({l1, l2}, {m1, m2}, {},
             {mk_TimeConstraint(t(s(m1)), t(r(m1)), nil, 1000),
              mk_TimeConstraint(t(s(m1)), t(s(m2)), 2000, nil)}),
      e1 = s(m1),
      e2 = r(m1),
      e3 = s(m2),
      e4 = r(m2),
      c1 = mkMessageTimed(2, mk_(l2, 2), mk_(l1, 2), "c1"),
      e5 = s(c1),
      e6 = r(c1),
      sd3 = mkInteraction({l1, l2}, {m1, m2, c1}, {})
  in
  (
    assertEqual({[e1, e2, e3, e4], [e1, e3, e2, e4]}, validTraces(sd1));
    assertTrue(isLocallyObservable(sd1));
    assertEqual({[e1, e3, e4]}, unintendedTraces(sd1));
    assertEqual({}, missingTraces(sd1)) ;

    assertEqual({[e1, e2, e3, e4]}, validTraces(sd2));
    assertEqual({}, unintendedTraces(sd2));

    assertEqual({[e1, e2, e5, e6, e3, e4]}, validTraces(sd3));
    assertEqual({}, unintendedTraces(sd3));
  )
);

public testNonLocalChoice() ==
(
  let l1 = mk_Lifeline("L1"),
      l2 = mk_Lifeline("L2"),
      l3 = mk_Lifeline("L3"),
      l4 = mk_Lifeline("L4"),
      o11 = mk_InteractionOperand(nil, {mk_(l1, 1), mk_(l2, 1), mk_(l3, 1), mk_(l4, 1)},
                                       {mk_(l1, 3), mk_(l2, 3), mk_(l3, 3), mk_(l4,
3)}),
      o12 = mk_InteractionOperand(nil, {mk_(l1, 3), mk_(l2, 3), mk_(l3, 3), mk_(l4, 3)},
                                       {mk_(l1, 5), mk_(l2, 5), mk_(l3, 5), mk_(l4,
5)}),
      f1 = mk_CombinedFragment(<alt>, [o11, o12], {l1, l2, l3, l4}),
      m1 = mkMessage(1, mk_(l1, 2), mk_(l2, 2), "m1"),
      m2 = mkMessage(2, mk_(l3, 2), mk_(l4, 2), "m2"),
      m3 = mkMessage(3, mk_(l1, 4), mk_(l2, 4), "m3"),
      m4 = mkMessage(4, mk_(l3, 4), mk_(l4, 4), "m4"),
      sd1 = mkInteraction({l1, l2, l3, l4}, {m1, m2, m3, m4}, {f1}),

      e1 = mkEvent(<Send>, "m1", l1),
      e2 = mkEvent(<Receive>, "m1", l2),
      e3 = mkEvent(<Send>, "m2", l3),
      e4 = mkEvent(<Receive>, "m2", l4),
      e5 = mkEvent(<Send>, "m3", l1),
      e6 = mkEvent(<Receive>, "m3", l2),
      e7 = mkEvent(<Send>, "m4", l3),
      e8 = mkEvent(<Receive>, "m4", l4)
  in
  (
    assertEqual({[e1, e2, e3, e4], [e1, e3, e2, e4], [e1, e3, e4, e2], [e3, e1, e2, e4],
                 [e3, e1, e4, e2], [e3, e4, e1, e2], [e5, e6, e7, e8], [e5, e7, e6, e8],
                 [e5, e7, e8, e6], [e7, e5, e6, e8], [e7, e5, e8, e6], [e7, e8, e5,
e6]},
             validTraces(sd1));
    assertTrue(not isLocallyObservable(sd1));
    assertEqual({[e1, e2, e7], [e1, e7], [e7, e1], [e7, e8, e1], [e5, e6, e3], [e5, e3],
                 [e3, e5], [e3, e4, e5]},
             unintendedTraces(sd1));
    assertEqual({}, missingTraces(sd1)) ;
  )
);

public testImpossible() ==
(
  let l1 = mk_Lifeline("L1"),
      l2 = mk_Lifeline("L2"),
      m1 = mkMessage(1, mk_(l1, 2), mk_(l2, 1), "m1"),
      m2 = mkMessage(2, mk_(l2, 2), mk_(l1, 1), "m2"),
      sd1 = mkInteraction({l1, l2}, {m1, m2}, {})
  in
  (
    assertEqual({}, validTraces(sd1));
    assertEqual({ l1 |-> {}, l2 |-> {}}, projectTraces(validTraces(sd1), {l1, l2}));
    assertTrue(isLocallyObservable(sd1));
    assertEqual({[]}, unintendedTraces(sd1));
    assertEqual({}, missingTraces(sd1)) ;
  )
);

public testUnintendedEmptyTrace() ==
(
  let l1 = mk_Lifeline("L1"),
      l2 = mk_Lifeline("L2"),
      l3 = mk_Lifeline("L3"),
      l4 = mk_Lifeline("L4"),
      o11 = mk_InteractionOperand(nil, {mk_(l1, 1), mk_(l2, 1), mk_(l3, 1), mk_(l4, 1)},
                                       {mk_(l1, 3), mk_(l2, 3), mk_(l3, 3), mk_(l4,
3)}),
      o12 = mk_InteractionOperand(nil, {mk_(l1, 3), mk_(l2, 3), mk_(l3, 3), mk_(l4, 3)},
                                       {mk_(l1, 5), mk_(l2, 5), mk_(l3, 5), mk_(l4,
5)}),
      f1 = mk_CombinedFragment(<alt>, [o11, o12], {l1, l2, l3, l4}),
      m1 = mkMessage(1, mk_(l1, 2), mk_(l2, 2), "m1"),
      m2 = mkMessage(2, mk_(l3, 4), mk_(l4, 4), "m2"),
      sd1 = mkInteraction({l1, l2, l3, l4}, {m1, m2}, {f1}),

      e1 = mkEvent(<Send>, "m1", l1),
      e2 = mkEvent(<Receive>, "m1", l2),
      e3 = mkEvent(<Send>, "m2", l3),
      e4 = mkEvent(<Receive>, "m2", l4)
  in
  (
    assertEqual({[e1, e2], [e3, e4]}, validTraces(sd1));
    assertTrue(not isLocallyObservable(sd1));
    assertEqual({[], [e1, e2, e3], [e1, e3], [e3, e4, e1], [e3, e1]},
                unintendedTraces(sd1));
    assertEqual({}, missingTraces(sd1)) ;
  )
);

public testUnintendedEmptyTrace2() ==
(
  let l1 = mk_Lifeline("L1"),
      l2 = mk_Lifeline("L2"),
      l3 = mk_Lifeline("L3"),
      o11 = mk_InteractionOperand(nil, {mk_(l1, 1), mk_(l2, 1), mk_(l3, 1)},
                                       {mk_(l1, 3), mk_(l2, 3), mk_(l3, 3)}),
      o12 = mk_InteractionOperand(nil, {mk_(l1, 3), mk_(l2, 3), mk_(l3, 3)},
                                       {mk_(l1, 5), mk_(l2, 5), mk_(l3, 5)}),
      f1 = mk_CombinedFragment(<alt>, [o11, o12], {l1, l2, l3}),
      m1 = mkMessage(1, mk_(l1, 2), mk_(l2, 2), "m1"),
      m2 = mkMessage(2, mk_(l3, 4), mk_(l2, 4), "m1"),
      sd1 = mkInteraction({l1, l2, l3}, {m1, m2}, {f1}),

      e1 = mkEvent(<Send>, "m1", l1),
      e2 = mkEvent(<Receive>, "m1", l2),
      e3 = mkEvent(<Send>, "m1", l3)
  in
  (
    assertEqual({[e1, e2], [e3, e2]}, validTraces(sd1));
    assertEqual({[e1, e2, e3], [e1, e3, e2], [e3, e1, e2], [e3, e2, e1]},
uncheckableLocally(sd1));
    assertEqual({[], [e1, e2, e3], [e1, e3], [e3, e1], [e3, e2, e1]},
unintendedTraces(sd1));
    assertEqual({}, missingTraces(sd1)) ;
  )
);

-- Example with unintendedTrace with invalidStop but not other problems (at least one
sends).
public testUnintendedEmptyTrace3() ==
(
  let l1 = mk_Lifeline("L1"),
      l2 = mk_Lifeline("L2"),
      l3 = mk_Lifeline("L3"),
      o11 = mk_InteractionOperand(nil, {mk_(l1, 1), mk_(l2, 1), mk_(l3, 1)},
                                       {mk_(l1, 3), mk_(l2, 3), mk_(l3, 3)}),
      o12 = mk_InteractionOperand(nil, {mk_(l1, 3), mk_(l2, 3), mk_(l3, 3)},
                                       {mk_(l1, 5), mk_(l2, 5), mk_(l3, 5)}),
      o13 = mk_InteractionOperand(nil, {mk_(l1, 5), mk_(l2, 5), mk_(l3, 5)},
                                       {mk_(l1, 11), mk_(l2, 11), mk_(l3, 11)}),
      f1 = mk_CombinedFragment(<alt>, [o11, o12, o13], {l1, l2, l3}),
      o21 = mk_InteractionOperand(nil, {mk_(l2, 6)}, {mk_(l2, 8)}),
      o22 = mk_InteractionOperand(nil, {mk_(l2, 8)}, {mk_(l2, 10)}),
      f2 = mk_CombinedFragment(<par>, [o21, o22], {l2}),
      m1 = mkMessage(1, mk_(l1, 2), mk_(l2, 2), "m1"),
      m2 = mkMessage(2, mk_(l3, 4), mk_(l2, 4), "m1"),
      m3 = mkMessage(3, mk_(l1, 7), mk_(l2, 7), "m1"),
      m4 = mkMessage(4, mk_(l3, 9), mk_(l2, 9), "m1"),
      sd1 = mkInteraction({l1, l2, l3}, {m1, m2, m3, m4}, {f1, f2}),

      e1 = mkEvent(<Send>, "m1", l1),
      e2 = mkEvent(<Receive>, "m1", l2),
      e3 = mkEvent(<Send>, "m1", l3)
  in
  (
    assertEqual({[e1, e2], [e3, e2], [e1, e2, e3, e2], [e3, e2, e1, e2],
                 [e3, e1, e2, e2], [e1, e3, e2, e2]}, validTraces(sd1));
    assertEqual(false, isLocallyObservable(sd1));
    assertEqual({[]}, unintendedTraces(sd1));
    assertEqual({}, missingTraces(sd1)) ;
    -- violates assumption of biunivoca relation between send and receive events
  )
);

public testWhoSends() ==
(
  let l1 = mk_Lifeline("L1"),
      l2 = mk_Lifeline("L2"),
      o11 = mk_InteractionOperand(nil, {mk_(l1, 1), mk_(l2, 1)},
                                       {mk_(l1, 3), mk_(l2, 3)}),
      o12 = mk_InteractionOperand(nil, {mk_(l1, 3), mk_(l2, 3)},
                                       {mk_(l1, 5), mk_(l2, 5)}),
      f1 = mk_CombinedFragment(<alt>, [o11, o12], {l1, l2}),
      m1 = mkMessage(1, mk_(l1, 2), mk_(l2, 2), "m1"),
      m2 = mkMessage(2, mk_(l2, 4), mk_(l1, 4), "m2"),
      sd1 = mkInteraction({l1, l2}, {m1, m2}, {f1}),

      e1 = mkEvent(<Send>, "m1", l1),
      e2 = mkEvent(<Receive>, "m1", l2),
      e3 = mkEvent(<Send>, "m2", l2),
      e4 = mkEvent(<Receive>, "m2", l1)
  in
  (
    assertEqual({[e1, e2], [e3, e4]}, validTraces(sd1));
    assertEqual({[e1, e3], [e3, e1]}, uncheckableLocally(sd1)); -- both send but
messages are lost
    assertEqual({[], [e1, e3], [e3, e1]}, unintendedTraces(sd1));
    assertEqual({}, missingTraces(sd1)) ;
  )
);

public testTimeConstraint() ==
(
  let l1 = mk_Lifeline("L1"),
      l2 = mk_Lifeline("L2"),
      t1 = mk_Variable("t1"),
      t2 = mk_Variable("t2"),
      t3 = mk_Variable("t3"),
      t4 = mk_Variable("t4"),
      m1 = mk_Message(1, mk_(l1, 1), mk_(l2, 1), "m1", t1, t2, <Asynch>, nil),
      m2 = mk_Message(2, mk_(l2, 2), mk_(l1, 2), "m2", t3, t4, <Asynch>, nil),
      sd1 = mkInteraction({l1, l2}, {m1, m2}, {},
                      {mk_TimeConstraint(t2, t3, 0, 2),
                       mk_TimeConstraint(t1, t4, 0, 5)}),
      e1 = mk_Event(<Send>, "m1", l1, t1),
      e2 = mk_Event(<Receive>, "m1", l2, t2),
      e3 = mk_Event(<Send>, "m2", l2, t3),
      e4 = mk_Event(<Receive>, "m2", l1, t4),

      te1 = mk_Event(<Send>, "m1", l1, 1),
      te2 = mk_Event(<Receive>, "m1", l2, 2),
      te3 = mk_Event(<Send>, "m2", l2, 3),
      te4a = mk_Event(<Receive>, "m2", l1, 6),
      te4b = mk_Event(<Receive>, "m2", l1, 7)

  in
  (
    assertEqual({[e1, e2, e3, e4]}, validTraces(sd1));

    assertEqual({}, uncheckableLocally(sd1));
      assertEqual({[e1, e2, e3, e4]}, unintendedTraces(sd1));
      assertEqual(<Pass>, finalConformanceChecking(sd1,
                                   {l1 |-> [e1, e4], l2 |-> [e2, e3]}));

      assertEqual(true, timedCheckNextEvent([te1], te4a, projectTraces(validTraces(sd1),
l1), getTimeConstraints(sd1, l1)));
      assertEqual(false, timedCheckNextEvent([te1], te4b,
projectTraces(validTraces(sd1), l1), getTimeConstraints(sd1, l1)));

      assertEqual({mk_(e3, mk_(2, 4))}, nextSendEventsTimed([te2],
projectTraces(validTraces(sd1), l2), getTimeConstraints(sd1, l2)));

             assertEqual(<Pass>, timedFinalConformanceChecking(sd1, {l1 |-> [te1, te4a],
l2 |-> [te2, te3]}));
             assertEqual(<Fail>, timedFinalConformanceChecking(sd1, {l1 |-> [te1, te4b],
l2 |-> [te2, te3]}));

    assertEqual({}, missingTraces(sd1)) ;
  )
);

-- Test case to check that, in the presence of multiple timed events refering to the
same timestamp variable
-- (e.g., in a loop), it is the the most recent occurrence that prevails
public testTimeConstraintInLoop() ==
(
  let l1 = mk_Lifeline("L1"),
      l2 = mk_Lifeline("L2"),
      t1 = mk_Variable("t1"),
      t2 = mk_Variable("t2"),
      t3 = mk_Variable("t3"),
      t4 = mk_Variable("t4"),

      m1 = mk_Message(1, mk_(l1, 2), mk_(l2, 2), "m1", t1, t2, <Asynch>, nil),
      m2 = mk_Message(2, mk_(l2, 3), mk_(l1, 3), "m2", t3, t4, <Asynch>, nil),
      o1 = mk_InteractionOperand(mk_InteractionConstraint(1, 2, nil),
                     {mk_(l1, 1), mk_(l2, 1)}, {mk_(l1, 4), mk_(l2, 4)}),
      f1 = mk_CombinedFragment(<loop>, [o1], {l1, l2}),
      sd1 = mkInteraction({l1, l2}, {m1, m2}, {f1},
                      {mk_TimeConstraint(t2, t3, nil, 2),
                       mk_TimeConstraint(t1, t4, nil, 5)}),
      e1 = mk_Event(<Send>, "m1", l1, t1),
      e2 = mk_Event(<Receive>, "m1", l2, t2),
      e3 = mk_Event(<Send>, "m2", l2, t3),
      e4 = mk_Event(<Receive>, "m2", l1, t4),

      te1 = mk_Event(<Send>, "m1", l1, 1),
      te2 = mk_Event(<Receive>, "m1", l2, 2),
      te3 = mk_Event(<Send>, "m2", l2, 3),
      te4a = mk_Event(<Receive>, "m2", l1, 6),
      te4b = mk_Event(<Receive>, "m2", l1, 7),

      te21 = mk_Event(<Send>, "m1", l1, 11),
      te22 = mk_Event(<Receive>, "m1", l2, 12),
      te23 = mk_Event(<Send>, "m2", l2, 13),
      te24a = mk_Event(<Receive>, "m2", l1, 16),
      te24b = mk_Event(<Receive>, "m2", l1, 17)

  in
  (
    assertEqual({[e1, e2, e3, e4], [e1, e2, e3, e4, e1, e2, e3, e4]}, validTraces(sd1));
    assertTrue(isLocallyObservable(sd1));
    assertEqual({[e1, e2, e3, e4], [e1, e2, e3, e4, e1, e2, e3, e4]},
unintendedTraces(sd1));

      assertEqual(true, timedCheckNextEvent([te1], te4a, projectTraces(validTraces(sd1),
l1), getTimeConstraints(sd1, l1)));
      assertEqual(false, timedCheckNextEvent([te1], te4b,
projectTraces(validTraces(sd1), l1), getTimeConstraints(sd1, l1)));

      assertEqual({mk_(e3, mk_(nil, 4))}, nextSendEventsTimed([te2],
projectTraces(validTraces(sd1), l2), getTimeConstraints(sd1, l2)));

             assertEqual(<Pass>, timedFinalConformanceChecking(sd1, {l1 |-> [te1, te4a],
l2 |-> [te2, te3]}));
             assertEqual(<Fail>, timedFinalConformanceChecking(sd1, {l1 |-> [te1, te4b],
l2 |-> [te2, te3]}));

             assertEqual(<Pass>, timedFinalConformanceChecking(sd1, {l1 |-> [te1, te4a,
te21, te24a], l2 |-> [te2, te3, te22, te23]}));
             assertEqual(<Fail>, timedFinalConformanceChecking(sd1, {l1 |-> [te1, te4a,
te21, te24b], l2 |-> [te2, te3, te22, te23]}));

    assertEqual({}, missingTraces(sd1)) ;
  )
);

public testInterLifelineTimeConstraints() ==
(
  let l1 = mk_Lifeline("L1"),
      l2 = mk_Lifeline("L2"),
      t1 = mk_Variable("t1"),
      t2 = mk_Variable("t2"),
      t3 = mk_Variable("t3"),
      t4 = mk_Variable("t4"),
      m1 = mk_Message(1, mk_(l1, 1), mk_(l2, 1), "m1", t1, t2, <Asynch>, nil),
      m2 = mk_Message(2, mk_(l2, 2), mk_(l1, 2), "m2", t3, t4, <Asynch>, nil),
      sd1 = mkInteraction({l1, l2}, {m1, m2}, {},
                      {mk_TimeConstraint(t1, t2, 0, 2000),
                       mk_TimeConstraint(t2, t3, 0, 2000),
                       mk_TimeConstraint(t3, t4, 0, 2000),
                       mk_TimeConstraint(t1, t4, 0, 5000)}),
      e1 = mk_Event(<Send>, "m1", l1, t1),
      e2 = mk_Event(<Receive>, "m1", l2, t2),
      e3 = mk_Event(<Send>, "m2", l2, t3),
      e4 = mk_Event(<Receive>, "m2", l1, t4),

      te1 = mk_Event(<Send>, "m1", l1, 1000),
      te2a = mk_Event(<Receive>, "m1", l2, 2000),
      te2b = mk_Event(<Receive>, "m1", l2, 4000),
      te3 = mk_Event(<Send>, "m2", l2, 4000),
      te4a = mk_Event(<Receive>, "m2", l1, 6000 - 10),
      te4b = mk_Event(<Receive>, "m2", l1, 7000),
      te4c = mk_Event(<Receive>, "m2", l1, 6000)

  in
  (
    assertEqual({[e1, e2, e3, e4]}, validTraces(sd1));
      assertEqual({[e1, e2, e3, e4]}, unintendedTraces(sd1));
    assertEqual({[e1, e2, e3, e4]}, uncheckableLocally(sd1));
      assertEqual(<Pass>, finalConformanceChecking(sd1, {l1 |-> [e1, e4], l2 |-> [e2,
e3]}));

      assertEqual(true, timedCheckNextEvent([te1], te4a, projectTraces(validTraces(sd1),
l1), getTimeConstraints(sd1, l1)));
      assertEqual(false, timedCheckNextEvent([te1], te4b,
projectTraces(validTraces(sd1), l1), getTimeConstraints(sd1, l1)));

      assertEqual({mk_(e3, mk_(2000, 4000))}, nextSendEventsTimed([te2a],
projectTraces(validTraces(sd1), l2), getTimeConstraints(sd1, l2)));

             assertEqual(<Pass>, timedFinalConformanceChecking(sd1, {l1 |-> [te1, te4a],
l2 |-> [te2a, te3]}));
             assertEqual(<Fail>, timedFinalConformanceChecking(sd1, {l1 |-> [te1, te4a],
l2 |-> [te2b, te3]}));
             assertEqual(<Fail>, timedFinalConformanceChecking(sd1, {l1 |-> [te1, te4b],
l2 |-> [te2a, te3]}));

             assertEqual(<Inconclusive>, timedFinalConformanceChecking(sd1, {l1 |->
[te1, te4c], l2 |-> [te2a, te3]}));

    assertEqual({}, missingTraces(sd1)) ;
  )
);

public testVerdictWithTimestamps() ==
(
  let l1 = mk_Lifeline("L1"),
      l2 = mk_Lifeline("L2"),
      l3 = mk_Lifeline("L3"),
      m1 = mkMessage(1, mk_(l1, 2), mk_(l2, 2), "m1"),
      m2 = mkMessage(2, mk_(l3, 4), mk_(l2, 4), "m2"),
      o1 = mk_InteractionOperand(nil, {mk_(l1, 1), mk_(l2, 1), mk_(l3, 1)}, {mk_(l1, 3),
mk_(l2, 3), mk_(l3, 3)}),
      o2 = mk_InteractionOperand(nil, {mk_(l1, 3), mk_(l2, 3), mk_(l3, 3)}, {mk_(l1, 5),
mk_(l2, 5), mk_(l3, 5)}),
      f1 = mk_CombinedFragment(<strict>, [o1, o2], {l1, l2, l3}),
      sd1 = mkInteraction({l1, l2, l3}, {m1, m2}, {f1}),
      e1 = mkEvent(<Send>, "m1", l1),
      e2 = mkEvent(<Receive>, "m1", l2),
      e3 = mkEvent(<Send>, "m2", l3),
      e4 = mkEvent(<Receive>, "m2", l2),
      te1 = mkEvent(<Send>, "m1", l1, 10),
      te2 = mkEvent(<Receive>, "m1", l2, 20),
      te3a = mkEvent(<Send>, "m2", l3, 20 + MaxClockSkew),
      te3b = mkEvent(<Send>, "m2", l3, 20 + MaxClockSkew + 1),
      te3c = mkEvent(<Send>, "m2", l3, 20 - MaxClockSkew - 1),
      te3d = mkEvent(<Send>, "m2", l3, 20 - MaxClockSkew),
      te4 = mkEvent(<Receive>, "m2", l2, 20 + MaxClockSkew + 2)
  in
  (
      assertEqual(<Inconclusive>, finalConformanceChecking(sd1, {l1 |-> [e1], l2 |->
[e2, e4], l3 |-> [e3]}));
      assertEqual(<Inconclusive>, timedFinalConformanceChecking(sd1, {l1 |-> [te1], l2
|-> [te2, te4], l3 |-> [te3a]}));
      assertEqual(<Pass>, timedFinalConformanceChecking(sd1, {l1 |-> [te1], l2 |-> [te2,
te4], l3 |-> [te3b]}));
      assertEqual(<Fail>, timedFinalConformanceChecking(sd1, {l1 |-> [te1], l2 |-> [te2,
te4], l3 |-> [te3c]}));
      assertEqual(<Inconclusive>, timedFinalConformanceChecking(sd1, {l1 |-> [te1], l2
|-> [te2, te4], l3 |-> [te3d]}));
    assertEqual({}, missingTraces(sd1)) ;
  )
);


-- Example of restricting valid traces based on time constraints.
public testFallDetection() ==
(
  let l1 = mk_Lifeline("Care Receiver"),
      l2 = mk_Lifeline("Fall Detection App"),
      l3 = mk_Lifeline("AAL4ALL Portal"),
      m1 = mkMessageTimed(1, mk_(l1, 1), mk_(l2, 1), "fall signal"),
      m2 = mkMessageTimed(2, mk_(l2, 2), mk_(l1, 2), "confirm?"),
      m3 = mkMessageTimed(3, mk_(l1, 4), mk_(l2, 4), "yes!"),
      m4 = mkMessageTimed(4, mk_(l2, 5), mk_(l3, 5), "notify fall"),
      m5 = mkMessageTimed(5, mk_(l1, 7), mk_(l2, 7), "no!"),
      m6 = mkMessageTimed(6, mk_(l2, 9), mk_(l3, 9), "notify possible fall"),
      o1 = mk_InteractionOperand(nil, {mk_(l1, 3), mk_(l2, 3), mk_(l3, 3)}, {mk_(l1, 6),
mk_(l2, 6), mk_(l3, 6)}),
      o2 = mk_InteractionOperand(nil, {mk_(l1, 6), mk_(l2, 6), mk_(l3, 6)}, {mk_(l1, 8),
mk_(l2, 8), mk_(l3, 8)}),
      o3 = mk_InteractionOperand(nil, {mk_(l1, 8), mk_(l2, 8), mk_(l3, 8)}, {mk_(l1,
10), mk_(l2, 10), mk_(l3, 10)}),
      f1 = mk_CombinedFragment(<alt>, [o1, o2, o3], {l1, l2, l3}),
      tcs =  {mk_TimeConstraint(t(s(m2)), t(r(m2)), 0, 1000),
                       mk_TimeConstraint(t(s(m3)), t(r(m3)), 0, 1000),
                       mk_TimeConstraint(t(s(m5)), t(r(m5)), 0, 1000),
                       mk_TimeConstraint(t(r(m2)), t(s(m3)), 0, 10000),
                       mk_TimeConstraint(t(r(m2)), t(s(m5)), 0, 10000),
                       mk_TimeConstraint(t(s(m2)), t(s(m6)), 13000, nil)},
      derivedTC = {mk_TimeConstraint(t(s(m2)), t(r(m3)), 0, 12000),
                            mk_TimeConstraint(t(s(m2)), t(r(m5)), 0, 12000) },
      sd1 = mkInteraction({l1, l2, l3}, {m1, m2, m3, m4, m5, m6}, {f1}, tcs),
      e1 = s(m1),
      e2 = r(m1),
      e3 = s(m2),
      e4 = r(m2),
      e5 = s(m3),
      e6 = r(m3),
      e7 = s(m4),
      e8 = r(m4),
      e9 = s(m5),
      e10 = r(m5),
      e11 = s(m6),
      e12 = r(m6),

      e1a = mk_Event(<Send>, "fall signal", l1, 0),
      e2a = mk_Event(<Receive>, "fall signal", l2, 2000),
      e3a = mk_Event(<Send>, "confirm?", l2, 4000),
      e4a = mk_Event(<Receive>, "confirm?", l1, 4200),
      e5a = mk_Event(<Send>, "yes!", l1, 14200),
      e6a = mk_Event(<Receive>, "yes!", l2, 14500),
      e7a = mk_Event(<Send>, "notify fall", l2, 14600),
      e8a = mk_Event(<Receive>, "notify fall", l3, 16000),

      e6b = mk_Event(<Receive>, "yes!", l2, 15200),
      e7b = mk_Event(<Send>, "notify fall", l2, 15600),

      e6c = mk_Event(<Receive>, "yes!", l2, 18000),
      e7c = mk_Event(<Send>, "notify fall", l2, 18600),
      e8c = mk_Event(<Receive>, "notify fall", l3, 19000),

      e4d = mk_Event(<Receive>, "confirm?", l1, 16800),
      e11d = mk_Event(<Send>, "notify possible fall", l2, 17000),
      e12d = mk_Event(<Receive>, "notify possible fall", l3, 18000)

  in
  (
    -- derived time constraints
   -- assertTrue(derivedTC subset sd1.timeConstraints);

      -- time constraints ensure that, in the third case, "notify all" is sent after
"Confirm?" is received by the user.
    assertEqual({[e1, e2, e3, e4, e5, e6, e7, e8], [e1, e2, e3, e4, e9, e10],  [e1, e2,
e3, e4, e11, e12]}, validTraces(sd1));

    MaxClockSkew := 500;
    assertEqual({[e1a, e2a, e3a, e4a, e5a, e6a, e7a, e8a]},
      joinActualTraces([], {l1 |-> [e1a, e4a, e5a], l2 |-> [e2a, e3a, e6a, e7a], l3 |->
[e8a]}));
    assertEqual(<Pass>, timedFinalConformanceChecking(sd1, {l1 |-> [e1a, e4a, e5a], l2
|-> [e2a, e3a, e6a, e7a], l3 |-> [e8a]}));
    assertEqual(<Inconclusive>, timedFinalConformanceChecking(sd1, {l1 |-> [e1a, e4a,
e5a], l2 |-> [e2a, e3a, e6b, e7b], l3 |-> [e8a]}));
    assertEqual(<Fail>, timedFinalConformanceChecking(sd1, {l1 |-> [e1a, e4a, e5a], l2
|-> [e2a, e3a, e6c, e7c], l3 |-> [e8c]}));

    assertEqual({[e1a, e2a, e3a, e4d, e11d, e12d], [e1a, e2a, e3a, e11d, e4d, e12d]},
      joinActualTraces([], {l1 |-> [e1a, e4d], l2 |-> [e2a, e3a, e11d], l3 |->
[e12d]}));
    assertEqual(<Fail>, timedFinalConformanceChecking(sd1, {l1 |-> [e1a, e4d], l2 |->
[e2a, e3a, e11d], l3 |-> [e12d]}));
    MaxClockSkew := 10;
    assertEqual({}, missingTraces(sd1)) ;

    assertEqual({}, unintendedTraces(sd1));
  )
);

-- Example of restricting valid traces based on time constraints.
public testIsLocallyObservableTimed() ==
(
  let l1 = mk_Lifeline("L1"),
      l2 = mk_Lifeline("L2"),
      l3 = mk_Lifeline("L3"),
      l4 = mk_Lifeline("L4"),
      m1 = mkMessageTimed(1, mk_(l2, 2), mk_(l1, 2), "m1"),
      m2 = mkMessageTimed(2, mk_(l2, 4), mk_(l3, 4), "m2"),
      m3 = mkMessageTimed(3, mk_(l3, 6), mk_(l4, 6), "m3"),
      m4 = mkMessageTimed(4, mk_(l4, 7), mk_(l3, 7), "m4"),
      m5 = mkMessageTimed(5, mk_(l3, 10), mk_(l2, 10), "m5"),
      o1 = mk_InteractionOperand(nil, {mk_(l1, 1), mk_(l2, 1)}, {mk_(l1, 3), mk_(l2,
3)}),
      o2 = mk_InteractionOperand(nil, {mk_(l3, 5), mk_(l4, 5)}, {mk_(l3, 8), mk_(l4,
8)}),
      o3 = mk_InteractionOperand(nil, {mk_(l2, 9), mk_(l3, 9)}, {mk_(l2, 11), mk_(l3,
11)}),
      f1 = mk_CombinedFragment(<opt>, [o1], {l1, l2}),
      f2 = mk_CombinedFragment(<opt>, [o2], {l3, l4}),
      f3 = mk_CombinedFragment(<opt>, [o3], {l2, l3}),
      sd1 = mkInteraction({l1, l2, l3, l4}, {m1, m2, m3, m4, m5}, {f1, f2, f3},
                      {mk_TimeConstraint(t(s(m1)), t(r(m1)), nil, 1000),
                       mk_TimeConstraint(t(s(m1)), t(s(m2)), 2000, nil),
                       mk_TimeConstraint(t(r(m3)), t(s(m4)), 10000,nil),
                       mk_TimeConstraint(t(s(m1)), t(r(m5)), nil, 5000)}),
      e1 = s(m1),
      e2 = r(m1),
      e3 = s(m2),
      e4 = r(m2),
      e5 = s(m3),
      e6 = r(m3),
      e7 = s(m4),
      e8 = r(m4),
      e9 = s(m5),
      e10 = r(m5)
  in
  (
    assertEqual({[e1, e2, e3, e4], [e1, e2, e3, e4, e5, e6, e7, e8], [e1, e2, e3, e4,
e9, e10],
      [e3, e4], [e3, e4, e5, e6, e7, e8],  [e3, e4, e5, e6, e7, e8, e9, e10], [e3, e4,
e9, e10]},
      validTraces(sd1));

    let t = [e1, e2, e3, e4, e5, e6, e7, e8, e9, e10] in
    (
      assertTrue(t  not in set uncheckableLocally(sd1));
      assertTrue(t in set uncheckableLocallyUntimed(sd1))
    );

    assertTrue( {[e1, e3, e2, e4], [e1, e3, e4, e2]} subset uncheckableLocally(sd1));
    assertTrue( {[e1, e3, e2, e4], [e1, e3, e4, e2]} subset
uncheckableLocallyUntimed(sd1));
    assertEqual({}, missingTraces(sd1));
    assertEqual({}, missingTraces(sd1));
  )
);

-- Example of non-controllability because of reception constraint.
public testNonLocallyControlableTimed() ==
(
  let l1 = mk_Lifeline("L1"),
      l2 = mk_Lifeline("L2"),
      m1 = mkMessageTimed(1, mk_(l1, 1), mk_(l2, 1), "m1"),
      m2 = mkMessageTimed(2, mk_(l2, 2), mk_(l1, 2), "m2"),
      sd1 = mkInteraction({l1, l2}, {m1, m2}, {}, {mk_TimeConstraint(t(s(m1)), t(r(m2)),
0, 1000)})
  in
  (
    assertEqual({[s(m1), r(m1), s(m2), r(m2)]}, validTraces(sd1));
    assertTrue(isLocallyObservable(sd1));
    assertEqual( {[s(m1), r(m1), s(m2), r(m2)]}, unintendedTraces(sd1));
    assertTrue(not isLocallyControllable(sd1));
    assertEqual({}, missingTraces(sd1))
  )
);

-- Example of non-controllability because of reception constraint.
public testStrangeControllableTimed() ==
(
  let l1 = mk_Lifeline("L1"),
      l2 = mk_Lifeline("L2"),

      m1 = mkMessageTimed(1, mk_(l1, 1), mk_(l2, 1), "m1"),
      m2 = mkMessageTimed(2, mk_(l1, 3), mk_(l2, 3), "m2"),
      m3 = mkMessageTimed(3, mk_(l2, 4), mk_(l1, 4), "m3"),
      m4 = mkMessageTimed(4, mk_(l2, 7), mk_(l1, 7), "m4"),

      o1 = mk_InteractionOperand(nil, {mk_(l1, 2), mk_(l2, 2)}, {mk_(l1, 5), mk_(l2,
5)}),
      o2 = mk_InteractionOperand(nil, {mk_(l1, 6), mk_(l2, 6)}, {mk_(l1, 8), mk_(l2,
8)}),
      f1 = mk_CombinedFragment(<opt>, [o1], {l1, l2}),
      f2 = mk_CombinedFragment(<opt>, [o2], {l1, l2}),

      sd1 = mkInteraction({l1, l2}, {m1, m2, m3, m4}, {f1, f2},
      {mk_TimeConstraint(t(s(m1)), t(r(m1)), 0, 1000),
       mk_TimeConstraint(t(s(m4)), t(r(m4)), 0, 1000),
       mk_TimeConstraint(t(s(m1)), t(s(m2)), 7000, nil),
       mk_TimeConstraint(t(r(m1)), t(s(m4)), 0, 4000),
       mk_TimeConstraint(t(s(m2)), t(r(m3)), 0, 5000)}),
      e1 = s(m1),
      e2 = r(m1),
      e3 = s(m2),
      e4 = r(m2),
      e5 = s(m3),
      e6 = r(m3),
      e7 = s(m4),
      e8 = r(m4)
  in
  (
    assertEqual({[e1, e2], [e1, e2, e3, e4, e5, e6], [e1, e2, e7, e8]},
validTraces(sd1));
    assertEqual( {[e1, e2, e3, e4, e5, e6]}, unintendedTraces(sd1));
    assertEqual({}, missingTraces(sd1))
  )
);

-- Example of non-controllability because of reception constraint.
public testSendableFirst() ==
(
  let l1 = mk_Lifeline("L1"),
      l2 = mk_Lifeline("L2"),
      l3 = mk_Lifeline("L3"),

      m1 = mkMessageTimed(1, mk_(l1, 1), mk_(l2, 1), "m1"),
      m2 = mkMessageTimed(2, mk_(l1, 2), mk_(l3, 2), "m2"),
      m3 = mkMessageTimed(3, mk_(l2, 3), mk_(l3, 3), "m3"),

      sd1 = mkInteraction({l1, l2, l3}, {m1, m2, m3}, {},
      {mk_TimeConstraint(t(s(m1)), t(r(m1)), 0, 1000),
       mk_TimeConstraint(t(s(m2)), t(r(m2)), 0, 1000),
       mk_TimeConstraint(t(s(m3)), t(r(m3)), 0, 1000),
       mk_TimeConstraint(t(s(m1)), t(s(m2)), 2000, 4000),
       mk_TimeConstraint(t(r(m1)), t(s(m3)), 8000, nil)})
  in
  (
    assertEqual({[s(m1), r(m1), s(m2), r(m2), s(m3), r(m3)]}, validTraces(sd1));
    assertEqual({}, unintendedTraces(sd1));
    assertEqual({}, missingTraces(sd1))
  )
);

public testSendableFirst2() ==
(
  let l1 = mk_Lifeline("L1"),
      l2 = mk_Lifeline("L2"),

      m1 = mkMessageTimed(1, mk_(l1, 1), mk_(l2, 1), "m1"),
      m2 = mkMessageTimed(2, mk_(l1, 3), mk_(l2, 3), "m2"),
      m3 = mkMessageTimed(3, mk_(l2, 5), mk_(l1, 5), "m3"),
      m4 = mkMessageTimed(4, mk_(l1, 6), mk_(l2, 6), "m4"),

      o1 = mk_InteractionOperand(nil, {mk_(l1, 2), mk_(l2, 2)}, {mk_(l1, 4), mk_(l2,
4)}),
      o2 = mk_InteractionOperand(nil, {mk_(l1, 4), mk_(l2, 4)}, {mk_(l1, 7), mk_(l2,
7)}),
      f1 = mk_CombinedFragment(<alt>, [o1, o2], {l1, l2}),

      sd1 = mkInteraction({l1, l2}, {m1, m2, m3, m4}, {f1},
      {mk_TimeConstraint(t(s(m1)), t(r(m1)), 0, 1000),
       mk_TimeConstraint(t(s(m1)), t(s(m2)), 2000, 5000),
       mk_TimeConstraint(t(s(m2)), t(r(m2)), 0, 1000),
       mk_TimeConstraint(t(r(m1)), t(s(m3)), 8000, nil),
       mk_TimeConstraint(t(s(m3)), t(r(m4)), 0, 5000)
--        mk_TimeConstraint(t(r(m1)), t(r(m2)), 1000, 6000) -- derived
--            mk_TimeConstraint(t(s(m1)), t(r(m3)), 8000, nil) -- derived
       }),
      sd2 = mkInteraction({l1, l2}, {m1, m2, m3, m4}, {f1},
      {mk_TimeConstraint(t(s(m1)), t(r(m1)), 0, 1000),
       mk_TimeConstraint(t(s(m1)), t(s(m2)), 2000, 5000),
       mk_TimeConstraint(t(s(m2)), t(r(m2)), 0, 1000),
       mk_TimeConstraint(t(r(m1)), t(s(m3)), 8000, nil),
       mk_TimeConstraint(t(s(m3)), t(r(m3)), 0, 1000),
       mk_TimeConstraint(t(s(m4)), t(r(m4)), 0, 1000),
       mk_TimeConstraint(t(r(m3)), t(s(m4)), 0, 3000),
       mk_TimeConstraint(t(s(m3)), t(r(m4)), 0, 5000)
      --  mk_TimeConstraint(t(r(m1)), t(r(m2)), 1000, 6000) -- derived
      --  mk_TimeConstraint(t(s(m1)), t(r(m3)), 8000, nil) -- derived
       })

  in
  (
    assertEqual({[s(m1), r(m1), s(m2), r(m2)], [s(m1), r(m1), s(m3), r(m3), s(m4),
r(m4)]}, validTraces(sd1));
    assertEqual({ [s(m1), r(m1), s(m3), r(m3), s(m4), r(m4)]}, unintendedTraces(sd1));
      assertEqual({}, missingTraces(sd1)) ;
    assertEqual({}, unintendedTraces(sd2))
  )
);

-- Example of intra-lifeline time constraint that causes controllability problems:
-- a maximum delay is defined between two send events, with an unconstrained event in
between
-- (in this case, a reception event).
public testSendRecvSendConstraint() ==
(
  let l1 = mk_Lifeline("L1"),
      l2 = mk_Lifeline("L2"),
      m1 = mkMessageTimed(1, mk_(l1, 1), mk_(l2, 1), "m1"),
      m2 = mkMessageTimed(2, mk_(l2, 2), mk_(l1, 2), "m2"),
      m3 = mkMessageTimed(3, mk_(l1, 3), mk_(l2, 3), "m3"),
      sd1 = mkInteraction({l1, l2}, {m1, m2, m3}, {}, {mk_TimeConstraint(t(s(m1)),
t(s(m3)),nil, 5000)}),
      sd2 = mkInteraction({l1, l2}, {m1, m2, m3}, {}, {
             mk_TimeConstraint(t(s(m1)), t(r(m1)), 0, 1000),
             mk_TimeConstraint(t(r(m1)), t(s(m2)), 0, 2000),
             mk_TimeConstraint(t(s(m2)), t(r(m2)), 0, 1000),
             mk_TimeConstraint(t(s(m1)), t(s(m3)), 0, 5000)})
  in
  (
    -- Problem in previous test case solved with the addition of time constraints
    --assertEqual({[s(m1), r(m1), s(m2), r(m2), s(m3), r(m3)]}, validTraces(sd2));
    --assertEqual({}, unintendedTraces(sd2));

    --assertEqual({[s(m1), r(m1), s(m2), r(m2), s(m3), r(m3)]}, validTraces(sd1));
    assertEqual({[s(m1), r(m1), s(m2), r(m2)]}, unintendedTraces(sd1));
    -- because cannot assure that s(m3) can be sent within the defined constraint

  )
);

-- Similar to testSendRecvSendConstraint, but now with a send event in between.
public testSendSendSendConstraint() ==
(
  let l1 = mk_Lifeline("L1"),
      l2 = mk_Lifeline("L2"),
      m1 = mkMessageTimed(1, mk_(l1, 1), mk_(l2, 1), "m1"),
      m2 = mkMessageTimed(2, mk_(l1, 2), mk_(l2, 2), "m2"),
      m3 = mkMessageTimed(3, mk_(l1, 3), mk_(l2, 3), "m3"),
      sd1 = mkInteraction({l1, l2}, {m1, m2, m3}, {}, {mk_TimeConstraint(t(s(m1)),
t(s(m3)), 0, 6000)}),
      sd2 = mkInteraction({l1, l2}, {m1, m2, m3}, {}, {
             mk_TimeConstraint(t(s(m1)), t(s(m2)), 0, 3000),
             mk_TimeConstraint(t(s(m1)), t(s(m3)), 0, 6000)}),
      sd3 = mkInteraction({l1, l2}, {m1, m2, m3}, {}, {
             mk_TimeConstraint(t(s(m1)), t(r(m1)), nil, 1000),
             mk_TimeConstraint(t(s(m2)), t(r(m2)), nil, 1000),
             mk_TimeConstraint(t(s(m1)), t(s(m2)), 2000, 3000),
        mk_TimeConstraint(t(s(m2)), t(s(m3)), 2000, 2000),
             mk_TimeConstraint(t(s(m1)), t(s(m3)), 0, 6000)})

  in
  (
    assertEqual({
          [s(m1), r(m1), s(m2), r(m2), s(m3), r(m3)],
          [s(m1), r(m1), s(m2), s(m3), r(m2), r(m3)],
          [s(m1), s(m2), r(m1), r(m2), s(m3), r(m3)],
          [s(m1), s(m2), r(m1), s(m3), r(m2), r(m3)],
          [s(m1), s(m2), s(m3), r(m1), r(m2), r(m3)]}, validTraces(sd1));

    assertEqual(validTraces(sd1), validTraces(sd2));

    let U = unintendedTraces(sd1) in (
      assertTrue([s(m1), s(m2), r(m2)] in set U); --message overtaking (ok)
      assertTrue([s(m1), r(m1), s(m2), r(m2)] not in set U);-- invalid termination
(fails) CHANGED
      assertTrue([s(m1), s(m2), r(m1), r(m2)] not in set U);-- invalid termination
(fails)CHANGED
  );

    assertTrue( [s(m1), r(m1), s(m2), r(m2)] not in set unintendedTraces(sd2));

    assertEqual( {}, unintendedTraces(sd3));
    -- because cannot assure that s(m3) can be sent within the defined constraint

    assertTrue( [s(m1), s(m2), r(m1), r(m2)] not in set unintendedTraces(sd2));

    assertEqual({}, missingTraces(sd3));
    assertEqual({}, missingTraces(sd2));
  )
);

public testBugFixCheckSatisfiability() ==
(
  let l1 = mk_Lifeline("L1"),
      l2 = mk_Lifeline("L2"),
      m1 = mkMessageTimed(1, mk_(l1, 1), mk_(l2, 1), "m1"),
      m2 = mkMessageTimed(2, mk_(l2, 2), mk_(l1, 2), "m2"),
      m3 = mkMessageTimed(3, mk_(l1, 3), mk_(l2, 3), "m3"),
      m4 = mkMessageTimed(4, mk_(l1, 4), mk_(l2, 4), "m4"),
      sd1 = mkInteraction({l1, l2}, {m1, m2, m3, m4}, {}, {
             mk_TimeConstraint(t(r(m2)), t(s(m3)), 0, 1000),
             mk_TimeConstraint(t(r(m2)), t(s(m4)), 0, 1000),
             mk_TimeConstraint(t(s(m1)), t(s(m3)), 0, 10000),
             mk_TimeConstraint(t(s(m1)), t(s(m4)), 12000, nil)})
  in
  (
    --assertEqual({[s(m1), r(m1), s(m2), r(m2), s(m3), r(m3), s(m4), r(m4)],
    --[s(m1), r(m1), s(m2), r(m2), s(m3), s(m4), r(m3), r(m4)]}, validTraces(sd1));
    assertEqual({}, validTraces(sd1));
    --assertEqual({}, unintendedTraces(sd1));
  )
);

public testMayRemainQuiescentTimed() ==
(
  let l1 = mk_Lifeline("L1"),
      l2 = mk_Lifeline("L2"),
      m1 = mkMessageTimed(1, mk_(l1, 1), mk_(l2, 1), "m1"),
      m2 = mkMessageTimed(2, mk_(l2, 3), mk_(l1, 3), "m2"),
      m3 = mkMessageTimed(3, mk_(l1, 5), mk_(l2, 5), "m3"),
      o1 = mk_InteractionOperand(nil, {mk_(l1, 2), mk_(l2, 2)}, {mk_(l1, 4), mk_(l2,
4)}),
      f1 = mk_CombinedFragment(<opt>, [o1], {l1, l2}),
      sd1 = mkInteraction({l1, l2}, {m1, m2, m3}, {f1}, {
      mk_TimeConstraint(t(s(m1)), t(r(m1)), nil, 1000)}), -- just to force using timed
version
      sd2 = mkInteraction({l1, l2}, {m1, m2, m3}, {f1}, {
      mk_TimeConstraint(t(r(m1)), t(s(m2)), nil, 2000),
      mk_TimeConstraint(t(r(m1)), t(r(m3)), nil, 4000)}),
      sd3 = mkInteraction({l1, l2}, {m1, m2, m3}, {f1}, {
      mk_TimeConstraint(t(s(m1)), t(r(m1)), nil, 1000),
      mk_TimeConstraint(t(r(m1)), t(s(m2)), nil, 2000),
      mk_TimeConstraint(t(s(m2)), t(r(m2)), nil, 1000),
      mk_TimeConstraint(t(s(m1)), t(s(m3)), 5000, 6000)})
  in
  (
    assertEqual({[s(m1), r(m1), s(m2), r(m2), s(m3), r(m3)], [s(m1), r(m1), s(m3),
r(m3)], [s(m1), s(m3), r(m1), r(m3)]}, validTraces(sd1));
    assertTrue([s(m1), r(m1)] in set unintendedTraces(sd1));
    assertTrue([s(m1), r(m1)] in set unintendedTraces(sd2));
    assertEqual({}, unintendedTraces(sd3));
    assertEqual({}, missingTraces(sd1));
    assertEqual({}, missingTraces(sd2));
    assertEqual({}, missingTraces(sd3));
  )
);


public testRcvConstraint() ==
(
  let l1 = mk_Lifeline("L1"),
      l2 = mk_Lifeline("L2"),
      m1 = mkMessageTimed(1, mk_(l1, 1), mk_(l2, 1), "m1"),
      m2 = mkMessageTimed(2, mk_(l2, 2), mk_(l1, 2), "m2"),
      sd1 = mkInteraction({l1, l2}, {m1, m2}, {}, {mk_TimeConstraint(t(s(m1)), t(r(m2)),
nil, 4000)})
  in
  (
    assertEqual({[s(m1), r(m1), s(m2), r(m2)]}, validTraces(sd1));
    assertTrue([s(m1), r(m1), s(m2), r(m2)] in set unintendedTraces(sd1));
    assertEqual({}, missingTraces(sd1));
  )
);
```

```
functions

public mkMsgTimeConstraints: set of Message * [Duration] * [Duration] -> set of
TimeConstraint
mkMsgTimeConstraints(messages, minDuratin, maxDuration) ==
      {mk_TimeConstraint(t(s(m)), t(r(m)), minDuratin, maxDuration) | m in set
messages};

operations
public testTrafficControl () ==
(
  let l1 = mk_Lifeline("Car"),
      l2 = mk_Lifeline("SensorA"),
      l3 = mk_Lifeline("SensorB"),
      l4 = mk_Lifeline("TMC"),
      l5 = mk_Lifeline("DMS"),
      l6 = mk_Lifeline("OCC"),
      l7 = mk_Lifeline("Operator"),
      m1 = mkMessageTimedSynch(1, mk_(l1, 1), mk_(l2, 1), "id_signal"),
      m2 = mkMessageTimed(2, mk_(l2, 2), mk_(l3, 2), "notify_id"),
      m3 = mkMessageTimedSynch(3, mk_(l1, 4), mk_(l3, 4), "id_signal"),
      m4 = mkMessageTimedSynch(4, mk_(l1, 6), mk_(l3, 6), "id_signal"),
      m5 = mkMessageTimed(5, mk_(l3, 7), mk_(l4, 7), "notify_speed_alert"),
      m6 = mkMessageTimed(6, mk_(l3, 9), mk_(l4, 9), "notify_traffic_alert"),
      m7 = mkMessageTimed(7, mk_(l4, 10), mk_(l5, 10), "warning_msg_on"),
      m8 = mkMessageTimed(8, mk_(l4, 11), mk_(l6, 11), "notify_traffic_alert"),
      m9 = mkMessageTimedSynch(9, mk_(l6, 12), mk_(l7, 12), "traffic_alert"),
      m10 = mkMessageTimedSynch(10, mk_(l7, 14), mk_(l6, 14), "msg_cancel"),
      m11 = mkMessageTimed(11, mk_(l6, 15), mk_(l4, 15), "msg_cancel"),
      m12 = mkMessageTimed(12, mk_(l4, 16), mk_(l5, 16), "warning_msg_off"),
      o1 = mk_InteractionOperand(nil, {mk_(l1, 3), mk_(l2, 3), mk_(l3, 3), mk_(l4, 3),
mk_(l5, 3), mk_(l6, 3), mk_(l7, 3)},
      {mk_(l1, 5), mk_(l2, 5), mk_(l3, 5), mk_(l4, 5), mk_(l5, 5), mk_(l6, 5), mk_(l7,
5)}),
      o2 = mk_InteractionOperand(nil, {mk_(l1, 5), mk_(l2, 5), mk_(l3, 5), mk_(l4, 5),
mk_(l5, 5), mk_(l6, 5), mk_(l7, 5)},
      {mk_(l1, 8), mk_(l2, 8), mk_(l3, 8), mk_(l4, 8), mk_(l5, 8), mk_(l6, 8), mk_(l7,
8)}),
      o3 = mk_InteractionOperand(nil, {mk_(l1, 8), mk_(l2, 8), mk_(l3, 8), mk_(l4, 8),
mk_(l5, 8), mk_(l6, 8), mk_(l7, 8)},
      {mk_(l1, 18), mk_(l2, 18), mk_(l3, 18), mk_(l4, 18), mk_(l5, 18), mk_(l6, 18),
mk_(l7, 18)}),
      o4 = mk_InteractionOperand(nil, {mk_(l4, 13), mk_(l5, 13), mk_(l6, 13), mk_(l7,
13)},
      {mk_(l4, 17), mk_(l5, 17), mk_(l6, 17), mk_(l7, 17)}),
      f1 = mk_CombinedFragment(<alt>, [o1, o2, o3], {l1, l2, l3, l4, l5, l6, l7}),
      f2 = mk_CombinedFragment(<opt>, [o4], {l4, l5, l6, l7}),
      sd1 = mkInteraction({l1, l2, l3, l4, l5, l6, l7},
                          {m1, m2, m3, m4, m5, m6, m7, m8, m9, m10, m11, m12},
                          {f1, f2},
                     mkMsgTimeConstraints({m1, m3, m4, m9, m10}, nil, 0) -- synchronous
messages
                     union
                     { mk_TimeConstraint(t(s(m1)), t(s(m4)), nil, 23),    -- more than
150 km/h
                          mk_TimeConstraint(t(s(m1)), t(s(m3)), 24, 72),  -- 50 to 150
km/h, 1 km between sensors
```

```
                       mk_TimeConstraint(t(r(m2)), t(s(m6)), 73, nil)  -- less than 50
km/h
                     }),

      sd2 = mkInteraction({l1, l2, l3, l4, l5, l6, l7},
                          {m1, m2, m3, m4, m5, m6, m7, m8, m9, m10, m11, m12},
                          {f1, f2},
                     mkMsgTimeConstraints({m1, m3, m4, m9, m10}, nil, 0) -- synchronous
messages
                     union mkMsgTimeConstraints({m2, m7}, nil, 1)
                     union
                     { mk_TimeConstraint(t(s(m1)), t(s(m4)), 8, 25),    -- more than 150
to 450 km/h
                       mk_TimeConstraint(t(r(m2)), t(r(m4)), 6, 23),
                         mk_TimeConstraint(t(s(m1)), t(s(m3)), 24, 72),  -- 50 to 150
km/h, 1 km between sensors
                         mk_TimeConstraint(t(r(m2)), t(r(m3)), 24, 72),

                       mk_TimeConstraint(t(r(m1)), t(s(m2)), nil, 1),
                       mk_TimeConstraint(t(r(m2)), t(s(m6)), 73, nil),  -- less than 50
km/h
              mk_TimeConstraint(t(r(m9)), t(s(m10)), 5, nil) -- can cancel after only
after 5s
                     })

  in
  (
    assertEqual(
      {-- Late reception of notify_id (m2):
       mk_([s(m1), r(m1), s(m3), r(m3), s(m2), r(m2)], mk_AndExp({mk_DC(1, 3, -24),
mk_DC(2, 1, 0), mk_DC(3, 1, 72), mk_DC(4, 3, 0)})),
       mk_([s(m1), r(m1), s(m2), s(m3), r(m3), r(m2)], mk_AndExp({mk_DC(1, 4, -24),
mk_DC(2, 1, 0), mk_DC(4, 1, 72), mk_DC(5, 4, 0)})),
       mk_([s(m1), r(m1), s(m4), r(m4), s(m2), r(m2)], mk_AndExp({mk_DC(2, 1, 0),
mk_DC(3, 1, 23), mk_DC(4, 3, 0)})),
       mk_([s(m1), r(m1), s(m2), s(m4), r(m4), r(m2)], mk_AndExp({mk_DC(2, 1, 0),
mk_DC(4, 1, 23), mk_DC(5, 4, 0)})),
       -- Missing notify_speed_alert (m5):
       mk_([s(m1), r(m1), s(m2), r(m2), s(m4), r(m4)], mk_AndExp({mk_DC(2, 1, 0),
mk_DC(5, 1, 23), mk_DC(6, 4, 72), mk_DC(6, 5, 0)})),
       -- Extraneous notify_speed_alert (m5):
       mk_([s(m1), r(m1), s(m2), r(m2), s(m3), r(m3), s(m5), r(m5)], mk_AndExp({mk_DC(1,
5, -24), mk_DC(2, 1, 0), mk_DC(5, 1, 72), mk_DC(6, 4, 23), mk_DC(6, 5, 0)})),
       -- Late reception of warning_msg_on (m7) (after warning_msg_on) :
       mk_([s(m1), r(m1), s(m2), r(m2), s(m6), r(m6), s(m7), s(m8), r(m8), s(m9), r(m9),
s(m10), r(m10), s(m11), r(m11), s(m12), r(m12), r(m7)], mk_AndExp({mk_DC(2, 1, 0),
mk_DC(4, 5, -73), mk_DC(11, 10, 0), mk_DC(13, 12, 0)}))},
       unintendedTracesTimedRaw(sd1));

     assertEqual(
       {[s(m1), r(m1), s(m2), r(m2), s(m3), r(m3)],
        [s(m1), r(m1), s(m2), r(m2), s(m4), r(m4), s(m5), r(m5)],
        [s(m1), r(m1), s(m2), r(m2), s(m6), r(m6), s(m7), r(m7), s(m8), r(m8), s(m9),
r(m9)],
        [s(m1), r(m1), s(m2), r(m2), s(m6), r(m6), s(m7), s(m8), r(m7), r(m8), s(m9),
r(m9)],
        [s(m1), r(m1), s(m2), r(m2), s(m6), r(m6), s(m7), s(m8), r(m8), r(m7), s(m9),
r(m9)],
        [s(m1), r(m1), s(m2), r(m2), s(m6), r(m6), s(m7), s(m8), r(m8), s(m9), r(m9),
r(m7)],
        [s(m1), r(m1), s(m2), r(m2), s(m6), r(m6), s(m7), r(m7), s(m8), r(m8), s(m9),
r(m9), s(m10), r(m10), s(m11), r(m11), s(m12), r(m12)],
        [s(m1), r(m1), s(m2), r(m2), s(m6), r(m6), s(m7), s(m8), r(m7), r(m8), s(m9),
r(m9), s(m10), r(m10), s(m11), r(m11), s(m12), r(m12)],
        [s(m1), r(m1), s(m2), r(m2), s(m6), r(m6), s(m7), s(m8), r(m8), r(m7), s(m9),
r(m9), s(m10), r(m10), s(m11), r(m11), s(m12), r(m12)],
        [s(m1), r(m1), s(m2), r(m2), s(m6), r(m6), s(m7), s(m8), r(m8), s(m9), r(m9),
r(m7), s(m10), r(m10), s(m11), r(m11), s(m12), r(m12)]
        }, validTraces(sd2) );

      assertEqual({}, unintendedTraces(sd2));
               -- alone: 1.7 seconds
               -- alone, with only assertions checking: 1.4 seconds
               --  (validTracesUntimed: 0.2 + ValidTimedTraces:0.2 + Project:0.2 +
SimulExec:0.7 + subtract: 0.1)
               -- after other teste cases: 0.8 seg
  )
);

public testTrafficControlWithGuards () ==
(
  let l1 = mk_Lifeline("Car"),
      l2 = mk_Lifeline("SensorA"),
      l3 = mk_Lifeline("SensorB"),
      l4 = mk_Lifeline("TMC"),
      l5 = mk_Lifeline("DMS"),
      l6 = mk_Lifeline("OCC"),
      l7 = mk_Lifeline("Operator"),
      m1 = mkMessageTimedSynch(1, mk_(l1, 1), mk_(l2, 1), "id_signal"),
      m2 = mkMessageTimed(2, mk_(l2, 2), mk_(l3, 2), "notify_id"),
      m3 = mkMessageTimedSynch(3, mk_(l1, 4), mk_(l3, 4), "id_signal"),
      c =  mk_TimeConstraint(t(r(m2)), t(r(m3)), nil, 23),
      m5 = mkMessageTimedGuarded(5, mk_(l3, 6), mk_(l4, 6), "notify_speed_alert", c),
      m6 = mkMessageTimed(6, mk_(l3, 9), mk_(l4, 9), "notify_traffic_alert"),
      m7 = mkMessageTimed(7, mk_(l4, 10), mk_(l5, 10), "warning_msg_on"),
      m8 = mkMessageTimed(8, mk_(l4, 11), mk_(l6, 11), "notify_traffic_alert"),
      m9 = mkMessageTimedSynch(9, mk_(l6, 12), mk_(l7, 12), "traffic_alert"),
      m10 = mkMessageTimedSynch(10, mk_(l7, 14), mk_(l6, 14), "msg_cancel"),
      m11 = mkMessageTimed(11, mk_(l6, 15), mk_(l4, 15), "msg_cancel"),
      m12 = mkMessageTimed(12, mk_(l4, 16), mk_(l5, 16), "warning_msg_off"),
      o1 = mk_InteractionOperand(nil, {mk_(l1, 3), mk_(l2, 3), mk_(l3, 3), mk_(l4, 3),
mk_(l5, 3), mk_(l6, 3), mk_(l7, 3)},
        {mk_(l1, 8), mk_(l2, 8), mk_(l3, 8), mk_(l4, 8), mk_(l5, 8), mk_(l6, 8), mk_(l7,
8)}),
      o2 = mk_InteractionOperand(nil, {mk_(l3, 5), mk_(l4, 5)}, {mk_(l3, 7), mk_(l4,
7)}),
      o3 = mk_InteractionOperand(nil, {mk_(l1, 8), mk_(l2, 8), mk_(l3, 8), mk_(l4, 8),
mk_(l5, 8), mk_(l6, 8), mk_(l7, 8)},
        {mk_(l1, 18), mk_(l2, 18), mk_(l3, 18), mk_(l4, 18), mk_(l5, 18), mk_(l6, 18),
mk_(l7, 18)}),
      o4 = mk_InteractionOperand(nil, {mk_(l4, 13), mk_(l5, 13), mk_(l6, 13), mk_(l7,
13)},
        {mk_(l4, 17), mk_(l5, 17), mk_(l6, 17), mk_(l7, 17)}),
      f1 = mk_CombinedFragment(<alt>, [o1, o3], {l1, l2, l3, l4, l5, l6, l7}),
      f2 = mk_CombinedFragment(<opt>, [o4], {l4, l5, l6, l7}),
      f3 = mk_CombinedFragment(<opt>, [o2], {l3, l4}),
      sd2 = mkInteraction({l1, l2, l3, l4, l5, l6, l7},
                          {m1, m2, m3, m5, m6, m7, m8, m9, m10, m11, m12},
                          {f1, f2, f3},
              mkMsgTimeConstraints({m1, m3, m9, m10}, nil, 0) -- synchronous messages
              union mkMsgTimeConstraints({m2, m7}, nil, 1)
              union
              { mk_TimeConstraint(t(s(m1)), t(s(m3)), 8, 72),   -- 50 to 450 km/h, 1 km
between sensors
                --mk_TimeConstraint(t(r(m2)), t(r(m3)), nil, 72),
                mk_TimeConstraint(t(r(m1)), t(s(m2)), nil, 1),
                mk_TimeConstraint(t(r(m2)), t(s(m6)), 73, nil),  -- less than 50 km/h
                mk_TimeConstraint(t(r(m9)), t(s(m10)), 5, nil) -- can cancel after only
after 5s
              })

  in
  (
      assertEqual({}, unintendedTracesTimedRaw(sd2));
  )
);

operations
public testAll() ==
(
             testSimple();
             testOpt();
             testAlt();
             testLoop();
             testAltNested();
             testStrict();
             testRace();
             testNonLocalChoice();
             testIndepMessages();
             testTimeConstraintInLoop();
             testImpossible();
             testUnintendedEmptyTrace();
             testWhoSends();
             testUnintendedEmptyTrace2();
        testUnintendedEmptyTrace3();
             testTimeConstraint();
             testInterLifelineTimeConstraints();
        testVerdictWithTimestamps();
             testRaceByMsgOvertaking();
             testNonLocallyControlableTimed();
             testRcvConstraint();
             testBugFixCheckSatisfiability();
             testIsLocallyObservableTimed();
             testStrangeControllableTimed();
        testSendSendSendConstraint();
             testFallDetection();
             testMayRemainQuiescentTimed();
             testSendableFirst();
             testSendableFirst2();
    testSendRecvSendConstraint();
             testTrafficControl();
             testTrafficControlWithGuards();
             -- total time for all test cases: 9 seconds
);
```

```
end TestCases
```

9. References

1. VDM-10 Language Manual, Peter Gorm Larsen *et al*, Overture Technical Report Series No. TR-001, Feb 2018
2. Local Observability and Controllability Analysis of Test Scenarios for Time-constrained Distributed Systems, *Anonymized authors*, January 2019 (*submitted for publication*)