# MBIT4DS

May 13, 2018

## Contents



# 1 SequenceDiagrams

```
/**
* 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.
* By Jo o Pascoal Faria & Bruno Lima, FEUP/INESC TEC, 2016-2018.
**/

class TSD is subclass of DC

/** Configurations **/

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

values
public static INFINITY = 1E10;

types

/** Values and Value Specifications (based on UML meta-model) **/

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

public Time = nat; -- in the desired sclase (seconds, millisenconds, etc.)
public Duration = int; -- may be negative, for convenience
```

```
public TimeInterval = [Time] * [Time];
public DurationInterval = [Duration] * [Duration];

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;


/** 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
  (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 Message ::
 id            : MessageId
  sendEvent     : LifelineLocation
  receiveEvent  : LifelineLocation
  signature     : MessageSignature
  sendTimestamp : [Variable]
  recvTimestamp : [Variable]
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 * DiffConstrExp; -- Time constrained trace

public Event ::
   type       : EventType
   signature  : MessageSignature
   lifeline   : Lifeline
   timestamp  : [Variable | Time];  -- Variable in formal 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 or querying things **/

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

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

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

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

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) ==
 mk_Interaction(lifelines, messages, combinedFragments, {});

protected mkMessage: MessageId * LifelineLocation * LifelineLocation * MessageSignature ->
    Message

mkMessage(id, sendEvent, receiveEvent, signature) == mk_Message(id, sendEvent, receiveEvent,
    signature, nil, nil);

protected mkMessageTimed: MessageId * LifelineLocation * LifelineLocation * MessageSignature ->
    Message

mkMessageTimed(id, sendEvent, receiveEvent, signature) == mk_Message(id, sendEvent, receiveEvent,
     signature,
  mk_Variable("s_" ^ signature), mk_Variable("r_" ^ signature));


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);


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_((lf), loc1) in set startLocs & loc1 < loc)
  and (exists mk_((lf), loc2) in set endLocs & 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_((lf), loc1) in set startLocs1 & loc1 < loc2)
  and (forall mk_(lf, loc2) in set endLocs2 &

         exists mk_((lf), loc1) in set endLocs1 & loc1 > loc2);

-- time constraints that are local to 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)};

end TSD
```

# 2 DifferenceConstraints

```
/**
 * Manipulation of difference constraints (DC).
 */

class DC

values

public INFINITY = 1E8;
public NEGINFINITY = -1E8;

types

public VariableId = int;
public Difference = int;
public Value = nat;

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

-- Expressions in Dijunctive-Normal-Form (DNF)
public MinExp :: args: set of TimeExpr;
public MaxExp :: args: set of TimeExpr;
public SumExp :: var  : VariableId
                 delta: Difference;
public TimeExpr = MinExp | MaxExp | SumExp | MaxValue | MinValue;

public MaxValue :: ;
public MinValue :: ;

public LeqExp :: lhs: TimeExpr
                 rhs: TimeExpr;
public AndExp :: args: set of DiffConstr;
public OrExp  :: args: set of (AndExp | DiffConstr);
public DiffConstrExp = DiffConstr | AndExp | OrExp | LeqExp;

private Weight = int;
```

```
private Distance = int;

functions

-- max of empty set is MinValue

public mkMaxExp: set of TimeExpr -> TimeExpr
mkMaxExp(args) == mk_MaxExp(args);

-- min of empty set is MaxValue

public mkMinExp: set of TimeExpr -> TimeExpr
mkMinExp(args) == mk_MinExp(args);


public mkSumExp: VariableId * Difference -> TimeExpr
mkSumExp(var, delta) == mk_SumExp(var, delta);


public mkVarExp: VariableId -> TimeExpr
mkVarExp(var) == mk_SumExp(var, 0);


public mkLeqExp: TimeExpr * TimeExpr -> DiffConstrExp
mkLeqExp(lhs, rhs) ==
 if is_MaxExp(lhs) then
    -- if lhs.args = {} the default is MinValue, which is <= than anything, so the result is true
    if lhs.args = {} then mkLeqExp(mk_MinValue(), rhs)
    else mkAndExp({mkLeqExp(a, rhs) | a in set lhs.args})
 else if is_MaxExp(rhs) then
    if rhs.args = {} then mkLeqExp(lhs, mk_MinValue())
    -- if rhs.args = {} the default is MinValue, and nothing (except MinValue) is <= than that,
        so the result is false
    else mkOrExp({mkLeqExp(lhs, a) | a in set rhs.args})
 else if is_MinExp(rhs) then
    if rhs.args = {} then mkLeqExp(lhs, mk_MaxValue())
    -- if rhs.args = {} the default is MaxValue, and everything is <= than that, so the result is
         true
    else mkAndExp({mkLeqExp(lhs, a) | a in set rhs.args})
 else if is_MinExp(lhs) then
    if lhs.args = {} then mkLeqExp(mk_MaxValue(), rhs)
    -- if lhs.args = {} the default is MaxValue, which is not <= than anything (except MaxValue),
         so the result is false
    else mkOrExp({mkLeqExp(a, rhs) | a in set lhs.args})
 else if is_SumExp(lhs) and is_SumExp(rhs) then
    mk_DiffConstr(lhs.var, rhs.var, rhs.delta - lhs.delta)
 else if is_MinValue(lhs) or is_MaxValue(rhs) then
    mkTrueExp()
 else if is_MaxValue(lhs) or is_MinValue(rhs) then
    mkFalseExp()
 else
    mkFalseExp();

-- Given an expression in DNF (exp), returns the negation in DNF

public mkNotExp: DiffConstrExp -> DiffConstrExp
mkNotExp(exp) ==
   if is_DiffConstr(exp) then mk_DiffConstr(exp.j, exp.i, -(exp.d + 1))
   else if is_AndExp(exp) then mkOrExp2({mk_DiffConstr(arg.j, arg.i, -(arg.d+1))| arg in set exp.
       args})
   else -- is_OrExp(exp)
      if exp.args = {} then mkTrueExp()
      else
        let arg in set exp.args in
           let left = mkNotExp(arg) in
```

```
            if exp.args = {arg} then left
            else mkAndExp({left, mkNotExp(mkOrExp2(exp.args \ {arg}))});

-- Given a set of expressions in DNF (args), returns the conjunction in DNF

public mkAndExp: set of DiffConstrExp -> DiffConstrExp
mkAndExp(args) ==
   if args = {} then mk_AndExp({})
   else let left in set args in
          let right = mkAndExp(args \ {left}) in
              if is_OrExp(left) or is_OrExp(right) then
                   mkOrExp2(
                      {mkAndExp2((if is_AndExp(e1) then e1.args else {e1})
                                 union (if is_AndExp(e2) then e2.args else {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})})
              else
                   mkAndExp2((if is_AndExp(left) then left.args else {left})
                              union (if is_AndExp(right) then right.args else {right}));


public mkTrueExp: () -> DiffConstrExp
mkTrueExp() == mk_AndExp({});


public mkFalseExp: () -> DiffConstrExp
mkFalseExp() == mk_OrExp({});


private implies: DiffConstrExp * DiffConstrExp -> bool
implies(e1, e2) ==
   if e1 = e2 then true
   else if is_AndExp(e1) and is_AndExp(e2) then
           (forall d2 in set e2.args & exists d1 in set e1.args & implies(d1, d2))
   else if is_DiffConstr(e1) and is_DiffConstr(e2) then
            e1.i = e2.i and e1.j = e2.j and e1.d <= e2.d
   else false;

private mkAndExp2: set of DiffConstr -> AndExp
mkAndExp2(args) ==

   mk_AndExp({mk_DiffConstr(i, j, d) | mk_DiffConstr(i, j, d) in set args &
                    (not exists mk_DiffConstr((i), (j), d2) in set args & d2 < d)
                    });

-- Given a set of expressions in DNF (args), returns the disjunction in DNF
public mkOrExp: set of DiffConstrExp -> DiffConstrExp
mkOrExp(args) ==
  mkOrExp2( dunion {if is_OrExp(arg) then arg.args else {arg} | arg in set args});


private mkOrExp2: set of DiffConstrExp -> DiffConstrExp
mkOrExp2(args) ==
   let args2 = {arg | arg in set args & not exists arg2 in set args & arg2 <> arg and implies(arg
       , arg2)}

   in  if card args2 = 1 then (let arg in set args2 in arg)
       else mk_OrExp(args2);

public static isSatisfiable: DiffConstrExp  -> bool
isSatisfiable(exp) == isFeasible(exp);


public static sat: DiffConstrExp  -> bool
sat(exp) == isFeasible(exp);
```

```
operations

-- assumes varids start in 1
public static pure isFeasible: DiffConstrExp  ==> bool
isFeasible(exp) == (
  dcl weights : map VariableId * VariableId to Distance;
  dcl minvar : int;
  dcl maxvar : int;

  if is_OrExp(exp) then
   return exists arg in set exp.args & isFeasible(arg);
  if is_DiffConstr(exp) then
    return exp.i <> exp.j or exp.d >= 0;
  if exp.args = {} then return true;
  minvar := INFINITY;
  maxvar := NEGINFINITY;
  for all mk_DiffConstr(i, j, -) in set exp.args do (
    if i > maxvar then maxvar := i;
    if j > maxvar then maxvar := j;
    if i < minvar then minvar := i;
    if j < minvar then minvar := j;
  );
  weights := { mk_(minvar-1, i) |-> 0 | i in set {minvar, ..., maxvar}};
  for all mk_DiffConstr(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};
  return BellmanFord({minvar-1, ..., maxvar}, weights, minvar-1)
);

functions

public static reduceToFeasible: DiffConstrExp  -> [DiffConstrExp]
reduceToFeasible(exp) == (
  if is_OrExp(exp) then
   let feasible = {arg | arg in set exp.args & isFeasible(arg)}
   in if feasible = {} then nil

      else if card feasible = 1 then let arg in set feasible in arg
      else mk_OrExp(feasible)
  else if isFeasible(exp) then exp else nil
);

operations
-- shortest paths from a source vertex to all vertices;
-- in the presence of negative weights.
public static pure BellmanFord: (set of VariableId) * (map VariableId * VariableId to Weight) *
    VariableId ==> bool
BellmanFord(vertices, weights, source) == (
   dcl dist : map VariableId to [Distance] := {v |-> INFINITY | v in set vertices} ++ {source |->
        0};
   for i = 1 to card vertices - 1  do
      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));
    return not exists mk_(u, v) in set dom weights & dist(v) > dist(u) + weights(mk_(u,v))
);

functions

public simplify: set of DC`DiffConstr -> set of DC`DiffConstr
simplify(C) ==
  {mk_DC`DiffConstr(i, j, d) | mk_DC`DiffConstr(i, j, d) in set C &
```

```
        not (i = j and d >= 0)
        and not (exists mk_DC`DiffConstr((i), (j), d2) in set C & d2 < d)};

-- Eliminates a variable v in a set C of difference constraints.
-- Assumes the set is satisfiable, so self loops with negative weight do not appear.
private static elimVar: VariableId * (set of DC`DiffConstr) -> set of DC`DiffConstr
elimVar(v, C) ==
  simplify(
    {mk_DC`DiffConstr(i1, j2, d1 + d2) | mk_DC`DiffConstr(i1, (v), d1), mk_DC`DiffConstr((v), j2,
         d2) in set C & i1<>v and j2<>v}
    union
    {mk_DC`DiffConstr(i, j, d) | mk_DC`DiffConstr(i, j, d) in set C & i <> v and j <> v}
  );

-- Eliminates a set V of variables in a set C of difference constraints
private static elimVars: (set of VariableId) * (set of DC`DiffConstr) -> set of DC`DiffConstr
elimVars(V, C) ==
  if V = {} then C
  else let v in set V in elimVars(V \ {v}, elimVar(v,C));

private static elimVars: (set of VariableId) * DC`DiffConstrExp -> DC`DiffConstrExp
elimVars(V, c) == mkAndExp(elimVars(V, c.args));

public static projectToVars: (set of DC`DiffConstr) * (set of VariableId) -> set of DC`DiffConstr
projectToVars(C, V) == elimVars(vars(C) \ V, C);

public static projectToVars: DiffConstrExp * (set of VariableId) -> DiffConstrExp
projectToVars(C, V) == elimVars(vars(C.args) \ V, C);

public static vars: set of DC`DiffConstr -> set of VariableId
vars(C) == dunion {{i, j} | mk_DiffConstr(i, j, -) in set C};

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


public static renumVars: ((map VariableId to VariableId) | (seq of VariableId)) * DiffConstrExp
    -> DiffConstrExp
renumVars(renum, c) ==
  mkAndExp({mk_DC`DiffConstr(renum(i), renum(j), d)| mk_DC`DiffConstr(i,j,d) in set c.args});

public static satisfies: (map VariableId to Value | seq of Value) * (set of DC`DiffConstr) ->
    bool
satisfies(v, c) ==
 forall mk_DC`DiffConstr(i, j, d) in set c & v(i) - v(j) <= d;


public static maxVal: VariableId * (map VariableId to Value | seq of Value) * (set of DC`
    DiffConstr) -> [Value]
maxVal(id, v, c) ==
 let S = {v(j) + d | mk_DC`DiffConstr((id), j, d) in set c}
 in if S = {} then nil else iota x in set S & not exists y in set S & y < x;

public static minVal: VariableId * (map VariableId to Value | seq of Value) * (set of DC`
    DiffConstr) -> [Value]
minVal(id, v, c) ==
 let S = {v(i) - d | mk_DC`DiffConstr(i, (id), d) in set c}

 in if S = {} then nil else iota x in set S & not exists y in set S & y > x;

end DC
```

# 3 ValidTraces

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

class ValidTraces is subclass of TSD


functions

-- Determine the valid formal traces defined by an Interaction (sd).
public static validTraces: Interaction -> set of Trace

validTraces(sd) ==
 {t | t in set validTracesUntimed(sd) & isSatisfiable(constraintExp(t, 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) ==
 {mk_(t, constraintExp(t, sd)) | t in set validTracesUntimed(sd) & isSatisfiable(constraintExp(t,
      sd))};


-- 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
public 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}
   };


public static constraintExp: Trace * Interaction -> DiffConstrExp
constraintExp(t, sd) == constraintSet2Exp(constraintSet(t, sd));


public static constraintSet2Exp: set of DiffConstr -> DiffConstrExp
constraintSet2Exp(S) == mkAndExp(S);

public static constraintSet: Trace * Interaction -> set of DiffConstr
constraintSet(t, sd) ==
  {mk_DiffConstr(i, i+1, 0) | i in set {1, ..., len t - 1}}
  union
  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 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}
   };

public ev2ocConstr: nat * nat * TimeConstraint ->  set of DiffConstr
ev2ocConstr(i, j, c) ==

   (if c.max = nil then {} else {mk_DiffConstr(j, i, c.max)})
   union (if c.min = nil then {} else {mk_DiffConstr(i, j, -c.min)});

protected removeExtraTraceInfo: set of TraceExt -> set of Trace

removeExtraTraceInfo(s) ==
 {[mkEvent(e.type, e.signature, e.lifeline, e.timestamp) | e in seq t] | t in set s};

protected validTracesExt: Interaction -> set of TraceExt
validTracesExt(sd) ==

 freeComb({{[e]} | e in set topLevelEvents(sd)}
        union {expandCombinedFragment(sd, c) | c in set topLevelCombFrag(sd)});

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

   m in set sd.messages &
    not exists c in set sd.combinedFragments & contains(c, m.receiveEvent)};

protected topLevelCombFrag: Interaction -> set of CombinedFragment
topLevelCombFrag(sd) ==

 {c | c in set sd.combinedFragments &
    not exists c2 in set sd.combinedFragments & contains(c2, c)};

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

protected freeComb: TraceExt * TraceExt -> set of TraceExt
freeComb(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 freeComb(tl t1, t2)})
       union

       (if exists e in seq t1 & precedes(e, hd t2) then {}
     else {[hd t2] ^ r | r in set freeComb(t1, tl t2)});

protected 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)));

protected 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 **/

protected 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;

protected 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.
protected 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.
protected 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.
protected 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)};


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


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

```
protected 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};

protected 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};


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

protected expandOperand: Interaction * InteractionOperand -> set of TraceExt
expandOperand(i, o) ==

 freeComb({{[e]} | e in set nestedEvents(i, o)}
        union {expandCombinedFragment(i, c) | c in set nestedCombFrag(i, o)});

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

   m in set sd.messages & contains(o, m.receiveEvent)
     and not exists c in set sd.combinedFragments & contains(o, c) and contains(c, m.receiveEvent
         )};

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

end ValidTraces
```

# 4 ConformanceChecking

```
/**
 * 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 (prevEvents) ^ [e] ^ - in set validLocalTraces & e = 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).
operations
protected static pure checkConstraint: Event * Event * TimeConstraint ==> Verdict
```

```
checkConstraint(e1, e2, c) ==
 return
  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)
  in cases intersect({mk_(c.min, c.max), ds}):
        (ds) -> <Pass>,
        nil -> <Fail>,
        others -> <Inconclusive>
     end;


functions
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};


public static prefixesTimedTraces: set of TCTrace -> set of TCTrace
prefixesTimedTraces(T) ==
  {mk_([], mkTrueExp())}
  union
  dunion {{projectTimedTrace(mk_(t,c), [k | k in set {1, ..., i}]) | i in set inds t} | mk_(t,c)
      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>};

-- 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};


-- Returns the set of next allowed events at a lifeline and time instant,
-- including nil if the absence of an event is valid.
public allowedEvents: Trace * (set of Trace) * (set of TimeConstraint) * Time -> set of [Event]
allowedEvents(a, V, C, time) ==
  dunion {
    if len f = len a then {nil}
    else let i = eventInterval(a, f, len a + 1, C) in if i = nil then {}
    else (if intersect({i, mk_(time,time)}) = nil then {} else {mu(f(len a + 1), timestamp |->
        time)})
         union (let max = i.#2 in if max = nil or max > time then {nil} else {})
   | f in set V & len f >= len a and matches(a, f(1, ..., len a), C) = <Pass>}
pre forall i in set inds a & a(i).timestamp <= time;

-- 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, (i), c) in set getConstrainedPairs(f, C)});

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
```

# 5 Observability

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

class Observability is subclass of ValidTraces


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 = projectTimedTraces(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 static 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 static 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 static projectTrace: Trace * Lifeline -> Trace
projectTrace(t, l) == [e | e in seq t & e.lifeline = l];

public static projectTimedTraces: (set of TCTrace) * (set of Lifeline) -> map Lifeline to set of
    TCTrace

projectTimedTraces(T, L) == {l |-> projectTimedTraces(T, l) | l in set L};

-- Projects a set of time constrained traces (T) onto a lifeline (l).

public static projectTimedTraces: (set of TCTrace) * Lifeline -> set of TCTrace
projectTimedTraces(T, l) == {projectTimedTrace(t, l) | t in set T};

-- Projects a time constrained trace (t, c) onto a lifeline (l).
public static projectTimedTrace: TCTrace * Lifeline -> TCTrace
projectTimedTrace(mk_(t,c), l) ==
   projectTimedTrace(mk_(t,c), lifelineInds(l, t));


-- Projects a time constrained trace (t,c) onto a sorted subset (I) of indices .
public static projectTimedTrace: TCTrace * (seq of nat) -> TCTrace
projectTimedTrace(mk_(t,c), I) ==
  mk_([t(i) | i in seq I], renumVars(inverse asMap[nat](I), projectToVars(c, elems I)));

asMap[@T]: seq of @T -> map nat to @T
asMap(s) == { i |-> s(i) | i in set inds s};

  -- example [-,e2,-,e4], lifelineInds = [2, 4]

public static subtractTimedTraces: (set of TCTrace) * (set of TCTrace) -> set of TCTrace
subtractTimedTraces(S1, S2) ==
  dunion {let c1 = mkOrExp({c1 | mk_((t), c1) in set S1}),

              c2 = mkOrExp({c2 | mk_((t), c2) in set S2}),
              c3  = reduceToFeasible(mkAndExp({c1, mkNotExp(c2)}))
          in if c3 <> nil then {mk_(t,c3)} else {}
         | mk_(t,-) 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) == joinTimedTraces(mk_([], mkTrueExp()), M);
-- join(m, mk_([], mkTrueExp()));
-- dunion {allInterleavings(c) | T in set allCombinations(M)};

-- non-executable specification:
```

```
-- {t | t: TCTrace & forall l in set dom m & isSubsetTimedTraces({projectTimedTrace(t,l)}, m(l))
    };
-- post forall t in set RESULT & forall l in set dom m & isSubsetTimedTraces({projectTimedTrace(t
    ,l)}, m(l));

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_([],-) in set m(l) & true then {mk_(t,c)} else {})
 union

 dunion {
     dunion {
       let newT = t ^ [e],
           r = lifelineInds(l, newT),
            C2 = projectToVars(lc.args, {1,..., len r}),
            newC = mkAndExp(c.args
                              union (if t <> [] then {mk_DiffConstr(len t, len t + 1, 0)} else
                                  {})

                              union renumVars(r, {c2 | c2 in set C2 & not isImplicitOrd(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};

-- Checks if it is an implicit ordering constraint

public isImplicitOrd: DiffConstr -> bool
isImplicitOrd(mk_DiffConstr(i,j,d)) == j = i+1 and d = 0;

-- Obtains the possible (feasible) extensions of a global time constrained trace (t,c), from
-- the perspective of a lifeline 'l' with a given set V of locally valid time constrained traces.
-- Each extension is a pair of an added event and added time constraints.
public static traceExtLf: TCTrace * Lifeline * (set of TCTrace) -> set of (Event * (set of
    DiffConstr))
traceExtLf(mk_(t, c), l, V) ==
  let pt = projectTrace(t, l)
  in dunion
       {let r = lifelineInds(l, t ^ [e]),
            C2 = projectToVars(lc.args, {1,..., len r}),
            newC = (if t <> [] then {mk_DiffConstr(len t, len t + 1, 0)} else {})
                   union renumVars(r, {c2 | c2 in set C2 & not isImplicitOrd(c2)})
         in if sat(mkAndExp({c} union newC)) then {mk_(e, newC)} else {}
       | mk_((pt) ^ [e] ^ -, lc) in set V}
       union
       dunion
       {let r = lifelineInds(l, t),
            newC = renumVars(r, {c2 | c2 in set lc.args & not isImplicitOrd(c2)})
         in if sat(mkAndExp({c} union newC)) then {mk_(mkStopEvent(l), newC)} else {}
       | mk_((pt), lc) in set V};

-- Similar to the above, but from the perspective of the transmission channels.
public traceExtCh: Interaction * TCTrace  -> set of (Event * (set of DiffConstr))
traceExtCh(sd, mk_(t,-)) ==
  {let s = t(i), -- send

       r = reception(sd, s),  -- receive
       C = {mk_DiffConstr(len t, len t + 1, 0)}
            union dunion {ev2ocConstr(i, len t + 1, c) | c in set sd.timeConstraints &
                          c.firstEvent = s.timestamp and c.secondEvent = r.timestamp}
   in mk_(r, C)

   | i in set inds t & isSend(t(i)) and count(t(i), t(1,...,i)) = count(reception(sd, t(i)), t) +
       1};
-- Note: assumes biunivoque relation between send and receive events
```

```
 -- 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 static isSend: Event -> bool
isSend(e) == e.type = <Send>;

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

-- Gets the reception event in an interaction (sd) corresponding to a send event (e).
-- Assumes that such event exists and is unique.
public static reception: Interaction * Event -> Event
reception(sd, e) ==
  let m in set sd.messages be st e = mk_Event(<Send>, m.signature, m.sendEvent.#1, m.
      sendTimestamp)
  in mk_Event(<Receive>, m.signature, m.receiveEvent.#1, m.recvTimestamp);

-- Checks if a set S1 of time constrained traces is a subset of another set S2,
-- in the sense that every timed trace defined by S1 is also a timed trace defined by S2.
public static isSubsetTimedTraces: (set of TCTrace) * (set of TCTrace) -> bool
isSubsetTimedTraces(S1, S2) == subtractTimedTraces(S1, S2) = {};

-- Obtains the sequence of indices of events in a trace 't' that occur at a lifeline 'l'.
public static 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>, (e.signature), -, -) in seq t] >
    len [ 0 | mk_Event(<Receive>, (e.signature), -, -) in seq t])
 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 Controllability

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

class Controllability is subclass of ConformanceChecking


functions

/*** Executable controllability checking procedures based on ***/
/**** simulated execution with time constrained traces      ****/


-- 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.
public static 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 static unintendedTraces: Interaction -> set of Trace
unintendedTraces(sd) ==
 if sd.timeConstraints = {} then unintendedTracesUntimed(sd) else unintendedTracesTimed(sd);


-- Determines the valid time traces that are not generated when lifelines
-- behave using local knowledge only (traces observed locally and traces valid locally).
public static missingTraces: Interaction -> set of Trace
missingTraces(sd) ==
  let V = validTimedTraces(sd),
      P = projectTimedTraces(V, sd.lifelines),

      S = simulExec(sd, V, P, mk_([], DC`mkTrueExp())),
      U = {mk_(t,c) | mk_(t,c,(<Pass>)/*-*/) in set S}
  in {t | mk_(t,-) in set subtractTimedTraces(V, U)};

/** Auxiliary private features - Utilities ***/

-- Checks (in a simplified way) if all message have been received in a trace (t).
-- It assumes the trace is defined consistently, that is, there no receptions
-- without corresponding emissions.
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))};
```

```
-- Checks if a difference constraint is a maximum duration constraint.
private static isMaxDuration: DC`DiffConstr -> bool
isMaxDuration(mk_DC`DiffConstr(i,j,d)) == i > j and d > 0;

-- Checks if a difference constraint is a minimum duration constraint.

private static isMinDuration: DC`DiffConstr -> bool
isMinDuration(mk_DC`DiffConstr(i,j,d)) == i < j and d <= 0;

-- Obtains an expression for the maximum value of a time variable 'v' defined by

-- a set 'D' of difference constraints.
private maxTimeInst: (set of DC`DiffConstr) * DC`VariableId -> DC`TimeExpr
maxTimeInst(D, v) ==
  DC`mkMinExp({DC`mkSumExp(j, d) | mk_DC`DiffConstr(i, j, d) in set D &
         isMaxDuration(mk_DC`DiffConstr(i,j, d)) and j < v and i = v});


-- Obtains an expression for the minimum value of a time variable 'v' defined by
-- a set 'D' of difference constraints.
private minTimeInst: (set of DC`DiffConstr) * nat -> DC`TimeExpr
minTimeInst(D, v) ==
  DC`mkMaxExp({DC`mkSumExp(i, -d) | mk_DC`DiffConstr(i, j, d) in set D &
          isMinDuration(mk_DC`DiffConstr(i,j, d)) and i < v and j = v});



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

-- Determines the invalid traces that can be generated 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).
-- Assumptions of lifeline behavior, depending on the set of next possible events at the lifeline
    :
--  (notation: ? - receive, ! - send, stop - there is a valid local trace that coincides with the
     observed trace):
--  {} -> fail, something went wrong (previous local trace isn't a prefix of any valid local
    trace)
--  {!x1, ..., !xn} -> choose one xi and send
--  {?y1, ..., ?ym} -> wait and fail if nothing is received after a sufficently long time
--  {stop} -> stop (fail if something is received)
--  {!x1, ..., !xn, stop} -> choose one xi and send or stop
--  {?y1, ..., ?ym, stop} -> wait and succeed if nothing is received after a sufficently long
    time
--  {!x1, ..., !xn, ?y1, ..., ?ym, stop} -> choose one xi and send, or wait and succeed if
    nothing is received after a sufficently long time
--  {!x1, ..., !xn, ?y1, ..., ?ym} -> choose one xi and send, or wait and fail if nothing is
    received after a sufficiently long time
private unintendedTracesUntimed: Interaction -> set of Trace
unintendedTracesUntimed(sd) ==
 let V = validTracesUntimed(sd),
     T = prefixes(V),

     L = sd.lifelines,
     P = projectTraces(V, L),
   us = {q ^ [e] | q in set T, p ^ [e] in set T & e.type = <Send>
       and projectTrace(q, e.lifeline) = projectTrace(p, e.lifeline)} \ T  ,
   ur =  dunion {{q  ^ [e] | q in set prefixes({p}) & isFeasibleAddition(q, e)}
                | p ^ [e] in set T & e.type = <Receive>} \ T,
  ut = {p | p in set T & allMsgsReceived(p) and mayRemainQuiescentUntimed(sd, p, P)} \ V
  in us union ur union ut;
```

```
-- 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>);

-- 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.
private static unintendedTracesTimed: Interaction -> set of Trace
unintendedTracesTimed(sd) ==
  let V = validTimedTraces(sd),
      L = sd.lifelines,

      P = projectTimedTraces(V, L),
      S = simulExec(sd, V, P, mk_([], DC`mkTrueExp()))
  in {t | mk_(t, -, (<Fail>)) in set S};


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

-- Recursively computes the sets of valid and invalid 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)
--  V - valid global time constrained traces
--  P - valid local time constrained traces per lifeline
--  (t, c) - time constrained trace generated so far (initially empty) (assumed to be valid)
private static simulExec: Interaction * (set of TCTrace) * (map Lifeline to set of TCTrace) *
    TCTrace -> set of (Trace *  DC`DiffConstrExp * Verdict)
simulExec(sd, V, P, mk_(t, c)) == (
 if is_DC`OrExp(c) then dunion {simulExec(sd, V, P, mk_(t, arg)) | arg in set c.args} --
     Optimization
 else
     let -- Set of all possible trace extensions, from the perspective of the lifelines:

         E =  dunion {traceExtLf(mk_(t,c), l, P(l)) | l in set sd.lifelines},
         -- Emission candidates at lifelines and respective local constraints (controllable):
         S =  {mk_(e, C, DC`mkTrueExp()) |  mk_(e, C) in set E & isSend(e)},
         -- Actual reception candidates, based on messages in transit, respective (controllable)
         -- transmission (inter-lifeline) constraints, and uncontrollable intra-lifeline
             constraints.
         R = {mk_(r, c2, DC`mkOrExp({DC`mkAndExp(c3)| mk_((r),c3) in set E}))
              | mk_(r, c2) in set traceExtCh(sd, mk_(t,c))},
        -- System overall emission deadline (for how long the system may remain quiescent):
        cS = quiescenceCond(sd, P, t, c, true, E),
         -- System overall reception deadline:
        cR = dunion {{ct | ct in set CC & isMaxDuration(ct)} | mk_(-,CC, -) in set R}
  in dunion
     {let CC2 = DC`reduceToFeasible(DC`mkAndExp({c} union CC union {cS} union cR))
      in if CC2 = nil
         then {}
         else if not exists mk_((t ^ [e]) ^ -, -) in set V & true
              then {mk_(t ^ [e], CC2, <Fail>)} -- invalid event (out of order)
              else
```

```
            (let c2 = DC`reduceToFeasible(DC`mkAndExp({CC2, DC`mkNotExp(UC)}))
            in if c2 = nil then {} else {mk_(t ^ [e], c2, <Fail>)}) -- invalid event (possible
                wrong timing)
            union
            (let c2 = DC`reduceToFeasible(DC`mkAndExp({CC2} union {UC}))
             in if c2 = nil then {} else simulExec(sd, V, P, mk_(t ^ [e], c2)))  -- valid event
                 (proceed recursively)
     | mk_(e, CC, UC) in set R union S}
     union
     -- termination
    if R = {} and DC`sat(quiescenceCond(sd, P, t, c, false, E))
     then if isSubsetTimedTraces({mk_(t,c)}, V)
          then {mk_(t, c, <Pass>)} -- valid termination
         else {mk_(t, c, <Fail>)} -- invalid termination
     else {}
);


-- If deadlineConstr is true, determines the difference constraint expression for the next
    emission event
-- in the system after a valid time constrained trace (t, c), assuming lifelines behave using
    local knowledge
-- only.
-- If deadlineConstr is false, determines the difference constraint expression for the
-- system to remain quiescent after a valid constrained trace (t, c) (without messages in transit
    ).
-- It are given the valid local traces per lifeline (P).
private quiescenceCond: Interaction * (map Lifeline to set of TCTrace) * Trace * DC`DiffConstrExp
     * bool * (set of (Event * (set of DC`DiffConstr)))-> DC`DiffConstrExp
quiescenceCond(sd, P, t, c, deadlineConstr, E) ==
    -- Optimization: if there are no possible emission events, the system will remain quiescent
    if not exists mk_(e,-) in set E & e.type = <Send> /*and e.lifeline in set L*/ then DC`
        mkTrueExp()
     else let -- Determine the satisfiability condition for each extension in E
              s = {C |-> DC`mkAndExp(DC`projectToVars(C, {1,...,len t})) | mk_(-, C) in set E},

              -- Determine the extensions that are optional, i.e., that may be not satisfiable (
                  optimization)
              opt = {mk_(e,C) | mk_(e,C) in set E & DC`isFeasible(DC`mkAndExp({c, DC`mkNotExp(s(C
                  ))}))}
          in DC`mkOrExp({
                DC`mkAndExp({c} union {s(C) | mk_(-,C) in set E \ rmv}
                                union {DC`mkNotExp(s(C)) | mk_(-,C) in set rmv}
                                union {getQuiescenceConstraint(sd, t,  E \ rmv, deadlineConstr)})
             | rmv in set power opt});

-- If deadlineConstr is true, gets the deadline constraint (if any) for the next emission event
    in the
-- system after a trace 't', assuming lifelines behave using local knowledge only.
-- If deadlineConstr is false, gives the constraint for the system to remain quiescent.
-- Receives the set of relevant lifelines (L), and possible next events (E).
-- A lifeline may remain quiescent if, for any emission event, there is a reception event with a
-- deadline greater or equal than the deadline of the emission event.
private getQuiescenceConstraint: Interaction * Trace * (set of (Event * (set of DC`DiffConstr)))
    * bool -> DC`DiffConstrExp
getQuiescenceConstraint(sd, t, E, deadlineConstr) ==
  DC`mkAndExp({
    let maxSend = DC`mkMaxExp({maxTimeInst(C, len t + 1) | mk_(e, C) in set E & e.type=<Send> and
         e.lifeline=l}),
        maxRecv = DC`mkMaxExp({maxTimeInst(C, len t + 1) | mk_(e, C) in set E & e.type=<Receive>
            and e.lifeline=l}),
        stopCond = DC`mkOrExp({DC`mkAndExp(C) | mk_(e, C) in set E & e.type = <Stop> and e.
            lifeline=l})

    in if deadlineConstr then DC`mkOrExp({stopCond,
```

```
                                          DC`mkLeqExp(maxSend, maxRecv),
                                          DC`mkLeqExp(DC`mkVarExp(len t + 1), maxSend)})
       else DC`mkOrExp({stopCond, DC`mkLeqExp(maxSend, maxRecv)})
  | l in set sd.lifelines});


 /****  Simulated execution with actual time instants  ****/



-- Simulates the execution of an interaction (sd), given the sets of valid timed traces (V),
-- and the maximum execution time. Returns the timed traces that can be generated.
private static simulExec: Interaction * (set of Trace)  -> set of Trace
simulExec(sd, V) == simulExec(sd, V, [], 0);


private static simulExec: Interaction * (set of Trace) * Trace * Time -> set of Trace
simulExec(sd, V, t, i) ==
   if not exists (t) ^ - in set V & true then {t} -- truncate on error
   else if allMsgsReceived(t) and forall l in set sd.lifelines &
            quiescent(projectTraces(V,l), projectTrace(t, l), i) then {t}

   else let N = {nextSend(projectTraces(V,l), projectTrace(t, l), i) | l in set sd.lifelines}
                 union {nextRecv(sd, m, t, i) | m in set sd.messages}
       in dunion {simulExec(sd, V, t ^ [e], i) | e in set dunion {x | mk_(x, -) in set N}}

      union (if not exists mk_(-, true) in set N & true then simulExec(sd, V, t, i + 1) else {});


private static nextSend: (set of Trace) * Trace * Time -> (set of Event) * bool
nextSend(T, t, i) ==
   mk_({e | (t) ^ [e] ^ - in set T & e.timestamp = i and e.type = <Send>},
       t not in set T and
       not exists (t) ^ [e] ^ - in set T &  e.timestamp > i or e.timestamp = i and e.type = <
           Receive>);


private static nextRecv: Interaction * Message * Trace * Time -> (set of Event) * bool
nextRecv(sd, m, t, i) ==
  let e = oldestPendingEmission(m, t), d = maxDelay(sd, m)
  in if e = nil then mk_({}, false) else mk_({reception(m, i)}, d <> nil and i - e.timestamp = d)
      ;

private static quiescent:  (set of Trace) * Trace * Time -> bool
quiescent(T, t, i) == not exists (t) ^ [e] ^ - in set T & e.timestamp >= i and e.type = <Send>;

-- Counts the number of occurrences of event 'e' in trace 't', ignoring timestamps
private count2: Event * Trace -> nat
count2(e,t) ==
   if t = [] then 0
   else (if mu(e, timestamp |-> 0) = mu(hd t, timestamp |-> 0) then 1 else 0) + count2(e, tl t);

private reception: Message * Time -> Event
reception(m, t) ==  mk_Event(<Receive>, m.signature, m.receiveEvent.#1, t);

private oldestPendingEmission: Message * Trace -> [Event]
oldestPendingEmission(m, t) ==
   let I = {i | i in set inds t &
                 t(i).type = <Send>

                 and t(i).signature = m.signature and t(i).lifeline =  m.sendEvent.#1
                 and count2(t(i), t(1,...,i)) = count2(reception(m, 0), t) + 1}
   in if I = {} then nil else let i in set I in t(i);

private maxDelay: Interaction * Message -> [Duration]
```

```
maxDelay(sd, m) ==

  let C = {c | c in set sd.timeConstraints & c.firstEvent = m.sendTimestamp and c.secondEvent = m
      .recvTimestamp}
  in if C = {} then nil else (let c in set C in c.max);




end Controllability
```

# 7 TestCases

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

class TestCases is subclass of Controllability


operations


operations

-- Simulates assertion checking by reducing it to pre-condition checking.
-- If 'arg' does not hold, a pre-condition violation will be signaled.

protected assertTrue: bool ==> ()
assertTrue(arg) ==
 return
pre 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}, {}),
      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({}, uncheckableLocally(sd1));
   assertEqual({}, unintendedTraces(sd1));
   assertEqual(<Pass>, finalConformanceChecking(sd1, {l1 |-> [e1, e4], l2 |-> [e2, e3]}));
    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 = 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], [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 = mkEvent(<Send>, "m1", l1),
      e2 = mkEvent(<Receive>, "m1", l2)
  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 = mkEvent(<Send>, "m1", l1),
      e2 = mkEvent(<Receive>, "m1", l2),
      e3 = mkEvent(<Send>, "m2", l1),
      e4 = mkEvent(<Receive>, "m2", l2)
  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 = mkEvent(<Send>, "m1", l1),
      e2 = mkEvent(<Receive>, "m1", l2),
      e3 = mkEvent(<Send>, "m2", l3),
      e4 = mkEvent(<Receive>, "m2", l2)
  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 = mkEvent(<Send>, "m1", l1),
      e2 = mkEvent(<Receive>, "m1", l2)
  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 = mkEvent(<Send>, "m1", l1),
      e2 = mkEvent(<Receive>, "m1", l2),
      e3 = mkEvent(<Send>, "m2", l3),
      e4 = mkEvent(<Receive>, "m2", l2)
  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),
      m2 = mk_Message(2, mk_(l2, 2), mk_(l1, 2), "m2", t3, t4),
      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),
      m2 = mk_Message(2, mk_(l2, 3), mk_(l1, 3), "m2", t3, t4),
      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, 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),

      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_(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(<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),
      m2 = mk_Message(2, mk_(l2, 2), mk_(l1, 2), "m2", t3, t4),
      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)), 0,
          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));
  )
);

public testAll() ==
(
  -- checking VDM++ language features in corner cases
   assertEqual({[0],[]}, {p | p ^ [1] in set {[0, 1], [1]}} );
   assertEqual([1, 2, 3], [1, 2, 3](1,...,8));

   assertEqual([],[1, 2, 3](1,...,0));

  testSimple();
  testOpt();
  testAlt();
  testLoop();
  testAltNested();
  testStrict();
  testRace();
  testNonLocalChoice();
  testIndepMessages();
  testTimeConstraintInLoop();
  testImpossible();
  testUnintendedEmptyTrace();
  testWhoSends();
  testUnintendedEmptyTrace2();
   testUnintendedEmptyTrace3();
  testTimeConstraint();
  testInterLifelineTimeConstraints();
   testVerdictWithTimestamps();
  testRaceByMsgOvertaking();
  testNonLocallyControlableTimed();
  testRcvConstraint();
  testBugFixCheckSatisfiability();
  testIsLocallyObservableTimed();
  testSendRecvSendConstraint();
  testStrangeControllableTimed();
```

```
   testSendSendSendConstraint();
  testFallDetection();
  testMayRemainQuiescentTimed();
  testSendableFirst();
  testSendableFirst2();
);

end TestCases
```