```
// Wood cutting model, finite state, "action-oriented" DSL style.
//
// (Potential) differences to actual machine model/optimization problem
// (to be considered in reduction of original problem):
//
// - the "Reorder" actions may have a different granularity than the actual ones
// - the "ReorderBoards" and "ReorderPieces" actions may leave a board/piece
//   in the buffer which is not considered in the optimization problem
// - the "Assembly" actions are performed on all pieces, even those that
//   may be not necessary any more to complete the desired number of beams
//   (this can be easily handled by introducing an additional state variable
//   that counts the number of beams produced)
//
system WoodCutting
{
  const LEN: Int; // maximum length of a board
  type Length = Int[0,LEN];
  type Interval = Record[from:Length,to:Length];

  // "good" intervals are not considered (have to be removed before)
  // also for a "curved" interval both endpoints are explicitly given
  type BoardIntervalType = { bad, curved };
  type BoardInterval = Record[type:IntervalType,interval:Interval];

  const BINUM: Int; // maximum number of intervals per board
  type BoardIntervalIndex = Int[0,BINUM];
  type BoardIntervals = Array[BINUM,BoardInterval];
  type Board = Record[length:Length,bintnum:BoardIntervalIndex,bints:BoardIntervals];

  const CNUM: Int = BINUM*2; // maximum number of cuts per board
  type CutIndex = Int[0,CNUM];
  type Cuts = Array[CNUM,Length];

  const NUM: Int; // number of boards
  type BoardIndex = Int[0,NUM];
  type Boards = Array[NUM,Board];

  type InPiece = Record[good:Bool, length:Length];
  type OutPiece = Length;

  const PLEN: Int; // minimal length of pieces

  const PNUM: Int = NUM*(CNUM+1); // maximum number of pieces
  type PieceIndex = Int[0,PNUM];
  type InPieces = Array[PNUM,InPiece];
  type OutPieces = Array[PNUM,OutPiece];

  // can be used to limit the decision search space
  const RBDNUM = NUM+1;  // number of reordering boards decisions (<= BNUM+1)
  const CBDNUM = NUM;    // number of cutting boards decisions (<= BNUM)
  const FPDNUM = PNUM;   // number of filtering pieces decisions (<= PNUM)
  const RPDNUM = PNUM+1; // number of reordering pieces decisions (<= PNUM+1)
  const APDNUM = PNUM;   // number of assembling decisions (<= PNUM)
```

```
const FNUM: Int; // maximum number of forbidden intervals
type ForbiddenIndex = Int[0,FNUM];
type ForbiddenIntervals = Array[FNUM,Interval];

const BLEN: Int;                // desired length of a beam
type BeamLength = Int[0,BLEN];  // actual length of beam
const BDEPTH: Int;              // desired number of layers
type BeamDepth = Int[0,BDEPTH]; // actual number of layers
const BNUM: Int;                // maximum number of pieces per beam
type BeamIndex = Int[0,BNUM];   // actual number of pieces
type BeamLengths = Array[PNUM,BeamLength];
const DIST: Length;             // minimum distance between two cuts

type Cost = Real; // need not be bounded

// the production line (consisting of multiple "stages")
pipeline Main(
  // input
  in inboards: Boards,

  // state (preserved from previous executions)
  inout bempty: Bool,
  inout buffer: Board,
  inout pempty: Bool,
  inout pbuffer: Board,

  // other parameters and cost
  in fints: ForbiddenIntervals,
  inout cost: Cost
)
{
  // requirement on input (add it to solver?)
  // constraint forall i:BoardIndex.
  //    value board: Board = inboards[i];
  //    board.interval.from < board.interval.to && // intervals are not empty
  //    if i+1 < BoardIndex
  //      then board.interval.to <= inboards[i+1].from
  //      else board.interval.to <= board.length;

  val outboards: OutBoards; // unconstrained at indices >= obnum
  var obnum: OutBoardIndex = 0;
  val inpieces: InPieces;   // unconstrained at indices >= ipnum
  var ipnum: InPieceIndex = 0;
  val outpieces: OutPieces; // unconstrained at indices >= ipnum
  var opnum: InPieceIndex = 0;
  in apieces: OutPieces; // unconstrained at indices >= apnum
  var apnum: OutPieceIndex = 0;

  // try at most RBDNUM reorder board decisions (if no action is possible,
  // perform a "dummy" action that leaves the state unchanged)
  // as a result with have *exactly* NUM boards in "outboards"
  // (may leave one board in "buffer")
  for i:Int[0,RBDNUM-1] do
  {
```

```
      try ReorderBoards(i,inboards,outboards,obnum,bempty,buffer);
    }

    // should be ensured by above
    // constraint obnum = NUM;

    // make exactly CBDNUM cut boards decisions (each with at most CNUM cut positions)
    // generates "ipnum" pieces in "inpieces"
    for i:Int[0,CBDNUM-1] do
    {
      Cut(i,outboards,inpieces,ipnum);
    }

    // try at most FPDNUM filtering decisions
    // generates "opnum" pieces in "outpieces"
    for i:Int[0,FPDNUM-1] do
    {
      try Filter(i,inpieces,ipnum,outpieces,opnum,cost);
    }

    // try at most RPDNUM reordering decisions (if no action is possible,
    // perform a "dummy" action that leaves the state unchanged)
    // as a result with have *exactly* "opnum" = "apnum" pieces in "apieces"
    // (may leave one piece in "pbuffer")
    for i:Int[0,RPDNUM-1] do
    {
      try ReorderPieces(i,outpieces,opnum,apieces,apnum,pempty,pbuffer);
    }

    // should be ensured by above
    // constraint apnum = opnum;

    // try at most APDNUM assembly decisions
    // (exactly "apnum" scheduling decisions *must* be performed)
    val blens: BeamLengths;
    var blen: BeamLength = 0;
    var bnum: BeamIndex = 0;
    var bdepth: BeamDepth = 0;
    var bnum0: BeamIndex = 0;
    for i:Int[0,APDNUM-1] do
    {
      try Assembly(i,apieces,apnum,fints,blens,blen,bnum,bdepth,bnum0);
    }
  }

  // the first reordering stage
  stage ReorderBoards(
    in i: BoardIndex,
    in inboards: Boards,
    in outboards: Boards, // unconstrained at indices >= obnum
    inout obnum: BoardIndex,
    inout bempty: Bool,
    inout buffer: Board
  )
```

```
{
  requires i < NUM && obnum < NUM;
  action forward()
  {
    in board: Board = inboards[i];
    outboards[obnum] = board; // equality, not assignment!
    obnum' = obnum+1;
    unchanged bempty, buffer;
  }
  action moveout()
  requires i < NUM && (bempty || onum < NUM);
  {
    in board: Board = inboards[i];
    !bempty => outboards[obnum] = buffer;
    obnum' = if bempty then obnum else obnum+1;
    bempty' = false;
    buffer' = board;
  }
  action movein()
  requires i = NUM && !bempty;
  {
    outboards[obnum] = buffer;
    obnum' = obnum+1;
    bempty' = true;
    // buffer' can be arbitrary
  }
}

// the cutting stage
stage Cut(
  in i: BoardIndex,
  in outboards: Boards,
  in inpieces: InPieces, // unconstrained at indices >= ipnum
  inout ipnum: PieceIndex
)
{
  action cut(cnum:CutIndex,cuts:Cuts)
  {
    val board: Board = outboards[i];
    // only allowed cuts
    constraint forall j: CutIndex with j < cnum.
      val start: Length = if j = 0 then 0 else cut[j-1];
      start < cut[j] && (j = cnum-1 => cut[j] < board.length) &&
      exists k: BoardIntervalIndex with k < board.bintsnum.
        val bint: BoardInterval = boards.bints[k].
        val from: Length = bint.interval.from;
        val to: Length = bint.interval.to;
        if bint.type = curved then
          from <= cut[j] && cut[j] <= to
        else // if bint.type = bad then
          (cut[j] = from && j+1 < cnum && cut[j+1] = to) ||
          (cut[j] = to && j-1 > 0 && cut[j-1] = from);
    // all necessary cuts
    constraint forall k: BoardIntervalIndex with k < board.bintsnum.
```

```
        val bint: BoardInterval = board.bints[k];
        if bint.type = curved then
          exists j: CutIndex with j < cnum.
            bint.interval.from <= cuts[j] && cuts[j] <= bint.interval.to;
        else // if cint.type = bad then
          exists j: CutIndex with j < cnum-1.
            bint.interval.from = cut[j] && bint.interval.to = cut[j+1];
      // the resulting piece (can be combined with "only allowed cuts")
      constraint forall j: CutIndex with j < cnum.
        var inpiece: InPiece = inpieces[ipnum+j];
        val start: Length = if j = 0 then 0 else cut[j-1];
        inpiece.length = cut[j]-start &&
        inpiece.good =
          ~exists k: BoardIndex with k < board.bintsnum.
            val bint: BoardInterval = boards.bints[k].
            bint.type = bad && start = bint.interval.from;
      ipnum' = ipnum+cnum;
    }
  }

  // the filtering stage
  stage Filter(
    in i: PieceIndex,
    in inpieces: InPieces,
    in ipnum: PieceIndex,
    in outpieces: OutPieces, // unconstrained at indices >= opnum
    inout opnum: OutPieceIndex,
    inout cost: Cost;
  )
  {
    requires i < ipnum;
    action keep()
    {
      val piece: Piece = inpieces[i];
      constraint piece.good && piece.length >= PLEN;
      outpieces[opnum] = piece.length; // equality, not assignment!
      opnum' = opnum+1;
      unchanged cost;
    }
    requires i < ipnum;
    action discard()
    {
      val piece: Piece = inpieces[i];
      cost' = if piece.good then cost+piece.length else cost;
      unchanged opnum;
    }
  }

  // the second reordering stage
  stage ReorderPieces(
    in i: PieceIndex,
    in outpieces: OutPieces,
    in opnum: PieceIndex,
    in apieces: OutPieces, // unconstrained at indices >= apnum
```

```
    inout apnum: PieceIndex,
    inout pempty: Bool,
    inout pbuffer: Piece
  )
  {
    action forward()
    requires i < opnum && apnum < PNUM;
    {
      val piece: OutPiece = outpieces[i];
      apieces[apnum] = piece; // equality, not assignment!
      apnum' = apnum+1;
      unchanged pempty, pbuffer;
    }
    action moveout()
    requires i < opnum && (pempty || apnum < PNUM);
    {
      val piece: InPiece = outpieces[i];
      !pempty => apieces[apnum] = pbuffer;
      apnum' = if pempty then apnum else apnum+1;
      pempty' = false;
      pbuffer' = piece;
    }
    action movein()
    requires i = opnum && !pempty;
    {
      apieces[apnum] = pbuffer;
      apnum' = apnum+1;
      pempty' = true;
      // pbuffer' can be arbitrary
    }
  }

  // the assembly stage
  stage Assembly(
    in i: PieceIndex,
    in apieces: OutPieces,
    in apnum: PieceIndex,
    in fints: ForbiddenIntervals,
    in blens: BeamLengths, // unconstrained at indices >= i
    inout blen: BeamLength,
    inout bnum: BeamIndex,
    inout bdepth: BeamDepth,
    inout bnum0: BeamIndex
  )
  {
    action accept()
    requires i < apnum;
    {
      val blen0: BeamLength = blen+apieces[i];
      blens[i] = blen0;
      constraint blen0 <= BLEN;
      constraint !exists j:ForbiddenIndex with j < FNUM.
        fints[j].from <= blen0 && blen0 <= fints[j].to;
      constraint forall j:BeamIndex with j < bnum0.
```

```
        value diff: BeamLength = blen0-blens[i-bnum-bnum0+j];
        DIFF <= if diff >= 0 then diff else -diff;
      if blen0 < BLEN then
        blen' = blen0;
        bnum' = bnum+1;
        unchanged bdepth, bnum0;
      else
        blen' = 0;
        bnum' = 0;
        if bdepth = BDEPTH then
          bdepth' = 0;
          bnum0' = 0;
        else
          bdepth' = bdepth+1;
          bnum0' = bnum;
    }
  }
}
```