PALM Technical Report

September 2020

1 Mining tool-independent specification

The mining algorithm is computed by the SchimmAlgorithm object while it is instantiated. While every method represents a step of the algorithm, only unifyAllBlockStructure aggregates 2 steps that are **4 th step - model for each cluster** and **5th step - unify all block structure**.

```
1  public SchimmAlgorithm(EventLog log) {
2   //Set containing all the loops found in this eventlog
3   this.loops = new HashSet<LoopSet>();
4   this.log = log;
5   //1st - search for loops
6   searchForLoops();
7   //2nd step - creation of clusters
8   Set<Trace> cluster = creationOfClusters();
9   //3rd step - identification and removal of
        pseudo-dependencies
10  cluster =
        identificationAndRemovalOfPseudoDependencies(cluster);
11  //4th step - model for each cluster && 5th step unify all
        block structure
12  BlockStructure blockStructure =
        unifyAllBlockStructure(cluster)
13  //6th step - restructuring the model
14  blockStructure = restructuringTheModel(blockStructure);
15  //7th step - replacing loop references
16  this.modelToTransf = replacingLoopReference(blockStructure);
17 }
```

Listing 1: SchimmAlgorithm class constructor

The first step executed in the computation is the **search for loops** method. This method detects all the loops in the event log, substituting them with their process reference. This process works as following: For each detected loop l=$\{t = \langle e_1 ... e_i ... e_j ... e_n \rangle$ s.t. $e_j > e_i \Rightarrow l = e_i ... e_j\}$ and for each instance of l in t we need to substitute the reference to the process l in t updating the happened-before relation of t in the following way $e_{i-1} > e_i$ and $e_j > e_{j+1}$, if there is more than one instance of l next to each other, just one reference will be inserted.

```
1  private void searchForLoops() {
2   for (Trace t : log) {
```

```
3     for (int i = 0; i < t.length(); i++) {
4      Event e = t.getEvent(i);
5      Set<Event> incomingEventsofE = t.getPreEventHB(e);
6      Trace tmp = t.getSubTraceFrom(i, t.length());
7       for (int j = 1; j < tmp.length(); j++) {
8        Event ei = tmp.getEvent(j);
9        if (incomingEventsofE.contains(ei)) {
10       // counts the number of loop's occurrences inside the trace
11        int occ = 0;
12        //If exists, retrieve a loop already in loops
13        LoopSet loopSet = retriveLoop(e, ei);
14        Trace loopTrace;
15        int lenght = 0;
16        do {
17         //The trace that is a loop
18         loopTrace = t.getSubTrace(e, ei);
19         lenght = loopTrace.length();
20         //Set the happened-before relation of the trace in the loop
21         loopTrace.setHBrel(t.getSubHBrel(loopTrace));
22         loopSet.addLoop(loopTrace);
23         loops.add(loopSet);
24         // Remove an instance of the loop trace
25         int start_index = t.removeSubTrace(loopTrace);
26         occ += 1;
27         if (!(((start_index - 1) < 0) && (start_index + 1) >=
               t.length())) {
28          if (!t.getEvent(start_index -
                1).equals(loopSet.getName())){
29           t.add(start_index, loopSet.getName());
30          if (start_index > 0)
31           t.addPreHBRelation(t.getEvent(start_index - 1),
                 loopSet.getName());
32          if ((start_index + 1) < t.length())
                          t.addPostHBRelation(t.getEvent(start_index
               + 1), loopSet.getName());
33         } else {
34          if (start_index < t.length())
35           t.addPostHBRelation(t.getEvent(start_index),loopSet.getName());
36         }
37        }
38       } while (!(loopTrace = t.getSubTrace(e, ei)).isEmpty());
39         //Sets how many time the loop is repeated in this trace
40         loopSet.setRepetition(occ);
41         //Update frequency of the loop int the trace
42         t.addLoopWithFrequency(loopSet.getName().getName(), lenght,
               occ);
43               break;
44 }}}}}
```

Listing 2: searchForLoop() method

From line 27 until line 32 we update the happened-before relation of the loop reference as following: If the a reference to this loop doesn't exists immediately on the left or of the trace then it is added to the trace and updates the happened-before relation with $e_i > e$ and $e > e_j$, otherwise just update the post relation of the reference with the element immediately at its right.

The second method groups all the trace with the same alphabet and the

same happened-before relation in one trace. The set of these traces creates the cluster that will be analyzed later.

```
private Set<Trace> creationOfClusters() {
 Set<Trace> cluster = new HashSet<Trace>();
 for (Trace t : log) {
  Map<Event, Set<Event>> hbt = t.getHBRel();
  boolean alreadyExist = false;
  for (Trace c : cluster) {
    //Check if a trace in the cluster has the hb-relation
        equal to the trace t
    if (c.equalHB(hbt)) {
       alreadyExist = true;
       break;
     }
   }
   if (!alreadyExist)
     cluster.add(t);
 }
 return cluster;}
```

Listing 3: Creation of cluster method

Now that the cluster has been computed we can identify and remove the traces that contains the pseudo-dependencies. In line 11-16 we check if the current trace has a relation of precedence s.t. $e_i > e_j$ or $ej > e_i$, than in every trace with the same alphabet should exists this relation, otherwise the current trace contains a pseudo-dependency and needs to be removed. In line 17-21 we check the opposite scenario, that is,if there is not a relation of precedence between e_i and e_j in the current trace then the same should be for all the other traces with the same alphabet in the cluster, the traces found with that relation are removed from the cluster.

```
1  private Set<Trace> identificationAndRemovalOfPseudoDependencie
2   (Set<Trace> cluster){
3  Set<Trace> clusterToRemove = new HashSet<Trace>();
4  for (Trace t1 : cluster) {
5   if (!Sets.difference(cluster, clusterToRemove).contains(t1))
6     continue;
7   for (int i = 0; i < t1.length(); i++) {
8    Event ei = t1.getEvent(i);
9    for (int j = i + 1; j < t1.length(); j++) {
10    Event ej = t1.getEvent(j);
11    if (t1.containsRelation(ei, ej) || t1.containsRelation(ej, ej))
        {
12     for (Trace t2 : Sets.difference(cluster, clusterToRemove)) {
13      if (!t2.equals(t1) &&
          t2.getAlphabet().equals(t1.getAlphabet()) &&
          !t2.containsRelation(ei, ej) && !t2.containsRelation(ej,
          ej)) {
14       clusterToRemove.add(t1);
15       }
16     }
```

```
17      } else if (!t1.containsRelation(ei, ej) &&
            !t1.containsRelation(ej, ej)) {
18       for (Trace t2 : Sets.difference(cluster, clusterToRemove)) {
19        if (!t2.equals(t1) &&
              t2.getAlphabet().equals(t1.getAlphabet()) &&
              (t2.containsRelation(ei, ej) || t2.containsRelation(ej,
              ej))) {
20         clusterToRemove.add(t2);
21   }}}}}}
22   if (!clusterToRemove.isEmpty())
23    cluster.removeAll(clusterToRemove);
24    return cluster;
25  }
```

Listing 4: Identification and removal of pseudo-dependencies

After that all the pseudo dependencies have been removed the first coarse block structure can be constructed. This method contains the 4th and 5th step of the algorithm, first compute the model of each cluster and then put them together using the choice operator if needed, i.e. if there is more than one model.

```
1   private BlockStructure unifyAllBlockStructure(Set<Trace>
        cluster) {
2    BlockStructure[] realFinalPath = new
         BlockStructure[cluster.size()];
3    int i = 0;
4    for (Trace c : cluster) {
5     //The model of cluster c is computed
6     BlockStructure p = modelForEachCluster(c);
7     realFinalPath[i] = p;
8     i++;
9     }
10    if (realFinalPath.length == 1)
11     return realFinalPath[0];
12    else
13     return new BlockStructure(realFinalPath, Operator.CHOICE);
14   }
```

Listing 5: Unifiy all block structure method

In line 6 is called the method to construct a block structure from a single cluster, (4th step - model for each cluster). Every path obtained from the cluster is represented as a sequence block, all the paths are then combined togheter into a parallel block structure.

```
1   private BlockStructure modelForEachCluster(Trace t) {
2    List<List<Event>> path = t.computePaths();
3    BlockStructure[] toPutInParallel = new
         BlockStructure[path.size()];
4    int i = 0;
5    for (List<Event> list : path) {
6     BlockStructure[] tmp = new BlockStructure[list.size()];
7     for (int j = 0; j < list.size(); j++)
8      tmp[j] = new BlockStructure(list.get(j));
```

```
9       toPutInParallel[i] = new BlockStructure(tmp,
            Operator.SEQUENCE);
10      i++;
11     }
12    if (toPutInParallel.length < 2)
13     return toPutInParallel[0];
14    else
15     return new BlockStructure(toPutInParallel,
           Operator.PARALLEL);
16  }
```

Listing 6: Model for each cluster

At the end of the unify method we have a coarse block structure that does not represents the real behaviour yet. In the restructuring method we have implemented the set of rules that we apply in order to obtain the final block structure.

```
1   private BlockStructure restructuringTheModel(BlockStructure
        b) {
2    BlockStructure newBlock = null;
3    if (b.hasEvent())
4     return b;
5    else if ((b.getOp().equals(Operator.CHOICE) ||
         b.getOp().equals(Operator.PARALLEL) ||
         b.getOp().equals(Operator.SEQUENCE)) && b.size() == 1) {
6     // S{B}->B , C{B}->B , P{B}-> B
7     newBlock = TransformationRule.removeOperator(b);
8    } else if (b.blockWithSamrOperator()) {
9     // Op{B1...Op{e1..en}...Bm} -> Op{B1..., e1 ,..., en ,...Bm}
10    newBlock = TransformationRule.identity(b);
11   } else if ((b.getOp().equals(Operator.PARALLEL) ||
         b.getOp().equals(Operator.CHOICE)) && b.getFirstRowOp()
         != null && b.getFirstRowOp().equals(Operator.SEQUENCE)) {
12    int number = 0;
13    // Commutative of parallel and choice operator
14    if ((number = howManyBlock(b, LEFT)) != 0)
15     newBlock = TransformationRule.mergeSide(b, LEFT, number);
16    else if ((number = howManyBlock(b, RIGHT)) != 0)
17     newBlock = TransformationRule.mergeSide(b, RIGHT, number);
18    }
19   if (newBlock == null) {
20    newBlock = new BlockStructure(b.getOp());
21    for (int i = 0; i < b.size(); i++)
22     newBlock.addBlockAtPosition(restructuringTheModel(b.getBlock(i)),
           i);
23   }
24   if (!b.equals(newBlock))
25    b = restructuringTheModel(newBlock);
```

```
26   return b;}
```

Listing 7: Restructuring the model method

At the start of the this computation we substituted the subtraces generating loops with a reference. Now it's time to substitute that reference with the result of the algorithm applied over the subtrace.

```
1  private BlockStructure replacingLoopReference(BlockStructure
       b) {
2   if (loops.isEmpty())
3    return b;
4   Map<Event, BlockStructure> loopNametoLoopBS = new
        HashMap<Event, BlockStructure>();
5   for (LoopSet l : loops) {
6    SchimmAlgorithm alg = new SchimmAlgorithm(l.getLoop());
7    BlockStructure bl = new BlockStructure(Operator.LOOP);
8    bl.addBlockAtPosition(alg.getFinalModel(), 0);
9    bl.setRepetition(l.getRepetition());
10   bl.setFrequency(log.getFrequencyLoop(l));
11   loopNametoLoopBS.put(l.getName(), bl);
12  }
13  //Method to substitute the loop reference with the
       corresponding block structure
14  return replaceReferences(loopNametoLoopBS, b);
15 }
```

Listing 8: Replacing loop reference method

At last, we generated a block structure for a single eventlog that can be retrieved by using the **getFinalModel()** method of the SchimmAlgorithm class.

2 Aggregation

When we want to analyze more than one eventlog and generate an overall specification of the system we can use the aggregate function. This method applies on a list of mcrl2 specification objects, this means that we apply the algorithm explained above for all the eventlog separately. Once that we a mcrl2 object for each eventlog we can unify their sets of actions and messages and also the allow, hide and communication sets. Every initial process in added to a common init set that will be represented as the parallel of initial processes.

```
1  public static String mergeMCRL2(List<MCRL2> mcrl2list) {
2  MCRL2 unicspec = new MCRL2();
3  mcrl2list.forEach(l -> {
4  unicspec.addActSet(l.getActSet());
5  if (l.getAllowedAction().isEmpty())
6   unicspec.addAllowedAction(l.getActSet());
7  else
8  //Union of act, comm, allow and hide sets
9   unicspec.addAllowedAction(l.getAllowedAction());
```

```
10   unicspec.addHideAction(l.getHideAction());
11   unicspec.addCommFunction(l.getCommFunction());
12   unicspec.addInitSet(l.getInitSet());
13   unicspec.appendMessage(l.getMessage());
14   unicspec.addProcSpec(l.getProcspec());
15  });
16  //A communication function among events with the same message
        is generated
17  for (Entry<Event, Collection<Event>> m :
        unicspec.getMessage().asMap().entrySet()) {
18   Event[] a = new Event[m.getValue().size()];
19   unicspec.addCommFunction(m.getValue().toArray(a),
         m.getKey());
20   unicspec.addActSet(m.getKey());
21   unicspec.addAllowedAction(m.getKey());
22   for(Event e : a)
23           unicspec.removedAllowedAction(e);
24  }
25  return generateMcrl2File(unicspec);
26  }
```

Listing 9: Aggregation

3 Running example

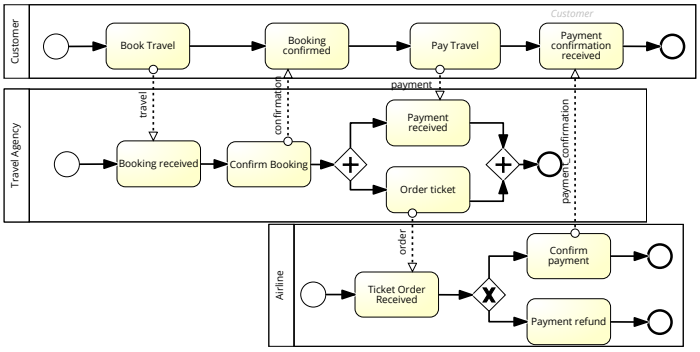

Figure 1: Running example

The following listings report each mCRL2 specification obtained from the event logs corresponding to each participant of the running example.

```
act
BookTravel, PayTravel, Paymentconfirmationreceived, Bookingconfirmed;
proc
```

```
P0=(BookTravel.Bookingconfirmed.PayTravel.Paymentconfirmationreceived);
init P0;
```

Listing 10: mCRL2 specification generated from the Customer log.

```
act
Confirmpayment,TicketOrderReceived,Paymentrefund;
proc
P1=(TicketOrderReceived.(Paymentrefund+Confirmpayment));
init P1;
```

Listing 11: mCRL2 specification generated from the Airline log.

```
act
Orderticket,t,Bookingreceived,Paymentreceived,t0,ConfirmBooking;
proc
P2=((Bookingreceived.ConfirmBooking.t0.Paymentreceived.t0)
    ||(t0.Orderticket.t0));
init hide({t},
    allow({Orderticket,t,Bookingreceived,Paymentreceived,ConfirmBooking},
          comm({t0|t0->t},
           P2)));
```

Listing 12: mCRL2 specification generated from the Travel agency log.

Following we have the specification of the aggregated logs

```
act
Confirmpayment,BookTravel,Bookingreceived,Paymentreceived,Paymentrefund,
Bookingconfirmed,ConfirmBooking,confirmation,Orderticket,
TicketOrderReceived,PayTravel,t,Paymentconfirmationreceived,payment,
payment_confirmation,t0,order,travel;
proc
P0=(TicketOrderReceived.(Paymentrefund+Confirmpayment));
P1=(BookTravel.Bookingconfirmed.PayTravel.Paymentconfirmationreceived);
P2=((Bookingreceived.ConfirmBooking.t0.Paymentreceived.t0)
    ||(t0.Orderticket.t0));
init
hide({t},allow({Paymentrefund,confirmation,t,payment,payment_confirmation,
                order,travel},
  comm({Bookingreceived|BookTravel->travel,
        Confirmpayment|Paymentconfirmationreceived->payment_confirmation,
        Orderticket|TicketOrderReceived->order,
        PayTravel|Paymentreceived->payment,
        Bookingconfirmed|ConfirmBooking->confirmation,t0|t0->t},
    P0||P1||P2)));
```

Listing 13: mCRL2 aggregate specification of the running example.

4 Replicate experiments

The tool is equipped with a way to replicate part of the experiments. That is, the execution of PALM to generate a mCRL2 specification measuring the execution time and its MC-fitness value. Given the same log is able to check which type of equivalence exist between the specification and the coverability graph, then transformed in a mCRL2 specification as well, obtained from the other algorithms. Still, is missing the integration on how to generate the coverability graph given an eventlog applying the IM, sHM and SM algorithms. For the moment, in order to replicate this step the user should follow these steps:

- Download and extract the logs.zip archive from the GitHub repository.
- Execute the TKDE benchmark tool using this line on the terminal.

```
java -jar tkde\_benchmark\_v2.0.jar -ext
    ".\logs-folder" -miners 0 8 2 -metrics 0
```

  The output will be a BPMN model for each log in the folder and algorithm used, and in the csv file there is the execution time.
- For each BPMN generated use the plug-ins in the ProM framework in the following order:
  1. "Convert BPMN diagram to Petri net"
  2. "Construct coverability graph of a Petri Net"
- The result will be a file ".sg" containing the coverability graph. Download the file inside the folder corresponding to its eventlog inside the "logs" folder and rename it as *namelog_namealgorithm.sg* (example: log1_IM.sg).
- Execute the PALM tool and select "[3] Repeat experiments" and wait for the "result.csv" to be generated.