Executing alignments outside the interface in ProM 6.x

This document covers the execution of alignments in the context of ProM being run without an interface (or inside Eclipse, or from the command line). The document comprises a set of requirements, and an example application.

Requirement #1:

- ILP libraries (because if the model contains much concurrency, it helps saving out a lot of time).

The ILP libraries needed to run alignments can be downloaded from the following URL:

https://svn.win.tue.nl/repos/prom/Packages/Alignment/Trunk/dll/

And need to be placed in the system path to be accessible from the system path:

- On Windows operating systems, they need to be placed under **C:\Windows\System32**. Moreover, **C:\Windows\System32** needs to be added to the path if already is not there.
- On Linux systems, they need to be placed in a folder contained in both the **LD_LIBRARY_PATH** and the path. To set these variables in Linux, the following commands need to be executed:

export LD_LIBRARY_PATH=%LD_LIBRARY_PATH:/folder/containing/the/libraries

export PATH=%PATH:/folder/containing/the/libraries

Requirement #2:

The “PNRepResult” package needs to be inserted in the **ivy.xml** as dependency. Example:

```
<ivy-module version="2.0">
    <info organisation="prom" module="AlphaMiner" revision="latest">
        <description>
            Version VERSION
        </description>
    </info>
    <dependencies>
       <dependency org="prom" name="ProM-Plugins" rev="latest" changing="true" transitive="true"/>
       <dependency org="prom" name="AcceptingPetriNet" rev="latest" changing="true" transitive="true"/>
       <dependency org="prom" name="Log" rev="latest" changing="true" transitive="true" />
       <dependency org="prom" name="LogAbstractions" rev="latest" changing="true" transitive="true" />
       <dependency org="prom" name="PetriNets" rev="latest" changing="true" transitive="true" />
             <dependency org="prom" name="XESLite" rev="latest" changing="true" transitive="true" />
                    <dependency org="prom" name="PNetReplayer" rev="latest" changing="true" transitive="true" />
    </dependencies>
</ivy-module>
```

Example of functioning:

Let's take the event log and Petri net reading functions from the **PromWithoutProm** document:

Log (imported from a XES file):

```
	public static XLog readSingleLog(String fileName) throws Exception {
		File initialFile = new File(fileName);
		InputStream inputStream = new FileInputStream(initialFile);
		XesLiteXmlParser parser = new XesLiteXmlParser(true);
		List<XLog> parsedLogs = parser.parse(inputStream);

		if (parsedLogs.size() > 0) {
			return parsedLogs.get(0);
		}
		return null;
	}
```

Petri net (imported from a PNML file):

```
	public static Pnml importPnmlFromStream(InputStream input,
			String filename, long fileSizeInBytes) throws
XmlPullParserException, IOException {
		FullPnmlElementFactory pnmlFactory = new FullPnmlElementFactory();

		XmlPullParserFactory factory = XmlPullParserFactory.newInstance();
		factory.setNamespaceAware(true);
		XmlPullParser xpp = factory.newPullParser();
		xpp.setInput(input, null);
		int eventType = xpp.getEventType();

		Pnml pnml = new Pnml();
		synchronized (pnmlFactory) {
			pnml.setFactory(pnmlFactory);

			/*
			 * Skip whatever we find until we've found a start tag.
			 */
			while (eventType != XmlPullParser.START_TAG) {
				eventType = xpp.next();
			}
			/*
			 * Check whether start tag corresponds to PNML start tag.
```

```
                     */
                    if (xpp.getName().equals(Pnml.TAG)) {
                          /*
                           * Yes it does. Import the PNML element.
                           */
                          pnml.importElement(xpp, pnml);
                    } else {
                          /*
                           * No it does not. Return null to signal failure.
                           */
                          pnml.log(Pnml.TAG, xpp.getLineNumber(), "Expected
pnml");
                    }
                    if (pnml.hasErrors()) {
                          return null;
                    }
                    return pnml;
              }
        }

        public static Object[] connectNet(Pnml pnml, PetrinetGraph net) {
              /*
               * Return the net and the marking.
               */
              Marking marking = new Marking();
              Collection<Marking> finalMarkings = new HashSet<Marking>();
              GraphLayoutConnection layout = new GraphLayoutConnection(net);

              pnml.convertToNet(net, marking, finalMarkings, layout);

              Object[] objects = new Object[2];
              objects[0] = net;
              objects[1] = marking;
              return objects;
        }

        public static Object[] importFromStream(InputStream input,
                    String filename, long fileSizeInBytes) throws
XmlPullParserException, IOException {
              Pnml pnml = importPnmlFromStream(input, filename, fileSizeInBytes);

              if (pnml == null) {
                    /*
                     * No PNML found in file. Fail.
                     */
                    return null;
              }

              PetrinetGraph net = PetrinetFactory.newPetrinet(pnml.getLabel());

              return connectNet(pnml, net);
        }

        public static Object[] importFromFile(String filename) throws Exception {
              File file = new File(filename);
              return importFromStream(new FileInputStream(file), filename,
file.length());
        }
```

Then, the next step is about creating the Petri net auxiliary objects and the mapping between the event log and the Petri net object. Then, the alignments can be easily performed.

Function to retrieve the final marking from a Petri net (just checks places without outer connections):

```
	public static Marking getFinalMarking(PetrinetGraph net) {
		Marking finalMarking = new Marking();

		for (Place p : net.getPlaces()) {
			if (net.getGraph().getOutEdges(p).size() == 0) {
				finalMarking.add(p);
			}
		}

		return finalMarking;
	}
```

Function that performs the alignments:

```
	public static PNRepResult executeAlignments(XLog log, PetrinetGraph net,
Marking initialMarking, Marking finalMarking) throws AStarException {
		XEventClass evClassDummy = new XEventClass("DUMMY", -1);

		TransEvClassMapping mapping = new
TransEvClassMapping(XLogInfoImpl.STANDARD_CLASSIFIER, evClassDummy);
		XLogInfo logInfo = XLogInfoFactory.createLogInfo(log);

		for (XEventClass ec : logInfo.getEventClasses().getClasses()) {
			for (Transition t : net.getTransitions()) {
				if (t.getLabel().equals(ec.toString().substring(0,
ec.toString().length()-1))) {
					mapping.put(t,  ec);
				}
			}
		}

		CostBasedCompleteParam parameter = new
CostBasedCompleteParam(logInfo.getEventClasses().getClasses(),
				evClassDummy, net.getTransitions(), 2, 5);
		parameter.setGUIMode(false);
		parameter.setCreateConn(false);

		parameter.setInitialMarking(initialMarking);
		parameter.setFinalMarkings(finalMarking);
		parameter.setMaxNumOfStates(200000);

		PluginContext context = null;

		PetrinetReplayerWithILP replWithoutILP = new
PetrinetReplayerWithILP();

		PNLogReplayer replayer = new PNLogReplayer();
		PNRepResult pnRepResult = replayer.replayLog(context, net, log,
mapping, replWithoutILP, parameter);
```

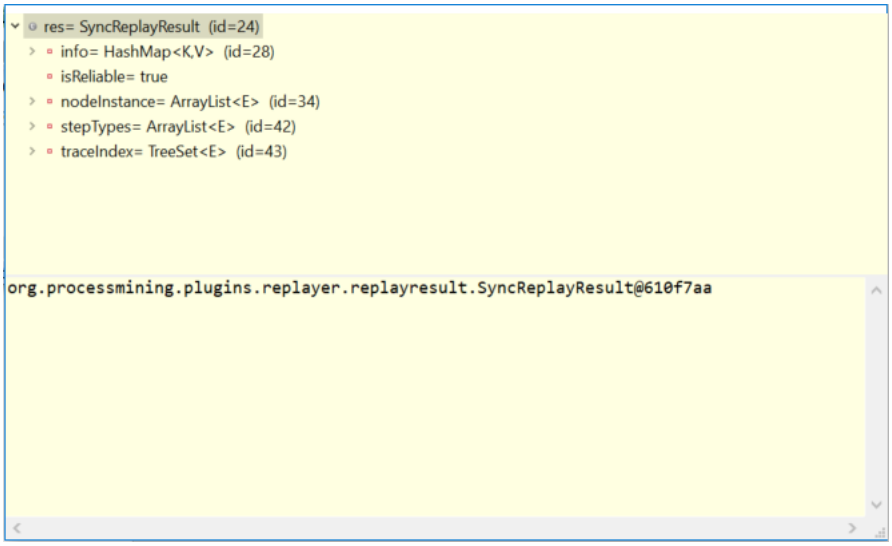
```
        return pnRepResult;
    }
```

Let's recap the contents of the function:

- A mapping is created associating each class (activity) of the log to the corresponding transition (that has the same label). The key of the TransEvClassMapping object is the transition; the value is the event class.
- Some parameters are set to start the replay. The GUI mode is turned off; the creation of connections also. Moreover, a limit to the number of states is set in such way that the replay is not unbounded in time.
- We create a "null" **PluginContext**.
- We create a replayer. We have some choices:
    - **PetrinetReplayerWithILP** => ILP based replay (faster with the models with concurrency).
    - **PetrinetReplayerWithoutILP** => Dijkstra-based replayer (faster with smaller models).
- We execute the replay getting a **PNRepResult** object containing the replay of the result of each variant of the log.

Indeed, to maximize the execution speed, only one alignment per variant is executed. The object for each variant has class **SyncReplayResult**. Let's see the structure of this object.

Structure of the SyncReplayResult object



- The **isReliable** boolean tells if the alignment of the trace can be trusted (when the replay succeeds, it is set to True; when for some reasons a

path from the initial marking to the final marking cannot be found or cannot be found in a reasonsable time, then is not reliable.

- **traceIndex** contains the indexes of the cases of the log sharing that variant. While the SyncReplayResult does not contain any direct reference to the list of the activities of the variant, the list of activities can be easily retrieved from the replay result.
- The **nodeInstance** contains a list of nodes that form a path in the model from the initial marking to the final marking. Important attributes of a transition are:
    - The label (from getLabel())
    - The visibility (checking isInvisible()
- The **stepTypes** contains the step of the alignment. Each step is of type StepTypes. To check the type of the move, the following code snippet may be useful:

```
for (StepTypes st : res.getStepTypes()) {
    if (st == StepTypes.LMGOOD) {
        System.out.println("Sync move");
    }
    else if (st == StepTypes.L) {
        System.out.println("Log move");
    }
    else if (st == StepTypes.MINVI) {
        System.out.println("Invisible step");
    }
    else if (st == StepTypes.MREAL) {
        System.out.println("Model move");
    }
}
```

Getting the time in which each transition fired

The following code snippet (unexplained) is proposed to get the time in which each transition fired from the alignments. The output of the method is a list (each item is referring to a different case; each case is a list of pairs where the first item is a transition, and the second item is the timestamp in which the transition is executed):

```
public List<List<Pair<Transition, Long>>> getTransitionsExecutionTimePerTrace(XLog log, Petrinet net, PNRepResult res) {

        Map<Integer, List<Pair<Transition, Long>>> intermediate = new HashMap<Integer, List<Pair<Transition, Long>>>();

        for (SyncReplayResult singleVariantReplay : res) {

            Set<Integer> allTraceIdxOfThisVariant = singleVariantReplay.getTraceIndex();
```

```
                        List<StepTypes> stepTypes = singleVariantReplay.getStepTypes();
                        List<Object> nodeInstance = singleVariantReplay.getNodeInstance();
                        for (Integer traceIdx : allTraceIdxOfThisVariant) {
                                List<Pair<Transition, Long>> thisTraceReplay = new
ArrayList<Pair<Transition, Long>>();
                                Integer idxOfModel = 0;
                                Integer idxOfLog = -1;
                                while (idxOfModel < nodeInstance.size()) {
                                        StepTypes thisStepType = stepTypes.get(idxOfModel);
                                        if (nodeInstance.get(idxOfModel) instanceof Transition) {
                                                if (thisStepType.toString().equals("Sync move")) {
                                                        idxOfLog++;
                                                }
                                                if (idxOfLog > 0) {
                                                        Transition thisTransition = (Transition)
nodeInstance.get(idxOfModel);
                                                        Long millisecondEventTime =
XExtendedEvent.wrap(log.get(traceIdx).get(idxOfLog)).getTimestamp().getTime();
                                                        thisTraceReplay.add(new Pair(thisTransition,
millisecondEventTime));
                                                }
                                        }
                                        else {
                                                idxOfLog++;
                                        }
                                        idxOfModel++;
                                }
                                intermediate.put(traceIdx, thisTraceReplay);
                        }
                }
                List<List<Pair<Transition, Long>>> listOfTransTimesPerTrace = new
ArrayList<List<Pair<Transition, Long>>>();
                return listOfTransTimesPerTrace;
        }
```