DiRPOMS: Automatic Checker of Distributed Realizability of POMSets

Roberto Guanciale

DiRPOMS: automatic checker of Distributed Realizability of POMSets

Roberto Guanciale[1 [0000−1111−2222−3333]]

KTH, Sweden robertog@kth.se

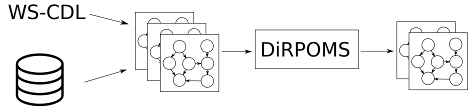
Abstract. DiRPOMS permits to verify if the specification of a distributed system can be faithfully realised via distributed agents that communicate using asynchronous message passing. A distinguishing feature of DiRPOMS is the usage of set of pomsets to specify the distributed system. This provides two benefits: syntax obliviousness and efficiency. By defining the semantics of a coordination language in term of pomsets, it is possible to use DiRPOMS for several coordination models. Also, DiRPOMS can analyze pomsets extracted by system logs, when the coordination model is unknown, and therefore can support coordination mining activities. Finally, by using sets of pomsets in place of flat languages, DiRPOMS can reduce exponential blows of analysis that is typical in case of multiple threads due to interleaving. [1] [2]

Keywords: Pomsets · choreography · realisability · CFSMs.

1 Introduction

Choreographic approaches advocate two views of the same distributed system: a *global view* that describes ordering conditions and constraints under which messages are exchanged, and *local views* that are used by each party to build their components. Here, the global view is a specification that is realised by combination of the local systems. As observed in [1], a source of problems is that there are some global specifications that are impossible to implement using distributed agents in a given communication model.

DiRPOMS is a tool designed to analyze realisability of choreographies. A choreography is formalized as a set of pomsets, were each pomset represents the causalities of events in one single branch of execution. Local views are modeled via finite state machines that communicate via asynchronous message passing. DiRPOMS checks realizability by verifying two closure conditions of the input pomsets and outputs the corresponding counterexamples:



[1] Demo video available at https://youtu.be/ISYdBNMxEDY

[2] Tool available at https://bitbucket.org/guancio/chosem-tools/

The first use case of our tool is design time analysis, where an architect checks if a choreography is realizable. In this case, violations of the closure conditions (i.e. the counterexamples) enable to identify behaviors that are not included in the choreography but are necessary in any distributed system that implements it (using finite state machines and asynchronous message passing). The usage of set of pomsets allows this analysis to be syntax oblivious, since the semantics of several existing choreographic models (i.e. [11], [6], [8]) can be expressed using set of pomsets.

The second use case is choreography mining. In this case an analyst extracts a hypotheses choreography from (partial) execution logs of a distributed system. Here, violations of the closure conditions enable to identify behaviors of the distributed system that are not included in the logs, so supplementing partial information regarding the system under test and reducing the number of executions needed to extract a model of the system.

The paper is organized as follows. In Section 2 we present the models for local and global views and in Section 3 we briefly recall the theory supporting our tool. Section 4 presents some examples of faulty choreographies, which cannot be implemented using communicating finite state machines. Section 5 shows an example of choreography mining, where the tool is used to identify missing traces from a partial execution log. Usage, implementation, and evaluation of the tool are presented in Sections 6, 7, and 8.

2 Local and global views of choreographies

We assume a set $\mathcal{P}$ of distributed *participants* (ranged over by A, B, etc.) and a set $\mathcal{M}$ of *messages* (ranged over by m, x, etc.). Participants communicate by exchanging messages over *channels*, that are elements of the set $\mathcal{C} = (\mathcal{P} \times \mathcal{P})$. The set of *(communication) labels* $\mathcal{L}$, ranged over by l and l', is defined by

$$\mathcal{L} = \mathcal{L}^! \cup \mathcal{L}^? \qquad \text{where (outputs) } \mathcal{L}^! = \mathcal{C} \times \{!\} \times \mathcal{M} \qquad \text{and (inputs) } \mathcal{L}^? = \mathcal{C} \times \{?\} \times \mathcal{M}$$

we shorten $(\mathsf{A}, \mathsf{B}, !, \mathsf{m})$ as $\mathsf{A}\,\mathsf{B}!\mathsf{m}$ and $(\mathsf{A}, \mathsf{B}, ?, \mathsf{m})$ as $\mathsf{A}\,\mathsf{B}?\mathsf{m}$. The *subject* of output and input are the sender ($\mathsf{sbj}(\mathsf{A}\,\mathsf{B}!\mathsf{m}) = \mathsf{A}$) and receiver ($\mathsf{sbj}(\mathsf{A}\,\mathsf{B}?\mathsf{m}) = \mathsf{B}$) respectively.

Local systems are modeled in terms of *communicating fine state machines* [1].

Definition 1. *An* A-communicating finite state machine (A-CFSM) $M = (Q, q_0, F, \rightarrow)$ *is a finite-state automaton on the alphabet* $\{l \in \mathcal{L} \mid \mathsf{sbj}(l) = \mathsf{A}\}$ *such that,* $q_0 \in Q$ *is the initial state, and* $F \subseteq Q$ *are the accepting states. A* (communicating) system *is a map* S *assigning an* A*-CFSM to each participant* $\mathsf{A} \in \mathcal{P}$.

Figure 1 presents a system with three participants: A, B, and C. Participant C always sends message x to B. Participant A sends two messages to B: the first message is x or y; the second message is always z. Participant B receives the first message from A and C in any order, then it receives the second message of A.

A *configuration* of a communicating system consists of a state-map $\vec{q}$, which maps each participant to its local state, and buffer-map $\vec{b}$, which maps each channel and message to the number of outputs that have been consumed. A configuration is *accepting* if all buffers are empty and the local state of each participant is accepting while it is a

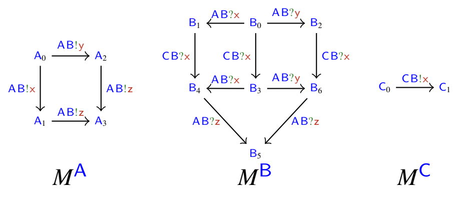


Fig. 1: A system consisting of CMFSs. Initial states are A_0, B_0, and C_0. Accepting states are A_3, B_5, and C_1.

deadlock if no accepting configuration is reachable from it. The *initial* configuration is the one where, for all $\mathsf{A} \in \mathcal{P}$, $\vec{q}(\mathsf{A})$ is the initial state of the corresponding CFSM and all buffers are empty.

The semantics of communicating systems is defined in terms of a labeled transition relation between configurations. Each transition models one action performed by one machine: an output, which adds a message to a channel, or an input, which consumed a pending message from a channel. Formally $\langle \vec{q}\ ;\ \vec{b} \rangle \overset{l}{\Rightarrow} \langle \vec{q}'\ ;\ \vec{b}' \rangle$ if there is a message $\mathsf{m} \in \mathcal{M}$ such that either (1) or (2) below holds:

1. $l = \mathsf{AB!m}$, $q(\mathsf{A}) \xrightarrow{l} q'(\mathsf{A})$, $q'(\mathsf{C}) = q(\mathsf{C})$ for all $\mathsf{C} \neq \mathsf{A} \in \mathcal{P}$, and $\vec{b}'(\mathsf{AB}) = \vec{b}(\mathsf{AB})[\mathsf{m} \mapsto \vec{b}(\mathsf{AB})(\mathsf{m})\ +\ 1]$
2. $l = \mathsf{AB?m}$, $q(\mathsf{B}) \xrightarrow{l} q'(\mathsf{B})$, $q'(\mathsf{C}) = q(\mathsf{C})$ for all $\mathsf{C} \neq \mathsf{B} \in \mathcal{P}$, $\vec{b}(\mathsf{AB})(\mathsf{m}) > 0$ and $b'(\mathsf{AB}) = b(\mathsf{AB})[\mathsf{m} \mapsto b(\mathsf{AB})(\mathsf{m}) - 1]$

where, $f[x \mapsto y]$ represents updating of a function f in x with a value y.

Definition 2. *The* language of *a communicating system S is the set $\mathbb{L}(S) \in \mathcal{L}^{\star}$ of sequences $l_0 \ldots l_{n-1}$ such that exist a trace labeled with $l_0 \ldots l_{n-1}$ that start in the initial configuration and ends in an accepting configuration.*

The notion of *realisability* is given in terms of the relation between the *language* of the global view and the one of a system of local views "implementing" it [1].

Definition 3 (Realisability). *A language $L \subseteq \mathcal{L}^{\star}$ is* weakly realisable *if there is a communicating system S such that $L = \mathbb{L}(S)$; when S is deadlock-free we say that L is* safely realisable.

We model the global views in terms of sets of pomsets, where each pomset models one branch of execution.

Definition 4 (Pomsets [4]). *A labelled partially-ordered set (lposet) is a triple $(\mathcal{E}, \leq, \lambda)$, with $\mathcal{E}$ a set of events, $\leq \subseteq \mathcal{E} \times \mathcal{E}$ a reflexive, anti-symmetric, and transitive relation on $\mathcal{E}$, and $\lambda : \mathcal{E} \to \mathcal{L}$ a labelling function mapping events in $\mathcal{E}$ to labels in $\mathcal{L}$.*

A partially-ordered multi-set (of actions), pomset for short, is an isomorphism class of lposets, where $(\mathcal{E}, \leq, \lambda)$ and $(\mathcal{E}', \leq', \lambda')$ are isomorphic *if there is a bijection $\phi : \mathcal{E} \to \mathcal{E}'$ such that $e \leq e' \iff \phi(e) \leq' \phi(e')$ and $\lambda = \lambda' \circ \phi$.*

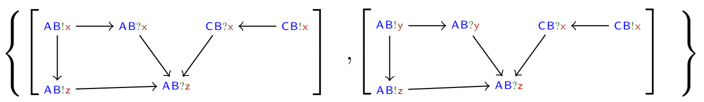


Fig. 2: A set of two pomsets that represents the global view of the system of Figure 1

Pomsets allow to represent scenarios where the same communication occurs multiple times. Intuitively, $\leq$ represents causality; if $e < e'$ then e' is caused by e. Note that λ is not required to be injective: $\lambda(e) = \lambda(e')$ means that e and e' model different occurrences of the same action. In the following, $[\mathcal{E}, \leq, \lambda]$ denotes the isomorphism class of $(\mathcal{E}, \leq, \lambda)$, symbols $r, r', \ldots$ (resp. $R, R', \ldots$) range over (resp. sets of) pomsets, and we assume that pomsets r contain at least one lposet which will possibly be referred to as $(\mathcal{E}_r, \leq_r, \lambda_r)$. The *projection* $r\lfloor_{\mathsf{A}}$ *of a pomset r on a participant* $\mathsf{A} \in \mathcal{P}$ is obtained by restricting r to the events having subject A. We will represent pomsets as (a variant of) Hasse diagrams of the immediate predecessor relation.

A pomset is *well-formed* if (1) for every output $\mathsf{A\,B!m}$ there is at most one immediate successor input $\mathsf{A\,B?m}$, (2) for every input $\mathsf{A\,B?m}$ there exists exactly one immediate predecessor output $\mathsf{A\,B!m}$, (3) if an event immediately precedes an event having different subjects then these events are matching output and input respectively, (4) ordered output events with the same label cannot be matched by inputs that have opposite order. A pomset is *complete* if there is no output event in without a matching input event.

Definition 5. *Given a pomset* $r = [\mathcal{E}, \leq, \lambda]$*, a* linearization *of r is a string in* $\mathcal{L}^\star$ *obtained by considering a total ordering of the events* $\mathcal{E}$ *that is consistent with the partial order* $\leq$ *, and then replacing each event by its label. The language of a pomset* ($\mathbb{L}(r)$) *the set of all linearizations of r. The* language *of a set of pomsets R is simply defined as* $\mathbb{L}(R) = \bigcup_{r \in R} \mathbb{L}(r)$.

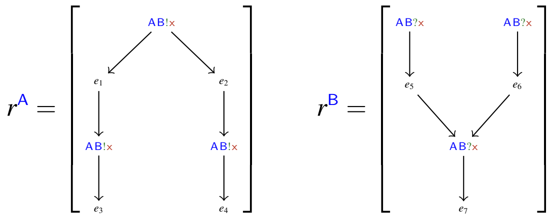
The set of pomsets of Figure 2 represents the global view of the system of Figure 1, i.e. the two views have the same language. The two pomsets represents two different scenarios (i.e. branches): in the left scenario A sends x, in the right scenario A sends y.

3 Realisability conditions

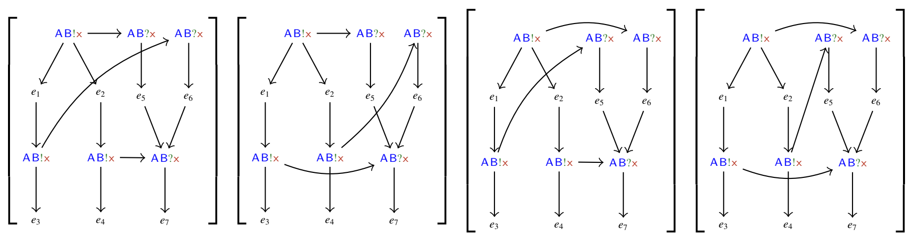
Our tool uses the verification conditions for realisability identified in [5]. These conditions requires to introduce the following definitions.

Definition 6 (Inter-Participant Closure). *Let* $(r^{\mathsf{A}})_{\mathsf{A} \in \mathcal{P}}$ *be the tuple where* $r^{\mathsf{A}} = r^{\mathsf{A}}\lfloor_{\mathsf{A}}$ *for all* $\mathsf{A} \in \mathcal{P}$*. The inter-participant closure* $\square((r^{\mathsf{A}})_{\mathsf{A} \in \mathcal{P}})$ *is the set of all well-formed pomsets* $[\bigcup_{\mathsf{A} \in \mathcal{P}} \mathcal{E}_{r\mathsf{A}}, \quad \leq_I \cup \bigcup_{\mathsf{A} \in \mathcal{P}} \leq_{r\mathsf{A}}, \quad \bigcup_{\mathsf{A} \in \mathcal{P}} \lambda_{r\mathsf{A}}]$ *where* $\leq_I \subseteq \{(e^{\mathsf{A}}, e^{\mathsf{B}}) \in \mathcal{E}_{r\mathsf{A}} \times \mathcal{E}_{r\mathsf{B}}, \mathsf{A}, \mathsf{B} \in \mathcal{P} \mid \lambda_{r\mathsf{A}}(e^{\mathsf{A}}) = \mathsf{A\,B!m}, \lambda_{r\mathsf{B}}(e^{\mathsf{B}}) = \mathsf{A\,B?m}\}$.

The inter-participant closure takes one pomset for every participant and generates all "acceptable" matches between output and input events. We use the following tuple of pomsets $(r^{\mathsf{A}}, r^{\mathsf{B}})$ to illustrate the inter-participant closure.



Pomset r^{A} represents a fork while pomset r^{B} represents a join. The inter-participant closure of $(r^{\mathsf{A}}, r^{\mathsf{B}})$ consists of four well-formed pomsets:



Definition 7 (More permissive relation). *A* pomset r' is *more permissive* than pomset *r, written* $r \sqsubseteq r'$*, when* $\mathcal{E}_r = \mathcal{E}_{r'}$*,* $\lambda_r = \lambda_{r'}$*, and* $\leq_r \supseteq \leq_{r'}$*.*

The more permissive relation guarantees language inclusion, i.e. if $r \sqsubseteq r'$ then $\mathbb{L}(r) \subseteq \mathbb{L}(r')$.

Definition 8 (Prefix pomsets). *A pomset* $r' = [\mathcal{E}', \leq', \lambda']$ *is a* prefix *of pomset* $r = [\mathcal{E}, \leq, \lambda]$ *if there exists a label preserving injection* $\phi : \mathcal{E}' \to \mathcal{E}$ *such that* $\phi(\leq') = \leq \cap (\mathcal{E} \times \phi(\mathcal{E}'))$*.*

A *prefix* of a pomset r is a pomset on a subset of the events of r that preserves the order and labelling of r.

The realisability conditions presented in [5] are two closure conditions, which are formalized by the following theorem

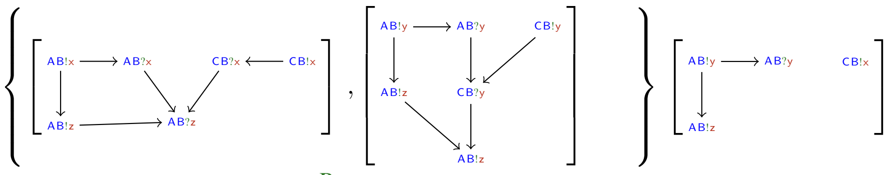
Theorem 1. *If* R *satisfies* ***CC2-POM*** *then* $\mathbb{L}(R)$ *is weak realisable, if* R *also satisfies* ***CC3-POM*** *then its language is safe realisable, where*

– ***CC2-POM***$(R) \triangleq$ *for all tuples* $(r^{\mathsf{A}})_{\mathsf{A} \in \mathcal{P}}$ *of pomsets of* R*, for every pomset* $r \in \square((r^{\mathsf{A}}\downharpoonright_{\mathsf{A}})_{\mathsf{A} \in \mathcal{P}})$*, there exists* $r' \in R$ *such that* $r \sqsubseteq r'$*.*
– ***CC3-POM***$(R) \triangleq$ *for all tuples of pomsets* $(\bar{r}^{\mathsf{A}})_{\mathsf{A} \in \mathcal{P}}$ *such that* $\bar{r}^{\mathsf{A}}$ *is a prefix of a pomset* $r^{\mathsf{A}} \in R$ *for every* A*, and for every pomset* $\bar{r} \in \square((\bar{r}^{\mathsf{A}}\downharpoonright_{\mathsf{A}})_{\mathsf{A} \in \mathcal{P}})$ *there is a pomset* $r' \in R$ *and a prefix* $\bar{r}'$ *of* r' *such that* $\bar{r} \sqsubseteq \bar{r}'$*.*

Intuitively **CC2-POM** requires that if all the possible executions of a pomset cannot be distinguished by any of the participants of R, then those executions must be part of the language of R. Similarly, **CC3-POM** requires that if all partial executions cannot be distinguished by any of the participants of R, then those executions must be a prefix of the language of R.

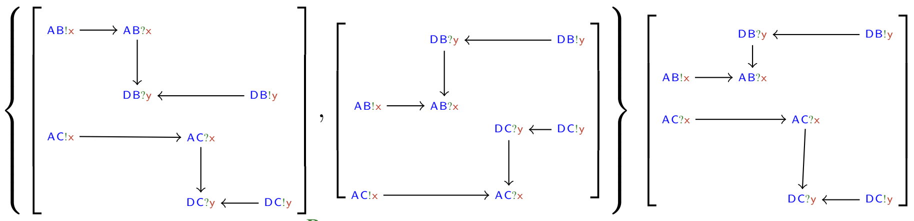
4 Realisability by examples

In this section we give some examples of the problems related to implementing pomset-based choreographers using CFSMs. Distributed choices can prevent faithful implementations in case of lack of coordination. For example, the set R_1 models two branches. Participants A and C should both send the message x or both send the message y. However, A and C do not coordinate to achieve this behaviour; this makes it impossible for them to distributively commit to a common choice. R_1 satisfies **CC2-POM**. However, pomset r_1, which represents the case A and C do not agree on the message to deliver, is in the inter-participant closure of prefixes and violates **CC3-POM**.



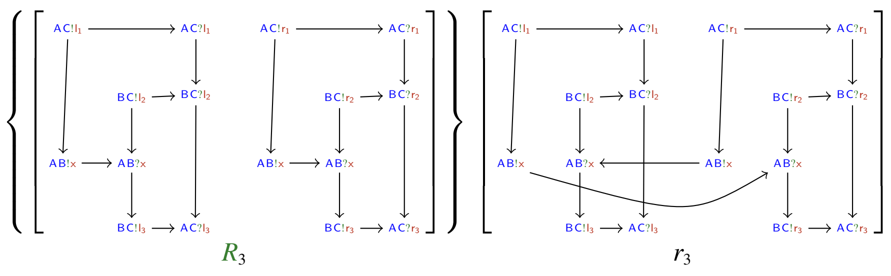
R_1 r_1

A different problem affects R_2. Here the two branches describe different orders of the same set of events. The behaviour of A (and D) is the same in both branches: A (resp. D) concurrently sends message x (resp. y) to B and C. The behaviours of B and C differ: in the left branch they first receive the message from A then the one from D, in the right branch, they have the same interactions but in opposite order. This choreography cannot be realised since, intuitively, it requires B and C to commit on the same order of reception without communicating with each other. Pomset r_2, which captures the case when B and C do not agree on the order of message reception, is in the inter-participant closure and violates **CC2-POM**.



R_2 r_2

The last example demonstrates problems led by the usage of the same message in the concurrent threads. The set R_3: consists of a single pomset, which represents two concurrent sub-choreographies. The usage of message x in both threads can cause the following problem: (1) the left thread of A executes AC!l_1 and AB!x; (2) after the output BC!r_2, the right thread of B executes the input AB?x, so "stealing" the message x generated by the left thread of A and meant for the left thread of B; (3) the right thread of B executes BC!r_3. Pomset r_3, which represents this case, is in the inter-participant closure and violates **CC2-POM**.



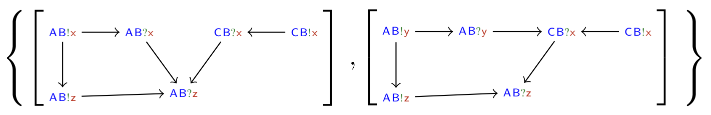
5 Identifying missing execution logs for choreography mining

Choreography (and process) mining [10] consists of extracting a hypothesis choreography from a partial execution log of a distributed system. In this section we show that violations of the closure conditions can be used to identify behaviors of the distributed system that are not included in the log. Therefore the closure conditions can support the mining and testing activities.

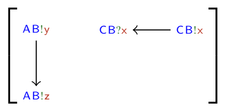
Let the partial execution log of the system of Figure 1 contains the following traces

```
A log        | B log              | C log
-------------+--------------------+-------
AB!x; AB!z   | AB?x;CB?x;AB?z     | CB!x
AB!x; AB!z   | CB?x;AB?x;AB?z     | CB!x
AB!y; AB!z   | AB?y;CB?x;AB?z     | CB!x
```

A choreography that precisely represents these traces is the following set of pomsets:



This set of pomsets satisfies **CC2-POM**, but it does not satisfy **CC3-POM**. The following pomset is in the inter-participant closure of prefixes and violates **CC3-POM**:



This pomset represents the fact that there must be an execution of the system where A sends y and B receives the first message from C, i.e.:

```
A log        | B log              | C log
-------------+--------------------+-------
AB!y; AB!z   | CB?x;...           | CB!x
```

This information can be used to fix the hypothesis choreography, by enabling the traces that are necessarily part of the behaviors of the distributed system. The set of pomsets of Figure 2 satisfies both closure conditions and its language includes the initial partial execution log.

6 Tool usage

DiRPROM is written in Python and provides a set of API to build and manipulate pomsets and to check the closure conditions. The API can be invoked by any Python development environment (in the demo video we use org-mode [9] for analyzing the examples using literate programming).

A typical DiRPOM session starts by defining the set of pomsets modeling the choreography. Pomsets can be loaded using the existing formats (including GEXF, GraphML, and JSON), be generated by translating other choreography models, or be dynamically generated. For example, the following snippet creates R_1 as input choreography:

```
# a choreography is a list of pomsets
global_view = []

# a pomset is a defined using a directed graph
# left pomset of R1
gr1 = nx.DiGraph()
# add_pair(gr1, A, B, n, m) creates two node "out-n" and "in-n"
# labeled with AB!m and AB?m, connects the two events and returns
# the pair (out-n, in-n)
abx = add_pair(gr1, "a", "b", 1, "x")
cby = add_pair(gr1, "c", "b", 2, "x")
abz = add_pair(gr1, "a", "b", 3, "z")
# Input pomsets do not need to be transitive (transitive closure
# is done internally)
gr1.add_edge(abx[1], abz[1])
gr1.add_edge(cby[1], abz[1])
gr1.add_edge(abx[0], abz[0])
global_view.append(gr1)

# right pomset of R2
gr2 = nx.DiGraph()
abx = add_pair(gr2, "a", "b", 1, "y")
cby = add_pair(gr2, "c", "b", 2, "y")
abz = add_pair(gr2, "a", "b", 3, "z")
gr2.add_edge(abx[1], cby[1])
gr2.add_edge(cby[1], abz[1])
gr2.add_edge(abx[0], abz[0])
global_view.append(gr2)
```

The closure condition **CC2-POM** can be checked using

```
cc2c = cc2closure(global_view)        # cc2c is the list of pomsets
cc2res = cc2pom(cc2c, global_view)
```

The result `cc2res` is a map that yields for each index `i` of `cc2c` the index of `global_view` matching it or `None` if `cc2c[i]` is a counterexample. Similarly closure condition **CC3-POM** can be checked using

```
(cc3c, pref) = cc3closure(global_view) # cc3c and prefix are lists
cc3res = cc3pom(cc3c, pref)
```

The list `pref` contains the list of prefixes of the input choreography, and the result `cc3res` maps each index of `cc3c` to an index of `pref` or `None`. The counter examples can be rendered using:

```
errors = counterexamples(cc3c, cc3res)
debug_graphs(errors, "output-folder")  # generates pictures of errors
```

DiRPOM also provides a command line utility, which uses GraphML format for input and output of pomsets. The left pomset of R_1 can be defined by the following GraphML file:

```
<?xml version='1.0' encoding='utf-8'?>
<graphml>
  <key attr.name="label" attr.type="string" for="node" id="d0" />
  <graph edgedefault="directed">
    <node id="b-2"><data key="d0">CB?x</data></node>
    <node id="b-3"><data key="d0">AB?z</data></node>
    <node id="b-1"><data key="d0">AB?x</data></node>
    <node id="a-1"><data key="d0">AB!x</data></node>
    <node id="a-3"><data key="d0">AB!z</data></node>
    <node id="c-2"><data key="d0">CB!x</data></node>
    <edge source="b-2" target="b-3" />
    <edge source="b-1" target="b-3" />
    <edge source="a-1" target="a-3" />
    <edge source="a-1" target="b-1" />
    <edge source="a-3" target="b-3" />
    <edge source="c-2" target="b-2" />
  </graph>
</graphml>
```

Each GraphML must contain a `key` element, specifying the existence of the node attribute `label` of type `string`. Each node has a unique identifier and a data sub-element, which defines the node label. The following command executes the analysis of a choreography:

```
dirpom [input] [output1] [output2] --draw --graphml
```

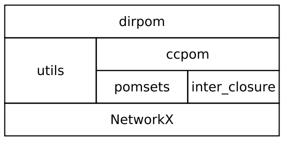
The parameter `input` specifies the path of a directory that contains one GraphML file for each pomset of the choreography. The tool produces one GraphML file in the `output1` and `output2` for each violation of **CC2-POM** and **CC3-POM** respectively. Additionally, if the `--draw` option is specified, the tool renders the counterexamples as `.png` in the same directories.

7 Tool implementation

DiRPROM relies on the NetworkX package for graph operations. In fact, pomsets are represented as direct labelled acyclic graphs. The tool consists of five modules:

- `utils`: provides export of pomsets to `png` and utilities to build pomsets
- `pomset`: provides functions to process pomsets, e.g. query lists of participants and messages, projections per participant or message, transitive closure and reduction, enumeration of prefixes, enumeration of linearizations

– `inter_closure`: implements inter-participant closure
– `ccpom`: generates the two closure sets and verifies the closure conditions
– `dirpom`: provides the command line utility



In order to demonstrate the implementation of the analyses and the internal API, we report the implementation of **CC3-POM**:

```
def cc3closure(graphs):
  # retrieves the list of principals in graphs
  principals = pomset.get_all_principals(graphs)
  # projects the input graphs on principals and yields a map mapping
  # principals to list of "local" pomsets (avoids duplicates)
  local_threads = pomset.get_principal_threads(graphs, principals)
  local_prefixes = {}
  for p in principals:
    # computes all prefixes of all graphs in local_threads[p]
    # (avoids duplicates)
    local_prefixes[p] = pomset.get_prefixes(local_threads[p])
  # generates all tuples in the product of local_prefixes
  tuples = inter_closure.make_tuples(local_prefixes)
  # computes the inter-participant closure of all the tuples
  # (avoids duplicates)
  ipc = inter_closure.inter_process_closure(tuples)
  # computes all prefixes of the input graphs (avoids duplicates)
  prefixes = pomset.get_prefixes(graphs)
  return (ipc, prefixes)

def cc3pom(ipc, prefixes):
  matches = {}
  for i in range(len(ipc)):
    matches[i] = None
    for j in range(len(graphs)):
      # checks if graph[j] is more permissive than ipc[i]
      if (pomset.is_more_permissive(graph[j], ipc[i])):
        matches[i] = j
        break
  return matches
```

8 Tool evaluation

The main primitive of NetworkX used by the tool is `subgraph_is_ismorphic`, which returns true iff r_1 is (label-preserving) isomorphic to a subgraph of r_2. If r_1 and r_2 have the same number of nodes and the predicates holds then $r_2 \sqsubseteq r_1$.

```
import networkx.algorithms.isomorphism as iso
nm = iso.categorical_node_match('label', '')

def is_more_permissive(g1, g2):
  if len(g1.nodes()) != len(g2.nodes()):
    return False
  m = iso.GraphMatcher(g1, g2, nm)
  return m.subgraph_is_isomorphic()
```

The complexity of finding a label-preserving graph isomorphism is in general exponential in the number of events. However, since the graphs are acyclic, the complexity can be bound to the number of *concurrently-repeated actions*: i.e. events that have the same label, are unordered, and have the same number of predecessor with the same label (e.g. AB!x in R_3). If there are no concurrently repeated actions then isomorphism of pomsetes can be checked in polynomial time with respect to the number of events.

We report the performance of our tool for the examples. The experiments have been executed on a Intel 2.2 Ghz i7 with 16 GB of RAM. The table reports the size of the closures, the number of counterexamples, and the processing time in milliseconds.

	CC2-POM	errors	ms	**CC3-POM**	errors	ms
R_1	2	0	3	38	10	64
R_2	2	1	9	100	18	340
R_3	2	1	16	668	258	9297

In general the evaluation of closure conditions is fast for simple examples. However, the number of prefixes to check in **CC3-POM** can be large when participant have several concurrent threads.

One of the advantages of checking **CC$\star$-POM** with respect to previous work [1] is that the former does not require the explicit computation of the language of the family of pomsets, which can lead to combinatorial explosion due to interleavings. In fact, in case of concurrency, the number of prefixes is usually smaller than the number of possible linearizations of a pomset. For example, the following pomset consists of two independent threads, each one consisting of n sequential and distinguished events

$$\left[\begin{tikzcd} e_1 \arrow[r] & e_2 \arrow[r] & \cdots \arrow[r] & e_n \\ e'_1 \arrow[r] & e'_2 \arrow[r] & \cdots \arrow[r] & e'_n \end{tikzcd}\right]$$

The closure condition in [1] requires to directly compute the language of the pomset, which has 2^n words. Instead, the prefix of the pomset are $(n+1)^2$.

As a further example, the set of pomsets R_3 contains one pomset and has two actions that occur in both threads: AB!x and AB?x. The inter-participant closure has exactly two pomsets: the element of R_3 itself and r_3. The left and right subpomsets of R_3, which represent the two threads, have 32 different linearizations, each one consisting of 8 events. Therefore the language of R_3 consists of $32*32*2^8 = 2^{18}$ words. On the other hand, analyzing **CC3-POM** for R_3 requires to check 668 prefixes.

9 Concluding remarks

Realisability of specifications is of concern for both practical and theoretical reasons. Several works (e.g., [2,3,7]) defined constraints to guarantee soundness of the implementation of choreographies. These approaches address the problem for specific languages and use conditions that rely on the syntactical structure of the specification. DiRPOMS provides a language independent tool to check realisability of choreographies. Therefore, it can be used for several choreographic models, as long as their semantics can be expressed via set of partial orders.

There two main limitations of DiRPOMS that we plan to address. First, our tool cannot analyze recursive choreographies, since their pomset based semantics is infinite. Even if loops are bounded, naive loop unrolling can easily generate large sets of pomsets which are intractable. Secondly, **CC$\star$-POM** conditions are sufficient but not necessary conditions for realisability. In fact, the same set of traces can be expressed using different sets of pomsets by exploring different interleavings. We are currently investigating a notion of normal forms for families of pomsets that can be used to guarantee that our conditions are necessary.

We are also working on optimizing our tool. In particular we think that it is possible to demonstrate equivalence between **CC3-POM** and a different formulation, which requires to check only a subset of prefixes. For instance, in verifying **CC3-POM** for R_3, the analysis of the prefix $\left[\ \mathsf{AC!l_1} \quad \mathsf{AC!r_1}\ \right]$ covers also the cases of the prefixes $\left[\ \mathsf{AC!l_1}\ \right]$ and $\left[\ \mathsf{AC!r_1}\ \right]$.

References

1. Alur, R., Etessami, K., Yannakakis, M.: Inference of Message Sequence Charts. IEEE Trans. Software Eng. **29**(7), 623–633 (2003). https://doi.org/10.1109/TSE.2003.1214326
2. Bocchi, L., Melgratti, H.C., Tuosto, E.: Resolving non-determinism in choreographies. In: ESOP. pp. 493–512 (2014). https://doi.org/10.1007/978-3-642-54833-8_26
3. Carbone, M., Honda, K., Yoshida, N.: A Calculus of Global Interaction based on Session Types. Electronic Notes in Theoretical Computer Science **171**(3), 127 – 151 (2007). https://doi.org/10.1016/j.entcs.2006.12.041
4. Gaifman, H., Pratt, V.R.: Partial order models of concurrency and the computation of functions. In: LICS. pp. 72–85 (1987)
5. Guanciale, R., Tuosto, E.: Realisability of pomsets via communicating automata. CoRR **abs/1810.02469** (2018), http://arxiv.org/abs/1810.02469
6. Gunter, E.L., Muscholl, A., Peled, D.A.: Compositional Message Sequence Charts. In: TACAS. pp. 496–511. Springer (2001). https://doi.org/10.1007/3-540-45319-9_34
7. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. Journal of the ACM **63**(1), 9:1–9:67 (2016). https://doi.org/10.1145/2827695, extended version of a paper presented at POPL08
8. Lange, J., Tuosto, E., Yoshida, N.: From Communicating Machines to Graphical Choreographies. In: POPL15. pp. 221–232 (2015)
9. Schulte, E., Davison, D., Dye, T., Dominik, C., et al.: A multi-language computing environment for literate programming and reproducible research. Journal of Statistical Software **46**(3), 1–24 (2012)
10. Van Der Aalst, W.: Process mining: discovery, conformance and enhancement of business processes, vol. 2. Springer (2011)
11. WSCDL Version 1.0. https://www.w3.org/TR/ws-cdl-10/ (2005)