Object Centric DCR Graph Discovery in OCPA User Manual and Documentation

Lasse Berkensträter, Xi Wang, Alois Hannen

May 2025

Contents

1 Introduction

This library is an extension of the Python library OCPA [1] (Object-Centric Process Analysis). OCPA is enhanced with support for Object-Centric Dynamic Condition Response (OC-DCR) Graphs (Fig.1), as well as their discovery from Object-Centric Event Logs (OCEL) using an adapted version of the OCDisCoveR algorithm [3].

Dynamic Condition Response (DCR) Graphs are an innovative and increasingly prominent business process notation [4]. By extending OCPA to support this notation, we enhance the range of modeling and discovery tools available in OCPA. Offering multiple process discovery notations and algorithms allows users to choose the most appropriate one for a given context, improving OCPA's applicability across a variety of domains and use cases. The implementation includes the discovery of a declarative process model that captures object lifecycles, one-to-many and many-to-many synchronization constraints between object types.

This user manual introduces the key components and features of the implementation, emphasizing the overall structure of the code, rather than explaining each function in detail. For specific details and explanations of functions, as well as their parameters, please refer to the docstrings, which provide comprehensive documentation. We highly recommend using the Jupyter notebook found in the project's root directory alongside this manual. Please also be aware that we only included pictures and code examples whenever necessary to keep this document as slim and useful as possible and organize the hands-on examples in the Jupyter notebook.

2 Getting Started

The code was developed and tested for Python version 3.12. It is strongly recommended to use a virtual environment with this version of Python.

2.1 Installing Requirements

To get started, create a virtual environment and install the requirements using Listing 1.

```
pip install -r requirements.txt
```

Listing 1: Install dependencies using pip

3 Features and Functions

3.1 OC-DCR Graph Data Structures (ocpa.objects.oc_dcr_graph)

The core objects of DCR Graphs are implemented in the `oc_dcr_graph` module. Those objects are used to create (OC-)DCR Graph structures and also serve as the output of the OCDisCoveR algorithm.

3.1.1 (OC)DCRRelation

In DCR Graphs activities are connected using four types of constraints. The constraint type is defined using the following enum values:

- `RelationTyps.C` (A $\rightarrow\bullet$ B): Activity A must occur before activity B can occur
- `RelationTyps.R` (A $\bullet\rightarrow$ B): After activity A occurs, activity B must eventually occur
- `RelationTyps.I` (A $\rightarrow +$ B): Activity A enables activity B
- `RelationTyps.E` (A $\rightarrow \%$ B): Activity A disables activity B

For constraints in OC-DCR graphs, the `OCDCRRelation` class also stores boolean values for the quantifiers used to represent many-to-many and one-to-many constraints.

3.1.2 DCRMarking

The `DCRMarking` class represents the runtime state of a DCR Graph. The different states of activities are described using the following enum values:

- `MarkingTyps.E`: The activity has been executed
- `MarkingTyps.I`: The activity is currently included
- `MarkingTyps.P`: The activity is pending

3.1.3 DCRGraph

Basic DCR Graphs, including support for nested groups, are implemented in the `DCRGraph` class. The functions used to add activities, relations, nested groups and so on are demonstrated in the Jupyter Notebook, as well as Listing 3

A graph object can be transferred into a string structured like a Python dictionary, containing all of the graph's data. In the following, this dictionary representation will be called template. In addition to XML export (see Section 3.3.2), this template format makes it easier to inspect, share and reconstruct DCR Graphs, as templates can be directly passed to the constructor to (re)create a graph.

```
dcr_template = {
    "events": {"Event3", "Event4", "Event5", "Event1", "Event2"},
    "marking": {
        "executed": set(),
        "pending": {"Event1"},
        "included": {"Event3", "Event4", "Event5", "Event1", "Event2"},
    },
    "includesTo": {"Event3": {"Event5"}},
    "excludesTo": {"Event1": {"Event4"}},
    "responseTo": {"Event1": {"Event3"}},
    "conditionsFor": {"Event2": {"Event1"}},
    "nestedgroups": {"Event2": {"Event5", "Event3"}},
}
```

Listing 2: Example DCR Template of the first DCR Graph in the Jupyter Notebook

3.1.4 OCDCRObject

The `OCDCRObject` class represents an object subgraph in an OCDCR Graph. It inherits from `DCRGraph` adding a spawn activity and the object type. Like the DCR Graph, it can be constructed manually using provided methods as in Listing 3 or a template like the one in Listing 2.

3.1.5 OCDCRGraph

The `OCDCRGraph` class inherits from `DCRGraph` adding object-centric capabilities:

- The subclass `OCDCRObject` of `DCRGraph` representing an object type
- Spawn Relations (A $\rightarrow *$ B) indicating that activity A spawns object type B and its corresponding activities
- One-to-Many and Many-to-Many synchronization constraints between object types and top level activities

The export of templates work the same as for the basic DCR Graphs, but are extended with keys to describe the object-centric features. Caution is advised since for creating object-centric graphs from templates, only templates of basic DCR Graphs can be used, due to the main focus of this project, which is the discovery of OC-DCR Graphs from logs and not primarily a visualization tool for already created graphs. More details on exporting to a standard format can be found in Section 3.3.2.

```
from ocpa.objects.oc_dcr_graph import DCRGraph, OCDCRGraph, RelationTyps, MarkingTyps,
                                        OCDCRObject, Event
# Create basic DCR Graph
dcr = DCRGraph()

# Add Activities
dcr.add_event('Event1', marking={MarkingTyps.P, MarkingTyps.I})
dcr.add_event('Event2', isGroup=True)
dcr.add_event('Event3', parent='Event2')

# Add relations
dcr.add_relation('Event1', 'Event2', RelationTyps.C)
dcr.add_relation('Event1', 'Event3', RelationTyps.R)

# Create an OCDCRGraph with the specifications of the graph created before
ocdcr = OCDCRGraph(dcr=dcr)

# Create an object type
spawn_event = Event('Create_Obj1')
object = OCDCRObject(spawn=spawn_event, type='Object1')

object.add_event('OBJ_Event1')
object.add_event('OBJ_Event2')
object.add_relation('OBJ_Event1', 'OBJ_Event2', RelationTyps.C)

# Add object type to the oc graph
ocdcr.add_object(object)

# Add one-to-many and many-to-many constraints
ocdcr.add_relation('Event1', 'OBJ_Event1', RelationTyps.C, False, True)
ocdcr.add_relation('OBJ_Event2', 'OBJ_Event2', RelationTyps.E, True, True)
ocdcr.add_relation('OBJ_Event1', 'OBJ_Event1', RelationTyps.C, True, True)
ocdcr.add_relation('OBJ_Event2', 'Event4', RelationTyps.E, True, False)
```

Listing 3: Example DCR Template of the first DCR Graph in the Jupyter Notebook

3.2 Visualization (ocpa.visualization.oc_dcr_vis)

Both DCR and OC-DCR graphs can be visualized using the `oc_dcr_vis` module, which implements three useful functions to handle the visual representation.
The data structures of graphs are converted into a graphviz Digraph using the `apply` function. Those Digraphs can then either be exported as PNG using the `save` function or viewed in a local image viewer using `view`.
In the visualization, spawned object types can be differentiated from static ones by the light blue background color instead of a light green one and the marking can be taken from the [I], [E] and [P] behind the activities for the corresponding marking type.

```
from ocpa.visualization.oc_dcr_vis import visualizer as dcr_viz
from ocpa.objects.oc_dcr_graph import DCRGraph

example_graph = DCRGraph(template=template)

# Convert to Digraph
graph = dcr_vis.apply(example_graph)

# View in local image viewer
dcr_vis.view(dcr_vis.apply(example_graph))

# Export as PNG
dcr_vis.save(dcr_vis.apply(example_graph), 'graph.png')
```

Listing 4: Applying a visualization to a DCR Graph

3.3 Utility Functions

To extract insights from the logs, the discovered graphs have to be analyzed. However, even small logs can lead to a lot of constraints. To manage this complexity, the implementation includes different filters and supports exporting graphs as XML, enabling import in other more interactive tools.

3.3.1 Constraint Filtering

There are three filter options in `ocpa.objects.oc_dcr_graph.filtering`, returning a new graph and not manipulating the original graph:

- `filter_by_relation_type`: Filters all constraints based on their type
- `filter_many_to_many`: Filters out all many-to-many constraints, if their type is not in the set of relation types to retain
- `filter_one_to_many`: Filters out all one-to-many constraints, if their type is not in the set of relation types to retain

This approach might seem drastic, but makes it possible to focus on the relevant constraints for a certain context, especially useful for visualizations.

3.3.2 Export

DCR Graphs can be exported into a standardized XML format [4], by calling its `export_to_xml` function.
This export functionality is especially useful for importing discovered graphs into other tools for further inspection or validation. However, since these tools do not share a publicly available standard XML format, XML files may require adjustments before they can be imported without any issues. For OC-DCR Graphs, we decided to expand the standard structure for normal DCR Graphs (see Listing 5) with attributes similar to the ones in the string representation as there is no official OC-DCR XML format, which could be used and tested in a publicly available tool.

```
<dcrgraph title="...">
  <specification>
    <resources>
      <events>
        <event id="...">
          <event id="..."/>
          ...
        </event>
        ...
      </events>
      <labels>
        <label id="..."/>
        ...
      </labels>
      <labelMappings>
        <labelMapping eventId="..." labelId="..."/>
        ...
      </labelMappings>
    </resources>
    <constraints>
      <conditions>
        <condition sourceId="..." targetId="..."/>
        ...
      </conditions>
      <responses/>
      <excludes/>
      <includes/>
    </constraints>
    <spawns>
      <spawn sourceId="..." targetId="..."/>
      ...
    </spawns>
    <objects>
      <object objectId="..."/>
      ...
    </objects>
    <object_mappings>
      <objectMapping eventId="..." objectId="..."/>
      ...
    </object_mappings>
  </specification>
  <runtime>
    <marking>
      <executed>
        <event id="..."/>
        ...
      </executed>
      <included/>
      <pendingResponses/>
    </marking>
  </runtime>
</dcrgraph>
```

Listing 5: XML Structure for OC-DCR Graphs

3.4 Discovery Algorithm (ocpa.algo.discovery.oc_dcr)

The discovery algorithm of OC-DCR graphs (OCDisCoveR) is based on the paper 'Discovery of Object-Centric Declarative Models' by Christfort et al.[3]. It adapts their approach to work with the Object-Centric Event Log (OCEL) format of OCPA instead of event knowledge graphs.

3.4.1 Apply

Before applying the algorithm, there are some decisions that need to be made to define how the algorithm should be applied to the log.

1. `spawn_mapping : Dict[str, str]`: Specifies which activity in the log spawns which object type
2. `activities_mapping : Dict[str, str]`: Maps activities to specific object types or to the top-level graph. This determines where each activity belongs in the graph structure.
3. `apply_nested : bool`: Indicates whether the algorithm should attempt to discover nested activities. Note that this feature depends on an external algorithm, which is why we can not fully guarantee for the results.
4. `derived_entities : List[Tuple[str,str]]`: Defines which entities are considered derived entities. The order of the object types in each tupel is not relevant.

The whole discovery pipeline is then applied to an OCEL using the algorithms `apply` function, as seen in Listing 8. For CSV and JSONOCEL files, first import to an OCEL object as shown in Listing 6 and 7.

```
from ocpa.util.dcr.import_export import csv_to_ocel

## import csv file
object_types = ["Order", "Item", "Customer"]
parameters = {
    "obj_names": object_types,
    "val_names": [],
    "act_name": "activity",
    "time_name": "timestamp",
    "sep": ","
}
ocel = csv_to_ocel(filepath="tests/logs/generated.csv", parameters=parameters)
```

Listing 6: Importing csv file to an OCEL

```
## import jsonocel file
ocel = ocpa.objects.log.importer.ocel.factory.apply("tests/logs/generated.jsonocel")
```

Listing 7: importing jsonocel file to an OCEL

```
from ocpa.algo.discovery.oc_dcr.algorithm import apply
from ocpa.visualization.oc_dcr_vis import visualizer as dcr_viz
from ocpa.objects.oc_dcr_graph import IN_TOP_GRAPH

## define spawn mapping
spawn_mapping = dict({
    ("offer", "Create␣offer"),
})

# Map activities to objects
activities_mapping = {
    'Call': 'application',   ## write IN_TOP_GRAPH instead of 'application', if in top graph
    'Deny': 'application',
    'Accept': 'application',
    'Personal␣loan␣collection': 'application',
    'Refuse␣offer': 'offer',
    'Cancel␣offer': 'offer',
    'Cancel␣application': 'application',
    'Accept␣offer': 'offer',
    'Return': 'application',
    'Send␣(mail␣and␣online)': 'application',
    'Send␣(online)': 'application',
    'Call␣incomplete␣files': 'application',
    'Handle␣leads': 'application',
    'Assess␣potential␣fraud': 'application',
    'Shorten␣completion': 'application',
    'Validate': 'application',
    'Create␣application': 'application',
    'Complete': 'application',
    'Submit': 'application',
    'Pending': 'application'
}


## apply algorithm with defined derived entities, without nesting:
graph = apply(ocel=ocel, spawn_mapping=spawn_mapping, activities_mapping=activities_mapping,
    derived_entities=[('application','offer')])

## apply algorithm with defined derived entities and nesting:
graph = apply(ocel=ocel, spawn_mapping=spawn_mapping, activities_mapping=activities_mapping,
    apply_nested = True, derived_entities=[('application','offer')])

## apply algorithm without defined derives entities, but with nesting:
graph = apply(ocel=ocel, spawn_mapping=spawn_mapping, activities_mapping=activities_mapping,
    apply_nested = True)
```

Listing 8: Applying the discovery pipeline to the bpi 2017 dataset

3.4.2 Steps

The key steps of the algorithm are:

1. Initial discovery
   (a) Extract lifecycle traces for each object type from the Object-Centric Event Log.
   (b) Apply a DisCoveR algorithm to derive a base DCR Graph.
   (c) Translate the DCR Graph into an Object-Centric DCR graph with spawns and objects as subgraphs and extract one-to-many relations between the system process and object instances.
2. Many-to-many synchronization discovery between object types
   (a) Identify transitive closures in the object graph to build merged traces across related objects.
   (b) Discover many-to-many excludes.
   (c) Discover many-to-many conditions and responses.
3. Optimizations and Filtering
   (a) Optimize relations

(b) Filter out the many-to-many constraints that are not between derived entities

While the implementation details of each step can be gathered from the code (docstrings), in the following we explain the differences in our implementation and the algorithm described in the paper.

3.4.3 Structure of the Algorithm Implementation

The main methods of the algorithm can be found in `ocpa.algo.discovery.oc_dcr.util`. The class `DiscoverData` in `discover_data` deals with all the data information about the event log and the mappings used in Object-Centric DCR discovery and the class `ErrorManager` is responsible for validating the given data inputs. The classes in `discover_logic` then implement the core logic of the algorithm, with an extra class `InitialDiscovery` for step 1 from 3.4.2 and `ManyToManyDiscovery` for step 2. For step 3, everything correlating to optimizations can be found in `graph_optimizations`, including nesting, filtering and handling derived entities. In `unspawned_obj_handler`, we define how we deal with objects that are not spawned. The whole algorithm pipeline is managed in `discover`.

3.4.4 Comparison of the Implementation with the Original Paper

- **Assumptions before OC-DisCoveR:**

  To implement the algorithm described in the paper, we assume it takes the two user-defined dictionaries as parameters: `spawn_mapping` and `activities_mapping`.

  Based on these mappings, we make the following assumptions in our implementation to derive the prerequisites required by the algorithm:

  - Activities defined as spawn activities in the `spawn_mapping` are automatically treated as top-level activities.
  - Activities not listed in the `activities_mapping` are also considered top-level by default.
  - All object types appearing in the `spawn_mapping` are treated as dynamically created; object types not included are assumed to be static.

  Additionally, we classify every activity that belongs to the system process, such as spawn activities, as system-level activities. This enables such activities to participate in both one-to-many and top-level synchronization constraints.

- **Derived Entities:**

  The paper requires derived entities to be specified prior to executing the algorithm, in order to include them in the Event Knowledge Graph (EKG). As they are not mentioned again in the algorithmic description, we treat derived entities primarily as a mechanism to allow users to define between which object types synchronization constraints should be represented. All other synchronizing constraints that do not correspond to the defined derived entities are subsequently filtered out.

- **Static Objects:**

  Since activities mapped to unspawned objects and top-level activities can be treated uniformly within the algorithm, we abstract this mapping prior to the algorithm's execution. Specifically, activities associated with unspawned objects are temporarily treated as top-level activities in the internal representation. This abstraction reduces the need for special-case logic during the algorithmic process.

  After the discovery phases, we restructure the resulting OC-DCR graph by reassigning activities that were temporarily treated as top-level—due to their association with unspawned objects—back to their corresponding unspawned object types. To improve clarity and structural coherence, we further enhance the visualization by explicitly grouping these activities under their respective unspawned objects. While this refinement is not explicitly discussed in the original OCDisCoveR approach, it facilitates a clearer representation and allows users to decide whether to interpret unspawned activities as top-level elements or as logically belonging to specific unspawned object types.

- **Log Abstraction and Integration of DisCoveR:**

  The log abstraction is implemented exactly as described in the paper, by extracting directly-follows paths per object instance for each entity type in the OCEL, resulting in one log per type, which are then merged using Polars to form a unified log abstraction.

  We then apply an implementation of DisCoveR from Back et al. [2].

  We adapted this implementation to fit our data structure, enabling us to obtain a DCR Graph that integrates seamlessly with our OCEL-based abstraction pipeline.

- **Translate to OC-DCR:**

  In line with the description in the paper (cf. Section A. *Initial Discovery*), we apply a filtering step to remove relations that violate the correlation assumptions between entities.

  Thus in the `translate_to_basic_ocgraph_structure` function, we implicitly assume that all discovered constraints in Steps 1 and 2 - when involving either top-level activities or activities mapped to unspawned objects - are to be interpreted as one-to-many constraints. This assumption, while crucial for the algorithm's behavior, is not explicitly stated in the original paper.

  To ensure semantic correctness, we apply an additional filtering step to both top-level and one-to-many constraints. In this step, we retain only those constraints where the involved activities share at least one object type in the actual OCEL, rather than relying solely on the abstracted mapping. This refinement extends the filtering logic defined in Definition 6 to include top-level constraints. Consequently, we guarantee that top-level activities are only connected by constraints when they are semantically related in the event log.

  Moreover, this filtering ensures that no constraint exists between a spawn activity and its corresponding spawned activity, even if the user has not explicitly mapped the spawned activity into the subgraph of the spawn activity. This design choice further reinforces the semantic integrity of the resulting OC-DCR graph.

- **Log from Closures:**

  The computation of the closures follows the procedure described in the paper. However, during the log abstraction phase of the resulting transitive closure, the paper states that the resulting event log should be projected onto the set of spawned activities. As this statement is somewhat ambiguous, our implementation does not perform this projection.

  Nonetheless, to preserve the intended semantics, we ensure in subsequent functions that operate on the event log and the discovered DCR graph—such as partition and the extraction of many-to-many responses and conditions—that any discovered synchronizing many-to-many constraints occur exclusively between spawned activities.

- **Many-To-Many Excludes:**

  For the many-to-many excludes, we follow the steps by the paper, and simply add the exclusions as many-to-many exclusions. Therefore, we filter all newly discovered exclusions that are not between two spawned activities and therefore cannot be Many-To-Many excludes out, even though not explicitly stated in the paper.

- **Many-To-Many Conditions and Responses:**

  As for this step, a concrete explanation in the paper is missing, we make a few assumptions about lines 7 to 22 of algorithm 1:
  In line 7 of Algorithm 1 in the paper, a mapping of all activities in the universe of activities Σ is described; however, since many-to-many conditions and responses are only possible

between two different activities that are both spawned, we initialize the mapping with only spawned activities and without self-loops. Therefore, the following in lines 11 and 12 will only be executed if the activity is spawned by an object.
For line 13, we interpret $\rightarrow *(a)$ as whether a is a spawn activity. If that is the case, we record all the activities of the object a spawns as spawned, which is how we interpret $\text{MAINTAIN}(a'_{\forall})$ in line 16. After all conditions and responses are found, we filter out all the redundant constraints due to transitivity to improve the readability of discovered models, as later we will only apply this filtering to the subgraphs themselves and not to constraints between subgraphs anymore.

- **Nesting:**

  The user can choose whether nesting should be applied to each object graph. If so, we apply the nested algorithm from the pm4py dcr extension, which we already used for DisCoveR [2]. It should be noted, that this nested algorithm is not deterministic, and therefore the resulting OCDCR graphs can differ between applications. Because that algorithm does not filter the redundant constraints after adding nested groups, we implemented a filtering method.
  Instead of having nested dcr graphs after the first discovery, we apply nesting after the main discovery process on all object graphs with only the one-to-one constraints, as there are only one-to-one constraints after the first discovery as well, and the nested algorithm does not differentiate between different synchronization types.

- **Optimizations:**

  After the discovery, we filter out different relations to improve readability. Firstly, we filter relation and condition and response constraints based on transitivity in the main graph and then in it's subgraphs. Secondly, we filter the condition and response constraints that have a parallel exclude relation.

4 Validation (Explanation of the Approach and the Results)

4.1 Validation on a Synthetic Data

ID	Activity	Order	Item	Customer	Timestamp
0	Register Customer			[C1]	2024-01-01 08:00:00
1	Verify			[C1]	2024-01-01 08:01:00
2	Create Order	[O1]		[C1]	2024-01-01 08:10:00
3	Add Item		[I1]		2024-01-01 08:15:00
4	Link Item to Order	[O1]	[I1]		2024-01-01 08:16:00
5	Ship Order	[O1]		[C1]	2024-01-01 08:30:00
6	Register Customer			[C2]	2024-01-01 08:40:00
7	Verify			[C2]	2024-01-01 08:41:00
8	Create Order	[O2]		[C2]	2024-01-01 08:50:00
9	Add Item		[I2]		2024-01-01 08:55:00
10	Link Item to Order	[O2]	[I2]		2024-01-01 08:56:00
11	Ship Order	[O2]		[C2]	2024-01-01 09:10:00
12	Accept	[O2]			2024-01-01 09:20:00

Table 1: Synthetic example log

For the very small example log 1 and the following mappings, we discover the OC-DCR Graph in Figure 1.
In our configuration, the object type `Order` is spawned by the activity `Create Order`, and `Item` is spawned by `Add Item`. Activities are mapped to object types as follows: `Ship Order` and `Accept` belong to `Order`, while `Link Item to Order` is assigned to `Item` (note that it could alternatively be

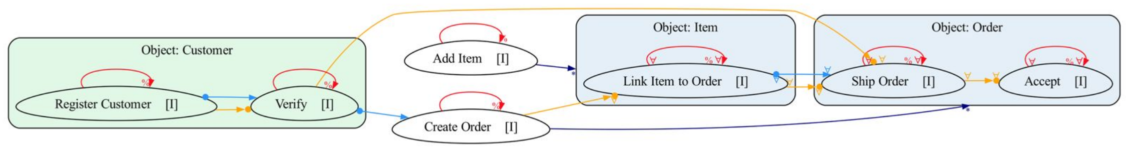

Figure 1: Discovered graph for the small example log

mapped to `Order`, but we opt for `Item` to better reflect the item lifecycle). Furthermore, `Register Customer` and `Verify` are associated with the object type `Customer`.
We observe the dynamic instantiation of the objects `Order` and `Item`, which are associated with the activities `Ship Order` and `Link Item to Order`, respectively. These objects are spawned by the activities `Create Order` and `Add Item`, as indicated by the corresponding spawn relations. The object `Customer`, in contrast, is classified as a static object, since there is no spawn activity in the log. For the purpose of analysis, we define the derived entity relationships as `(Item, Order)` and `(Customer, Order)`, as these pairs represent key interactions within the process and reveal particularly insightful interplay.
Relations:

- Excludes Relations ($\rightarrow \%$)
  - In the discovered OC-DCR Graph, all activities are associated with self-exclusion constraints. This observation is rooted in the event log, where each activity appears only once per object instance. As a result, the model enforces that once an activity has been executed for a given object, it cannot occur again within the same lifecycle.
- Response Relations ($\bullet \rightarrow$)
  - `Create Order` responds to `Verify`
    This behavior is reflected in the event log, as each `Create Order` event is consistently preceded by a `Verify` event involving the corresponding customer
- Condition Relations ($\rightarrow \bullet$)
  - `Create Order` is a condition for `Link Item to Order`
    An order must exist before items can be linked to it. This can be seen in the log, where linking always follows order creation for the same object.
  - `Verify` is a one to many condition for `Ship Order`
    In the event log, this behavior is reflected by: each `Ship Order` event is preceded by a `Verify` event involving the corresponding customer.
  - `Link Item to Order` is a condition for `Ship Order`
    The log shows that shipping never occurs before all items are linked. This implies that no shipping should occur before the completion of linking activities across all item instances.
  - `Ship Order` is a condition for `Accept`
    In the log, `Accept` always follows shipping for orders. This enforces a logical sequence: Orders must be shipped before they can be accepted, as modeled by this condition.

4.2 Validation on the BPI 2017 Dataset

Figure 2 shows the result after discovering an OC-DCR Graph on the BPI 2017 dataset with the mappings used in Listing 8. In the following, a few selected constraints will be explained:

- One-to-One Relations

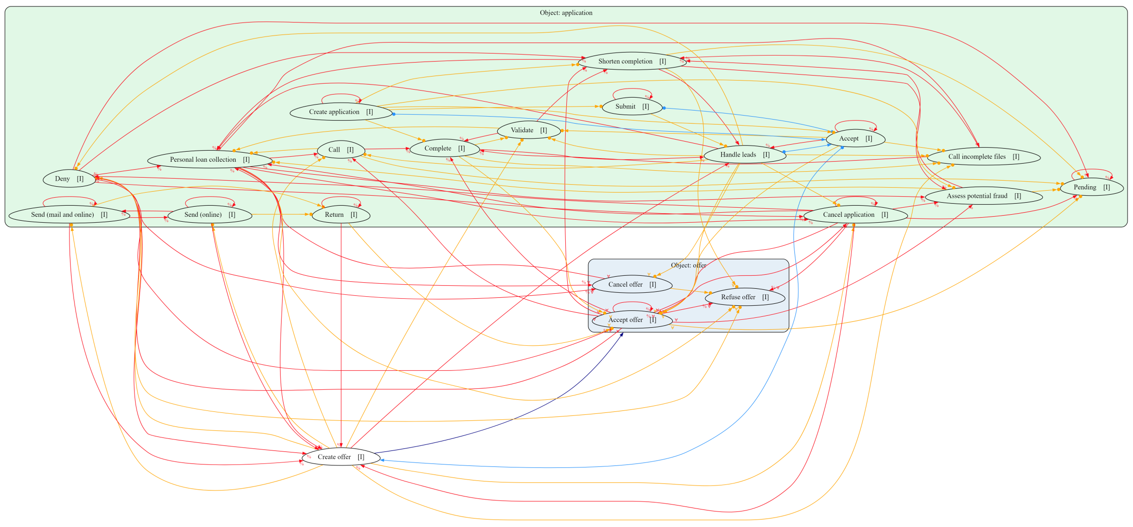

Figure 2: Discovered graph for the BPI 2017 log

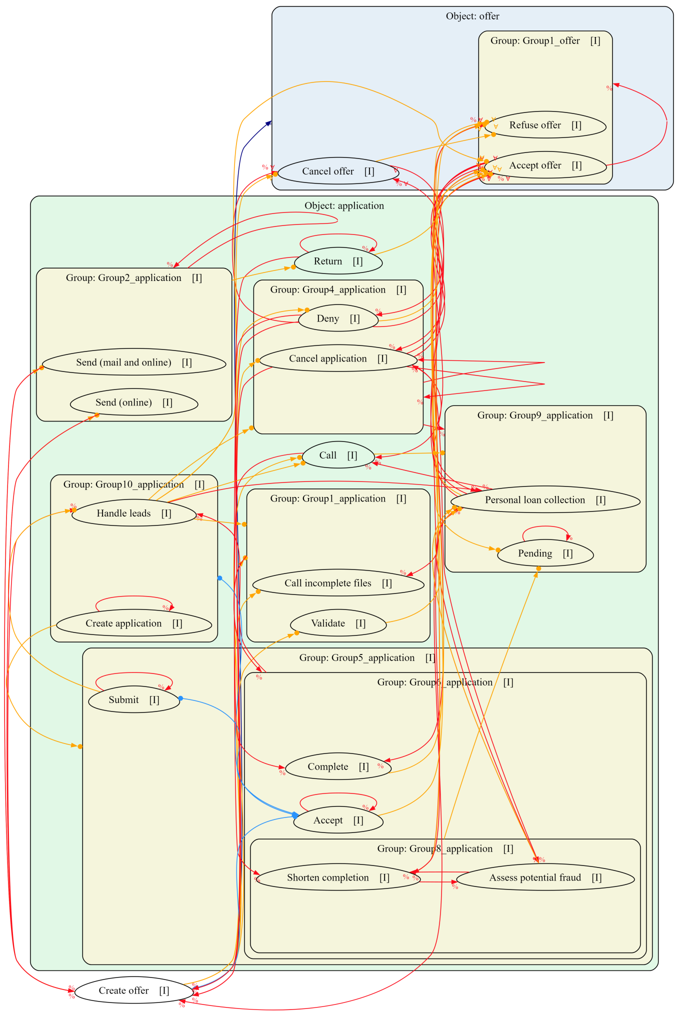
Object: offer
Group: Group1_offer [I]
Refuse offer [I]
Accept offer [I]
Cancel offer [I]
Object: application
Return [I]
Group: Group2_application [I]
Send (mail and online) [I]
Send (online) [I]
Group: Group4_application [I]
Deny [I]
Cancel application [I]
Group: Group9_application [I]
Call [I]
Group: Group10_application [I]
Handle leads [I]
Create application [I]
Group: Group1_application [I]
Personal loan collection [I]
Pending [I]
Call incomplete files [I]
Validate [I]
Group: Group5_application [I]
Group: Group6_application [I]
Submit [I]
Complete [I]
Accept [I]
Group: Group8_application [I]
Shorten completion [I]
Assess potential fraud [I]
Create offer [I]

Figure 3: Discovered graph for the bpi 2017 log with nested groups

  - `Send (mail and online)` excludes `Send (online)` and vice versa: Only one communication method is selected per message.
  - `Create offer` is a condition for `Call`, `Call incomplete files`, `Validate`, `Send (mail and online)`, `Send (online)`, and `Complete`. This reflects that an offer is necessary first to execute activities related to an offer. For example, an offer must exist to complete the process, send mails or call.
  - `Create offer` is a response to `Accept`: Once an application is accepted, sooner or later an offer has to be created in the process.
  - `Cancel offer` is a condition for `Refuse offer`: This life-cycle constraint shows that to refuse an offer, the offer has to be cancelled first. Note that here, as it is a life-cycle constraint, it does not hold for all offers, but is local to the lifecycle of each offer.

- One-to-Many Relations
  - `Cancel application` excludes `Refuse offer`, which shows that a cancellation of an application is propagated onto all offers by disallowing refusing them.
  - `Deny` excludes `Cancel offer` and `Accept offer`, which shows that once an application is denied, no offers can be cancelled or accepted any more.
  - `Deny` is a condition for `Refuse offer`, which shows that any offer can only be refused if the application is denied.

- Many-to-Many Relations
  - `Accept offer` excludes `Accept offer`: Once any offer is accepted, all other offers cannot be accepted as well.
  - `Accept offer` excludes `Refuse offer`: As, once any offer is accepted, all other offers cannot be refused, all other offers have to then be cancelled.

The result with nested applied can be seen in Figure 3, which shows the same relations as in 2, but the graph appears more compressed in the visualization due to the nested groups.

4.3 Further Remarks

We applied the OCDisCoveR algorithm on the BPI_2017.csv file provided in OCPA's sample logs. The algorithm executes in a reasonable amount of time. However, the log referenced in the given paper [3] is a different BPI file, available only in XML format. Since OCPA supports import and conversion to OCEL format only for logs in CSV format, we decided to use the provided CSV file. Despite this, the performance of our OCDisCoveR implementation on large datasets is still effectively demonstrated. Validation of the results was ensured by comparing with desired results of short, well-understood event logs and the BPI 2017 dataset.

5 Comparison with OCPA's Petri Nets

To further validate the results of the OCDisCoveR algorithm, we compare the OC-DCR Graph in Figure 1 with the Petri net in Figure 4, which was discovered by OCPA from the same event log. While Petri nets rely on tokens and transitions and OC-DCR Graphs on markings and constraints, a direct comparison seems non-trivial. However, examining the dependencies between activities and object types, by extracting the actual meaning of the models features, reveals a lot of similarities.

We can consider `Register Customer` and `Add Item` as starting points in both models. In the OC-DCR graph's Customer object type, as well as in the Petri net, `Register Customer` is followed by `Verify`. Similarly, for the `Item` object, `Add Item` has to occur before `Link Item to Order` can be executed. In the Petri net, this dependency is encoded the token flow: A token is placed before `Link Item to Order` only after `Add Item` is fired. In the OC-DCR Graph, this is represented by spawning a new `Item` object instance.

In the Petri net, firing the transition `Link Item to Order` requires that `Create Order` has already occurred. This causality corresponds to a condition relation (Sec. 3.1.1), in the OC-DCR Graph, where `Link Item to Order` is conditioned by `Create Order`.
`Create Order` consumes a token produced by `Verify`. In the OC-DCR Graph, this behavior is captured as a response relation, indicating that once `Verify` occurs, `Create Order` must eventually follow.

This pattern of comparing activity dependencies can be continued until the very last activities happening in these models. In the Petri net the end of process traces is more obvious, e.g. `Ship Order` is optionally followed by `Accept`. The same behavior is described in the graph, through a condition: `Ship Order` conditions `Accept`, meaning if `Accept` occurs, `Ship Order` had to occur before that. After `Accept` nothing else can happen anymore.

Considering the self-loop exclude relations one can argue that in the Petri net transitions can not fire twice, since the token is not passed to the place the transition consumes tokens from.

Overall we can describe very similar process behaviour. While in the Petri net the chronological sequence of activities and their dependencies are visually easier to extract, the OC-DCR Graphs allow for a more object-centric perspective, as the object-types are clearly separated. This brings the inconvenience, that activities can only belong to one object. Transitions in the Petri net can consume from places of different objects, clearly visualizing that for example `Link Item to Order` corresponds to `Order` as well as `Item`.

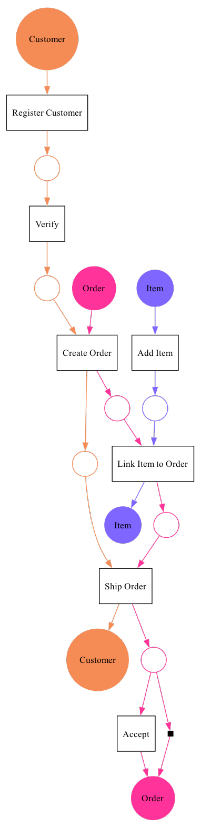

Figure 4: Petri-Net discovered by OCPA, from the same log as the OC-DCR Graph in Fig.1

6 Troubleshooting

This section covers some issues one might encounter. In case of any unexpected behavior, leading to relevant issues, please contact us, allowing us to further improve this extended version of OCPA.

6.1 Testing

We recommend running the unittests before using this implementation, using the command in Listing 9. These tests cover the core objects, as well as most parts of the algorithm.

```
python -m unittest discover -s tests/dcr -p "test_*.py"
```

Listing 9: Command used to run all Unittests

6.2 Visualization

If the visualizations appear incorrect (for example constraints overlapping along their whole path), rerunning the visualization often resolves this issue. Still be aware that the focus of this project is the algorithm and its underlying data structures. As a result, very complex graphs may not always render perfectly. In addition, as graphviz does not currently support cluster to cluster edges, our workaround with creating invisible ”bridge” nodes might result in quite confusing representations of those constraints.

6.3 Requirement Conflicts with OCPA

The original OCPA library has dependencies that differ from those used in this implementation. Some OCPA functions may produce unexpected errors due to the syntax changes introduced by updates in Python or third-party libraries.

6.4 OCPA's Tests fail

If the tests for the original OCPA implementation fail, this is most likely due to issues with the file paths used. We successfully ran all tests using absolute paths, configured via Python's os module. However, we did not alter or modify any of the original test code in the final version.

6.5 Validation of User Input

The algorithm performs extensive validation of user-provided mappings. In cases of semantic errors, it raises an exception with a descriptive error message, if the issue is non-fatal, a warning is logged instead.

6.6 Failure in first Step

A failure in the first step of the algorithm is most likely caused by an improperly formatted event log. In such cases, the resulting list of logs per object type contains empty DataFrames, which prevents the subsequent concatenation into a unified event log.

References

[1] Jan Niklas Adams, Gyunam Park, and Wil M.P. van der Aalst. “ocpa: A Python library for object-centric process analysis”. In: *Software Impacts* (2022), p. 100438.

[2] C.O. Back, A.R. Højen, and L.S. Vestergaard. “DCR4Py: A PM4Py Library Extension for Declarative Process Mining in Python”. In: *CEUR Workshop Proceedings*. Vol. 3783. 2023.

[3] Axel K.F. Christfort et al. “Discovery of Object-Centric Declarative Models”. In: *2024 6th International Conference on Process Mining (ICPM)*. 2024, pp. 121–128.

[4] Tijs Slaats et al. “Exformatics declarative case management workflows as DCR graphs”. In: *Proceedings of the 11th International Conference on Business Process Management*. BPM’13. Beijing, China: Springer-Verlag, 2013, pp. 339–354.