Discovering guard stage milestone models through hierarchical clustering

Leyla Moctar M'baba, Mohamed Sellami, Nour Assy, Walid Gaaloul, Mohamedade Farouk Nanne

▶ **To cite this version:**

Leyla Moctar M'baba, Mohamed Sellami, Nour Assy, Walid Gaaloul, Mohamedade Farouk Nanne. Discovering guard stage milestone models through hierarchical clustering. International Conference on Cooperative Information Systems (CoopIS), Oct 2023, Groningen, Netherlands. pp.239-256, 10.1007/978-3-031-46846-9_13 . hal-04257487

HAL Id: hal-04257487
https://hal.science/hal-04257487v1

Submitted on 25 Oct 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution - NonCommercial - ShareAlike 4.0 International License

Discovering Guard Stage Milestone models through hierarchical clustering

Leyla Moctar M'Baba[1,2] [0000−0002−1724−7937]
leyla.moctar$_m$baba@telecom − sudparis.eu
, Mohamed Sellami[1] [0000−0002−7547−1857], Nour Assy[3] [0000−0002−6181−410X],
Walid Gaaloul[1] [0000−0003−0451−532X], and Mohamedade Farouk NANNE[2] [0000−0002−4079−8286]

[1] Télécom SudParis, SAMOVAR, Institut Polytechnique de Paris, France
[2] University of Nouakchott Al Aasriya, Mauritania
[3] Bonitasoft

Abstract. Processes executed on enterprise Information Systems (IS), such as ERP and CMS, are artifact-centric. The execution of these processes is driven by the creation and evolution of business entities called artifacts. Several artifact-centric modeling languages were proposed to capture the specificity of these processes. One of the most used artifact-centric modeling languages is the Guard Stage Milestone (GSM) language. It represents an artifact-centric process as an information model and a lifecycle. The lifecycle groups activities in stages with data conditions as guards. The hierarchy between the stages is based on common conditions. However, existing works do not discover this hierarchy nor the data conditions, as they considered them to be already available. They also do not discover GSM models directly from event logs. They discover Petri nets and translate them into GSM models. To fill this gap, we propose in this paper a discovery approach based on hierarchical clustering. We use invariants detection to discover data conditions and information gain of common conditions to cluster stages. The approach does not rely on domain knowledge nor translation mechanisms. It was implemented and evaluated using a blockchain case study.

Keywords: Guard-Stage-Milestone · Artifact-Centric Processes · Process mining · Artifact-Centric Event Logs.

1 Introduction

In recent times, there has been a growing interest in artifact-centric systems, leading to the development of artifact-centric process mining techniques. These techniques include artifact-centric process discovery, conformance checking, and enhancement. Artifact-centric process discovery techniques aim to uncover the lifecycles of involved artifacts and their interactions [5]. Most current artifact-centric discovery approaches use event data stored in a relational database containing information about data creation, modification, and deletion [5]. Alternatively, they utilize the Object-Centric Event Log (OCEL) standard format [6].

Some approaches can also be applied to the traditional activity-centric event log format called eXtensible Event Log (XES) [4] with additional processing and filtering steps. Additionally, these techniques discover flat procedural models, focusing on the order of execution, while neglecting the influence of data on execution and interactions between artifacts. These models are typically represented using directly-follows graphs [2] or Petri nets [18]. Some approaches propose translating Petri nets into the Guard-Stage-Milestone (GSM) language [16,18], which is a well-known declarative approach for modeling artifact-centric processes. This work also concentrates on discovering GSM models. Data conditions and hierarchical abstractions are essential to artifact-centric languages like GSM [8]. In fact, GSM relies on data conditions to model authorized behaviors and parallelism within and between the lifecycles of artifacts. GSM also supports hierarchy between groups of activities which allows the representation of different levels of abstraction of the business operations. The discovery of data conditions was not described in existing approaches where GSM models discovery is based on translation mechanisms from Petri nets [16,18]. These conditions were considered as provided by domain experts or extracted using existing tools (e.g. decision miner). These approaches also omit the discovery of the hierarchical structure of a lifecycle as supported by GSM and do not consider interactions between artifacts.

Furthermore, the input of existing approaches cannot be directly used to discover GSM models. In case of OCEL logs, the data changes depicting the evolution of objects are not stored which hinders the discovery of GSM data conditions. As for classic XES logs, a processing is required to generate a log for each artifact which may cause convergence and divergence problems [1]. The discovery of the relational model from these logs requires using databases and or domain knowledge. To solve these issues we previously introduced ACEL (Artifact-Centric Event Log) [12], an extension of OCEL which is specific for artifact-centric event data. An ACEL log supports multiple case notions and contains information about artifacts, their evolution (lifecycle) and their relations.

To address the aforementioned limitations, we propose a discovery technique that takes as input an ACEL log and gives as output a GSM model. Taking ACEL as input avoids classic convergence and divergence problems and alleviates the pain of processing, translating and fetching additional external knowledge. The novelty of our proposed technique compared to existing ones is i) its ability to discover stages' data conditions (i.e. guards) by analyzing the data changes of objects' evolution stored in ACEL logs; ii) discovering nested stages based on the idea of hierarchically clustering stages according to the Information Gain [19] obtained by grouping their data conditions and iii) discovering interactions between different artifacts as well as between instances of the same artifact type using data conditions. The approach has been evaluated in terms of feasibility using an ACEL log generated from a blockchain application. We used blockchain logs because they are an immutable trustworthy source of data for process min-

[4] `https://xes-standard.org/`

ing. Process mining shows the business process perspective of smart contracts and it has been demonstrated for activity-centric processes [7]. However, to the best of our knowledge, no work used blockchain logs to discover artifact-centric processes.

The paper is organized as follows. Section 2, provides some background on GSM and ACEL. The approach is briefly presented in section 3 along with a running example. The approach is detailed in section 4 and 5. Section 6 presents the implementation and evaluation through a case study and a discussion. Section 7 reviews the related work. Finally, Section 8 concludes the paper.

2 Preliminaries

In this section we provide an overview of the GSM model and the artifact-centric event log format (ACEL) we consider in this work.

2.1 The Guard-Stage-Milestone Model

Artifact-centric processes revolve around the progression of business entities known as artifacts as they undergo various business operations. Each artifact possesses an information model and a lifecycle. The information model stores data in the form of attributes that capture information about the artifact throughout its existence. On the other hand, the lifecycle represents a "micro process" model that outlines the sequence of operations or tasks that can be performed on the artifact to transition it from one state to another. In addition, artifacts have the ability to interact with one another and establish relations. These relations are represented by nested foreign key attributes, allowing for the establishment of connections between artifacts. GSM is a declarative approach to specifying artifact lifecycles[9]. It represents *(i)* the lifecycle of an artifact in terms of guards, stages, and milestones, and *(ii)* the information model in terms of data and state attributes. Stages are groups of tasks (activities) which modify an artifact in order to achieve a certain business goal. A stage with only one task is called an atomic stage. A stage can have multiple guards and milestones. Guards consist of sentries which are comprised of triggering external or internal events and/or conditions on data. Internal events and data conditions may refer to the modeled artifact or other artifacts. External events can come from external services or human actions. Milestones correspond to business-relevant operational objectives and are achieved (or invalidated) based on a sentry. A stage becomes active/open when the sentry of one of its guards becomes true and inactive/closed when the sentry of one of its milestones becomes true. Stages can be nested, i.e., a stage can contain several stages [8]. Fig. 1 illustrates a sample GSM model for a blockchain-based application (Section 3.1 details this running example).

2.2 The Artifact-Centric Event Log Format

Artifact-Centric Event Log (ACEL) [12] is an enhanced version of the Object-Centric Event Log (OCEL) standard [6] designed for storing event data in

artifact-centric business processes. ACEL extends OCEL by supporting the storage of object relations and attribute changes of both objects and relations. In ACEL, each event contains information about the execution of a business process activity, including the objects and relations modified by that activity. Moreover, ACEL captures attribute-level changes, meaning that the new value of an attribute reflects the specific alteration made. Objects in ACEL represent relevant business entities or artifacts, while relations represent the connections between artifacts, including one-to-one, one-to-many, and many-to-many relationships. Within an ACEL log, events, objects, and relations are uniquely identified and can possess multiple attributes. Certain attributes are mandatory, such as the lifecycle attribute for objects, which indicates their state, and the source and target attributes for relations, which identify the objects involved in the relation. Each object or relation in ACEL is associated with a specific object or relation type, and multiple objects or relations can be associated with a single event [12]. Table 1 provides an excerpt of a sample ACEL log.

3 Approach overview

3.1 Running Example

Cryptokitties [5] is an Ethereum Dapp [6] where cats are auctioned for sale or breeding purposes. The sale auction procedure can be considered as a GSM stage. A cat is auctioned for sale when its owner triggers the opening of the `CreateAuction` stage, which creates a new auction in the Created state. Other users can buy the auctioned cat by making bids. When a satisfying bid is made, the `CompleteAuction` stage is opened which triggers the transfer of the cat to the new owner and the auction becomes in the Successful state. The user who initiated the auction can also cancel it by triggering the `CancelAuction` stage which puts the auction in the Cancelled state. A user can also obtain a new cat by breeding one of his cats with another cat. After Breeding, the mother and the father are expecting, i.e., the mother is in the Pregnant state and the father is in the FutureFather state. After a prefixed amount of time, called cooldown period, the user can trigger the opening of the `Birth` stage and the mother gives birth to a new cat which is in the Born state. After the Birth, the mother and the father become in the BecameMother and BecameFather states, respectively.

Fig. 1 shows an excerpt of the representation of a cat's and an auction's lifecycles in GSM. Both artifacts, i.e., cat and auction, have data attributes constituting their information model, e.g., `tokenId`, `cooldownPeriod` and `startingPrice`. These attributes also indicate relations between artifacts by referring to other artifacts or other instances of the same artifact, e.g., `SiringWithId` and `KittyId`. Artifacts are also characterized by status attributes indicating a stage's or a milestone's status, e.g., Pregnant and Successful, is achieved. Fig.1 also illustrates the

[5] `https://www.cryptokitties.co/`

[6] A Dapp is a decentralized application running on a blockchain platform.

possible nesting of stages, it shows two nested stages, Procreation and SaleAuction. The Procreation stage has one guard whose sentry is one data condition (`k.'cooldown'`≤`currentTime`) and one milestone (BecameMother) whose sentry is an internal event about the cat's milestone (`k.'Pregnant'.achieved()`). When its guard is true, the Procreation stage is opened and its sub-stages, Breeding and Birth, can be activated when their respective guards are true. For example the sentry of Birth's guard is composed of one external event (`k.'giveBirth'.onEvent()`), to be sent by a user, and a data condition (`k.'Pregnant'`). The previous data condition is an internal event but expressed as a data condition because sentries can have only one event. In our previous paper [12], we proposed an approach to generate ACEL logs for blockchain applications. We used it to generate a log for our running example, as shown in Table1.

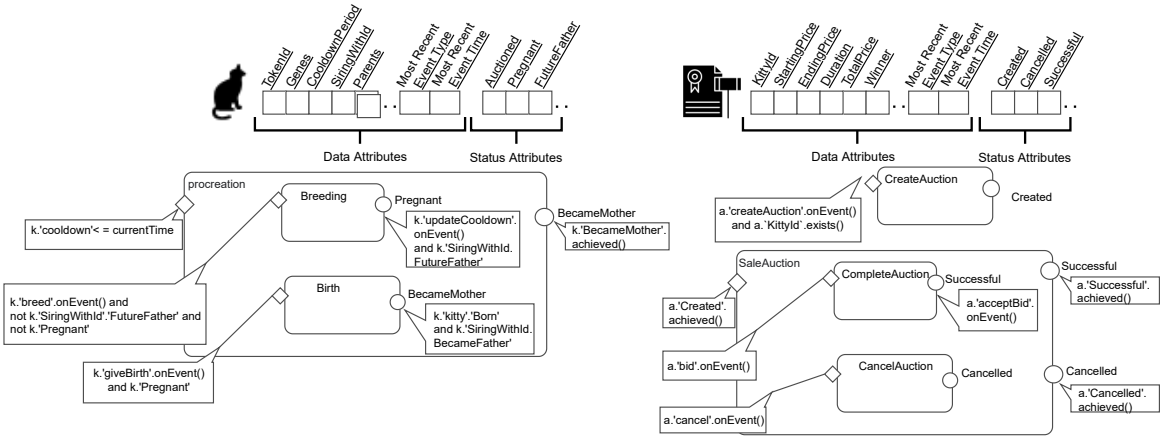

Fig. 1: GSM process model associated with the Cryptokitties example

3.2 Discovering GSM models from ACEL logs

Discovering a GSM model, similar to Fig.1, consists of discovering *(i)* the information model of each involved artifact, *(ii)* its different stages and their hierarchical structure, *(iii)* the guard(s) and milestone(s) of each stage, and *(iv)* the interaction between the different artifacts.

Discovering a GSM information model from an ACEL log is simply an extraction process because ACEL stores the artifact relational model. The extraction consists of collecting the attribute names for each artifact. Then, following the relations, foreign keys are added to the information model. Therefore, the required steps can be reduced to discovering stages' guards, their interactions and hierarchy. An overview of our proposed approach is illustrated in Fig. 2.

Table 1: Sample of Cryptokitties ACEL log

EventId	Activity	Timestamp	Attribute		Objects	Relations
			Name	Value		
e1	Breeding	23/10/2021 04:11:51	Resource	0x22D1A..	1960326	r1

ObjectChanges		
ObjectId	Attribute	NewValue
1960326	lifecycle	Pregnant
1960326	CooldownPeriod	11115513
1688830	lifecycle	FutureFather
1688830	CooldownPeriod	11115513

RelationChanges		
RelationId	Target	ChangeStatus
r1	1688830	addedTarget

(a) Events

ObjectId	Type	genes
1960326	kitty	5321000..
1688830	kitty	62855942..

(b) Objects

RelationId	Type	Source
r1	siringWith	1960326

(c) Relations

We can discover GSM stages by discovering their guards because two stages cannot have the same guards. Sub-stages have their own guards, and in addition they inherit the guards of their parent stage. This creates a hierarchy between sub-stages and their parent, e.g., in Fig.1 the Procreation stage is the parent of the Breeding and Birth stages. Furthermore, we can consider that the parent stage is a cluster of sub-stages and each sub-stage can also be a cluster of its own sub-stages. Therefore, we can model the discovery of stages and their hierarchical structure as a hierarchical clustering problem where similarity is based on common guards. Additionally, the clusters have to be loosely coupled, i.e., the guards of one stage should not allow the activation of another stage.

As mentioned above, discovering guards is a prerequisite to the discovery of stages and their hierarchy. A guard consists of a sentry which contains an event, internal or external, e.g., (k.'Pregnant'.achieved()) and (k.'giveBirth'.onEvent()) in Fig.1, and data conditions, e.g., (k.'cooldown'$\leq$ currentTime). Since ACEL logs do not contain external events, we only consider internal events for the discovery of guard sentries. Furthermore, the internal events we consider are about achieving milestones, the latter are stored in ACEL as values of the object attribute lifecycle. Therefore, we can discover them by discovering data conditions and since in GSM, internal events can be expressed through data conditions, e.g., in Fig.,1 (k.'Pregnant'), we represent them as such. Thus, the discovery of guard sentries (internal event and data conditions) becomes a problem of discovering data con-

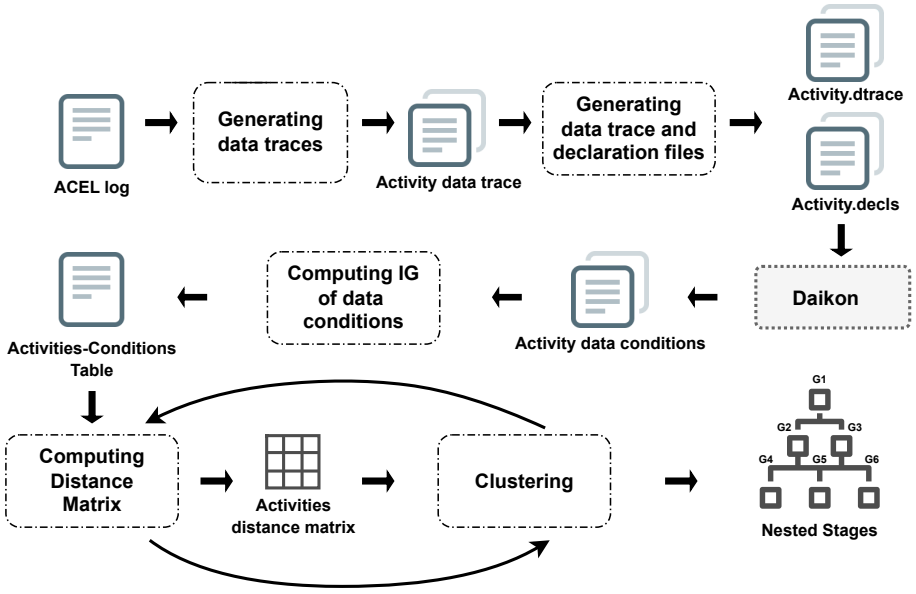

Fig. 2: Overview of the GSM models discovery approach

ditions on artifact attributes and their milestones. Milestones of a stage can be some or all milestones of its sub-stages but they can also be new milestones. In the case of an atomic stage we consider that its milestones are those of its task (activity) and the milestones of its parent stage can be independent of the task's milestones. Task independent milestones are not supported by ACEL. In ACEL, an activity (task) is linked to one lifecycle (milestone) change, since it is not GSM specific. Stage specific milestones can be introduced through an optional custom object attribute [12]. However, since we rely on classic ACEL logs we will not introduce stage specific milestones and we will define the milestones of an atomic stage as the milestones of its task.

In GSM, interactions between artifacts can be represented by internal events about achieved milestones of other artifacts. We discover these events through data conditions, as mentioned above.

The following sections detail these different steps: **discover guards and interactions** as data conditions (Section 4) and clustering activities to **discover nested stages** (Section 5).

4 Discovering guards and interactions

As stated in section 3.2, for an ACEL log, discovering guards and interactions amounts to discovering data conditions. Data conditions are properties verified by observed attribute values before the execution of an activity. They are similar to invariants that hold at a certain point in a program [4]. Techniques used to

dynamically detect program likely invariants can be applied to data conditions in a process as demonstrated in [10]. We adapted the latter approach to our context as detailed in the following subsections. We first generate data traces to store the values of attributes before an activity. Then we rely on invariant detection to discover the data conditions that hold in these traces.

4.1 Generating Data Traces

In the context of program likely invariants detection, a data trace contains the values of variables at points of interest in a program. In the case of a business process (BP), a data trace contains the values of variables before or after the execution of an activity. Systems like Daikon [4] generate data traces for a program given its source code. For a BP on the other hand, these traces are extracted from the BP event logs. The method for extracting data traces followed by [10] consists of replaying the events of a process instance against a reference model to update the value of each variable and stopping before the execution of the target activity to get one data trace. We propose a method to extract data traces from an ACEL log without relying on a reference model. We define in the following the concepts we used to generate data traces.

Artifact-centric process instances. We consider that a process instance should be defined for each artifact's lifecycle as our aim is to discover the stages of each artifact. Since we aim to discover interactions between artifacts as well, we consider events associated with related artifacts. Thus, we propose the following simple definition: "*A process instance is the sequence of events linked to one artifact instance and its related instances up until the end of their relations*". Identifying ACEL events of related artifact instances through relations is possible because ACEL stores the evolution of relations. This is done through the `changeStatus` attribute, associated to the relation's `target`, whose value `'deletedTarget'` indicates the end of a relation. The relation's end marks the end of an interaction between two artifact instances which reduces the possibility of irrelevant data conditions. For example, after a breeding event, a relation `breedingWith` links a pregnant cat pk and the future father fk. After a birth event this relation is deleted. fk is not allowed to breed again while it is in a `breedingWith` relation. Thus it is affected by the events of pk, i.e., it is waiting for a birth event to occur for pk. After the end of the relation, fk is no longer affected by the events of pk. For instance, if after the end of the `breedingWith` relation a breeding event occurs for pk with a third cat this has no consequence for fk. We also consider the case where objects do not "die" and the log contains several iterations of the lifecycle for one artifact instance. Previous approaches reviewed in [5] consider the cases where artifact instances go through their lifecycle only once (i.e.,created→updated... →terminated). We take into consideration the cases where artifact instances can revisit parts of their lifecycle several times or indefinitely, e.g., a cat never dies and can go through breeding endlessly.

Artifact-centric activity specific traces. Given the previous definition of an artifact-centric process instance and the possibility of revisiting artifacts' lifecycle states, we define an activity specific trace as: "*a sequence of events, from*

a process instance, delimited by two events related to the activity". The events of the activity are of course not included in the activity specific trace since we aim to discover data conditions preceding it.

Data Trace Generation. To generate a data trace from an activity specific trace, we apply a reverse traversal of the events to get data values instead of replaying the events. Starting from the last event we get the first value we encounter for each attribute of each artifact instance. We rename the attributes to improve readability, to quantify the artifact interactions, and shed light on eventual new types of interactions, as detailed in the following.

In GSM, related artifacts are referenced in the information model of the main artifact through a foreign key attribute. However, the name of this attribute might not provide information about the artifact's type. Thereby, we prefix it (in the data trace) by the latter's artifact type. For example in Fig.1, the `k.siringWithId` attribute references another cat and to make this explicit we prefix it with its type (i.e., `k.kitty.siringWithId`). However, in ACEL an artifact does have such foreign key attribute since the relations are stored and thus the notion of foreign key is unnecessary. Therefore we extend the previous prefix with the type of the relation (`k.breedingWith.kitty.siring-WithId`) to provide business relevant semantics and enhance the readability of the discovered model. This also allows us to indicate the cardinality of interactions[7] and define a new type of interaction which were not considered in previous works, to the best of our knowledge. Indeed, previous works considered only interactions between different artifacts, while we consider interactions between instances of the same artifact which we call reflexive interactions.To illustrate interaction cardinalities and reflexive interactions, we refer again to the Cryptokitties example and specifically to the `Birth` stage. The birth of a cat relates him to a father and a mother cat. In this case, the born cat is the main artifact and it interacts twice (cardinality) with instances of the same type (reflexive interaction).

4.2 Discovering Data Conditions

For the discovery of data conditions from data traces we use the same implementation of dynamic detection of likely invariants as [10], named Daikon. Along with data traces, Daikon [4] requires declaration files specifying the corresponding program points and variables in a data trace. In these declaration files, we also specify the comparability[8] property of the variables. This property assists Daikon in discovering relevant invariants, i.e., invariants involving only comparable variables, and in our case relevant data conditions. We can infer correlations between variables from the ACEL, e.g, which variables appear frequently together in the ObjectChanges list 1a. However this statistical analysis is out of the scope of this paper, we consider that the comparability is provided. Specifically, to discover data conditions using Daikon, we merge the activity specific

[7] Cardinality of interaction is the number of artifact instances of the same type interacting with the main artifact.

[8] A signed integer that indicates to Daikon comparable variables. Two variables with the same value for comparability are considered comparable.

data traces generated in Section 4.1 for each artifact instance in one Daikon data trace file. We then generate for it a declaration file. We use these two files as input to generate invariants for the activity data trace.

The steps, described in section 4.1 and 4.2, to generate data traces and discover data conditions are illustrated in Algorithm 1.

5 Discovering Nested Stages

To discover nested stages we use common data conditions with the same information gain. Information gain ($\mathcal{IG}$) measures how well a feature helps predict a label. It is based on the entropy which is, in the context of classification in machine learning, the measure of the diversification of the labels in a data set. The lowest entropy is equal to zero and correspond to a pure data set, i.e., all elements have the same label. The highest entropy is equal to one and corresponds to a data set with equal subsets for each label. $\mathcal{IG}$ is inversely proportional to the entropy, i.e., an entropy equal to zero corresponds to an $\mathcal{IG}$ equal to one.

We consider that stages having a common parent will be discriminated equally by this parent's guard from other stages of the artifact's lifecycle. This means that the guard will have the same $\mathcal{IG}$ each time it discriminates one activity of the sub-stages from other stages in the artifact. This is true because we can consider the parent stage as a label and the data condition as a feature. For example in Fig. 1, if we label all activities as `partOfProcreation` or `notPartOfProcreation`, the data condition (`k.'cooldown'` $\leq$ `currentTime`) that best splits the population, i.e., the activities, will be the guard of the `Procreation` stage.

Furthermore, sub-stages share the guard of their parent stage. Thus, discovering nested stages amounts to grouping stages with the same $\mathcal{IG}$ for their common data conditions. These common conditions represent the guard of the parent which can be merged with other stages when their guards overlap. The discovery of nested stages is therefore a hierarchical clustering where similarity is based on common data conditions with the same $\mathcal{IG}$. In the following, we propose a similarity function that we consider for clustering activities (Section 5.2) and an approach for hierarchically clustering the stages (Section 5.3).

5.1 Limitations of branching conditions for discovering GSM stages

The authors of [10] use $\mathcal{IG}$ to discover conditions that discriminate between two tasks in a branching point. This is not our case because GSM supports parallelism between stages and activities of a same stage. They also use $\mathcal{IG}$ to simplify the conditions, i.e., they only keep the condition or conjunction of conditions with the highest $\mathcal{IG}$. For example, in the `Breeding` stage (Fig. 1), $if\ IG($`!k.'Pregnant'`$) == IG($`!k.'Pregnant'` && (`k.'cooldown'` $\leq$ `currentTime`$))$, the condition (`!k.'Pregnant'&& k.'cooldown'` $\leq$ `currentTime`) will be discarded according to [10]. Such simplification makes the discovery of the `procreation` stage impossible. Therefore, we do not discard any condition since conditions

Algorithm 1: Data Conditions discovery

Data: acelLog $< E, O, R >$, E set of events, O set of objects and R set of relations. $OT \leftarrow \emptyset$, set of object types and $A \leftarrow \emptyset$, set of activities.

Result: DT, a function which associates to each object and activity a set of data traces

Let *TA be a function whose domain is* OT $\forall ot \in OT$, $\exists (a_1, ..., a_n) \in A^n, TA(ot) \leftarrow (a_1, ..., a_n)$.

Let *OI be a function whose domain is* $OT \times A$ $\forall ot, a \in OT, A$, $\exists (o_1, ..., o_n) \in O^n, OI(ot, a) \leftarrow (o_1, ..., o_n)$.

Let *T be a function whose domain is* $OT \times A \times O$ $\forall ot, a, o \in OT, A, O$, $\exists n, m \in \mathbb{N}, T(ot, a, o) \leftarrow (E^n)^m$.

ForEach *e of E*
  $a \leftarrow activityName(e)$;
  ForEach *o of objectList(E)*
    $ot \leftarrow type(o)$;
    ForEach *act of TA(ot)*
      If *a = act*
        Close last set of T(ot, a, o) ;
      else
        If *last set of T(ot, a, o) closed*
          Open new set in T(ot, a, o);
        Add e to last set of T(ot, a, o);
      end
ForEach *ot of OT*
  ForEach *a of TA(ot)*
    ForEach *o of OI(ot, a)*
      ForEach *set of Reverse(T(ot, a, o))*
        Open new set in $DT(ot, a)$;
        ForEach *e of Reverse(set)*
          ForEach *ob in objectChangeList(e)*
            ForEach *att in objectAttributes(ob)*
              If *ob = o*
                If *name(att) = 'lifecycle'*
                  Add (ot.'milestone', value(att)) to last set in $DT(ot, a)$;
                else
                  Add (ot.name(att), value(att)) to last set in $DT(ot, a)$;
                end
              else
                If *ob in relation with o and relation not ended*
                  If *name(att) = 'lifecycle'*
                    Add (ot.relationName(o,ob).type(ob).'milestone', value(att)) to last set in $DT(ot, a)$;
                  else
                    Add (ot.relationName(o,ob).type(ob)name(att), value(att)) to last set in $DT(ot, a)$;
                  end
              end
        Close last set in $DT(ot, a)$;
return DT

with the lowest $\mathcal{IG}$ might represent guards for parent stages. Nevertheless, the use of $\mathcal{IG}$ for branching conditions in [10] inspired our approach to detect stages.

5.2 Similarity Between Activities

Similarity is measured through distance, two points of a cluster are similar when they are the closest. We consider that two activities are close when they share common data conditions with the same $\mathcal{IG}$. The closest activities will have the same data conditions with the same $\mathcal{IG}$ and their distance is equal to zero. The furthest apart share no common data conditions or their common data conditions do not have the same $\mathcal{IG}$ and their distance is equal to one. In the following, we define a similarity function to compute the distance between two activities based on their data conditions and their $\mathcal{IG}$.

Definition 1 ($\mathcal{IG}$ of an Activity Condition). *Let $\mathcal{A}$ be the set of an artifact's activities, $\mathcal{C}$ the set of these activities' data conditions, $\mathcal{DT}$ the set of all data traces, and $adt\colon \mathcal{A} \longrightarrow \mathcal{DT}$ a mapping associating activities $\in \mathcal{A}$ to their data traces $\in \mathcal{DT}$. The information gain of an activity's condition is defined as:*

$$\forall a \in \mathcal{A}, \forall c \in \mathcal{C}, \mathcal{IG}_a(c) = \mathcal{IG}(adt(a), adt(\mathcal{A} \setminus a), c)$$

Definition 2 (Similarity Function). *Let $a, b \in \mathcal{A}, C_a, C_b \subset C, CC_{ab} = \{c | c \in C_a \wedge c \in C_b \wedge \mathcal{IG}_a(c) = \mathcal{IG}_b(c)\}$ the set of common data conditions with the same $\mathcal{IG}$ between two artifact's data conditions a and b. $\forall c_k \in CC_{ab}, k \in \{1 \dots n\}, n = |CC_{ab}|, \mathcal{IG}(c_k) = \mathcal{IG}_a(c_k) = \mathcal{IG}_b(c_k)$. The distance between a and b is given by:*

$$dist(a,b) = \begin{cases} 1 & |CCab| = 0 \\ 1 - \dfrac{1}{\log \dfrac{|Ca| + |Cb|}{2 \times |CCab|}} & |CCab| \neq 0 \\ \quad 1 + \dfrac{}{\sum\limits_{k=1}^{n} IG(c_k)} & \end{cases}$$

5.3 Hierarchical Clustering to Discover Nested Stages

We rely on a hierarchical agglomerative clustering [13] with a distance matrix computed using the similarity function *dist* and a different linkage criterion to determine similarity between clusters. As a linkage criterion in our context, we consider that all points, i.e., activities, of one cluster must have the same distance with all the points of the other cluster. We optimize this criterion by measuring the distance between two random activities, one of each cluster, based on the data conditions shared by all activities of each cluster. The merging condition we consider is that the similarity between two clusters is different from one. Indeed, because activity with no data condition in common, or different $\mathcal{IG}$ for their common data condition, cannot be in the same stage.

The clustering starts with the computation of a distance matrix between all activities of an artifact. Then (first iteration), the two closest activities are

merged. After this merge, the distance matrix between the rest of the activities and this new cluster is computed using the common data conditions shared by its activities. Next (second iteration), the next two closest activities, or one activity with the previous cluster if their distance is the shortest, are merged. The clustering continues with the re-computation of the distance matrix before each iteration and then merging of clusters until only one cluster or when the distance between all clusters is equal to one (stopping condition). The result is a hierarchical structure indicating the nesting of the stages.

6 Evaluation

In the following we present the implementation and the evaluation of the approach through a case study.

6.1 Implementation

We implemented the approach as several python modules, accessible via `https://gitlab.com/disco5/Gsm/-/tree/main/discovery`. The first module takes as input an ACEL log and generates a data trace file for each activity of each artifact. The second module converts this data trace file into the daikon .dtrace format and generates a declaration file in the daikon .decls format for each data trace. It then runs Daikon with the previous files to discover data conditions for each activity. The third module takes as input the data trace and the discovered data conditions of all activities of an artifact and computes for each activity the $\mathcal{IG}$ of data conditions, including their conjunctions. It outputs a table with as header all data conditions and a line for each activity with the $\mathcal{IG}$ of its data conditions. The previous table is the input of the fourth module in charge of the clustering, which first uses the table to run the clustering algorithm and discover the nested stages, then assigns to each stage its guard.

6.2 Case Study

As a case study we chose the blockchain application Cryptokitties and extracted its corresponding ACEL from Ethereum using [12]. The model discovered using our approach is accessible via `https://gitlab.com/disco5/Gsm`. For the cat artifact, the discovered lifecycle is briefly presented in Table 2 and illustrated in Fig. 3. The lifecycle presents two atomic stages $S1$ and $S2$ containing the activities Birth and Breeding, respectively. The data conditions (K.milestone = Pregnant) and (K.breedingWith.Kitty.milestone = FutureFather) of the Birth stage are a result of the closing of the Breeding stage which indicates that the latter always precedes the Birth stage. The condition ('K.breedingWith.Kitty'.milestone) is an internal event indicating a reflexive interaction with cardinality one, i.e., a cat instance has a relation 'breedingWith' with the main cat instance. For the auction artifact, a sample of the discovered lifecycle is presented in Table 3 and illustrated by Fig. 4. The lifecycle presents one stage $S1$ containing the two

atomic stages with the activities CompleteAuction and CancelAuction. The data condition (A.milestone = Created) indicates that an auction needs to be created to be completed or cancel and guards both from occurring to the same auction. Indeed, as indicated in Fig. 1, once an auction is created, it can only be completed or cancelled.

Table 2: Sample of a discovered cat lifecycle.

Stages	Guards
S1 (Birth)	(K.cooldownPeriod < Timestamp and K.milestone == Pregnant and K.breedingWith.Kitty.milestone == FutureFather)
S2 (Breeding)	(K.milestone == Transferred or K.milestone == Sold or K.milestone == BecameMother)

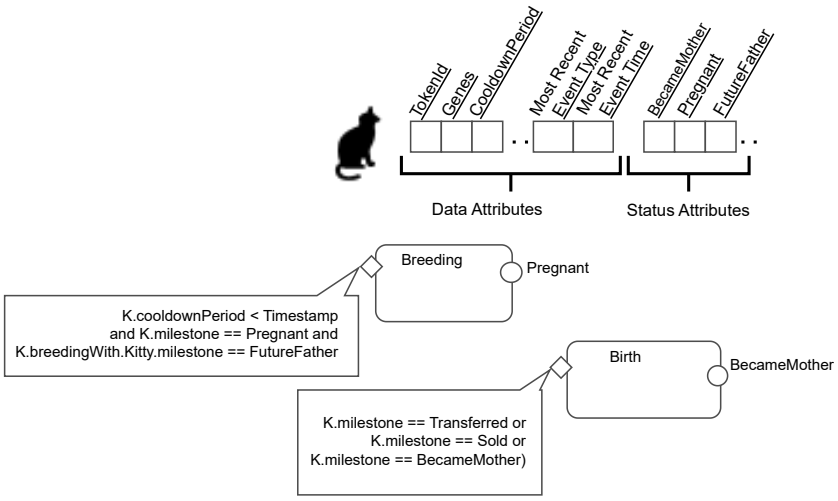

Fig. 3: Discovered GSM model: cat lifecycle

6.3 Evaluation and Discussion

We evaluate in this section our approach in terms of its ability to discover data conditions and interactions as well as the performance of the clustering algorithm.

Table 3: Sample of a discovered Auction lifecycle.

Stages	Guards
S1 (CompStagesleteAuction, CancelAuction)	(A.milestone == Created)

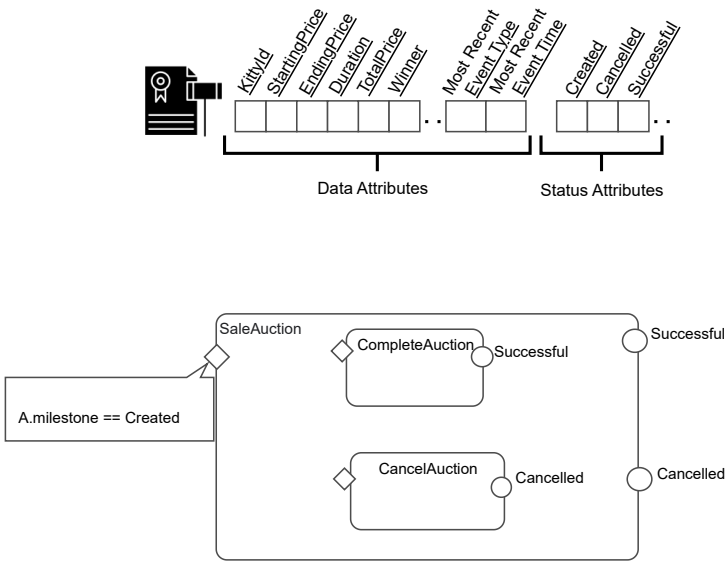

Fig. 4: Discovered GSM model: auction lifecycle

Guards and interactions discovery evaluation. To evaluate our approach, we refer to the GSM model of Cryptokitties derived from the whitepaper [9] of the application and illustrated in Fig.1. For the Birth stage, we denote two data conditions (k.'Pregnant' and K.cooldownPeriod < Timestamp) and one external event (k.'giveBirth'.onEvent()). Using our approach, the data conditions were discovered in addition to one more condition (K.breedingWith.Kitty.milestone = FutureFather). This additional condition is the opposite of the condition (not k.'siringWithId'.'FutureFather') of the Breeding stage, thus its discovery is accurate since it differentiates between the two stages. External events were not considered as explained in Section 3.2. Regarding the interactions, in Fig.1 no interactions are shown for the `Birth` stage but our approach detected a reflexive interaction represented by the attribute K.breedingWith.Kitty.milestone. This shows that the Father's lifecycle affects that of the Mother.

However, the `Procreation` stage was not discovered since the condition (K.cooldownPeriod < Timestamp) was only discovered for the Birth activity. This is due to the fact that in the extracted log, the attribute 'cooldown' only appears in events related to Breeding

[9] `https://www.cryptokitties.co/technical-details`

and not to Birth. When we examined the source code of Cryptokitties we found that it checked and updated the value of 'cooldown' for both Birth and Breeding activities, but only logged it for Breeding related events. Therefore, a more precise logging mechanism would have allowed us to discover the `Procreation` stage.

This deduction is valuable for the redesign phase of the DApp and offers appropriate data for conformance checking techniques. Indeed, if the logging was accurate DApp developers would have noticed the need to add more guards before the execution of certain activities.

Furthermore, our approach accurately discovered the SaleAuction stage, as it appears in Fig.1, with two atomic stages CompleteAuction and CancelAuction. The guards of the atomic stages were not discover as they are solely composed of external events which are out of the scope of this paper.

Nested stages discovery evaluation. We evaluated our hierarchical clustering based approach using the silhouette coefficient (Definition 3), a metric for evaluating the performance of clustering algorithms in terms of clusters' cohesion and separation [15].

Definition 3 (Silhouette Coefficient). *The silhouette coefficient of one sample point in a cluster is given by:*

$$S = \frac{b - a}{max(a, b)}$$

Where, a is the average distance between the sample point and all the other points in the same cluster; and b is the minimum average distance between the sample point and the points of the other clusters.

The silhouette coefficient has a value in $[-1, 1]$ for each point. Incorrect clustering will give a score between 0 and -1, while a correct clustering will give a score between 0 and +1 and a score of zero indicates overlapping clusters. The silhouette coefficient of the discovered stages $S1$ and $S2$ gives a score of 1 since each each cluster contains only one point and the distance between the two points is 1 because they have no condition in common. Therefore, our algorithm produced dense well separated clusters.

Discussion. The accuracy of the discovered lifecycles is due to our definition of artifact-centric process instance (Section 4.1) and also to our use of ACEL logs. Normally convergence and divergence problems arise when events of related artifacts objects are duplicated in an artifact-centric process instance [5]. This did not happen in our case since we only collect data from related events without considering their related activities in the stage discovery. However, in the case of reflexive interactions this problems could still arise. For example the Birth event is linked to three instances of the cat artifact and can be duplicated for all of them. However, since ACEL supports transition relations, i.e., an event can affect an instance without being part of its trace, the event relating to the Birth activity is linked to the mother only and affects the father and the new born cat. One of the limitations of our GSM models discovery approach is the fact that it does not discover stage specific milestones, and hence milestone sentries (post data conditions). It also allows for discovering only one guard per stage.

7 Related Work

Several approaches in the literature have explored the discovery of artifact-centric processes. Most of them focus on the discovery of artifact types, their relations and the

lifecycle of each artifact. They use classic discovery techniques to discover the artifact lifecycles [5]. The resulting lifecycles are represented as procedural processes not suited for the declarative nature of GSM. These approaches do not consider data conditions in their discovery and only few of them consider interactions between lifecycles [11,3,17]. Interactions between artifacts are essential to determine behavioral dependencies in an artifact-centric process. Furthermore, to the best of our knowledge, no work considers blockchain logs as a source to discover artifact-centric processes. To the best of our knowledge, only one work attempted to discover GSM models from event logs [18] and they also used classic discovery techniques with a translation step. They discover a petri-net for each lifecycle and then translate it to a GSM model by considering that each transition is a stage. They do not take into account the interactions between different artifacts and consider the data conditions of the petri-net as provided. They also do not discover the hierarchy between stages, i.e., they only consider atomic stages. The different levels of abstraction of operations are thus not discovered. The approach of [17] discovers unbounded synchronization conditions between artifacts in GSM models. These conditions are the number of related artifact instances that need to reach a certain milestone before the stage of the main artifact can be opened. They represent these conditions as part of the data conditions of a guard. However, they do not discover other data conditions nor consider reflexive interactions. In [14], the authors propose to group activities as stages through graph cuts. However they do not discover stages' nesting and rely only on directly follows relations, not data conditions, when discovering stages. This is not applicable with GSM models because two stages can be active at the same time and all their activities will have strong directly follows relations.

8 Conclusion and Future Work

In this paper we presented a technique to discover GSM models from ACEL logs using hierarchical clustering where similarity is determined by common data conditions with the same information gain. We implemented and tested it on Cryptokitties, a blockchain application. The results show that the technique discovers stages according to the data recorded in the log. It also discovers interactions between artifacts, including reflexive interactions. Future work will be focused on addressing limitations discussed in section **??**. We will explore the use of post conditions in the discovery of nested stages. We will also work on discovering several guards per stage using disjunctive data conditions. Furthermore, we will investigate the refining of the clustering by setting a threshold to IG when computing the distance. We are also planning a larger study to further evaluate our approach using more case studies with more complex processes and richer larger logs. We will also apply our approach to data source other than blockchain.

References

1. van der Aalst, W.M.P.: Object-centric process mining: Dealing with divergence and convergence in event data. In: SEFM. LNCS, vol. 11724, pp. 3–25 (2019)
2. Berti, A., van der Aalst, W.M.P.: Extracting multiple viewpoint models from relational databases. CoRR **abs/2001.02562** (2020)
3. van Eck, M.L., Sidorova, N., van der Aalst, W.M.P.: Guided interaction exploration in artifact-centric process models. In: IEEE CBI, Thessaloniki, Greece, July 24-27. pp. 109–118. IEEE Computer Society (2017)

4. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., et al.: The daikon system for dynamic detection of likely invariants. Sci. Comput. Program. **69**, 35–45 (2007)
5. Fahland, D.: Artifact-centric process mining. In: Encyclopedia of Big Data Technologies (2019)
6. Ghahfarokhi, A.F., Park, G., Berti, A., van der Aalst, W.M.P.: OCEL: A standard for object-centric event logs. In: New Trends in ADBIS, Tartu, Estonia. CCIS, vol. 1450, pp. 169–175 (2021)
7. Hobeck, R., Klinkmüller, C., Bandara, H.M.N.D., Weber, I., van der Aalst, W.M.P.: Process mining on blockchain data: A case study of augur. In: BPM, Rome, Italy, Sep 06-10. LNCS, vol. 12875, pp. 306–323. Springer (2021)
8. Hull, R., Damaggio, E., Fournier, F., Gupta, M., et al.: Introducing the guard-stage-milestone approach for specifying business entity lifecycles. In: WS-FM - 7th Workshop, Hoboken, NJ, USA, September 16-17. LNCS, vol. 6551, pp. 1–24 (2010)
9. Hull, R., Damaggio, E., Masellis, R.D., Fournier, F., et al.: Business artifacts with guard-stage-milestone lifecycles: managing artifact interactions with conditions and events. In: 5th ACM on DEBS, New York, NY, USA, July 11-15. pp. 51–62 (2011)
10. de Leoni, M., Dumas, M., García-Bañuelos, L.: Discovering branching conditions from business process execution logs. In: 16th FASE, Rome, Italy, March 16-24. LNCS, vol. 7793, pp. 114–129 (2013)
11. Lu, X., Nagelkerke, M., van de Wiel, D., Fahland, D.: Discovering interacting artifacts from ERP systems. IEEE Trans. Serv. Comput. **8**, 861–873 (2015)
12. Moctar M'Baba, L., Assy, N., Sellami, M., Gaaloul, W., Nanne, M.F.: Extracting artifact-centric event logs from blockchain applications. In: IEEE ICSC, SCC, Barcelona, Spain, July 10-16. pp. 274–283. IEEE (2022)
13. Murtagh, F.: A survey of recent advances in hierarchical clustering algorithms. The computer journal **26**, 354–359 (1983)
14. Nguyen, H., Dumas, M., ter Hofstede, A.H.M., Rosa, M.L., et al.: Stage-based discovery of business process models from event logs. Inf. Syst. **84**, 214–237 (2019)
15. Palacio-Niño, J., Berzal, F.: Evaluation metrics for unsupervised learning algorithms. CoRR **abs/1905.05667** (2019)
16. Popova, V., Dumas, M.: From petri nets to guard-stage-milestone models. In: BPM Workshops, Tallinn, Estonia, September 3. LNBIP, vol. 132, pp. 340–351 (2012)
17. Popova, V., Dumas, M.: Discovering unbounded synchronization conditions in artifact-centric process models. In: BPM Workshops, Beijing, China, August 26. LNBIP, vol. 171, pp. 28–40 (2013)
18. Popova, V., Fahland, D., Dumas, M.: Artifact lifecycle discovery. CoRR **abs/1303.2554** (2013)
19. Quinlan, J.R.: Induction of decision trees. Mach. Learn. **1**, 81–106 (1986)