Mining Activity Clusters From Low-level Event Logs

Christian W. Günther and Wil M.P. van der Aalst

Department of Technology Management, Eindhoven University of Technology
P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands
{c.w.gunther, w.m.p.v.d.aalst}@tm.tue.nl

Abstract. Process mining techniques have proven to be a valuable tool for analyzing the execution of business processes. They rely on logs that identify events at an activity level, i.e., most process mining techniques assume that the information system explicitly supports the notion of activities/tasks. This is often not the case and only low-level events are being supported and logged. For example, users may provide different pieces of data which together constitute a single activity. The technique introduced in this paper uses clustering algorithms to derive activity logs from lower-level data modification logs, as produced by virtually every information system. This approach was implemented in the context of the ProM framework and its goal is to widen the scope of processes that can be analyzed using existing process mining techniques.

1 Introduction

Business Process Management (BPM) technology [2, 16, 17, 19] has become an integral part of the IT infrastructure of modern businesses, most notably in knowledge-intensive fields (e.g. public administration or the financial sector). The potential to create digital copies of a document, allowing multiple persons to work with these simultaneously, can greatly improve the performance of process execution. *Process-Aware Information Systems* (PAISs) make it possible to control this distribution of work, based on a process definition. This process definition contains *activities*, i.e. self-contained partitions of the work to be done, between which causal relationships are defined. Once a process is started, the PAIS creates a process instance, assigns activated tasks to appropriate users, and controls the activation of subsequent tasks based on the process definition.

In contrast to an industrial process, where one can literally follow the product through all stages of its manufacturing process, monitoring and evaluating the execution of an informational process is a complex endeavor. In recent years, several techniques to this end have been proposed, subsumed under the term *Business Process Intelligence* (BPI).

A subset of these methods belongs to the field of *process mining* [4, 8, 5], which deals with the analysis of process execution logs. From these logs, process mining techniques aim to derive knowledge in several *dimensions*, including the process model applied, the organizational structure, and the social network. This information about the actual situation can then be compared to the intended process definition and the organizational guidelines. Discovered discrepancies between the process definition and the real situation can be used to better align process and organization, to remedy performance bottlenecks and to enforce security guidelines.

One of the drawbacks current Process Mining techniques [6, 13, 3, 4] face is that their requirements towards execution logs to be mined are satisfied only by PAIS. These are designed around the idea of having a defined process model, comprising of atomic activities. However, a large number of BPM systems exists which do not enforce a strictly prescribed process model, but rather provide users with a generic environment to collectively *manipulate and exchange information*. An example of such unstructured BPM systems are most Enterprise Resource Planning (ERP) systems (e.g. SAP R/3).

Although some ERP systems include a workflow management component which enables the implementation of a PAIS, its use is not enforced. Thus, ERP logs typically do not contain events referring to the execution of activities, rather they refer to *low-level modifications of atomic data objects* in the system. However, this does not imply that the concept of an activity does not exist in an ERP system. Although they are not explicitly defined and logged, users tend to modify semantically related data within one logical step. The *form* metaphor, which is implemented in almost every information system, allowing users to fill in a number of data fields combined on one screen, strongly supports this paradigm.

It is highly desirable, and often necessary, to make these tacit activity patterns explicit, in order to analyse aspects of process execution on a more abstract level. The technique presented in this article uses clustering techniques to discover these "implicit" activities in low-level logs. It is based on the notion of *proximity* between low-level events, deriving from that measure a semantical relationship between modified data objects.

The fundamental idea of this approach is that an activity is a recurring pattern of low-level events, modifying the same set of data types in a system. Clusters of data modification events from an initial scan are subsequently aggregated to higher-level clusters, which are finally filtered to yield the patterns most likely to represent activities. By abstracting from the very low level of data modification logs to the activity level, this technique has the potential to substantially extend the field of systems that can be analyzed with Process Mining techniques.

This paper is organized as follows. The following section investigates the semantic and structural relationships between high- and low-level logs, followed by the introduction of a lifecycle model for data objects in Section 3. Section 4 introduces the basic notion of an event log and describes the pieces of information typically contained in a log. The subsequent three sections describe the three stages of the actual algorithm: the *initial scan*, the *aggregation pass* and the *final selection* of the most fit candidates. Section 8 describes implementation-specific aspects, followed by an overview on applications in Section 9. The article closes by linking to related work in Section 10, and a concluding discussion in Section 11.

2 Relationships Between High- and Low-level Logs

A business process describes the handling of a given case, where the case is the primary object to be manufactured, e.g. a car insurance claim. From a *top-down* perspective this can be interpreted as breaking down the high-level goal (“Assess whether we pay for the damage caused”) into a set of lower-level sub-goals (e.g., “Review client history”, “Investigate the accident“), and establishing suitable ordering relations between them. Executing the *tasks*, i.e. accomplishing these sub-goals, in the specified order will then yield the accomplishment of the high-level goal, i.e. finishing the case.

Process modeling can, however, also be interpreted as a *bottom-up* design process when approached from a different perspective. Especially in knowledge-intensive domains, e.g. administration, a business process is essentially a structured way of creating, modifying and processing large amounts of data. In the above example, a large set of atomic data objects (e.g., name and address of the client, and a detailed description of the accident) need to be collected. By successively processing and combining these objects into higher-level data (e.g., whether the accident has been caused by the client), the desired end product (e.g. “$2,500 will be paid”) is eventually created.

The relation between the manufactured low-level data types and the higher-level tasks (i.e., sub-goals) of the business process can be described as follows: *A task groups a set of data-modifying operations that are closely semantically related, and which typically occur together*. For example, the task “record client data” will typically always consist of the modifications of data types “Name”, “Street”, “Customer number”, and so forth.

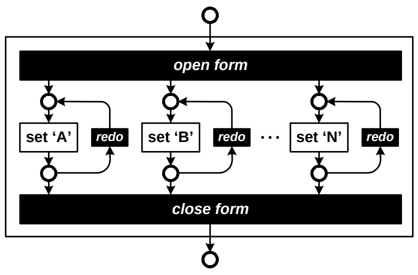


Fig. 1. Activity data modification model

Most user-centered PAISs use the concept of forms, which are interface masks containing an input field for each data type required. When a user executes a task he will be presented with the respective form, on which he can fill out the information for this task. While the ordering of tasks is controlled and enforced by the PAIS, the order in which to provide the low-level data is usually up to the user (i.e., he can freely navigate within the form). This concept is shown in Figure 1 in terms of a Petri net. Notice that, in addition to the concurrency of data modification operations, there is also the possibility that data objects are (potentially repeatedly) deleted and set to a different value within a task (e.g., when correcting a misspelled name).

This relationship of the higher-level task incorporating lower-level data modification operations on a semantic level is likewise exhibited on a temporal level, as shown in Figure 2. Data modification events occur within the realm of their high-level tasks,

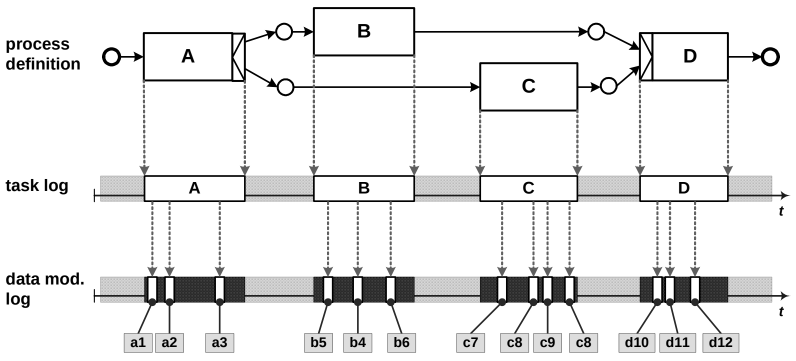


Fig. 2. Relations between multiple levels of abstraction

i.e. between their start and end events. In a system like FLOW*er* (Pallas Athena, [11]), a case handling system [1, 7] which can create logs on both the task and the data object level, this obvious property can be easily observed. Each execution of a task leaves a distinctive trace in the data modification log, i.e. a typical set of events referring to modifications of the set of data types accessible in this task. This typical set of data types modified by an activity shall be referred to as the activity's *footprint*.

Table 1. Example excerpt of a low-level event log

#	Timestamp	PID	Event	Originator
423	12.07.05;14:24:03	37	cust_firstname_p37	Brian
424	12.07.05;14:26:22	37	cust_lastname_p37	Brian
425	12.07.05;14:26:33	34	complaint_customerid_p34	Stewie
426	12.07.05;14:26:55	37	cust_street_p37	Brian
427	12.07.05;14:27:20	34	complaint_orderid_p34	Stewie
428	12.07.05;14:27:52	37	cust_city_p37	Brian
429	12.07.05;14:28:23	37	cust_zip_p37	Brian
430	12.07.05;14:28:44	34	complaint_value_p34	Stewie
431	12.07.05;14:29:34	34	complaint_status_p34	Stewie
432	12.07.05;14:29:34	34	complaint_handlerid_p34	Stewie
433	12.07.05;15:44:06	38	cust_lastname_p38	Brian
434	12.07.05;15:44:33	38	cust_firstname_p38	Brian
435	12.07.05;15:45:52	38	cust_street_p38	Brian
436	12.07.05;15:47:04	34	service_date_p38	Peter
437	12.07.05;15:47:15	38	cust_zip_p38	Brian
438	12.07.05;15:48:34	38	cust_city_p38	Brian
439	12.07.05;15:55:36	34	service_technid_p38	Peter
440	12.07.05;16:01:01	34	service_sysid_p38	Peter
441	12.07.05;16:03:22	34	service_result_p38	Peter

An example excerpt of a low-level log is given in Table 1. Each line represents one event, with its sequence number, time of occurrence, process instance id, name, and originator[1] information. Judging from a first glimpse, the log appears to be confusing and scattered. A great number of cryptic events occur during a relatively short time, and their relation is not clear.

After taking a closer look at Table 1 there are several hints about the relationships between the single events. It appears to be a commonplace pattern that events within the same process instance, triggered by the same originator, occur within a relatively small time frame. For example, it seems that Stewie has triggered all events whose names start with "complaint", and these have all occurred within roughly three minutes. Earlier in the log, there are some events triggered by Brian, each starting with "cust" and also exhibiting similar characteristics (limited time span, same process id), partly interleaved with Stewie's events. Notice that the same set of event names occurs again later on, also triggered by Brian but in a different order.

From all we have seen, this looks very much like a low-level log which has resulted from the execution of higher-level processes. Brian's and Stewie's events at the beginning of the log seem to have resulted from their concurrent execution of two higher-level activities. In order to analyze what has been going on in a more abstract fashion, and to rediscover the underlying, higher-level process model, it is crucial to reconstruct task-level events from these low-level patterns.

The subsequent two sections discuss the specific properties of data objects in a business process, and their associated low-level logs. Based on these properties and their typical relations to higher-level logs and processes, an algorithm for the discovery of activity patterns is introduced in Sections 5, 6, and 7. After introducing the algorithm we describe the implementation in the ProM framework [14] and discuss possible applications.

3 Data Object Lifecycle

The incentive for using a PAIS is usually that in an organization a standard set of business processes exist, which are repeatedly executed in a large volume. Therefore, *process definitions* are created which describe all the potential paths that the execution of one *type* of process can follow. When such a process is being executed, a *process instance* is created, which binds the abstract process definition to an actual set of resources and data for the specific case.

This relationship is depicted in Figure 3. The `<<instanceof>>` relation, which describes the connection between the type and instance level, is also apparent for the elements of a process definition: Abstract tasks defined in the process definition, once instantiated and bound to an executing resource, spawn activities. Data types, like the fields of a customer address defined in a process definition, are also instantiated to objects for each process instance.

The *data types* defined in a process definition are not actual data objects, but rather templates that describe the potential values of their instantiations (e.g., in programming

[1] The originator refers to the person, or resource, having caused the respective event (e.g. an employee filling out a form).

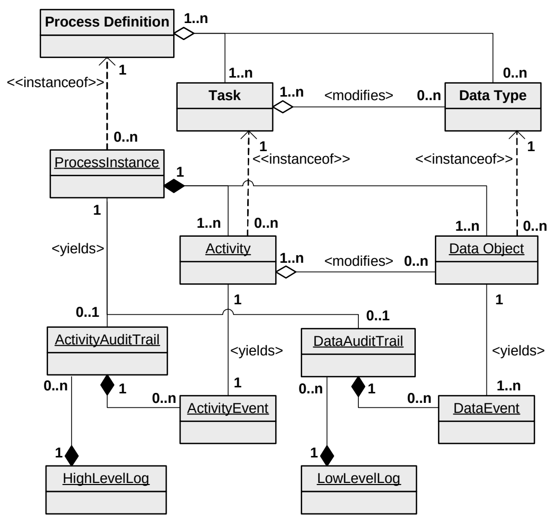


Fig. 3. Type-instance relationships on process, task and data level (UML 2.0 class diagram)

languages, defining a variable as “Integer” does not describe its value, but rather its nature and range). When a process is started, for each of these data types corresponding *data objects* are instantiated, which are subsequently set to actual values during execution.

The UML diagram shown in Figure 3 further describes the relationship between the task and data dimensions of a process, and their respective log events. Activities, being singular executions of tasks, always refer to exactly one task-level event in the log. Data instances, on the other hand, can be modified repeatedly, with transitions in their lifecycle resulting in data-level log events. Both on the task and data level, audit trails group all events having occurred within one process instance. Logs can be interpreted as containers for audit trails: they can contain multiple audit trails for the same process or even audit trails for multiple processes, as they are often capturing all events occurring in one information system.

The lifecycle of data objects during the execution of a process instance is described in the state-transition system shown in Figure 4.

Each transition in this model, i.e. *define* (DEF), *delete* (DEL), *update* (UPD), *rollback* (RBK), and *confirm* (CNF), is associated with a respective *event* occurring within the system. These events, referring to either a change in value or validity of the data object, are the elements of a low-level log.

The subsequent section introduces these low-level logs, and the pieces of information contained in each event, in more detail.

4 Low-level Logs

Log files were originally introduced as a means for administrators to monitor system operation and to trace back errors. These logs are essentially sequences of significant

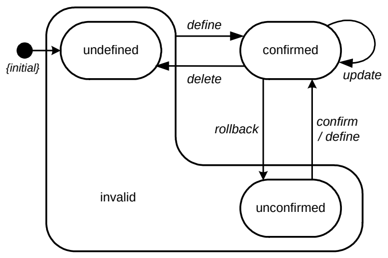


Fig. 4. Generic state-transition system for data objects

events which have occurred in the system so far, listed in chronological order. The notion of an event log can be formalized as follows.

Definition 1 (Sequence, non-repeating sequence). *Let A be a set. A finite sequence on A is a mapping $\sigma \in \{1, \ldots, n\} \to A$ where n is the length of the sequence. $len(\sigma) = n$ denotes the length and $\sigma(i)$ the i^{th} element (for $1 \leq i \leq len(\sigma)$). ϵ denotes the empty sequence, i.e., $len(\epsilon) = 0$ and $dom(\epsilon) = \emptyset$.*

A sequence $\sigma \in \{1, \ldots, n\} \to A$ can be denoted by $\langle\, \sigma_1, \sigma_2, \ldots, \sigma_n \,\rangle$ where $\sigma_i = \sigma(i)$ for $1 \leq i \leq n$. $set(\sigma) = \{\sigma(i) \mid i \in dom(\sigma)\}$ is the set of all elements. As a short cut we can write $a \in \sigma$ to denote $a \in set(\sigma)$ for some $a \in A$.

A sequence $\sigma \in \{1, \ldots, n\} \to A$ is a non-repeating sequence if and only if $\forall_{1 \leq i < j \leq n} \sigma(i) \neq \sigma(j)$.

For any non-repeating sequence $\sigma \in \{1, \ldots, n\} \to A$ and $a \in A$, $pos(\sigma, a)$ provides the sequence number of a in σ, i.e., $\sigma(pos(\sigma, a)) = a$.

In a non-repeating sequence the elements are totally ordered, i.e., for any $a_1, a_2 \in \sigma$: $a_1 <_\sigma a_2$ if and only if $pos(\sigma, a_1) < pos(\sigma, a_2)$.

Definition 2 (Event, log). *Let E be a* set of log events. *l is a* log *over E if and only if l is a non-repeating sequence on E.*

As has been stated in the above definition, the fundamental property of a log is the requirement of being a strictly ordered sequence of events. All events contained in a log are unique, which is expressed by the sequence being non-repeating. What differentiates several kinds of logs is mainly the typical set of event attributes, i.e. data which is provided for every event in the particular log. In this article we focus on low-level logs, in particular fine-grained logs which describe the ordered modification of a set of data objects in a distributed environment (i.e. executed by multiple resources or persons). In these logs, the following attributes can typically be found for each event.

Definition 3 (Attributes of an event). *Let $e \in E$ be an event. An event may have the following attributes:*

– *$tp \in E \not\to \{DEF, DEL, UPD, RBK, CNF\}$ provides the event type,*[2]

[2] Note that $f \in A \not\to B$ is a partial function, i.e., $dom(f) \subseteq A$. If $a \in A \setminus dom(f)$, then $f(a) = \perp$ denotes that a is not in the domain. For any function $f : f(\perp) = \perp$.

– *$o \in E \not\rightarrow O$ provides the originator of each event, where O is the set of originators (i.e., the set of possible resources),*
– *$ts \in E \not\rightarrow \mathbb{R}_0^+$ provides the timestamp of each event,*
– *$p \in E \not\rightarrow P$ provides the process instance of each event, where P is the set of process instances,*
– *$dt \in E \not\rightarrow D^t$ provides the data type of each event, where D^t is the set of data types, and*
– *$di \in E \not\rightarrow D^i$ provides the data object of each event, where D^i is the set of data objects.*

A low-level data modification log describes the lifecycle of data objects in the modified set, with events corresponding to transitions in the lifecycle model presented in Section 3. The type of state transition (e.g. DEF) an event refers to is stored in the attribute *event type*. A PAIS, which is supposed to be the source of logs in question, typically allows for multiple resources (e.g. workers) to be involved in handling one process instance. Each modification of a data object was initiated by a particular resource, whose identification is stored in the *originator* attribute. The *timestamp* attribute stores the exact time at which an event occurred. Every event occurs within the realm of one specific case, referenced by the *process instance* attribute.

In the example log given in Table 1, each row corresponds to one event e. The value in the first column denotes the event sequence number, i.e. $pos(\sigma, e)$. The timestamp given in the second column would correspond to $ts(e)$, the process ID in column three corresponds to $p(e)$, and the originator in the fifth column denotes the value for $o(e)$. From looking at the values in the fourth column, it can be seen that they correspond to data objects (i.e. instances), as the given strings have the process ID appended. Thus, the values in this column correspond to $di(e)$.

Note that the functions tp, o, dt, di, p and ts are partial. This indicates that not all attributes need to be present, e.g., $ts(e) = \perp$ if no timestamp is given.

Definition 4 (Mapping between data type and instance attribute). *Let D^t be a set of data types and D^i be the set of data objects. $c \in D^i \rightarrow D^t$ maps each instance on its corresponding type. Note that for every event $e \in E$ the following should hold: $dt(e) \neq \perp \wedge di(e) \neq \perp \implies dt(e) = c(di(e))$.*

The data objects whose modifications are recorded in logs are inherently *instances*. Each event holds the identification of the modified data object in the attribute *data object*. However, in order to compare and relate events of different process instances, it is also necessary to know the *type* of the modified data object, which is stored in attribute *data type*. The distinction between data object and type is mainly important on a conceptual level, which is why most systems do not explicitly record both. In a system recording only the data type identifier, several instances can be distinguished by the respective process instance of the event in question. Other systems will only record data object identifiers, such as “cname_p12” and “cname_p65”. In most of these cases one can relate instances to their given types without too much effort (in the given example, it can be assumed that both identifiers refer to instances of the same data type “cname”).

5 Modification Cluster Scanning

As discussed in Section 2, the hypothesis is that for every low-level log there exists an associated high-level process, with its enactment having resulted in this log. It is, generally speaking, rather irrelevant if this higher-level process actually exists in an explicit form. Procedural execution of activities does not necessarily result from an explicit process prescription. It can just as well result from practical restrictions or guidelines enforcing a certain process, from standard behavior, or just daily routine.

The basic assumption of modification cluster scanning is that, based on the fundamental relations between the distinct levels, it is possible to re-discover higher-level activity executions in a low-level log. In order to determine which low-level events constitute one execution of the same activity, this technique makes use of the following hypotheses:

– Each activity and its resulting low-level events occur within the *same process instance*.
– All low-level events having resulted from the same, higher-level activity have the *same originator*.
– Each execution of an activity typically involves modification of the *same set of data types*, i.e. the resulting footprints are (largely) identical.
– The execution of an activity takes place in a comparably *short time span*. Thus, all resulting low-level events occur in each other's *proximity*.
– In hierarchical transactional systems, all low-level events of a higher-level activity have the *same event type*. That is, if the higher-level activity was a roll-back task then all resulting lower-level events should also be of type “roll-back”. Note that this is an optional requirement which should only be imposed in systems with a hierarchical transactional semantics.

These assumptions appear to be natural and valid in almost any given setting. They directly follow the common perception of an activity: being performed in a process instance context, being performed by one person, involving a fixed set of artifacts, having occurred during a limited amount of time. Notice further that in a PAIS, the time a case spends waiting to be executed usually exceeds the time spent on actually executing activities. Thus, it can be safely assumed that in a typical log there are large voids between comparably short bursts of activity.

In order to use proximity as a means to decide whether low-level events belong to a certain activity, it is necessary to define a suitable metric for it.

Definition 5 (Proximity Function). *Let l be a* log *over E. A function p is a proximity function if $p \in E \times E \to \mathbb{R}_0^+$*

Depending on the information contained in a specific log, the metric for proximity can rely on different pieces of information; i.e., it is possible to use different proximity functions, either based on real time (i.e. using timestamp information), or using a logical clock based on the discrete distance between events:

– $p(e_1, e_2) = |ts(e_1) - ts(e_2)|$ (assuming $e_1, e_2 \in dom(ts)$),

– $p(e_1, e_2) = |pos(\sigma, e_1) - pos(\sigma, e_2)|$.

Based on the definition of a proximity function, the decision whether two given events are in each other's proximity can be made based on an appropriately chosen parameter p_{range}, denoting the maximal proximity of two related events.

Definition 6 (Proximity). *Let l be a* log *over E, p some proximity function ($p \in E \times E \to \mathbb{R}_0^+$), and $p_{range} \in \mathbb{R}_0^+$ the maximum proximity. Two events $e_1, e_2 \in l$ are in each other's proximity if $p(e_1, e_2) \leq p_{range}$.*

The first pass of the algorithm scans the log for an initial set of clusters, each composed of low-level events which are likely to have resulted from an activity execution. It is assumed that the majority of activities have been executed within p_{range}, e.g. within a certain time span, so that all low-level events of each activity are in each others' proximity. The basic idea of the algorithm is to create an event cluster for each event e in the log, where the *reference event* e is the first event in the cluster. All clusters *having occurred after* e, and which are *still in* e*'s proximity*, are also potentially contained in the cluster.

Thus, the definition of proximity is an example of a *clustering function*, grouping events from a log into potentially related subsets. In fact, the requirement of proximity is the most integral part of a clustering function used for activity mining, as it limits the set of potentially related events to a smaller neighborhood.

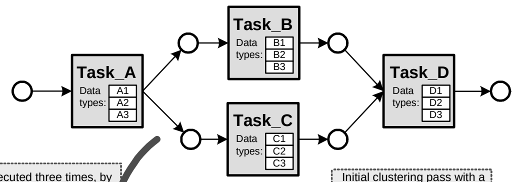
A scan window of length p_{range} is aligned to the first event e_1 in the log, i.e. the scan window contains event e_1 plus all subsequent events in the proximity of e_1. From the set of events visible in the scan window, a relevant subset can be selected by further requirements of the employed clustering function. The result of this operation is a cluster of potentially related low-level events.

Repeating this procedure for every event in the log, the result is a set of initial event clusters, exactly one for each event in the log. Each cluster has a different reference event e, which is the first[3] element of the cluster, and which is used to define the requirements of this cluster.

Figure 5 shows a step-by-step procedure of applying the algorithm to an example log. The original low-level log is shown in the leftmost table. Assuming a scan window size p_{range} of 10 minutes, the cluster for event 1 would comprise events 1, 2, 3, and 4, based on their proximity to event 1. For event number 2, the cluster contains events 2, 3, 4, 5, 6 and 7, and so on. Note that Figure 5 lists clusters which are smaller than the examples given here, which is due to the fact that the initial clustering function can contain further requirements extending mere proximity.

Relying solely on proximity in scanning clusters does not fully take advantage of the hypothesized properties of low-level events stated above. In order to filter out irrelevant events from this set, the cluster can be restricted further, e.g. to enforce equality of originator or process id within a cluster. The combination of all requirements to be enforced is specified with a suitable *clustering function*.

[3] Referring to the order of appearance in the log.



Executed three times, by three resources.

Initial clustering pass with a scan window size of **10 minutes,** enforcing **equality of originator** and **process instance**.

Aggregation of the initial clusters, using the **tolerant aggregation** method.

Low-level log:

#	Time-stamp	PID	Data Inst.	Data Type	Origin-ator
1	14:00:00	1	A1_1	A1	Barney
2	14:06:22	1	A2_1	A2	Barney
3	14:06:28	2	A1_2	A1	Homer
4	14:08:55	1	A3_1	A3	Barney
5	14:10:12	2	A3_2	A3	Homer
6	14:14:23	3	A1_3	A1	Lenny
7	14:15:47	2	A2_2	A2	Homer
8	14:18:14	3	A3_3	A3	Lenny
9	14:23:57	3	A2_3	A2	Lenny
10	14:24:04	3	C1_3	C1	Homer
11	14:27:22	3	C2_3	C2	Homer
12	14:32:40	1	C2_1	C2	Lenny
13	14:32:43	3	C3_3	C3	Homer
14	14:39:45	3	B1_3	B1	Homer
15	14:40:10	1	C1_1	C1	Lenny
16	14:42:04	3	B2_3	B2	Homer
17	14:42:10	1	C3_1	C3	Lenny
18	14:43:02	2	B1_2	B1	Barney
19	14:44:43	3	B3_3	B3	Homer
20	14:48:50	2	C1_2	C1	Homer
21	14:49:00	2	B2_2	B2	Barney
22	14:50:23	2	B3_2	B3	Barney
23	14:52:55	2	B2_2	B2	Barney
24	14:53:45	2	C2_2	C2	Homer
25	14:54:02	2	C3_2	C3	Homer
26	14:57:14	1	C3_1	C3	Lenny
27	14:58:00	2	C2_2	C2	Homer
28	15:04:43	1	B1_1	B1	Homer
29	15:10:22	3	D1_3	D1	Lenny
30	15:12:19	1	B2_1	B2	Homer
31	15:13:10	3	D3_3	D3	Lenny
32	15:14:42	1	B3_1	B3	Homer
33	15:22:05	1	D1_1	D1	Homer
34	15:28:02	3	D2_3	D2	Lenny
35	15:28:04	2	D1_2	D1	Barney
36	15:28:05	1	D2_1	D2	Homer
37	15:31:45	1	D3_1	D3	Homer
38	15:35:15	2	D3_2	D3	Barney
39	15:36:00	1	D3_1	D3	Homer
40	15:37:22	2	D2_2	D2	Barney

Initial clusters:

#	Events in cluster	Footprint	Conflicting
1	1, 2, 4	A1, A2, A3	2, 4
2	2, 4	A2, A3	1, 4
3	3, 5, 7	A1, A2, A3	5, 7
4	4	A3	1, 2
5	5, 7	A3, A2	3, 7
6	6, 8, 9	A1, A3, A2	8, 9
7	7, 10	A2, C1	3, 5, 10
8	8, 9	A3, A2	6, 9
9	9, 12	A2, C2	6, 8, 12
10	10, 11, 13	C1, C2, C3	7, 11, 12
11	11, 13	C2, C3	10, 13
12	12, 15, 17	C2, C1, C3	9, 15, 17
13	13, 14, 16	C3, B1, B2	10, 11, 14, 16
14	14, 16, 19, 20	B1, B2, B3, C1	13, 16, 19, 20
15	15, 17	C1, C3	12, 17
16	16, 19, 20	B2, B3, C1	13, 14, 19, 20
17	17	C3	12, 15
18	18, 21, 22, 23	B1, B2, B3	21, 22, 23
19	19, 20, 24, 25	B3, C1	14,16,20,24,25
20	20, 24, 25, 27	C1, C2, C3	14,16,19,24,25,27
21	21, 22, 23	B2, B3	18, 22, 23
22	22, 23	B3, B2	18, 21, 23
23	23	B2	18, 21, 22
24	24, 25, 27	C2, C3	19, 20, 25, 27
25	25, 27	C3, C2	20, 24, 27
26	26	C3	--
27	27, 28	C2, B1	20, 24, 25, 28
28	28, 30, 32	B1, B2, B3	27, 30, 32
29	29, 31	D1, D3	31
30	30, 32, 33	B2, B3, D1	28, 32, 33
31	31	D3	29
32	32, 33	B3, D1	28, 30, 33
33	33, 36, 37	D1, D2, D3	30, 32, 36, 37
34	34	D2	--
35	35, 38, 40	D1, D3, D2	38, 40
36	36, 37, 39	D2, D3	33, 37, 39
37	37, 39	D3	33, 36, 39
38	38, 40	D3, D2	35, 40
39	39	D3	36, 37
40	40	D2	35, 38

Aggregated clusters:

#	Footprint	Clusters	Conflicting	val (a=0.5)
1	A1, A2, ,A3	1, 3, 6	2, 3, 4, 5	3.0
2	A2, A3	2, 5, 8	1, 3, 4, 5	● 2.5
3	A3	4	1, 2	● 1.0
4	A2, C1	7	1, 2, 6	● 1.5
5	A2, C2	9	1, 2, 6	● 1.5
6	C1, C2, C3	10, 12, 20	4,5,7,9,10, 11,12,14,17	3.0
7	C2, C3	11, 24, 25	6, 8, 14, 17	● 2.5
8	B1, B2, C3	13	6, 7, 9, 11	2.0
9	B1, B2, B3, C1	14	6, 8, 11, 14	● 2.5
10	C1, C3	15	6, 12	● 1.5
11	B2, B3, C1	16	6, 8, 9, 14	● 2.0
12	C3	17, 26	6, 10	● 1.5
13	B1, B2, B3	18, 28	15, 16, 17, 19, 21	2.5
14	B3, C1	19	6, 7, 9, 11	● 1.5
15	B2, B3	21, 22	13, 16	● 2.0
16	B2	23	13, 15	● 1.0
17	B1, C2	27	6, 7, 13	● 1.5
18	D1, D3	29	20	1.5
19	B2, B3, D1	30	11, 21, 22	● 2.0
20	D3	31, 37, 39	18, 22, 24	● 2.0
21	B3, D1	32	13, 19, 22	● 1.5
22	D1, D2, D3	33, 35	19, 20, 21, 23, 24	2.5
23	D2	34, 40	22, 24	● 1.5
24	D2, D3	36, 38	20, 22, 23	● 2.0

● = victim of conflict resolution.

Conflict resolution, using **a=0.5**, yields the minimal conflict-free set. When ordered **by their value**, the desired clusters appear at the top of the list.

Minimal conflict-free set:

#	Footprint	Clusters	val (a=0.5)
1	A1, A2, ,A3	1, 3, 6	3.0
6	C1, C2, C3	10, 12, 20	3.0
13	B1, B2, B3	18, 28	2.5
22	D1, D2, D3	33, 35	2.5
8	B1, B2, C3	13	2.0
18	D1, D3	29	1.5

$\left.\begin{array}{l}1\\6\\13\\22\end{array}\right\}$ desired result

Fig. 5. All phases of the algorithm, applied to an example log

Definition 7 (Cluster). *Let l be a* log *over E. A cluster C is a set of events in l, i.e., $C \subseteq set(l)$. A clustering function is a function cf mapping l onto a set of clusters, i.e., $cf(l) \subseteq \mathbb{P}(set(l))$.*[4]

The clustering function to be used in scanning the initial set of clusters can be tailored to the specific application and type of log. Examples of such clustering functions include:

– $cf(l) = \{prox(e) \mid e \in l\}$, where $prox(e_1) = \{e_2 \in E \mid |ts(e_1) - ts(e_2)| \leq p_{range} \wedge pos(l, e_1) < pos(l, e_2)\}$
  (all events having occurred within a maximal time span of p_{range} after the reference event $e1$; introducing the requirement of proximity),
– $cf(l) = \{prox(e) \mid e \in l\}$, where $prox(e_1) = \{e_2 \in E \mid |pos(l, e_1) - pos(l, e_2)| \leq p_{range} \wedge tp(e_1) = tp(e_2) \wedge pos(l, e_1) < pos(l, e_2)\}$
  (logical clock based proximity limit; limits the events contained to those of the same type as $e1$; for logs from hierarchical transactional systems),
– $cf(l) = \{prox(e) \mid e \in l\}$, where $prox(e_1) = \{e_2 \in E \mid |ts(e_1) - ts(e_2)| \leq p_{range} \wedge o(e_1) = o(e_2) \wedge pos(l, e_1) < pos(l, e_2)\}$
  (timestamp-based proximity; replaces the requirement for event type equality with requiring the same originator),
– $cf(l) = \{prox(e) \mid e \in l\}$, where $prox(e_1) = \{e_2 \in E \mid |ts(e_1) - ts(e_2)| \leq p_{range} \wedge pos(l, e_1) < pos(l, e_2) \wedge o(e_1) = o(e_2) \wedge p(e_1) = p(e_2)\}$
  (timestamp-based proximity limit, enforcing the same originator and process instance for all events in one cluster),
– $cf(l) = \{\{e\} \mid e \in l\}$
  (every cluster is a singleton),
– etc.

In the example shown in Figure 5 the third proximity function listed above has been chosen (time-based proximity limit of 10 minutes, equality of originator enforced within clusters). It cuts the set of events contained in each scan window down further, including only those with an originator and process ID identical to the reference event. Thus, the cluster aligned to event 1 contains events 1, 2 and 4; the 2nd cluster contains events 2 and 4. These clusters, created for each event in the log, constitute the initial set of clusters, which is shown in its entirety in the middle table in Figure 5.

The optimal scan window size p_{range} is the maximal proximity between the first and last event of all activities recorded in the log. If p_{range} is set too small, activities with a longer duration (or a larger number of low-level events) cannot be captured completely. On the other hand, a too large scan window will lead to events from distinct activities being comprised in one cluster (when the scan window size exceeds the mean idle time a process spends between activities). The clustering function has to be chosen with respect to the system having produced the log under consideration (e.g., enforcing uniform event types within clusters will only work in transactional systems, otherwise it will distort the results).

[4] $\mathbb{P}(A)$ is the powerset of A, i.e., $\mathbb{P}(A) = \{B \mid B \subseteq A\}$.

6 Cluster Aggregation

The initial set of clusters contains a great share of clusters which do not correspond to actual activity executions. Moving the scan window over an actual cluster that has resulted from one activity, which e.g. comprises six lower-level events, will yield at least six scanned clusters (as the scan window is being moved event-wise). One of these clusters is correct, i.e. where the scan window covered all events derived from the activity. The remaining clusters have captured only a subset of the involved events. The latter kind of clusters, having resulted from an incorrect scan window position, shall be denoted as *fragmentary clusters*.

Another problem is the sheer amount of clusters yielded by the initial scan pass, as each event in the log will result in exactly one cluster. It is highly desirable to group similar clusters into *meta-clusters*, which constitutes the second pass of the presented algorithm. With each occurrence of a similar cluster found in the initial set, the probability that this cluster corresponds to the execution of an activity increases.

In order to group, or *aggregate*, the initial clusters, it is necessary to have a means for determining their similarity. The essence of what defines a cluster is the *footprint*, i.e. the set of data objects modified by its contained events.

Definition 8 (Footprint). *Let l be a* log *over E, C a cluster of l, and for any $e \in l$: $dt(e) \neq \perp$ (i.e., a data type is defined for each event in the log). $fp(C) = \{dt(e) \mid e \in C\}$.*

Using the footprint for evaluating the similarity of two clusters is consistent with the activity model presented in Section 2. Multiple events having resulted from setting and clearing one data object repeatedly do not affect the footprint, and thus equality between two different executions of the same activity is being preserved.

The aggregation pass combines sets of initial clusters with a compatible footprint to meta-clusters, which are identified by this very footprint. It can be characterized as grouping syntactically related patterns, assuming that these are related on a semantic level as well. From a set of initial clusters, the aggregation pass results in a set of aggregated meta-clusters. The decision whether two footprints are compatible, i.e. whether the associated clusters are to be aggregated, is performed by an *aggregation method*.

Definition 9 (Aggregation Method). *Let l be a* log *over E and $cf(l)$ be a clustering function over this log. An* aggregation method *$ag(l)$ groups the clusters of $cf(l)$ to sets of similar elements, based on their footprints. $ag(l) \in \mathbb{P}(\mathbb{P}(D^t) \times \mathbb{P}(cf(l)))$*

Note that if $(F, CS) \in ag(l)$, then CS is a set of initial clusters that somehow belong together based on a set of data types F (e.g. , all clusters in CS have an identical footprint). Which specific aggregation method to choose depends, once again, largely on the specific log under consideration. Based on many experiments, the following aggregation methods have been found most suitable.

Definition 10 (Tolerant Aggregation).
$ag(l) = \{(F, CS) \in \mathbb{P}(D^t) \times \mathbb{P}(cf(l)) \mid \exists\, C_i \in CS : fp(C_i) = F \wedge CS = \{C_j \in cf(l) \mid fp(C_j) = F\}\}$

The tolerant aggregation method groups sets of clusters which have exactly the same footprint, i.e. the set of modified data types is identical for each element of a meta-cluster. This common footprint is also used to describe the resulting aggregated meta-cluster.

Definition 11 (Strict Aggregation).
$ag(l) = \{(F, CS) \in \mathbb{P}(D^t) \times \mathbb{P}(cf(l)) \mid \exists\, C_i \in CS : fp(C_i) = F \wedge CS = \{C_j \in cf(l) \mid fp(C_j) = F\} \wedge \forall\, C_k, C_l \in CS : (C_k \cap C_l = \emptyset \vee C_k = C_l)\}$

The strict aggregation method also enforces the identity of footprints among all clusters within an aggregated meta-cluster. On top of this, it further includes a conflict-resolution requirement. The issue of conflicts will be explained in detail in the next section

Definition 12 (Greedy Aggregation).
$ag(l) = \{(F, CS) \in \mathbb{P}(D^t) \times \mathbb{P}(cf(l)) \mid CS \neq \emptyset \wedge \forall\, C_i, C_j \in CS : fp(C_i) \subseteq fp(C_j) \subseteq F \vee fp(C_j) \subseteq fp(C_i) \subseteq F\}$

Although the first two aggregation methods make no effort to filter out fragmentary clusters, the resulting meta-clusters are very precise. Greedy aggregation, on the other hand, only requires the footprint of all contained clusters to share a certain subset. It is up to the specific implementation of this aggregation method to define a minimal overlap for the contained clusters' footprints, as well as the method to determine the footprint of meta-clusters. Greedy aggregation is quite sensitive to these peculiarities, however, it can improve results in very diverse, or scattered, event logs (i.e., logs which do not exhibit a lot of exactly recurring patterns).

The result of the aggregation pass in the example is shown in the upper right table of Figure 5. As the *tolerant aggregation* function has been used in this example, the set of aggregated meta-clusters still contains a considerable number of elements which do not correspond to actual data modification patterns of activities, e.g. aggregated cluster number 8, containing a mixture of low-level events resulted from the execution of tasks B and C.

Nevertheless it is clear to see that the footprint is indeed a valid criteria for comparing initial clusters. For example, initial clusters 10, 12 and 20 have been aggregated to meta-cluster 6 according to their common footprint (C1, C2, C3), although initial cluster 20 contains one superfluous event. Still, the aggregated set also contains a large number of meta-clusters which represent fragmentary clusters.

The tables for initial and aggregated clusters in Figure 5 each include a column denoted "Conflicting", which lists the (meta-) clusters the current cluster is in conflict with. The next section introduces the concept of conflicts and the last pass of the algorithm, which is able to pick from the aggregated set the subset most likely to represent actual activity execution patterns.

7 Maximal Conflict-free Set of Clusters

It is easy to see that each event in a low-level log must have resulted from exactly one higher-level activity. However, the problem of fragmentary clusters introduced in the previous section already shows that events will be included in multiple clusters. The aggregation pass can successfully decrease the number of fragmentary clusters, given the fact that a suitable aggregation method (e.g. greedy aggregation) is chosen. However, fragmentary clusters can potentially still result in multiple aggregated meta-clusters, thus distorting the result.

The very core of the problem can be defined as two clusters containing the same event. If this is the case, these clusters are *in conflict*.

Definition 13 (Conflict between Clusters). *Let l be a* log *over E and C_i, C_j two clusters of l. C_i and C_j are in conflict iff $C_i \cap C_j \neq \emptyset$.*

This definition of conflict can be extended onto aggregated meta-clusters in a straightforward manner. Two meta-clusters are in conflict, if any of their aggregated initial clusters are in conflict.

Definition 14 (Conflict between Meta-Clusters). *Let $(F_1, CS_1), (F_2, CS_2) \in \mathbb{P}(D^t) \times \mathbb{P}(cf(l))$ be two aggregated meta-clusters. (F_1, CS_1) and (F_2, CS_2) are in conflict if at least two initial clusters $C_i \in CS_1$ and $C_j \in CS_2$ are in conflict, i.e. $C_i \cap C_j \neq \emptyset$.*

The nature of conflicts between initial clusters and their propagation into the aggregated set is also illustrated in the example in Figure 5: Clusters 1 and 2 from the initial set are in conflict, because they share events 2 and 4. As initial cluster 1 is aggregated into meta-cluster 1, and initial cluster 2 becomes part of meta-cluster 2, these two resulting meta-clusters are effectively in conflict as well. This “inheritance” of conflicts from the initial clusters leads to a great number of conflicts between meta-clusters of the aggregated set.

The strict aggregation method introduced in Section 6 ensures that each aggregated meta-cluster contains a conflict-free set of initial clusters. However, this has no influence on the existence of conflicts between meta-clusters. Thus, the purpose of the third pass of the presented algorithm is to resolve these conflicts, i.e. cut down the set of aggregated clusters to a maximal, conflict-free subset.

This step is of utmost importance with respect to the intended goal, i.e. discovering activity patterns. Given the fact that the scan window size and clustering functions are suitable for the analyzed log, and that a correct aggregation method has been chosen, all footprints referring to actual activity executions must be represented in the set of aggregated meta-clusters. As these correct meta-clusters must contain all events of the log, they will be in conflict with the illegal meta-clusters (as they have to share events then). If conflict resolution is performed correctly, the maximal set of conflict-free meta-clusters should equal the set of activity occurrences in the observed process.

For each two conflicting meta-clusters, the algorithm has to select one to be promoted to the maximal conflict-free set; the other one is discarded. To this end, a conflict evaluation function is used, which is defined as follows.

Definition 15 (Conflict Evaluation). *Let $(F, CS) \in ag(l)$ be an aggregated cluster of the log l. The evaluation function $val(F, CS)$ determines the relevance of an aggregated cluster for the maximal conflict-free set of clusters, when it conflicts with another aggregated cluster. $val \in \mathbb{P}(D^t) \times \mathbb{P}(cf(l)) \rightarrow \mathbb{R}$. The weighed evaluation*

$val(F, CS) = \alpha \cdot |F| + (1 - \alpha) \cdot |CS|$ uses the factor $\alpha \in [0, 1]$ to derive the value of an aggregated cluster from its footprint size and the number of aggregated clusters.

The conflict evaluation function used takes into account both the size of the aggregated cluster's footprint, as well as the number of contained initial clusters, weighed by a parameter α. Setting α to 1 will make the algorithm choose the aggregated cluster with the largest footprint. Conversely, a value of 0 for α will give preference to the meta-cluster having aggregated the most initial clusters. As a rule of thumb, the best results have generally been achieved by choosing a value of $0.6 – 0.8$ for α, as an expressed preference for larger footprints effectively eliminates a large share of fragmentary clusters.

The derivation of the maximal conflict-free set of aggregated clusters can be defined as follows.

Definition 16 (Maximal Conflict-Free Set). *Let l be a* log *over E and $ag(l)$ be an aggregation method over this log. The function $mcf(l)$ selects the maximal subset of conflict-free aggregated clusters from $ag(l)$. $mcf(l) = \{(F_i, CS_i) \in ag(l) \mid \forall\, (F_k, CS_k) \in ag(l) :\ \forall\, C_m \in CS_i, C_n \in CS_k : C_m \cap C_n \neq \emptyset \Rightarrow val(F_i, CS_i) > val(F_k, CS_k)\}$*

Each meta-cluster contained in the maximal conflict-free set should ideally correspond to one activity in the (envisioned) higher-level process. Aggregated clusters are characterized by their footprint, describing the set of data objects modified by the respective activity. From this point on, one can use the set of initial clusters aggregated in each element of the maximal conflict-free set in order to determine the occurrences of the described activity in the log. The boundaries of each activity occurrence are specified by the timestamps (or, respectively, the log indices) of the first and last low-level event contained in the respective initial cluster.

In the example shown in Figure 5, conflict evaluation has been performed with a weight of $\alpha = 0.5$, i.e. giving equal preference to the number of contained clusters and footprint size. This evaluation is where fragmentary clusters show, because they both have a lower chance of being repeatedly represented in the log, and their footprint is usually smaller than footprints of “correct” clusters. The conflict evaluation values are shown in the rightmost column of the table representing the set of aggregated clusters. A black dot marks those meta-clusters which have been victim to conflict evaluation, i.e. they are in conflict with another meta-cluster that scores a higher value in the evaluation.

The resulting minimal conflict-free set of meta-clusters is shown in the bottom right table of Figure 5. It looks a little disappointing at first sight, as 33% of the set's elements are clearly not corresponding to actual activity executions: Meta-cluster 18 is obviously a fragmentary cluster, while meta-cluster 8 seems to cover low-level events from the overlap of two activity executions.

However, the algorithm actually performed better than one might have thought. On the one hand, the “correct” meta-clusters have scored better in the evaluation, which supports the accuracy of the evaluation function. When the maximal conflict-free set is thus sorted by the conflict evaluation value of contained meta-clusters, the accurate results will reside on the top of the scale. Note that the algorithm has discovered the correct results even without taking into account the process ID information. Further, it has to be noted that the low-level log used in this example does not satisfy our initial assumptions at all: The time spent on executing the activities is not much shorter than the time spent idle between activities within a process instance (sometimes even considerably longer!). Hence, the scan window size could not be chosen in an optimal fashion. The algorithm has thus performed remarkably well, considering this problematic log as a starting point.

Constructing a high-level log from the minimal conflict-free set is fairly straightforward. Regarding the example, one would start with the meta-cluster having scored the highest evaluation value, i.e. meta-cluster 1. The envisioned task which this cluster represents could be called “A1A2A3”, based on the footprint. As it has been mentioned, one can now use the aggregated initial clusters to reproduce the occurrences of this task, i.e. activities, as events in the higher-level log. Meta-cluster 1 contains the initial clusters 1, 3, and 6, so the initial cluster 1 would be the first occurrence of activity “A1A2A3” in the higher-level log to be created. From the first and last event contained in initial cluster 1, the start and end of this event can be derived: The activity has started at 14:00:00 (cf. event 1) and ended at 14:08:55 (cf. event 4), so this constitutes our first event in the higher-level log. This process is then repeated for each initial cluster in this meta-cluster, and then accordingly for all meta-clusters which are considered “valid” activities.

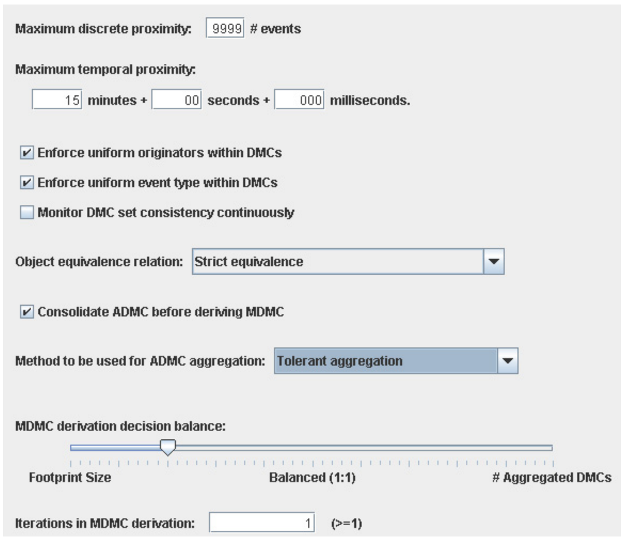
When the minimal conflict-free set stil contains “invalid” meta-clusters, it would be appropriate to set a limit for the evaluation value a meta-cluster has to score, in order to be included in the high-level log. In the given example, one would want to set this limit to $val(F, CS) \geq 2.5$. This is, however, a matter of fine-tuning, and one cannot give a general rule of thumb on how to set this limit. Given this limit, the resulting high-level log from the example would have correctly reconstructed two out of three process instances.

8 Implementation

As a proof of concept, and to allow experimentation with the concepts presented, the modification cluster scanning algorithm presented in the previous sections has been implemented as a plugin for ProM [14].

Figure 6 shows the configuration pane of the plugin. Rather than choosing a predefined clustering function, the user can configure each aspect of the initial clustering pass separately: The proximity threshold for initial clusters can be provided both in real time and as a logical number of events—the algorithm will then dynamically choose the more restrictive setting. Enforcing the equality of originator and event type fields within a cluster can be toggled independently.

A great share of systems do not log the name of the data type per event, but rather record only the data object, i.e. instance, identifier. As it is necessary to compare low-level events on a type level, the user can choose an equivalence relation for the specific log type under consideration. Equivalence relations are simple modules which can take any two data objects and decide, whether they are derived from the same data type or not. In the aggregation pass, the algorithm will rely on the chosen equivalence relation to compare footprints on a type level, rather than on an instance level.



Maximum discrete proximity: 9999 # events

Maximum temporal proximity:

15 minutes + 00 seconds + 000 milliseconds.

☑ Enforce uniform originators within DMCs

☑ Enforce uniform event type within DMCs

☐ Monitor DMC set consistency continuously

Object equivalence relation: Strict equivalence

☑ Consolidate ADMC before deriving MDMC

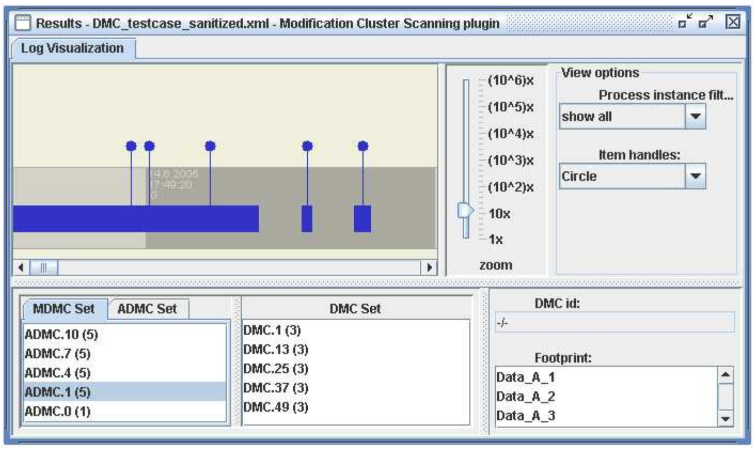
Method to be used for ADMC aggregation: Tolerant aggregation

MDMC derivation decision balance:

Iterations in MDMC derivation: 1 (>=1)

Fig. 6. Configuration pane of the Activity Miner plugin for ProM

Further options include the choice of an aggregation method to be used, and setting the weighing factor α for conflict evaluation.



Results - DMC_testcase_sanitized.xml - Modification Cluster Scanning plugin

Process instance filt...: show all

Item handles: Circle

MDMC Set	DMC Set
ADMC.10 (5)	DMC.1 (3)
ADMC.7 (5)	DMC.13 (3)
ADMC.4 (5)	DMC.25 (3)
ADMC.1 (5)	DMC.37 (3)
ADMC.0 (1)	DMC.49 (3)

DMC id: -/-

Footprint:
Data_A_1
Data_A_2
Data_A_3

Fig. 7. Result visualization of the Activity Miner plugin for ProM

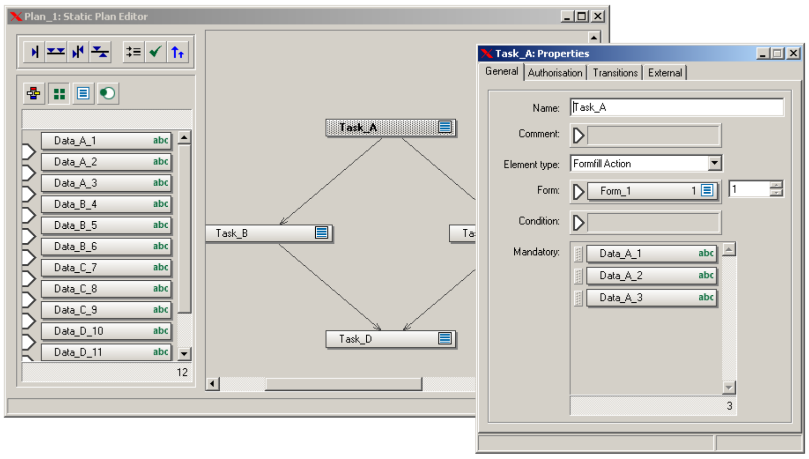
After all three passes of the algorithm have been performed successfully, the plugin will display its results, as shown in Figure 7. The lower part of the result dialog is the “Cluster Browser”: Aggregated meta-clusters are displayed in the leftmost column. The user can choose whether to display elements of the aggregated set (ADMC) or only the subset contained in the maximal conflict-free set (MDMC). When one or multiple meta-clusters are selected, the middle column displays the contained initial clusters. The rightmost column shows the current footprint, either of an aggregated or initial cluster (depending on the current selection). In Figure 7, the meta-cluster “ADMC.1” from the maximal conflict-free set has been selected, which contains five initial clusters and has a footprint of three data objects.

The upper left part of the result dialog shows the log as a linear ribbon, advancing to the right. On this pane the clusters currently selected in the browser are displayed with their temporal position in the log. When one or multiple *meta*-clusters are selected, this view will display all contained initial clusters, for they can be interpreted as occurrences. In Figure 7, the five initial clusters aggregated in “ADMC.1” are displayed on the log pane. Although their temporal positions are overlapping, the “handles” on top show their distinct, median positions.

9 Applications

The presented clustering algorithm has shown to be able to successfully rediscover a set of high-level activity patterns from a low-level log in the previous sections. This section investigates potential fields of application for the algorithm and shows that it can also be employed successfully within scopes which transcend the original intent.

In order to accurately rediscover low-level event patterns referring to activities on a higher-level process, the system having generated the low-level logs must satisfy certain requirements. As the algorithm is based on the notion of proximity, there has to be a significant gap between the duration of activity executions and the waiting time between distinct activities within the execution of a process instance. In general, all systems natively employing the activity metaphor (e.g. using forms) do satisfy this requirement.



Data_A_1 abc
Data_A_2 abc
Data_A_3 abc
Data_B_4 abc
Data_B_5 abc
Data_B_6 abc
Data_C_7 abc
Data_C_8 abc
Data_C_9 abc
Data_D_10 abc
Data_D_11 abc
12

Name: Task_A
Comment:
Element type: Formfill Action
Form: Form_1 1 | 1
Condition:
Mandatory: Data_A_1 abc, Data_A_2 abc, Data_A_3 abc
3

Fig. 8. Process and task definition in FLOW*er*

One interesting object for activity mining is the case handling system FLOW*er*. Case handling systems are data-driven, i.e. the availability of information determines the progress within the executed process. Thus, they provide logs both on a high level, identifying activity executions, as well as on a lower data modification level. Figure 8 shows the process designer of FLOW*er*. The displayed process model was actually used to generate the low-level log whose activity mining analysis is shown in Figure 7. In the window on the lower right of Figure 8 the properties for the first task "Task_A" are displayed, for whose completion three data objects (listed in the lower half) are required. These data objects indeed correspond to the footprint of the aggregated meta-cluster "ADMC.1" in Figure 7.

Case handling provides the end user of a process with a significant amount of freedom during execution, including custom deviations from the standard path. Even more important, the boundary between activities is significantly lowered, as there is no strict coupling between forms and activities. Activities can be executed only partly, and be finished later on, potentially by another person. On the other hand, it is also possible for users to execute multiple activities at once, i.e. by filling out only one form. In such an environment, activity mining can yield interesting results, by revealing the actual chunks of work which are frequently executed, in contrast to the ones proposed by the process definition. This analysis can both be used to gain interesting insights into the way people perform their work, and it can also serve as a perfect guideline for a redesign of the process definition.

Another important field of application for activity mining are *Enterprise Resource Management* (ERP) systems. In contrast to workflow management, emphasizing the control flow perspective, these systems are centered around large databases which can be modified in an application-specific manner. As a consequence of the emphasis on data, ERP systems usually create event logs on a low-level. Process definitions and notions of activities are often only existent on an application layer, which is not reflected in the log. Activity mining can effectively bridge this gap, by providing the necessary high-level abstraction. However, it has to be noted that there are further problems to be resolved with respect to logs from ERP systems, e.g. missing references to process instances, which the presented algorithm does not address.

While ERP systems only feature process orientation on the application layer, there are also systems which do not support the notion of a defined process at all. Nevertheless, a great share of these systems, e.g. document management systems or expert systems, are effectively used to support processes. While some organizations prescribe a process definition *off-line*, e.g. in printed form, others rely on their users' *tacit knowledge* to perform the right steps in the correct order.

It is obvious that in such less structured settings, there is even greater demand for abstraction and analysis. These systems have no notion of a high-level process, hence they can only produce low-level logs. Transforming these to higher-level logs, using activity mining, allows to use sophisticated analysis methods, e.g. process mining, which are a premise for discovering and monitoring these implicit, tacit processes.

This field of application can be extended onto *Enterprise Application Integration* (EAI) architectures, which are about to become commonplace in modern companies. In order to connect all kinds of incompatible systems from different vendors (often including legacy installations), most EAI implementations rely on a central *message broker*, or *message bus*, solution. These central hubs relay and transparently translate messages between otherwise incompatible components, thus enabling company-wide integration and workflow. Often, this architecture is supported by meta-processes, e.g. implemented in BPEL, which are orchestrating smaller processes within component systems. Logs from such message brokers usually feature singular interactions between component systems in the form of message events. Clustering these logs can unveil common patterns of interaction, potentially unforeseen by system designers. Based on these discovered patterns, the architecture can be better understood and optimized for performance and quality.

Apart from business processes, it is important to note that the presented algorithm is generic enough to provide useful insights from basically any sort of low-level log. Its application to change logs from source code management systems, like CVS or Subversion, can yield popular subsets of the repository which are frequently modified together. This information can subsequently be used to e.g. reorganize the repository layout for easier navigation. Further, the clustering algorithm has also been successfully applied to access logs from web servers. In this context, rather small values for the scan window size can yield clusters containing pages and their associated resources (e.g. images). On the other hand, increasing the scan window size, such that it spans an entire visit's duration, can be used to group visitors according to which subset of the site they have frequented.

Another interesting application for activity mining is the healthcare domain. In a great number of hospitals the supporting information systems transmit events related to patient treatment activities to a central data-warehousing system. Due to the organization of a hospital, it is usual to concentrate a number of examinations (e.g. blood tests) or treatments in a short time span, in order to minimize transportation between the wards involved. This practice leads to event logs containing bursts of events referring to lower-level activities. Clustering the fine-grained event data from a patient's treatment process can provide useful abstraction from single activities, and reveal logical tasks which describe the logged procedure more adequately.

When the clustering algorithm fails to successfully rediscover activity patterns present in the process definition, this does not necessarily mean that there is a problem with the log or the algorithm's configuration. It can rather be a valuable hint that the process definition is in fact not in line with the current practices within the organization. If the algorithm groups low-level events of two distinct activities into one cluster, this is a strong indication that these activities are frequently executed directly one after another. Such information can provide valuable information for a redesign effort, i.e. in the above case one would want to combine the affected activities into one.

Finally, it is also possible to apply the presented algorithm to regular high-level logs. By setting the scan window size to the typical throughput time of a case, the resulting meta-clusters represent typical sets of activities performed in a process[5]. The initial clusters contained in these meta-clusters can correspondingly be interpreted as cases, or process instances. Thus, if the high-level log does not contain the process ID

[5] Note that different execution orders for parallel parts of the process do not confuse the algorithm, as the footprint is considered an unordered set.

attribute for events, an activity mining analysis can be of great help in rediscovering this information.

10 Related Work

The presented work is closely related to the area of process mining, describing a family of a-posteriori process analysis techniques based on event logs. For example, the alpha algorithm [6] can construct a Petri net model describing the behavior observed in the log. The Multi-Phase Mining approach [13] can be used to construct an Event Process Chain (EPC) based on similar information. In the meantime there are mature tools such as the ProM framework [14] to construct different types of models based on real process executions.

Process mining research so far has mainly focussed on issues related to control flow mining. Different algorithms and advanced mining techniques have been developed and implemented in this context (e.g., making use of inductive learning techniques or genetic algorithms). Tackled problems include concurrency and loop backs in process executions, but also issues related to the handling of noise (e.g., exceptions). Furthermore, first work regarding the mining of other model perspectives (e.g., organizational aspects) and data-driven process support systems (e.g., case handling systems) has been conducted [3].

Activity mining is different from traditional process mining in various respects. It does not attempt to derive information about the process definition, organization, or execution in general, but rather concentrates on a logical, activity-based abstraction *within* the realm of event logs. In order to analyze event logs in a meaningful manner, process mining algorithms require the process instance ID for each event. This requirement does not hold for activity mining, as it is primarily based on the notion of proximity. While process mining algorithms are based on analyzing high-level logs, activity mining does provide this very information as an abstraction from lower-level logs. Thus, activity mining is a valuable means for preprocessing low-level event logs to higher-level logs, in order to perform meaningful process mining.

The results obtained from applying process mining techniques on these reconstructed higher-level logs can provide the most interesting insights, when the system having produced the initial low-level logs in the first place is not designed to strictly enforce a rigid process definition. One system which provides outstanding support for flexible changes of the process model, both on a casual and evolutionary basis, is ADEPT[20]. Also the case handling approach[7], implemented in the commercial system FLOW*er*[10], is especially interesting, as it rather limits the possible execution paths, in contrast to prescribing a fixed set of paths.

There exists a large amount of similar work also from outside the process mining field. Most notably there have been numerous approaches from the data mining domain, which are also focused on clustering sequential event data. Our approach is distinct in that it takes full advantage of the peculiarities of the sequence and events under consideration, i.e. it is far more closely tailored towards the application domain of business process event logs.

Clustering techniques are commonplace in the area of data mining. However, most approaches address the problem of *conceptual clustering* [15], i.e. they strive not to derive higher-level entities from the input data but to derive a classification which can be applied to the initial entities (i.e. events).

Another related field in the data mining domain deals with the discovery of patterns from sequential data. Agrawal et al. [9] also look at events (transactions) from the same process instance (customer), however, their observation is global, i.e. items to be compared are not constrained by their temporal distance. While Bettini et al. [12] also use the idea of focusing on small parts (granules) of the sequence, these do not overlap, and the "interesting" patterns to be discovered are supposed to be defined a-priori.

These approaches are interested in the (partial) order between elements of a pattern, as their focus lies on deriving implication rules from sequences of events. This contradicts the basic assumption of our clustering algorithm, namely that the order of events within an activity cluster is not significant.

Perhaps the approach most related to ours is from Mannila et al. [18], also using a "time window" to restrict the subset of events under consideration. Although this work also mainly focuses on the partial ordering of elements, they also consider the trivial case of an empty partial order, corresponding to our approach. Nevertheless, the absence of using any event attributes other than name (i.e. modified data type) and timestamp, and the focus on implication rules, e.g. for the prediction of future events, poses a significant difference to our approach.

11 Discussion

Activity mining, as it has been motivated and presented in this paper, describes the process of extracting recurring patterns from event logs, which point to the existence of common tasks on a higher level of abstraction. The need for activity mining is driven by multiple use cases.

Process mining techniques have evolved on to a stage where their industrial application does not only seem feasible but truly beneficial. However, they interpret log events as corresponding to the execution of abstract tasks, which conflicts with most real-life systems' logging on a far more fine-grained, and thus lower, level. Consequently this mismatch accounts for overly complex and detailed models which make it hard to derive meaningful information from.

Conducting process mining in a meaningful manner becomes even more difficult when the system from which the logs are derived is not process-aware. In most of these cases it is nevertheless safe to assume the existence of an implicit, higher-level process, following the hypothesis stated in Section 2. People tend to follow certain patterns (i.e. implicit processes) when accomplishing recurring tasks, and they usually also divide them into similarly sized chunks of work (which is our definition of an activity). This property even extends onto automatically executed processes, as these are designed by humans unconsciously applying these paradigms.

In such situations activity mining can provide the abstraction necessary to apply process mining techniques. Once a high-level log has been derived it can be analyzed with the set of process mining algorithms already available, in order to e.g. discover the tacit high-level process which has generated the log. If, however, the process appears to make no sense from a semantic point of view, then this is a strong hint that unsuitable parameters have been used for activity mining. Thus, performing the subsequent process mining analysis can also be used to verify the correctness of the activity mining pass.

In comparison to these process mining algorithms, activity mining does not rely solely on the log itself to derive these abstractions. Configuring the algorithm with appropriate parameters requires domain knowledge, such as the maximal time used to handle activities, in order to accurately discover the activity clusters. We are currently working on heuristics to automatically find suitable parameters based on the log, as a means to aid the user in finding the correct configuration.

The current state of affairs is that a large share of process-aware information systems do not provide activity-level logs, and are thus not suitable for the application of process mining. Activity mining has the potential to bridge this gap, bringing both new fields of application to process mining and, conversely, an ample toolkit of scientifically well-founded analysis methods to owners of such systems.

We have also shown that this technique can be used in settings different from its original content. The modular design of the algorithm allows for tailoring it to different applications, e.g. by using a custom clustering function or aggregation method. We are thus convinced that there are plenty of applications for this technique even outside of the intended scope.

12 Acknowledgements

This research is supported by the Technology Foundation STW, applied science division of NWO and the technology programme of the Dutch Ministry of Economic Affairs.

References

1. W.M.P. van der Aalst and P.J.S. Berens. Beyond Workflow Management: Product-Driven Case Handling. In S. Ellis, T. Rodden, and I. Zigurs, editors, *International ACM SIGGROUP Conference on Supporting Group Work (GROUP 2001)*, pages 42–51. ACM Press, New York, 2001.
2. W.M.P. van der Aalst and K.M. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT press, Cambridge, MA, 2002.
3. W.M.P. van der Aalst and M. Song. Mining Social Networks: Uncovering Interaction Patterns in Business Processes. In J. Desel, B. Pernici, and M. Weske, editors, *International Conference on Business Process Management (BPM 2004)*, volume 3080 of *Lecture Notes in Computer Science*, pages 244–260. Springer-Verlag, Berlin, 2004.
4. W.M.P. van der Aalst, B.F. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A.J.M.M. Weijters. Workflow Mining: A Survey of Issues and Approaches. *Data and Knowledge Engineering*, 47(2):237–267, 2003.
5. W.M.P. van der Aalst and A.J.M.M. Weijters, editors. *Process Mining*, Special Issue of Computers in Industry, Volume 53, Number 3. Elsevier Science Publishers, Amsterdam, 2004.
6. W.M.P. van der Aalst, A.J.M.M. Weijters, and L. Maruster. Workflow Mining: Discovering Process Models from Event Logs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1128–1142, 2004.

7. W.M.P. van der Aalst, M. Weske, and D. Grünbauer. Case Handling: A New Paradigm for Business Process Support. *Data and Knowledge Engineering*, 53(2):129–162, 2005.
8. R. Agrawal, D. Gunopulos, and F. Leymann. Mining Process Models from Workflow Logs. In *Sixth International Conference on Extending Database Technology*, pages 469–483, 1998.
9. R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In P. Buneman and S. Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 207–216, Washington, D.C., 26–28 1993.
10. Pallas Athena. *Case Handling with FLOWer: Beyond workflow*. Pallas Athena BV, Apeldoorn, The Netherlands, 2002.
11. Pallas Athena. *Flower User Manual*. Pallas Athena BV, Apeldoorn, The Netherlands, 2002.
12. C. Bettini, X. S. Wang, and S. Jajodia. Mining temporal relationships with multiple granularities in time sequences. *Data Engineering Bulletin*, 21(1):32–38, 1998.
13. B.F. van Dongen and W.M.P. van der Aalst. Multi-Phase Process Mining: Building Instance Graphs. In P. Atzeni, W. Chu, H. Lu, S. Zhou, and T.W. Ling, editors, *International Conference on Conceptual Modeling (ER 2004)*, volume 3288 of *Lecture Notes in Computer Science*, pages 362–376. Springer-Verlag, Berlin, 2004.
14. B.F. van Dongen, A.K. de Medeiros, H.M.W. Verbeek, A.J.M.M. Weijters, and W.M.P. van der Aalst. The prom framework: A new era in process mining tool support. In G. Ciardo and P. Darondeau, editors, *Proceedings of the 26th International Conference on Applications and Theory of Petri Nets (ICATPN 2005)*, volume 3536 of *Lecture Notes in Computer Science*, pages 444–454. Springer-Verlag, Berlin, 2005.
15. D. H. Fisher. Knowledge acquisition via incremental conceptual clustering. *Mach. Learn.*, 2(2):139–172, 1987.
16. S. Jablonski and C. Bussler. *Workflow Management: Modeling Concepts, Architecture, and Implementation*. International Thomson Computer Press, London, UK, 1996.
17. F. Leymann and D. Roller. *Production Workflow: Concepts and Techniques*. Prentice-Hall PTR, Upper Saddle River, New Jersey, USA, 1999.
18. H. Mannila, H. Toivonen, and A. I. Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1(3):259–289, 1997.
19. D.C. Marinescu. *Internet-Based Workflow Management: Towards a Semantic Web*, volume 40 of *Wiley Series on Parallel and Distributed Computing*. Wiley-Interscience, New York, 2002.
20. M. Reichert and P. Dadam. ADEPTflex: Supporting Dynamic Changes of Workflow without Loosing Control. *Journal of Intelligent Information Systems*, 10(2):93–129, 1998.