KeyValueSets: Event Logs Revisited

Michael Westergaard[1,2] and Boudewijn F. van Dongen[1]

[1] Department of Mathematics and Computer Science,
Eindhoven University of Technology, The Netherlands
m.westergaard,b.f.v.dongen@tue.nl
[2] National Research University Higher School of Economics,
Moscow, 101000, Russia

Abstract. In process mining, event logs have traditionally been considered as strictly hierarchical lists of events, where each event belongs to exactly one case, refers to exactly one activity and has a timestamp. Based on this assumption, the XES standard has been developed to describe event logs. In this paper, we reconsider the notion of an event log, by focussing on events as the primary entity. Furthermore, we do not assume the presence of traditional notions like cases and timestamps (or even ordering), but we introduce mappings for sorting and grouping our *KeyValueSets*. This allows us to provide a generic transformation from a wide variety of formats to standard XES logs.

1 Introduction

Traditional wisdom says that programs should be generous on input and strict on output. This means they should accept anything as input and produce only results according to some schema. Applied to process mining this means that a mining or analysis algorithm should be able to accept any data source as input. This is of course impractical, so most algorithms expects a structured log of events, e.g., specified using the eXtensible Event Stream (XES) [1] format. This instead puts the stress on conversions from other less structured data sources, such as relational databases or comma separated values (CSV) files, to generate structured data. Examples of such import programs are Nitro [2], which can convert CSV files and Excel spreadsheets to XES, XESame [1, 3], which can generate XES logs from database sources, and ProMimport [4, 5] which is an extensible tool for translating data sources to a predecessor of XES. Common for these tools is that they have to specify how their source data must be translated to XES. The reason is that XES logs are quite structured and not only assigns a semantics to attributes but also imposes a strict ordering on events. If such import plug-ins furthermore have to transform their data, this results in a lot of redoubled effort to transform unstructured data to a structured log.

We introduce a hierarchy of structuredness of logs. At the most structured level, we have a traditional XES log, providing random access to all events. We also introduce less structured versions of these, one is a log without random access, i.e., it is only possible to scan through the log from the start to the end, and one is a log without random access and without the ability to scan the data more than once, i.e., it can be seen as a live stream of things happening in real life.

At an even more abstract level, we introduce a new data structure, which has even less structure. Instead of separating events into traces, it is just a soup of events with associated data. This structure comes in two versions, one with and one without rewind ability. As we have no structure on the events, random access is meaningless.

As our new event representation has no or very little structure, it is very easy to transform other unstructured data sources to our format. We decompose the transformation and translation phases into two phases: a transformation phase and a translation phase. The transformation phase simply translates from our data structure to our data structure, making the phase optional, compositional, and easily extensible. The translation phase only has to be done once for all input sources, so we can put more effort into making a user-friendly and truly scalable translation.

Thus, our approach makes it possible for import programs to be generous on the output, as our data structure has very little structure, and for analysis and mining algorithms to be strict on input, as we provide a generic structuring mechanism. This reverses traditional wisdom and makes it easier to write import programs which no longer need to transform and structure their output themselves, and also easier to write analysis and mining algorithms as they can make strong assumptions on their input, simplifying case handling and moving focus to the actual task at hand.

Normally, a trace is an ordered sequence of events, each having a number of properties. A log is then a set of traces. We can think of an event as a mapping from property names to property values, making a log a set of sequences of maps. Our data structure simply removes the middle layer, so our representation considers sets of maps from keys to values. As we assign no semantics to our mapping, we use the more general terms keys and values instead of property names and property values. We call our data structure a *KeyValueSet* (it is a set of key/value mappings).

In the remainder of this paper, we first formally introduce logs and our new data structure in Sect. 2, and put it into context with the less structured log formats. We then give some examples of translations from common unstructured sources to our new data structure in Sect. 3. This includes a maybe surprising application of reinterpreting existing logs to focus on another perspective, e.g., reinterpreting a log from focusing on customers to focusing on employees. In Sect. 4 we present our implementation in ProM 6 [1] and finally, in Sect. 5 we sum up our conclusions.

2 Unstructured Event Data

Traditionally, event logs in process mining consist of events which refer to the execution of activities. These events are timestamped and belong to cases, i.e. an event log is a list of cases and each case is a list of events. This assumption forms the basis of the XES standard for storing event logs. However, some algorithms require less structure than what is provided by XES by default, e.g. they do not require timestamps or total orderings on events etc.

The open-source implementation of XES, OpenXES, provides a number of classes to read, write and edit logs in a random access fashion. However, we have noticed that random access to traces is rarely needed. For example, a genetic mining algorithm only needs to scan through a log in order to replay each event of each trace of the

log. This is an incredibly useful fact, as a disk-based implementation of logs is much more efficient if it is only scanning and does not provide random access. Furthermore, some algorithms only need to look at each trace once, so they do not even need to be able to scan the log more than once. One such example is the alpha algorithm, which only needs to look at each trace once in order to compute the footprint necessary to synthesise a model. This is useful as we no longer need to store the log; instead we can just feed traces into the algorithm as they occur. We have summarized this in Fig. 1, where the XLog is as usual in XES: full structure and random access. The XExternalLog has all operations of the XLog except for the ability to get an element at a random position in the log. The XTraceStream furthermore removes the ability to rewind the log.

The motivation for guaranteeing less structure is that it allows more efficient implementations and for other data sources to be viewed as event logs. In Fig. 1 we provide implementation suggestions for each of the levels. Any of the abstraction levels can be implemented using the implementation strategies of more specific levels, but can not efficiently be represented by one of the more generic levels. For example, providing random access using a disk-based backing is possible but inefficient as seeking in external memory is expensive. The reason for including databases as implementation suggestion in multiple places is that depending on the assumptions we make on the database backing, different implementations can be made. For example, without an index a database would work for scanning and rewinding, but not for random access.

In Fig. 1, we also see our data structure, the KeyValueSet. It can be thought of as a table in a relational database or as a log where all structure has been removed. We have a number of data items each representing an event. For each data item we have a set of named attributes. In Table 1, we see an example of a simple ordering process. We see that not all fields are set (marked with a dash) and there is no particular order to the data. Our data structure considers such a table as a set of rows (optionally introducing an identifier to make rows unique). Each column corresponds to a key (e.g., Customer and Action). Each row contains values for (some of) the keys (e.g., the first row contains values for all keys but Employee). We especially note that (a) not all keys necessarily have any value for each row, and (b) there is no obvious sorting of the elements.

The KeyValueSet in Fig. 1 is a more specific version of the KeyValueStream. The KeyValueStream just provides unstructured information about events as they happen without any rewind ability. This is a very realistic situation, but provides so little information, it is very hard to retain or rediscover a structure without significant assumptions and/or

Table 1. Simple table showing customer handling.

Customer	Employee	OrderID	Items	Action	Time
Alizee	-	1	2	Create Order	10:00
-	Howie	1	2	Send Order	10:10
-	Howie	3	1	Send Order	10:15
Britney	-	2	2	Create Order	10:05
Britney	-	3	1	Create Order	10:07
Britney	-	2	2	Cancel Order	10:09
Alizee	Jon	1	2	Pay Order	15:00

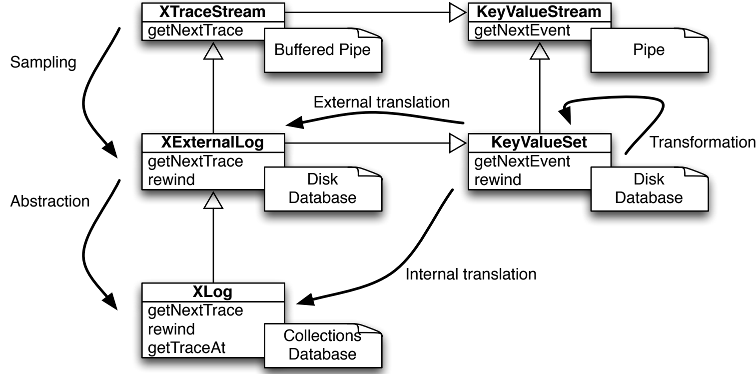

Fig. 1. Different levels of structure of logs.

caching. For this reason we focus on the KeyValueSet, as it provides a very flexible foundation and we can translate it into any of the more structured representations.

Formally, we consider a finite set $\mathbb{K}$ of *keys*, a set V of possible values, and define a KeyValueSet as:

Definition 1 (KeyValueSet). *Given a finite set of* ***keys*** $\mathbb{K}$ *and a set of values* V*, a* ***KeyValueSet*** *over* $(\mathbb{K}, V)$ *is a set* KVS *of partial functions from* $\mathbb{K}$ *to* V*,* $\iota : \mathbb{K} \to V \uplus \{\perp\}$ *where* $\perp$ *denotes the bottom element where the function does not return a real value, i.e.,*

$$KVS \subset \{\iota \mid \iota : \mathbb{K} \to V \uplus \{\perp\}\} = \mathcal{KVS}$$

In our example in Table 1, we have $\mathbb{K} = \{$Customer, Employee, OrderID, Items, Action, Time$\}$ and $V = \{$Alizee, Britney, Howie, Jon, 1, 2, 3, Create Order, Send Order, Cancel Order, Pay Order, 10:00, 10:05, 10:07, 10:09, 10:10, 10:15, 15:00$\}$. We can of course consider any superset of $\mathbb{K}$ and V as well. We can write the first item of the example as $\{$Customer $=$ Alizee, OrderID $=$ 1, Items $=$ 2, Action $=$ Create Order, Time $=$ 10:00$\} \in KVS$ using the notation $\iota = \{key_1 = value_1, key_2 = value_2\}$ for the function

$$\iota(k) = \begin{cases} value_1 & \text{if } k = key_1, \\ value_2 & \text{if } k = key_2, \\ \perp & \text{otherwise.} \end{cases}$$

We may want to transform our KeyValueSet. In our example, we may want to add a unique identifier to each row or we may want to add 1000 to each OrderID. The last operation is useful if we want to merge (compute the union) of two KeyValueSets and ensure the OrderIDs do not overlap.

A transformation from one KeyValueSet to another is just a function, $t : \mathcal{KVS} \to \mathcal{KVS}$. This is of course impractical as it in theory requires us to fully specify the function for each input. Instead we often prefer to apply transformations item by item, i.e., we search a function $t_i : \mathbb{K} \times V \to \mathbb{K}' \times V'$ and define the transformation as:

$$t(KVS) = \{t_i(\iota) \mid \iota \in KVS\},$$

which for a KeyValueSet over $(\mathbb{K}, V)$ yields a new KeyValueSet over $(\mathbb{K}', V')$.

We note that when specified by item, a transformation cannot remove items. We can however introduce a new key, say ToDelete, and only define it for items to delete and then use a generic transformation to delete the desired items. We can, e.g., mark all items taking place before 10:10 for deletion using

$$t_{delete}(\iota) = \begin{cases} \iota \cup \{\mathsf{ToDelete} = \mathsf{true}\} & \text{if } \iota(\mathsf{Time}) < \mathsf{10{:}10}, \\ \iota & \text{otherwise}. \end{cases}$$

Our example in Table 1 does not have enough information to consider it a log. A log consists of individual traces all pertaining to a single case or instance of the process we look at. Each event has a number of *properties*, which are similar to values for keys, but often augmented with a semantics and limited in nature. For example, the XES standard defines a set of *extensions* each defining a set of properties. This set is well-defined and while extensible subject to standardization (or nobody can use your new property). We shall use $\mathbb{P}$ to denote a set of properties supported by our log formalism. As an example, we shall stick to the (subset of the) properties of XES, $\mathbb{P} = \{$org:resource, concept:name, time:timestamp$\}$. An *event*, e, is a partial map from $\mathbb{P}$ to some value set V (actually, the value set depends on the property in question, but we ignore this for simplicity; our implementation correctly handles this).

Furthermore, a trace needs to be sorted according to the order the events took place. This makes it possible to also treat logs with no timestamps. A trace may optionally include a trace identifier (and other attributes), but we shall ignore this for simplicity. A log representation of our example from Table 1 can be seen in Table 2. Here we consider a case to coincide with the OrderID and order events according to their timestamp. We consider the Employee the resource and have to omit the other values – although, we could include them as custom properties, few algorithms could make use of them. We see that all events occur together with the other events belonging to the same order and the timestamps are all ordered chronologically.

Formally, a trace is a sequence of events, which are similar to items from KeyValueSets except they do not use a generic set for keys but the predefined set of properties.

Definition 2 (Trace). *A **trace** over a set of values V is a sequence of **events**, $\tau = e_1 e_2 \ldots e_n$, where e_i ($1 \leq i \leq n$) is a partial function from $\mathbb{P}$ to V. If we set $\Sigma = \{e \mid e : \mathbb{P} \to V \uplus \{\bot\}\}$, we have that*

$$\tau \in \Sigma^*.$$

A **log** is just a sequence of traces:

Definition 3 (Log). *A **log** over a set of values V is a set of traces over V, $L = \{\tau_1, \tau_2, \ldots, \tau_m\}$, where τ_j ($1 \leq j \leq m$) is a trace $\tau_j \in \Sigma^*$ where Σ is as defined in Def. 2.*

Our log in Table 2 is a log, where the individual traces are separated by horizontal lines, the order of each trace is top-down, and the value of each event can be read from the table.

Comparing the definition of KeyValueSets (Def. 1) and Logs (Defs. 2 and 3), we notice that items and events are very similar, though items use a more general set of

Table 2. Log of example from Table 1.

org:resource	concept:name	time:timestamp
-	Create Order	10:00
Howie	Send Order	10:10
Jon	Pay Order	15:00
-	Create Order	10:05
-	Cancel Order	10:09
-	Create Order	10:07
Howie	Send Order	10:15

keys as domain. We can translate from logs to KeyValueSets by removing the extra layer consisting of traces and make a flat structure. In order to translate KeyValueSets to Logs, we need a means to order events, a means to recognize traces, and a means of translating from general keys to properties.

In our example, we ordered events by the key Time to go from the KeyValueSet in Table 1 to the log in 2. We grouped traces by the key OrderID, and mapped org:resource to Employee, concept:name to Action, and time:timestamp to Time. Here we have only used one-to-one mappings, but there is nothing preventing us from mapping multiple properties to multiple keys, using multiple keys for grouping, and using multiple keys for sorting. Thus a mapping is formally defined as:

Definition 4 (Mapping). *Given properties $\mathbb{P}$ and values $(\mathbb{K}, V)$, a* ***mapping*** *translating KeyValueSets over $(\mathbb{K}, V)$ to Logs over V^* is a function m mapping properties of $\mathbb{P}$ and the two distinct values Sorting and Trace to sequences of keys of $\mathbb{K}$,*

$$m : \mathbb{P} \uplus \{\textit{Sorting}, \textit{Trace}\} \to \mathbb{K}^*.$$

We extract the value of an item of a KeyValueSet using a slightly particular method. For each property, we get a sequence of keys. The intuition is that the value of property is a sequence containing all the values of the item in the place corresponding to the position of the key in the sequence. The particularity is that we ignore any element which is $\perp$ and identify the empty tuple with $\perp$. Thus, if $m(\text{org:resource}) = \langle \text{Customer}, \text{Employee} \rangle$ we would extract the values Alizee and Howie for the first two items of Table 1 and the value Alizee Jon for the last item. The advantage is that we can merge multiple columns easily. For example, if we stored regular employees using key Employee and managers using the key Manager, we could easily merge the two. We denote by *value* $: \mathbb{K}^* \times (\mathbb{L} \to V) \to V^* \uplus \{\perp\}$ the function thus extracting the value of an item.

We can now define the translation from a KeyValueSet to a Log:

Definition 5 (Translation). *Given a KeyValueSet KVS over $(\mathbb{K}, V)$ and a mapping m translating KeyValueSets over $(\mathbb{K}, V)$, the translation of KVS to a Log L_{KVS} over V^* is defined as:*

- *$T_{KVS} = \{value(m(\text{Trace}), \iota) \mid \iota \in KVS\}$ is the set of* ***trace identifiers****,*
- *properties of events are mapped according to m, i.e., for each $j \in T_{KVS}$ we have that $e_i \in \tau_j$ iff $\iota_i \in KVS$ and $e_i(p) = value(m(p), \iota)$,*

– *events of individual traces are ordered according to* $m(\textsf{Sorting})$*, i.e., for each* $j \in T_{KVS}$ *and* $1 \leq i, i' \leq |\tau_j|$ *we have that value*$(m(\textsf{Sorting}), \iota_i) \prec$ *value*$(m(\textsf{Sorting}), \iota_{i'}) \implies i < i'$*,*
– *the result is the set of all such traces, i.e.,* $L_{KVS} = \{\tau_j \mid j \in T_{KVS}\}$.

We note that we may get a trace corresponding to the trace identifier $\perp$. This will often contain garbage, but can easily be filtered out subsequently, so we do not define our way out of this as there may be cases where this is useful. We also notice that any property not explicitly mentioned is left undefined as is any property for which no values were found in the original item.

3 Uses

Despite the simplicity of the translation outlined in Defs. 4 and 5 and the transformations defined in the previous section, they prove immensely powerful as it is trivial to translate from several common unstructured data sources to our KeyValueSets. In this section we first outline some simple examples of translations to KeyValueSets, we then turn to translating in the other direction, from a log to a KeyValueSet, and see some interesting applications of this.

3.1 CSV Files and Relational Databases

Translating from CSV files to logs is a task which has previously been quite complex. Recently, Nitro [2] has made this neigh-trivial, but using our KeyValueSets it becomes even easier. Most CSV files have column names embedded in the first line. If a file does not have that, we need to ask users for key names. Other than that, all we need to do is read each line, split it at the comma (or semi-colon in some cases), match the values to the keys from the first line and produce an item for each line. A fully functional translator implemented in ProM 6 uses 227 lines of Java code including a graphical user interface to select the separator and altering key names. Empty fields are translated to $\perp$.

Translating from relational databases to logs is also complex a separate tool, XESame [1] has been created to do it. It uses a complex XML description of transformation and mapping between different attributes. It would be fairly trivial to make an importer loading a table or even the result of a query into a KeyValueSet, using a similar strategy as for CSV files. We can even handle outer joins where fields may be undefined as our data structure explicitly supports this. A better way would be to create a KeyValueSet backed by a database as this would conserve internal memory and prevent double representation.

3.2 EDI Messages

The Electronic Data Interchange (EDI) [6, 7] protocol allows interenterprise communication. EDI messages are basically a standardized way of describing common documents and each message is at its base a set of key/value pairs and hence lends itself

perfectly to translation to KeyValueSets. The challenge with EDI is that it assigns semantics to many of the defined keys and in order to get something meaningful out of the translation, care must be taken to make a sensible mapping [8]. As soon as an EDI message has been translated to a KeyValueSet it can be transformed using generic transformation and a mapping can be set up and used for translation to a log with no further effort on the part of the implementer.

3.3 CPN Tools Simulation Logs

CPN Tools [9] is a tool for creating, simulating, and analyzing colored Petri nets (CPNs) [10], a formalism for discrete event simulation extending classical Petri nets with data. An important point of colored Petri nets compared to classical Petri nets is that tokens are distinguishable, which makes it easy to simulate multiple concurrent scenarios. When a transition, modeling an event, is executed it is executed in a particular binding of variables. The number and nature of variables vary from transition to transition. CPN Tools has for the past two decades emitted a simulation log. The contents of the log is the step number (a counter), the model-timestamp, the name of the transition (and a few modeling language-specific tidbits irrelevant here), and the value of all variables at the point of execution of the transition.

Previously, a library, the ProM CPN library, made it possible to annotate CPN models and get output which had to be run through ProMimport prior to analysis. The annotations are fairly cumbersome and does not make it easy to quickly make a model, get simulation result, and try out an algorithm.

Using KeyValueSets it is very easy (128 lines in ProM 6) to load in the simulation log native to CPN Tools. The output is unstructured and needs to be configured by the user, but most of the time that is simple. Along with the transition name and all bindings of variables, also the step counter and the timestamp are created as keys; the KeyValueSet contains one item for each transition execution. Often the transition name is mapped to concept:name and a particular variable (or 2-3 similarly named variables) serves as trace identifier. Finally, if applicable, a resource is often visible as a variable and more attributes can be added as necessary.

3.4 Flattening: Translating Event Logs to KeyValueSets

Most log formats allow adding auxiliary information. In our example from Table 1, we include customer information even though this is not present in the final log in Table 2. The reason we do not include this information is that unless an algorithm is specially tailored to recognize the information or the algorithm explicitly handles "unknown" information, it is useless. Using KeyValueSets we can, however, make this information available. In Table 3, we see an extended version of the log.

We now get the same basic log (the first 3 columns) and also some auxiliary information, the 3 last columns. In fact, we see that we have the full original information available; by shifting around columns and rows and ignoring that we renamed some of the columns, Table 3 contains the exact same information as the original data from Table 1. It is trivial to go from a log to a KeyValueSet: simply reinterpret all events as

Table 3. Log of example from Table 1.

org:resource	concept:name	time:timestamp	OrderID	Customer	Items
-	Create Order	10:00	1	Alizee	2
Howie	Send Order	10:10	1	-	2
Jon	Pay Order	15:00	1	Alizee	2
-	Create Order	10:05	2	Britney	2
-	Cancel Order	10:09	2	Britney	2
-	Create Order	10:07	3	Britney	1
Howie	Send Order	10:15	3	-	1

items, taking $\mathbb{K} = \mathbb{P}$, and flatten the data structure. In our figure we would do that by removing the two lines separating the three traces, formally we would do as so:

Definition 6 (Flattening). *Given a log $L = \{\tau_1, \tau_2, \ldots, \tau_m\}$ over V, a **flattening** of L is a KeyValueSet KVS_L over $(\mathbb{P}, V)$ defined as*

$$KVS_L = \{e_i \mid e_i \in \tau_j \textit{ for some } \tau_j \in L\}.$$

Here we have explicitly included the OrderID, which was also used as trace identifier. If the log formalism also saves this information for each trace, this is of course not necessary; our tool automatically extract all available or derivable information about each event/item.

3.5 Flattening in the CPN-XES Library

For making CPN logs available to any tool that supports the XES format, flattening is an important concept. We already mentioned the existing ProM CPN library. This library records events from CPN Tools by recording the events for each case in a separate file. When CPN Tools is done, the recorded log files have to be imported in ProMimport to merge the individual files into a single log. Storing intermediate files on disk rather than in memory is good practise and is known as *streaming*: instead of keeping information in memory, it is directly written to a file and can later be sequentially merged or, in this case, concatenated, without ever having all data in memory.

While this is a good strategy – in many cases even a necessary strategy – it does introduce an annoying phase of merging the logs using ProMimport. We could instead abuse the extensibility of the XES format and introduce a custom attribute, Trace. That way we could simply save events one-by-one as they arrive. Now, when we load in the log, we get one with all the necessary data but concentrated in a single trace. By translating this to a KeyValueSet using the reverse translation, we can translate it back using an identity mapping which in particular maps Trace to Trace. This splits up the log into traces, exactly as desired. As the translation from XES log to KeyValueSet takes no parameters and the mapping we use always is the identity mapping, we do not even have to ask the user, and can seamlessly import streaming XES by going via KeyValueSet, even though a user never sees that. This is exactly how the CPN-XES library, which supersedes the ProM CPN library, works. Instead of going thru ProMimport, CPN-XES files created by CPN Tools, or by any other tool wishing to do streaming logging using

XES, can be loaded directly in ProM 6. And the entire import plug-in comprises just 61 lines of code and is very efficient as all the hard lifting is done by the generic code.

3.6 Further Uses of Flattening

Flattening also has other less obvious but at least as interesting applications. If we look at the log in Table 3, we see that this log is clearly tailored for analyzing the ordering process. Maybe, for quality assurance, it would make more sense to analyze the customer experience, i.e., focus on the customer instead of the order? Until now, we needed to write special algorithms or do other tricks to facilitate that. Using KeyValueSets this is neigh trivial. All we do is flatten the log to a KeyValueSet and then translate it back to a log, this time using the Customer as the trace identifier.

If we have the original KeyValueSet from Table 1, we can of course also use that, but it is also likely we just have the log in some log format with auxiliary information. Applying this mapping, yields the log in Table 4. Now, we no longer focus on the flow individual orders, but on the experience of customers. We see that Alizee has a good experience; she orders and subsequently pays. Britney, on the other hand may have a frustrating experience; she first makes two orders and then cancels the one. This may indicate that it would be a good idea to make it possible to alter orders. The payment is not part of this log as the order has not completed. This should be possible to catch using any off-the-shell control-flow mining algorithm with no alteration, just by reinterpreting the log. Similarly, we can focus on the workflow of individual employees and so on. Thus, removing all structure allows us to reassemble a new log with a new structure and gain new insights.

4 Implementation

We have implemented our KeyValueSet data structure in ProM 6 in Java. We have taken special care to allow very efficient implementations. The most important part of the implementation is the setup of the mapping. Everything else is done with no user-intervention, so this is the only part a user sees (if they see anything at all). The view shows at the top a random extract of the KeyValueSet. The extract is chosen as random to avoid empty or dummy elements in the beginning of the set. Next is the setup of the mapping and sorting. For both, we can select any number of the available keys. To

Table 4. Log of example from Table 1.

concept:name	time:timestamp	OrderID	Customer
Create Order	10:00	1	Alizee
Pay Order	15:00	1	Alizee
Create Order	10:05	2	Britney
Create Order	10:07	3	Britney
Cancel Order	10:09	2	Britney
Send Order	10:10	1	-
Send Order	10:15	3	-

keep the user-interface simple, we use a canonical ordering of the keys instead of any ordering as in Def. 4. The key(s) chosen for time:timestamp is always at the beginning followed by the values selected explicitly. At the bottom we have a text-field, where we can add new parameters if we so desire.

As we translate to XES and not to a unspecified abstract log format, we know that each parameter has a specific type. For example, a concept:name is always a String and a time:timestamp is a Date. This is also reflected in Fig. 2, where we see that to the left of the textfield for adding new parameters we have a dropdown box of types.

When we to the translation from KeyValueSet to a log, we use similar heuristics; if we know something is supposed to be a date, we try converting it to a date (if it a string, we try parsing it, if it is an integer, we interpret it as Unix time, if it is already a date, we are good), and similar conversion rules for the other available types (Date, Integer, Double, String, Boolean).

The KeyValueSet is part of ProM 6 and can be downloaded from the ProM package manager. The entire package comprises 1695 lines of Java (plus some test code) and includes most of the importers mentioned in the previous section, a simple user interface for transforming KeyValueSet as well as for merging two or more KeyValueSets, and of course plug-ins for flattening logs and translating KeyValueSets to logs.

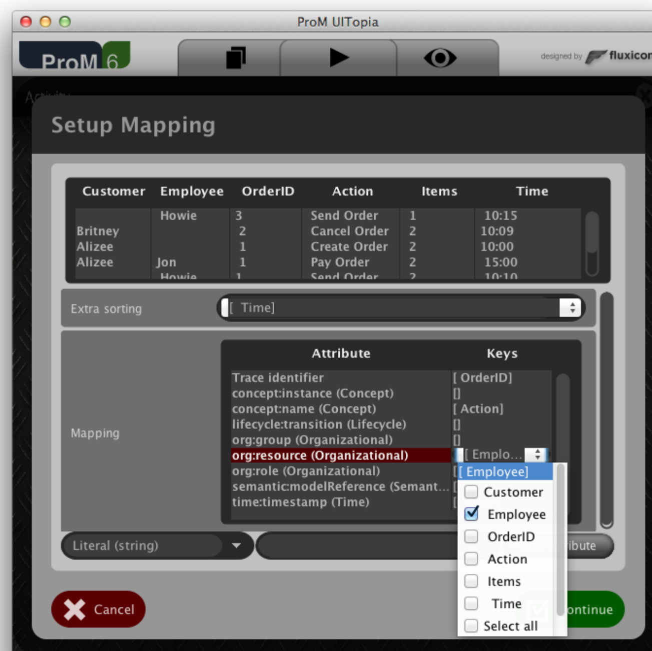

Fig. 2. Screenshot from ProM 6 showing the mapping setup.

5 Conclusion

We started this paper by quoting the traditional wisdom to write code that is generous on input and strict on output. We argued that being generous on input would make process mining and analysis algorithms unnecessarily complex and being strict on output would make converters from unstructured data unnecessarily complex. We have introduced a very simple data structure intended to sit between the two, allowing converters to be generous on output and algorithms to be strict on input, completely reversing the traditional wisdom and making both converters and algorithms much easier to write.

We have exemplified and formally defined our data structure and given several examples where translating to our data structure is much easier than to fully structured logs. We have demonstrated that translating the other way may be useful as we can break down old structure and build new ones, allowing us to analyze process data free of old structure. We have shown how our implementation in ProM.

References

1. Verbeek, H.M.W., Buijs, J.C.A.M., van Dongen, B.F., van der Aalst, W.M.P.: Xes, xesame, and prom 6. In Soffer, P., Proper, E., eds.: CAiSE Forum. Volume 72 of Lecture Notes in Business Information Processing., Springer (2010) 60–75
2. : http://www.fluxicon.com/nitro/
3. : http://www.xes-standard.org/
4. van der Aalst, W.M.P., van Dongen, B.F., Günther, C.W., Rozinat, A., Verbeek, E., Weijters, T.: Prom: The process mining toolkit. In de Medeiros, A.K.A., Weber, B., eds.: BPM (Demos). Volume 489 of CEUR Workshop Proceedings., CEUR-WS.org (2009)
5. : http://www.promtools.org/promimport/
6. Aggarwal, R.: Electronic data interchange. In: Wiley Encyclopedia of Computer Science and Engineering. (2008)
7. Choudhary, K., Pandey, U., Nayak, M.K., Mishra, D.K., Mishra, D.K.: Electronic data interchange: A review. In: CICSyN. (2011) 323–327
8. Engel, R., Krathu, W., Zapletal, M., Pichler, C., van der Aalst, W.M.P., Werthner, H., Werthner, H.: Process mining for electronic data interchange. In: EC-Web. (2011) 77–88
9. Ratzer, A.V., Wells, L., Lassen, H.M., Laursen, M., Qvortrup, J.F., Stissing, M.S., Westergaard, M., Christensen, S., Jensen, K., Jensen, K.: Cpn tools for editing, simulating, and analysing coloured petri nets. In: ICATPN. (2003) 450–462
10. Jensen, K., Kristensen, L.M.: Coloured Petri Nets - Modelling and Validation of Concurrent Systems. Springer (2009)