

"Log data compliance"

Technical Field

[1] This disclosure relates to analysing log data and more specifically, to methods and systems for analysing log data.

Background

[2] Technical systems are becoming more and more complex in the number of operations that they perform as well as the number of interacting modules they contain. While it is possible to define the behaviour of each module, it is difficult to check, at runtime, that all modules together perform as expected. For example, some modules may access shared resources, such as database servers or a cryptographic public keys. As a result, compliance of the entire system depends on the temporal aspects of the behaviour of each module.

[3] Some approaches primarily focus on *conformance checking* instead of *compliance checking*: the behaviour as recorded in the event log is evaluated against the intended behaviour as specified in a process model. With compliance checking, on the other hand, the behaviour is checked with the behaviour as required by a set of rules stemming from regulations. As a result, an execution of a process can be compliant but not conformant (i.e. the behaviour is not in the model, but does not violate any of the rules) or conformant but not compliant (the observed behaviour is allowed by the model but violates one or more rules).

[4] For example, the system may be compliant if module A accesses the public key storage before module B but not if module B access the public key storage before module A. If rules are encoded directly for such scenarios, the number of potential combinations quickly exceed practical constraints as the number of potential scenarios can grow combinatorically, also referred to as "combinatorial explosion". Due to this explosion, it is difficult, if not impossible to design compliance checkers. Therefore,

there is a need for methods and systems that can analyse log data, potentially from many different processes, in an efficient manner. It may be particularly useful, if complexity is reduced to a point where a compliance checker can check for compliance before the next event occurs, so that real-time monitoring becomes possible.

[5] It is noted that the terms 'rules' and 'regulations' herein do not necessarily relate to legal or statutory rules, but may also relate to technical requirements, such as limits on the capabilities of a physical system. That is, compliance checking provides the assurance that the physical system, with its limited resources, is able to safely perform the desired functions from different sources without overloading the system or risking malfunctioning.

[6] Throughout this specification the word "comprise", or variations such as "comprises" or "comprising", will be understood to imply the inclusion of a stated element, integer or step, or group of elements, integers or steps, but not the exclusion of any other element, integer or step, or group of elements, integers or steps.

[7] Any discussion of documents, acts, materials, devices, articles or the like which has been included in the present specification is not to be taken as an admission that any or all of these matters form part of the prior art base or were common general knowledge in the field relevant to the present disclosure as it existed before the priority date of each of the appended claims.

Summary

[8] A method for analysing log data comprises:
receiving log data comprising traces having multiple log events from multiple different respective process executions, each of the multiple log events being associated with an event time;
creating a single stream of log events comprising the multiple log events from the multiple respective different process executions, wherein the single stream of log events is sorted by the associated event time;

iterating over the single stream of log events, and for each log event, executing one or more update functions that define updates of a set of variables based on the log events, the set of variables comprising at least one cross-trace variable to calculate an updated value of one or more of the set of variables, wherein the one or more update functions define updates of the at least one cross-trace variable in response to the log events of more than one of the traces; and

executing one or more evaluation functions on the set of variables to determine compliance in relation to the log data based on the updated value, the one or more evaluation functions representing compliance rules based on the set of variables including the cross-trace variable.

[9] It is an advantage that there is a set of variables that is updated by the update functions, and then the evaluation functions test these variables. This allows efficient checking of cross-process compliance, where otherwise temporal dependencies would make the direct processing of the potentially fully connected evaluation graph computationally difficult, with a potentially combinatorial complexity.

[10] The method may comprise executing the one or more evaluation functions for each log event. The method may comprise performing the steps of creating, iterating and executing in real-time to determine compliance while receiving further log data.

[11] The compliance rules may comprise a conditional obligation. The compliance rules may be defined across multiple processes. The update functions may define an update of one of the set of variables in response to log data from multiple processes or multiple process instances.

[12] The log data may comprise log data generated by a computer system executing an operating system. The log data may be generated by different processes executed by the operating system. The multiple log events may comprise start log events that indicate the beginning of a task and stop events that indicate the end of a task.

[13] The set of variables may be indicative of a number of currently active tasks. The one or more update functions may increment one of the set of variables in response to one of the multiple log events being a start log event; and the one or more update functions may decrement one of the set of variables in response to one of the multiple log events being a stop log event.

[14] The one or more evaluation functions may be based on an upper threshold of one of the set of variables. The one or more evaluation functions may be represented by evaluation predicates. The evaluation predicates may be associated with a logical value indicating a predetermined occurrence of log events. The evaluation predicates may be defined on graph-like structures. The graph-like structures may define a precedence among the evaluation predicates.

[15] The method may further comprise determining a set of evaluation functions that require execution based on the graph structure and executing only the set of evaluation functions in that iteration.

[16] The graph structure may represent a combination of the state that has been checked and the evaluation functions that require execution.

[17] The method may further comprise:
generating an instance of an update function or evaluation function or both to represent a rule;
storing the generated instance in volatile computer memory;
executing the generated instance in the volatile computer memory; and
discarding or overwriting the generated instance in the volatile computer memory while further determining compliance.

[18] The method may further comprise determining compliance of multiple traces in parallel against multiple rules.

[19] The evaluation functions may represent an in-force time interval defined by the rules.

[20] Software, when installed on a computer, causes the computer to perform the above method.

[21] A computer system for monitoring compliance of another system by analysing log data, comprises a processor configured to:

receive the log data comprising traces having multiple log events from multiple respective different process execution, each of the multiple log events being associated with an event time;

create a single stream of log events comprising the multiple log events from the multiple respective different process executions, wherein the single stream of log events is sorted by the associated event time;

iterate over the single stream of log events, and for each log event, executing one or more update functions that define updates of a set of variables based on the log events, the set of variables comprising at least one cross-trace variable to calculate an updated value of one or more of the set of variables, wherein the one or more update functions define updates of the at least one cross-trace variable in response to the log events of more than one of the traces; and

execute one or more evaluation functions on the set of variables to determine compliance in relation to the log data based on the updated value, the one or more evaluation functions representing compliance rules based on the set of variables including the cross-trace variable.

Brief Description of Drawings

[22] An example will now be described with reference to the following figures:

[23] Fig. 1 illustrates a computer network comprising log event generating computers and a log processing server.

[24] Fig. 2 illustrates a graphical example of the event logs from Fig. 1.

[25] Fig. 3 graphically illustrates an in-force interval of a conditional obligation over a trace of a business process model between a trigger and a deadline.

[26] Fig. 4 shows a graphical overview of an example of the method disclosed herein.

[27] Fig. 5 provides a graphical overview of the sequentialised events of the event log depicted in Fig. 1.

[28] Fig. 6 illustrates the events from Fig. 5 as a single trace.

[29] Fig. 7 illustrates an example excerpt of the evolution of the variables after each replayed event.

[30] Fig. 8 illustrates a method for analysing log data.

[31] Fig. 9 illustrates a computer system for monitoring compliance of another system.

[32] Fig. 10 illustrates an example graph for the hypothetical rule *InvoicePay*.

[33] Fig. 11 illustrates an example graph for the rules given herein.

[34] Fig. 12 illustrates an example graph for the hypothetical rule *InvoicePay* scaling over the domain of invoice id's.

Description of Embodiments

Application domains

[35] Methods disclosed herein can be applied to a ranged of different application domains. In particular, the disclosed methods analyse log data that comprises traces. Each trace has multiple log events from multiple different respective process executions. That is, each trace has log events generated by one process execution. A process execution may also be considered as a process instance. For example, in an operating system, a compiled software program may be stored on program memory as a binary file and the steps for the processor to execute are considered as a process. When the processor executes the binary file, the processor assigns a process identifier to that execution. That is, the processor creates an instance of the process. In that sense, there can be multiple instances of the same process. For example, there can be two instances of an Apache webserver being executed by the same processor, each with their respective process identifier.

[36] These process executions are listed separately as an output of admin tools, such as ps in UNIX based system of Task Manager in Microsoft Windows based systems. Each of these executions write to their own separate log file or write into the same common log file with process identifiers added to each log event. It can be appreciated, that the number of log events from multiple process executions can be significant.

[37] Further, it is important to check compliance of the process executions for detecting abnormalities. These compliance checks are more relevant and more robust, if they check compliance across multiple process executions instead of only for each process execution individually. For example, it is possible that each of the multiple webserver mentioned above are compliant individually, but they breach monitoring rules that are defied for both servers together. However, checking compliance across different process executions, that is, across traces, leads to a combinatorial complexity that is difficult to handle with existing computer hardware. Additionally, direct implementation of cross-trace compliance checking using existing well-known computer functions, would result in program code that requires more random access memory (RAM) that is typically available in current computer architectures.

Therefore, the disclosed methods reduces the computational complexity and potentially the amount of RAM that is required for cross-trace compliance checking.

[38] In other examples, the process executions are of a physical nature, such as manufacturing processes. This is becoming an increasingly pressing problem under the label “Industry 4.0”, which relates to data-driven industry. While a huge amount of data on manufacturing processes is being collected, it is difficult to check this data for compliance given the computational complexity for cross-trace compliance checking. For example, in vehicle manufacturing, there may be a process for pressing door panels and a process for pressing bonnet panels. The executions of both of these can be checked separately. However, both processes may share the same press, or share the same transport robot. Therefore, there is a need for a cross-trace compliance check. However, cross-trace compliance checking presents a difficult combinatorial problem that is addressed by the disclosed methods.

Computer network

[39] Fig. 1 illustrates a computer network 100 comprising three computers 101, 102 and 103 which each execute their own processes. Computer network 100 further comprises a log processing server 104. In this example, each computer 101, 102, 103 executes one process with identifiers (IDs) 1, 2, 3, respectively and sends generated event logs to log processing server 104. Each of these processes generate log events, which are shown in individual logs 111, 112, 113, respectively. So the term ‘event log’ refers to a collection of ‘log events’.

[40] Each computer 101, 102, 103 may store event logs 111, 112, 113 locally as shown in Fig. 1 and then send them as to log processing server 104. In other examples, computers 101, 102, 103 ‘stream’ log events in the sense that every new log event is sent directly to log processing server 104 without local collection. In other examples, computers 101, 102, 103 may be replaced by other agents that execute log-generating processes, such as processors executing binaries, or virtual machines operating on a shared resource. Server 104 may also be implemented as a process on a single

computer system or processor together with computers 101, 102 and 103 or on a cloud computing environment. When log processing server 104 receives the log events in event logs 111, 112, 113, server 104 may combine them into one table, or collection of event log.

Log events

[41] Each log event, such as example log event 121, comprises a process ID 122, Event ID 123, event label 124, timestamp 125 and lifecycle 126. In other examples, the event logs 111, 112, 113 may have different columns or a different structure.

[42] It is noted here that the process ID remains unchanged as the log events are generated by the same process executed by a respective computer. In other examples, however, each computer may generate events with different process IDs if the computer executes multiple processes. The event ID is here labelled with a sequential index but computer 102 does not know which index computer 101 has last used, so the event IDs shown in Fig. 1 are simply for illustrative purposes. They may be chosen as unique labels, such as a random number, or they may be an auto-increment integer starting from '0' for each computer 101, 102, 103 and prefixed by a computer or processor ID. The event label 124 is a label that uniquely identifies an event or a 'task'. In this regard, a log event relates to an execution instance of a task and the event label refers to a definition of that task. So computer 101 has two log events corresponding to each of tasks 'A', 'B', 'C', marking the start and end of each of these tasks. It is noteworthy that computers 102 and 103 generate instances of the same tasks, so they also have 'A', 'B', 'C' in their event logs.

[43] The timestamp used in Fig. 1 is such that the index of t represents a point in time, which could be seconds or milliseconds. So for example, the first event in log 111 has occurred at 0 ms (index '0' of t_0), while the first event in log 112 has occurred at 2 ms (index '2' of t_2). In other examples, the timestamp is a UNIX timestamp or other timestamp. The lifecycle indicates whether the event of label 124 has started or

completed. Lifecycle column 126 may hold one of more than two states (e.g., ‘start’, ‘complete’, ‘terminated’, ‘aborted’, ‘suspended’, ‘resumed’, etc.) or may not be used at all.

[44] More specifically, events are recordings of important milestones in the execution of information systems. They may contain, for example, recordings of information regarding tasks in business processes, the creation and alterations of artefacts of that business process, the task or artefact ID, the time at which the event was triggered, and the resource involved in triggering the event. An event occurrence within a trace consists of a set of data variables relevant to that event. Given a trace, the same data variables can appear and change over multiple event occurrences. In addition, there are variables that hold on a trace/instance level. A log event (or simply ‘event’) may be defined formally as follows (but different definitions may be used):

[45] **Definition 1 (Event)** *An event $e \in E$ is a tuple $e = \langle D, v \rangle$ such that: D is a set of data variables such that $\{id, timestamp, state, resource\} \subseteq D$, and v is a function $v: D \rightarrow d(x)$ that obtains the value of the variable $x \in D$, where $d(x)$ denotes the domain of x .*

Event log

[46] Given a set of events, an event log consists of a set of traces, each containing the sequence of events produced by one execution of a process. Events in an event log are related via a total order induced by their timestamps. In other words, server 104 sorts the event log by the timestamps. Sorting may be achieved by entering the events into a linked tree data structure of the Java language, noting that many other implementations in other programming languages are possible. In this context, the terms ‘ordering’ and ‘sorting’ are used synonymously. An event log may be defined formally as follows (but different definitions may be used):

[47] **Definition 2 (Event log, Trace)** Let L be an event log and E a set of event occurrences. An event trace $\sigma \in L$ is defined in terms of an order $O_\sigma = [0, n-1]$ and a set of events $E_\sigma \subseteq E$ with $|E_\sigma| = n$ such that $\sigma = e_0 e_1 \dots e_{n-1}$ is a sequence of events with $\forall i \in O_\sigma : e_i \in E_\sigma$, and $\forall i, j \in O_\sigma : v_{e_i}(\text{timestamp}) \leq v_{e_j}(\text{timestamp})$ iff $i \leq j$.

[48] An example of an event log L is provided in Fig. 1. Every trace is a sequence, where every event has a unique identifier (e_i), an event label (e.g. A), a timestamp (t_k) and a lifecycle event ($start, complete$). In real life event logs, there are typically much more event properties (i.e. variables) recorded with each event, but these are omitted in Fig. 1 for readability purposes.

[49] Event properties here could be the customer ID, order amount, approval etc. The values for each of these variables may be altered by a subsequent event. Certain variables are trace specific, whereas others are valid across traces (e.g. total order amount a manager has to approve, total cases under investigation for a certain resource etc.).

[50] Fig. 2 illustrates a graphical example of the event logs from Fig. 1. Together, event logs 111, 112, 113 are referred to as event log L . In Fig. 2 the traces are visualised as a sequence of events.

Compliance rules

[51] Server 104 may employ a regulatory framework to check compliance of the event log combined from 111, 112, 113 to check that the requirements that a business process model needs to follow to be considered compliant. In one example, server 104 uses the Process Compliance Logic, introduced by Guido Governatori, and Antonino Rotolo. "Norm compliance in business process modeling." International Workshop on Rules and Rule Markup Languages for the Semantic Web. Springer, Berlin, Heidelberg, 2010, which is incorporated herein by reference. Conditional obligations

are described below, to provide an intuition on the types of rules that can be formulated in the regulatory framework.

[52] **Definition 3 (Conditional Obligation)** A local obligation e is a tuple $\langle o, r, t, d \rangle$, where $o \in \{a, m\}$ and represents the type of the obligation. The elements c, t and d are propositional literals in L . r is the requirement of the obligation, t is the trigger of the obligation and d is the deadline of the obligation.

[53] The notation $e = O^o \langle r, t, d \rangle$ is used herein to represent a conditional obligation.

[54] The *requirement*, *trigger*, and *deadline* of a conditional obligation are evaluated against the states of a trace. Given a trace of a business process model and a conditional obligation, if a state of the trace satisfies the obligation's triggering element, then the obligation is set in force. Additionally, when an obligation is in force over a trace, and the deadline element is satisfied by a state of a trace, then the conditional obligation ceases to be in force.

[55] Fig. 3 graphically illustrates the in-force interval 300 of a conditional obligation over a trace of a business process model between a trigger 301 and a deadline 302. It is noted that the conditional obligation is encoded in one or more evaluation functions. Therefore, the evaluation function is said to be executed for that in-force interval between the trigger 301 and the deadline 302s.

[56] Note that when a conditional obligation is in force, then the requirement element of the obligation, i.e. the evaluation function, is evaluated within the in force interval 300 to determine whether the obligation is satisfied or violated in that particular in force interval. How the requirement element is evaluated depends on the type of the obligation. We consider two types of obligations, *achievement* and *maintenance*:

[57] Achievement: When this type of obligation is in force, the requirement specified by the regulation must be satisfied by at least one state within the in force interval, in other words, before the deadline is reached. When this is the case, the obligation in force is considered to be satisfied, otherwise it is violated.

[58] Maintenance: When this type of obligation is in force, the requirement must be satisfied *continuously* in every state of the in force interval, until the deadline is reached. Again, if this is the case, the obligation in force is satisfied, otherwise it is violated.

[59] Potentially multiple in force intervals of an obligation can co-exist at the same time. However, multiple in force intervals can be fulfilled by a single event happening in a trace.

Logic expressions

[60] Deontic defeasible logic may be more expressive than just control flow and may impose more complicated constraints not only on variables in isolated traces, but also across traces. As such, evaluation of rules across different traces in an event log is considered.

[61] Simply combining all possible permutations of traces to verify those rules is intractable, so there is a need for a different setup for a more efficient evaluation of a rule set R over multiple traces, where each trace may even originate from a different process definition.

[62] Method for analysing an event log

[63] Fig. 4 shows a graphical overview of an example of the method disclosed herein. The method consists of the following steps:

1. Order/sort the events from all traces in an event log as a continuous stream, ordered by timestamp.
2. Transform each rule r_i in rule set R into:
 - a (set of) variables V
 - a set of *update functions* U that update each respective variable $v \in V$ after execution of the event
 - a set of corresponding *evaluation functions* F that can be evaluated given the set of input variables, such that: if $\forall f_j \in F, f_j \rightarrow true$ then $r_i \rightarrow true$, else $r_i \rightarrow false$.
3. Replay the newly created stream, updating each $v_j \in V$ after every replayed event using the update functions $u_j \in U$.
4. Evaluate the relevant evaluation functions after a change of a variable.

[64] It is noted that the set of variables V refers to data objects stored on computer memory (volatile or non-volatile). They may be declared as integer, float, string, boolean or other data objects.

[65] A function refer to computer program code comprising one or more computer instructions grouped into a logical unit with one or more input variables and one or more output variables. These variables may be provided by reference or by value. The output variables the update functions U comprise the set of variables V . The set of variables V are the input variables of the evaluation functions.

[66] Fig. 5 provides a graphical overview of the sequentialised events of the event log depicted in Fig. 1, which are represented as a single trace in Fig. 6. Note that server 104 still keeps track of the originating trace ("pid:") an event belongs to.

Example

[67] Original rules:

1. A manager can work on only two instances of event B at the same time.
2. A has to be completed before B is allowed to start.

[68] Variables: $X_0(array)$, $X_1(bitstring)$, $X_2(bitstring)$

Where variable X_0 is used in the computation of rule 1 and variables X_1 and X_2 are used in computation of rule 2.

[69] Update functions:

- $e.name = B, e.lifecycle = start, e.resource = manager \Rightarrow X_0[e.mid] = X_0[e.mid] + 1$
- $e.name = B, e.lifecycle = complete, e.resource = manager \Rightarrow X_0[e.mid] = X_0[e.mid] - 1$
- $e.name = A, e.lifecycle = complete \Rightarrow X_1 = X_1 \vee 2^{p_{id}}$
- $e.name = B, e.lifecycle = start \Rightarrow X_2 = X_2 \vee 2^{p_{id}}$

[70] Where $2^{p_{id}}$ is the number two to the power of the process instance identifier, which is ascending starting from 1. In other words, $2^{p_{id}}$ is a bit-string with a one-hot encoding of the process instance. So the 'OR' operation ($\vee$) creates a bit string for the entire event log, where each bit represents the occurrence of a specific event for a trace. For example X_1 contains a set of bits that indicate for each bit whether for the respective P_{id} , A has been completed. As an example, an event log has 5 process

instances: P_{id} between 0 and 4. Then, server 104 creates the following bit string for X_1 : 00000. If for $P_{id} = 1$ A is completed, the bit string for X_1 would be: 00010. If then for $P_{id} = 2$ A is also completed, the bit string for X_1 would be: 00110. If B starts for $P_{id} = 4$, then the bit string of X_2 would be: 10000. Now if server 104 executes the evaluation function $X_1 \vee X_2 = X_1$, we obtain a bit string $X_1 \vee X_2$: 10110, which is not equal to $X_1 = 00110$ and therefore not compliant (B was started before A was completed for $P_{id} = 4$). In other words, in the evaluation functions below, a logical OR operation is then all that is required to evaluate whether the requirement has been fulfilled that A is before B. If not, the difference between the bit-strings will indicate the violating traces.

[71] Evaluation functions:

- $(\forall mid)(X_0[mid] \leq 2)$
- $X_1 \vee X_2 = X_1$

[72] An excerpt of the evolution of the variables after each replayed event is shown Fig. 7.

[73] The disclosed approach enables the use of more complicated logic to be checked and provides a highly efficient **cross-trace** and **cross-process** analysis. As such, advantages of this approach can be summarised as follows:

- Log compliance checking *across* different process instances
- Log compliance checking *across* different processes
- Automated rule conversion:

- Automated derivation of variables
- Automated transformation to update functions
- Automated transformation to evaluation functions
 - Automated replay and compliance evaluation using a combined trace

Update and evaluation functions

[74] As described above, server 104 executes update functions on each log event. These update functions update variables, such as by assigning a new number value or Boolean value to these variables. The update functions may be arithmetic functions, such as including but not limited to:

- incrementing or decrementing a variable value;
- adding to or subtracting from a variable value; or
- performing an iterative function on the variable value.

[75] It is noted that there are typically multiple variables that are being updated. But each variable is updated by different event types so that not every variable is updated at every event. For example, there may be 100 variables that are continuously updated by 1,000 events per second. In one example, the variables are updated after each log event is received and before the next log event is received.

[76] Then, server 104 can execute evaluation functions to test whether the variable value corresponds to the rules. For example, this evaluation can be test for:

- equal to an exact value
- greater than a given lower threshold

- less than a given upper threshold

[77] Again, server 104 can evaluate the evaluation functions after receiving each log event, which enables a real-time compliance check.

[78] In one example, there is exactly one evaluation function for each variable. However, there may be multiple evaluation functions for each variable. In further examples, there may be functions that combine two or more variables, such as, for example, to test that the sum of two variables is below a given threshold.

[79] As shown in Fig. 1, server 104 implicitly keeps track of active tasks. For example, at t_0 there is an event with event label 'A' and lifecycle 'start'. So this indicates that task 'A' has now started. Server 104 keeps track of this running task by maintaining a variable that is indicative of the number of parallel instances of task 'A'. So at time t_2 another instance of task 'A' starts. However, at time t_1 the first instance was already completed so only one instance is running at one time. Through the use of variables and update functions, server 104 can keep track of the number of task despite them running on different computers 101 and 102, respectively, and may potentially overlap in execution.

[80] So the way the server 104 represents tasks with a duration, is through start and stop events and then update variables at each start and stop event.

[81] In some examples, server 104 connects the start event with a resource, such as connecting a manager to a contract start event. In this example, there may be two managers who can both work on 5 contracts but if the first manager has 6 contracts and the second manager has 2 contracts, the system would be not compliant.

[82] It is noted that this approach significantly reduces complexities, such as compared to constructing an evaluation graph, because every variable would be connected to the event and the evaluation event. Further, the disclosed method prevents backtracing, which is also resource intensive.

[83] It is further noted that server 104 operates on a mapping between regulations and variables, such that the variables represent the salient characteristics of the regulations and such that update functions and evaluation functions can be formulated on the variables to represent the regulations. In other words, regulations are transformed into a set of variables, which together would be sufficient to provide a proof value indicating compliance of that particular event log.

[84] In further examples, each variable is connected to a specific event in the event log, which means there is a bridge between the event in the event log and the corresponding regulations. The update functions keeps that bridge updated, while progressing over the event log. The evaluation functions are a set of complicated procedures that extracts the truth value over the values to have real time update on compliance of event logs against regulation.

[85] In one example, the regulation is formulated in a logic form, such as deontic defeasible logic. The regulations could represent business rules, but are transformed to a logic form. These regulations can come from contract, business rules, or other sources.

[86] In further examples, the rules may comprise atomic literals that are the same as the variables to be updated by the update functions. In other examples, however, the atomic literals in the rules are not the same as the updated variables.

[87] In one example, the transformation comprises obtaining variables from a rule, manual mapping to determine the updated functions where event logs do not have standardised way of presenting variables. In other cases, where the use of variables is standardised, the transformation from rules to update functions and evaluation functions can be automated.

[88] In further example implementations, all variables are global over all processes, so that any variable that is updated as a result of a log event from a first process, can then be further updated as a result of a subsequent log event from a second process.

Evaluation function/predicates

[89] In one example, evaluation functions are modelled by evaluation predicates. An evaluation predicate captures the occurrence of a condition by means of a Boolean function that returns true when the condition is fulfilled and false otherwise. As such, the evaluation function is represented by an evaluation predicate that occurs at a specific point in time and that can be recorded together with that time.

[90] Evaluation predicates may be defined on a graph-like structure with logical connectives to evaluate complex conjunctions in a single iteration. This means that server 104 stores the graph-like structure and evaluates the graph-like structure for each event occurrence in order to calculate the value associated with the evaluation predicate.

[91] As a result of using a graph-structure, evaluation predicates may have precedence among each other. That is, a certain evaluation predicate may need to occur after another evaluation predicate. Further, evaluation predicates may also have restrictions. For example, a certain evaluation predicate may only be allowed to be considered once another evaluation predicate holds true.

[92] Finally, evaluation predicates allow the notion of deferred variables. This means that the required value of the variable is not known during design time and is to be obtained from the execution during evaluation.

Method

[93] Fig. 8 illustrates a method 800 for analysing log data. The method 800 may be performed by server 104. In that example, server 104 receives 801 the log data, which comprises multiple log events from multiple different processes. The log data may be in a text file, database or in the form of a data pipe or stream. Each of the multiple log events is associated with an event time indicative of when an event occurred that caused the generation of that log event.

[94] Server 104 creates 802 a single stream of log events comprising the multiple log events from the multiple different processes. The single stream of log events is sorted by the associated event time. This single stream (or 'trace') may be stored as a new text file, or may overwrite the original log file or may be kept on volatile storage, such as RAM or as database entry, such as records of a SQL database. In further examples, the original log data may be stored in any order in the SQL database and server 104 sorts the log events by issuing a 'SELECT * FROM logdata' command including an 'ORDER BY timestamp' directive.

[95] Server 104 iterates 803 over the single stream of log events, such as by calling a 'next' routine of a collection data object. For each log event, server 104 executes one or more update functions that define updates of the set of variables based on the log events. Thereby, server 104 calculates an updated value of one or more of the set of variables. For example, server 104 increments or decrements a variable that maintains the number of running instances of a particular process. At this point in method 800 server 104 may loop back to obtaining the next log event and iterate over only step 804 of executing the update function and then executes 805 one or more evaluation functions on the set of variables to determine compliance of the log data.

[96] In most examples, however, server 104 executes 805 the one or more evaluation functions after each update step 804. That is, there is iteration loop 807 shown as a dotted line which iterates from after executing the evaluation functions back to obtaining the next log event 803. This means that server 104 executes the evaluation function after processing each log event, that is, after each execution of the update function. This also means that an up-to-date compliance value is available after each log event.

[97] The one or more evaluation functions represent compliance rules based on the set of variables. For example, the evaluation function evaluates to TRUE if a condition is met, such as the number of running instances of a particular process is below an upper threshold. If there are multiple evaluation functions, the overall operation is compliant if all evaluation functions evaluate to TRUE 806. In other words, the

evaluation value of all evaluation functions are 'AND' connected to create a final TRUE/FALSE or 0/1 compliance value.

[98] It is noted that a hybrid version is also possible where for some log events the iteration loops back after the update step 804 and for other log events the iteration loops back after the evaluation function 805. This may be used to adjust the processing load in the sense that the number of evaluations can be reduced. For example, if the rate of log event generation is higher than a human can perceive a change in compliance value, such as a generation of log events of 1,000 per second, server 104 may execute the evaluation function at 805 once every second or after executing the update functions for 1,000 log events.

Applications

[99] The disclosed method may be used in a variety of different applications. For example, the disclosed method may be used to analyse logs of computer systems executing an operating system, such as iOS, Windows or Linux. In this scenario, there may be a common log file, stored persistently on non-volatile storage and new events are appended to the log file. More particularly, multiple processes run on the computer system (such as by executing different binaries of software programs) and each process appends log events to the common log file. Each process may append start and stop events and server 104 processes these as described above.

[100] In a further application scenario, there may be a complex high-reliance system, such as an aircraft or a nuclear power plant. Again, in these scenarios, there is a large number of interconnected modules that each generate log events. With the methods and systems disclosed herein, these operations can be monitored in real time and a granular assessment of proper functioning guaranteed. For example, there may be evaluation functions for different categories of events, such as, for an aircraft, there may be a set of evaluation functions for only the engines, such that the compliance of engines with respect to their operating ranges can be assessed separately from in-cabin

systems, for example. It is also possible to formulate complicated compliance conditions using the conditional obligation concept described above.

Computer system

[101] Fig. 9 illustrates a computer system 900 for monitoring compliance of another system 101. In that sense, computer system 900 is an example implementation of processing server 104 in Fig. 1. Computer system 900 monitors compliance of system 101 by analysing log data. Computer system 900 comprises a processor 901, program memory 902, data memory 903 and a communication port 904. Processor 901 is configured to perform method 800 in Fig. 8.

[102] More particularly, program memory 902 is a non-transitory memory with program code stored thereon. The program code is a software implementation of method 800 and may be stored in compiled form as a binary or in another interpretable format. Processor 901 reads the program code, which includes instructions for processor 901 to perform method 800. It is noted that log data, log events, update functions and evaluation functions as well as variables are data structures that are stored either on program memory 902 (for functions) or data memory 903 (for variables).

[103] Processor 901 receives the log data comprising multiple log events from multiple different processes, each of the multiple log events being associated with an event time. Processor 901 then creates a single stream of log events comprising the multiple log events from the multiple different processes, and stores the stream on data memory 903. The single stream of log events is sorted by the associated event time. Processor 901 then iterates over the single stream of log events. For each log event, processor 901 executes one or more update functions that define updates of the set of variables based on the log events to calculate an updated value of one or more of the set of variables. Finally, processor 901 executes one or more evaluation functions on the set of variables to determine compliance of the log data. The one or more evaluation functions represent compliance rules based on the set of variables.

[104] Processor 901 may also perform an action in response to the determined compliance. For example, processor 901 may initiate a mitigation action when non-compliance is detected, such as by raising an alarm or sending an alert message, or stop the operation of the other system 101 altogether.

Graph structure

[105] The following disclosure provides: 1) how the graph-like structure is defined and; 2) the method of interpreting the structure to decide whenever the modelled rule is compliant.

[106] The graph structure is given as,

- A set of collections, each collection comprising a set of update functions, evaluation functions and variables where each collection is grouped by a single variable type. For example, a collection may contain all update functions and evaluation functions that update a variable X.
- A set of logical operators. Predominately AND/OR, however, this may also include variations of these such as XOR.
- A precedence relation defined over collections and logical operators (i.e., edges between logical nodes and collection nodes).
- A restriction relation defined over collections that describe whether a collection should be considered (at the point in time) once another collection has "occurred".
- An entry point that describes what nodes should be initially considered.

[107] A precedence relation is not reflexive and each collection is only related to a logical operator. For the sake of convenience, it is allowable to feature an edge directly

between two collections. This is purely syntactic and can be interpreted implicitly as having an AND operator as a middleman.

[108] Here, the "occurrence" of a collection is defined as the point in time when a variable is updated to a value that satisfies the evaluation predicate and is no longer in force (i.e., we are looking for an event B and then observe an event B). A subtlety to note is that maintenance rules must be interpreted differently. For example, the "occurrence" of a collection that is compliant when a bank account balance is positive is meaningless if the account is always positive. In this case, the collection will be in force until the point of evaluation.

[109] It may be desirable to consider the negation of the occurrence of a collection as, for example, if a rule requires some reparation event to occur, then we do not want to record the variable's data twice.

[110] Each node in the graph structure may represent one of two types: a logical operator or a collection. Both types may be connected interchangeably through a precedence relation. A collection can be defined within multiple nodes. Each collection-based node may represent the negation of the collection. That is, if the representative collection occurs, we take the negation of the evaluation predicates as the result. From the entry point of the graph, the depth of a collection-based node establishes a precedence among each other. That is, certain collections may need to occur after another collection.

[111] The graph-like structure is evaluated left-to-right from the given entry point and continues until the graph-like structure is pronounced as "incompliant" or "compliant". Such outcomes are a consequence of whether the variable/s inside each collection satisfy all respective evaluation predicates, no longer in force, and satisfy the graphs logical structure.

Graph example 1

[112] Fig. 10 provides an example of a graph structure where the "Start" node represents the entry point, nodes "Invoice" and "Pay" are collections, nodes "or" and "and" are logical connectives, each black solid line represents a precedence relation and each dotted line represents a restriction relation.

[113] Note that in Fig. 10, each restriction relation is considered from the entry point. It is common to omit these lines for the sake of simplicity, and has been done so in Fig. 11.

[114] Hypothetically, one may have the following collections:

[115] Invoice

- Variable: X_0 (boolean)
- Update function: $e.name = invoice, e.lifecycle = complete \Rightarrow X_0 = true$
- Evaluation function: $X_0 = true$

[116] Pay

- Variable: X_1 (float)
- Update function: $e.name = payment, e.lifecycle = complete \Rightarrow X_1 = X_1 + e.amount$
- Evaluation function: $X_1 = 100$

[117] This example is small and is analogous to the evaluation function $\phi = \neg X_0 \vee (X_1 = 100)$. However, a difference to note is that ϕ is checked on every single update to X_1 (pay), and so there is a redundant check whether $\neg X_0$ is true every time X_1 changes. This is inefficient if for example there are several redundant checks being made on most or all events. In this case, a graph structure can circumvent this issue by: 1) listening to when a collection has occurred, and 2), at this point of occurrence, traversing the graph to check compliance of the rule by checking related collections.

[118] Furthermore, once a collection has occurred, the set of update functions contained within the collection can be discarded, further reducing the number of checks on each event. Similarly, there are cases where the currently used variables pertaining to each collection can be optionally discarded as well.

[119] Another benefit is that such graph structures can be easily manipulated to modify or extend a rule.

Graph example 2

[120] Fig. 11 illustrates an example where the graph cannot be reduced to a single evaluation function without using more complex update functions (i.e., update functions that depend on multiple variables). This example contains three collections: "A", "AtMostTwoB" and "AtMostTwoC" and is similar to the example given herein. The red dotted lines represent a restriction relation between collections "A" and "AtMostTwoB"/"AtMostTwoB" and specifies that these collections should only be updated once collection "A" has occurred. The graph structure encodes the following rules:

[121] Rules

- If event A occurs then a manager can work on at most two instances of event B at the same time. Only count the event B's that occur after event A.
- If event A occurs then a manager can work on at most two instances of event C at the same time. Only count the event B's that occur after event A.

[122] In this case, if collection "A" has not occurred then both collections "AtMostTwoB" and "AtMostTwoB" can be ignored, reducing the number of updates required. Therefore, we can iteratively traverse the graph as a means to keep in memory what collections are relevant and need to be considered.

[123] Without using such a structure, one may formulate the update functions as follows:

- Variables: X_0 (boolean), X_1 (integer), X_2 (integer)
- Update functions:
 - $e.name = A, e.lifecycle = complete \Rightarrow X_0 = true$
 - $e.name = B, e.lifecycle = start, X_0 = true \Rightarrow X_1 = X_1 + 1$
 - $e.name = B, e.lifecycle = complete, X_0 = true \Rightarrow X_1 = X_1 - 1$
 - $e.name = C, e.lifecycle = start, X_0 = true \Rightarrow X_2 = X_2 + 1$
 - $e.name = C, e.lifecycle = complete, X_0 = true \Rightarrow X_2 = X_2 - 1$
- Evaluation function: $\neg X_0 \vee (X_1 \leq 2 \wedge X_2 \leq 2)$

[124] However, it can be seen that if X_0 is false, then the last four update functions (i.e., ones updating X_1, X_2) do not need to be checked. These functions are checked on every event which leads to a lot of redundant checks.

Scaling the graph over a domain

[125] The graph structure enables scaling over a domain of instances. An example case is to consider all processes as the domain to scale over. If we consider the example herein, each manager corresponds to a unique process execution. However, since the method has all events sorted as a continuous stream, the method can consider other domains as well. For instance, a rule may want to verify that for each day, the maximum number of staff on a worksite must be less than 20. In this case, we want to verify the rule for each day and hence our domain is the set of all relevant days.

[126] Condensed notation has been developed to support this notion. In continuation of Graph Example 1, we may want to verify the rule for every invoice instance that is created. The graph representation for this is illustrated in Fig. 12. It is important to note that this notation is purely syntactic and represents an "infinite" conjunction of copies of Fig. 10, one for each id .

[127] In terms of implementation, this is viewed as creating a new instance for every invoice id. The complexity of this problem does increase when processor 901 considers more complex domains and have a collection that occurs for part of this domain. For example, consider the domain $(branch_id, manager_id)$ and a collection that updates a single branch - which is a one-to-many relationship with all managers in the branch. It is hard to guess at what "copies" pairs should be created, especially if we require some preemptive calculation.

[128] However, since this notation is purely syntactic, these additional complexities should already be captured by the graph structure definition.

Method of traversing the graph structure

[129] The method of traversal is an adaption of the method presented herein. A collection is a set of update functions, evaluation functions and variables where each collection is grouped by a single variable type (as defined above). Given a graph structure that models a rule set, the method can be briefly described as follows:

1. Maintain a set of collections C that are to be considered during evaluation of the event log. From the entry point of the graph, add each collection related to the entry point by a restriction relation.
2. Replay the stream of events, updating each collection in C using their respective update functions.
3. Evaluate the relevant evaluation functions after a change of variable. If a node in the graph "occurs", that is, the node is pronounced true based on its underlying collection, perform the following:
 - i. Update set C with the next set of collections that are required to be checked.
 - ii. Traverse the graph structure and validate the graph's logical structure to assess compliance. If the graph structure is pronounced as provably compliant or noncompliant, then terminate.

- iii. Otherwise, remove from C any collections that are no longer required to the search.
4. After completing iteration of the event log, repeat step 3 for any collection in C that is still undecided and whose in-force interval is specified as the point of evaluation.

[130] The set of collections C may be an object of a computer program, such as a class or variable instance, which occupies space in volatile random access memory (RAM), such that the collections in C can be accessed quickly without resorting to persistent block-storage, which is slow and inefficient for the read and write access of small pieces of data. However, as the set C grows, it can quickly outgrow the amount of RAM available on a typical computer system. By dynamically adding and removing collections to and from C, it is possible to keep the amount of RAM that is used at any one time, within the limits of physical RAM that is available.

Merge

[131] Instead of evaluating a set of rules R on a rule per rule basis (or in groups of rules), R can be merged into a single graph structure. The graph structure allows for state or collections to be shared, reducing the need to duplicate any computation and memory. Furthermore, there may be cases where update functions can be modified/decoupled to share state or collections more efficiently.

[132] It will be appreciated by persons skilled in the art that numerous variations and/or modifications may be made to the above-described embodiments, without departing from the broad general scope of the present disclosure. The present embodiments are, therefore, to be considered in all respects as illustrative and not restrictive.