
Peer reviewed version

Link to published version (if available):
10.1109/FUZZ-IEEE.2016.7737808

Link to publication record in Explore Bristol Research
PDF-document

This is the author accepted manuscript (AAM). The final published version (version of record) is available online via IEEE at http://ieeexplore.ieee.org/document/7737808/. Please refer to any applicable terms of use of the publisher.

University of Bristol - Explore Bristol Research

General rights

This document is made available in accordance with publisher policies. Please cite only the published version using the reference above. Full terms of use are available:
http://www.bristol.ac.uk/pure/about/ebr-terms
A Virtual Machine for Event Sequence Identification using Fuzzy Tolerance

Trevor Martin \(^{a,b}\)

\(^a\) Machine Intelligence and Uncertainty Lab, Engineering Maths, University of Bristol, Bristol, BS8 1UB, UK

Ben Azvine \(^b\)

\(^b\) Security Futures Lab

BT TSO

Adastral Park, Ipswich, IP5 3RE, UK

Abstract—Analysing event logs and identifying multiple overlapping sequences of events is an important task in web intelligence and in other applications involving data streams. It is ideally suited to a collaborative intelligence approach, where humans provide insight and machines perform the repetitive processing and data collection. A fuzzy approach allows flexible definition of the relations which link events into a sequence. In this paper we describe a virtual machine which enables a previously published expandable sequence pattern format to be represented as virtual machine instructions, which can filter event streams and identify fuzzily related sequences.

Keywords—Fuzzy Event Sequence Identification, Fuzzy Virtual Machine, Collaborative Intelligence

I. INTRODUCTION

Collaborative web intelligence is a combination of human expertise (to provide insight) with machine power (to provide repetitive processing and data gathering capabilities). Current web intelligence - in the form of applications such as search engines, recommender systems, e-commerce systems, etc. - is essentially machine-based, relying on the availability of a large quantity of data and sophisticated statistical machine learning methods to produce a predictive model. Such models have been successful in a range of fields, but can be criticised on a number of grounds. They generally do not enable human understanding of the underlying mechanisms, and exist essentially as black boxes where a set of attributes in a specific case leads to a predicted outcome for that case. Secondly, they rely on the existence of large collections of reliable data. We argue that statistical machine learning is not adequate in situations where human expertise is required (either to build or to understand the model of a process), or where reliable data is not available. For example, in detecting and combating cyber-attacks, reliance on statistical machine learning is often inadequate. Almost by definition, a successful cyber-attack needs to involve novel (hitherto unseen) features and is thus out of scope for systems which require large scale data collection - for example, spectrum.ieee.org/telecom/security/the-real-story-of-stuxnet describes how a number of so-called zero-day vulnerabilities were exploited.

In such cases, collaborative intelligence offers an improvement by combining the processing powers and visualisation provided by machines with the interpretive skills, insight and lateral thinking provided by human analysts. In order to successfully implement a collaborative intelligent system, it is necessary to exchange knowledge between the components - in particular between humans and machines. We argue that there is a fundamental difference in the knowledge representations, where machine processing is usually centred on well-defined entities and relations, ranging from the flat table structures of database systems through graph-based representations and up to ontological approaches involving formal logics. On the other hand, human language and communication is based on a degree of vagueness and ambiguity that leads to an efficient transmission of information between humans without the need for precise definition of every term used. Even quantities that can be measured precisely (height of a person or building, volume of a sound, amount of rainfall, colour of an object, etc.) are usually described in non-precise terms such as tall, loud, quite heavy, dark green, etc. More abstract properties such as beautiful landscape, delicious food, pleasant weather, clear documentation, corporate social responsibility, are essentially ill-defined, whether they are based on a holistic assessment or reduced to a combination of lower-level, measurable quantities. Zadeh’s initial formulation of fuzzy sets [1] was inspired primarily by the flexibility of definitions in natural language.

Linking events into sequences is an area in which collaborative intelligence can play a role. The notion of linkages between events is inherently uncertain in many cases - example such internet logs, physical access logs, transaction records, email and phone records all contain multiple overlapping sequences of events related by different attributes. Clearly in the case of phone records, the calls made by a specific user (or from a specific phone) would form a sequence - but it is often possible to link individual events in different ways. Specific problems in extracting sequences of related events include determination of what makes events “related”, how to find groups of “similar” sequences, identification of typical sequences, and detection of sequences that deviate from previous patterns. This is strongly linked to the concept of information granulation introduced by Zadeh [2] to formalise the process of dividing a group of objects into sub-groups (granules) based on “indistinguishability, similarity, proximity or functionality”. In this view, a granule is a fuzzy set whose members are (to a degree) equivalent. In a similar manner, humans are good at dividing events into related groups, both from the temporal perspective (event A occurred a few minutes before event B but involves the same entities) and from the...
Multiple sequences of events can be compactly represented using a directed graph (DASG), such that common initial and final sub-sequences are combined. User-supplied code defines the similarities between events (allowing them to be grouped together on the same path) and relations which indicate that one event follows another in a sequence.

We assume that a DASG representation of various ordered sequences of events is available, together with a suitable source of data and user-supplied code to categorise events and sequence steps. Each method in this code takes specified arguments from the data, performs appropriate computation (for example, to determine that two events are sufficiently close in time to belong to the same sequence, or whether one event is allowed to follow another). Each method returns a value in the range \([0, 1]\) using the normal fuzzy interpretation. Since the fuzzy matching is under control of the user, we do not cover this aspect in depth. It is important to note that fuzziness is fundamental to the virtual machine operation.

The virtual machine is capable of reading an interleaved series of events and matching them to the sequence patterns, producing (on demand) lists of event sequences that match complete sequence patterns, lists of event sequences that match initial phases of sequence patterns, and lists of events that do not match any known patterns. The process of translating the DASG to virtual machine code is not described here, but can be achieved using standard techniques.

### B. Related Research

The major research fields linked to this work are

(i) compilation of finite state machines (FSM) into executable code or a virtual machine. This is a well-developed area of computer science and is used extensively in compilers, string processing, speech recognition, etc. See [5] for a tutorial description of finite state machines. Open source software such as [http://smc.sourceforge.net](http://smc.sourceforge.net) exists to convert FSM specifications into most common programming languages.

(ii) implementation of virtual machines for logic programs, such as the Warren Abstract Machine (WAM) [6]. The Fril Abstract Machine [7, 8] is of particular relevance to the work described here as the representation used in SIFT incorporates fuzzy matching. Fril is the only logic programming compiler with a mechanism to handle uncertainty integrated at the lowest level.

(iii) compilation of graphical models, although the notion of "compilation" here refers to translation into library routine calls, rather than to a dedicated virtual machine [9].

Our work uses a much more general representation than a finite state machine. In particular, the use of fuzzy values and multiple labels to define an edge distinguishes the approach from finite state machines. Fuzzy labels mean it may be necessary to revisit a node and test alternative edges from it. Multiple labels allow more complex branching behaviour than is possible in a finite state machine. As a consequence of these differences, the execution model described here is very different to a finite state machine. The notion of restarting computation and multiple threaded execution is not common in finite state machines.
Additionally, it is a relatively simple task to dynamically alter the virtual machine code to reflect changes in the DASG model of event sequences. There is a close (essentially, one-to-one) correspondence between the DASG representation of event sequences and sections of code for the virtual machine. Broadly speaking, edges correspond to short instruction sequences and nodes correspond to points at which execution may be suspended.

Finally, the virtual machine allows reconstruction of event data corresponding to recognised sequences and to unrecognised sequences by examination of the thread execution records.

Virtual machines for logic programs are complex, reflecting the fact that they implement complete programming languages. This DASG machine is a much simpler design, meaning that the use of multiple execution threads is easier.

Graphical models (particularly Bayesian nets) are normally implemented using specialist code, and “compilation” in this context typically refers to a translation process, whereby a specification of a graphical model is converted to a sequence of program calls to pre-written library functions. Initial construction of the graphical model and its use in simulation is reliant on the user’s statistical knowledge, the collection of large quantities of data and an assumption that past performance can be used to predict future behaviour. In contrast, the work described here does not rely on statistics to form the initial network of event sequences, and allows the graph to reconfigure easily, as new patterns are incorporated. The virtual machine gives a simple execution model corresponding to the DASG and does not rely on complex library functions.

C. Sample Data

A small subset of data from the 2009 VAST challenge was used in [3] to illustrate DASG formation. A similar subset (Table 1) is used here, but events are listed in time order (and event IDs are changed to reflect the ordering). To illustrate features of the system, row 8 has been changed so that the sequence for employee 10 no longer matches any pattern and an event has been added at row 20 which cannot be matched to any initial pattern step. The data is drawn from attributes:

- Employee = set of employee ids = \{10, 11, 12\}
- Date, Time = date / time of event
- Entrance points = \{B - building, C - classified section\}
- Access direction = \{in, out\}

and represents movement of employees in and out of building (B) with a swipecard barrier on entrance but not on exit. The building contains a classified area (C) with swipecard access on entrance and exit. Tailgating (following another employee without swiping a card) is possible. We use the same user-defined relations as [3]. For a candidate sequence of \( n \) events:

\[ S_1 = (o_{11}, o_{12}, \ldots, o_{1n}) \]

we define the following computed quantities:

- \( \text{ElapsedTime} \)
- \( \Delta T_{ij} = \text{Time} (o_{ij}) - \text{Time} (o_{i,j-1}) \)
- \( \Delta T_{i1} = \text{Time} (o_{i1}) \)

and restrictions (for \( j > 1 \)):

\[ \text{Date} (o_{ij}) = \text{Date} (o_{i,j-1}) \]
\[ 0 < \text{Time} (o_{ij}) - \text{Time} (o_{i,j-1}) \leq T_{thresh} \]
\[ \text{Emp} (o_{ij}) = \text{Emp} (o_{i,j-1}) \]
\[ (\text{Action} (o_{i,j-1}), \text{Action} (o_{ij})) \in \text{AllowedActions} \]

where \( \text{Action} (o_{ij}) = (\text{Entrance} (o_{ij}), \text{Direction} (o_{ij})) \)

and \( T_{thresh} \) specifies how close events must be to form part of the same sequence. The relation \( \text{AllowedActions} \) is given by the following table (row = first action, column = next action):

<table>
<thead>
<tr>
<th>Action</th>
<th>b,in</th>
<th>c,in</th>
<th>c,out</th>
</tr>
</thead>
<tbody>
<tr>
<td>b,in</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>c,in</td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>c,out</td>
<td>x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

These constraints can be summarised as:

- events in a single sequence refer to the same employee
- successive events in a single sequence conform to allowed transitions between locations and are on the same day, within a specified time of each other. We choose \( T_{thresh} = 8 \) (this ensures anything more than 8 hours after the last event is a new sequence). Note that

1 http://hcil2.cs.umd.edu/newvarepository/benchmarks.php
the allowed transitions are defined by a human expert. In an environment where “tailgating” occurs commonly, it is likely that learning from data would see this as normal behaviour.

We see from events 2, 3, 4, 7, 9 that employee 11 enters the building at approx 9:00 (rounding times to the hour), enters the classified area at 10:00 and leaves it after 3 hours, re-enters at 14:00, leaving 2 hours later. This corresponds to path S-3-6-12-13-14-E in the graph (Fig 2).

III. THE VIRTUAL MACHINE

The DASG is compiled to a sequence of instructions for the virtual machine. The instructions contain an initialisation section (labelled $LS$), an acceptance section (labelled $L0$), and code corresponding to the nodes and edges in the graph. A distinguished code label $<E>$ denotes the final edge in the graph. Given a valid graph, the corresponding virtual machine code can be generated straightforwardly. It is easy to make small optimisations (such as re-ordering operations so that instructions most likely to fail are executed first). Other obvious enhancements (not described here) include

- use of an index table or switching code to select best threads, given event data
- time-out scheduler (assuming events arrive in real time or in temporal order, this causes threads to fail when the time since their last event exceeds a threshold)
- addition of arbitrary code to graph nodes (for instance, to raise an alert)

Each sequence of events is represented by an execution thread. A partially recognised sequence corresponds to a suspended or executing thread; fully recognised sequences or rejected (unrecognised) sequences correspond to terminated threads.

A thread is represented by a small set of registers and a stack plus queue, and suspends execution once it has consumed relevant data. Execution of a thread terminates successfully if a complete sequence is identified. Unsuccessful termination represents a set of events which was not recognised as a sequence. Lists of executing / suspended threads and terminated threads are maintained. If not terminated, the thread is either executing or suspended (on the open sequence list). The return values from thread execution are:

- SUSPENDED-SUCCESS
- SUSPENDED-FAIL
- TERMINATE-SUCCESS
- TERMINATE-FAIL

Each thread has an associated degree of match, depending on how well the event data matches the fuzzy patterns used to describe the sequence. If this value falls below a specified threshold during thread execution, the computation is unwound and restarted at a previously unconsidered path from a branching node (with outdegree >2).

The virtual machine consists of registers, storage areas for runtime structures (stacks etc.) and code made up of instructions which operate on the registers and storage. The registers and runtime structures are described below, with the virtual machine instructions listed in Fig 1.

Registers

$\text{args}[0...n-1, n...m]$ (typed) argument registers corresponding to a row of event data. Registers from $n$ upwards are used as working storage but are not saved on the stack

$N$ NextChoice : instruction label, gives alternative execution address if current instruction fails. Can be null.

$C$ ContinuationInstruction instruction label, indicating the next step for execution when new data arrives and is accepted by this thread. Can be null

$M$ MatchDegree number in the interval $[0,1]$ giving the membership of the sequence on its matched path. Set by XOF instruction.

$CP$ code pointer, indicates current instruction (not saved on stack)

$StackTop$ top frame on stack.

$UR$ UserReturn : returned result (match) from user code

$TS$ thread status

$nodeArgs[0...n-1]$ saved arguments in top stack frame, accessed via StackTop

$n$ number of arguments in data table

$types[0...n-1]$ data types in the data table

$Threshold$ minimum value for $M$ (MatchDegree)

Runtime Structures

$Stack$ storage area for execution records (last in, first out stack)

$RematchQueue$ 0 or more sets of $n-1$ argument registers stored as a queue (first in, first out).

$USL$ unidentified sequence list

$OSL$ open sequence list

$ISL$ identified sequence list

Each of the lists $USL$, $OSL$, $ISL$ is initially empty and enables addition and removal of specified sequence threads from the list.

IV. VIRTUAL MACHINE EXECUTION

Execution consists of the steps shown in Fig 3, for each data row. For simplicity, the algorithm does not cater for threads that “time out” i.e. partial event sequences that were last modified at a point exceeding the time threshold. With the assumption that all events arrive in the correct temporal order, a simple extension to the execution model described makes it possible to identify threads that can no longer be extended and they can be failed (and moved to the USL).
DFR \textit{DequeueFromRematch} \\
Copy content of argument registers from the front of rematch queue, and de-allocate the space used.

EXEC \texttt{<label>}  \\
If \texttt{<label>} is null, execute \texttt{FAIL}. Otherwise, continue execution from the instruction labelled by \texttt{<label>}.

FAIL  \\
reset \textit{MatchDegree} to value saved in top stack frame  \\
IF \texttt{<NextChoice>} is not null THEN  \\
continue execution from address given by \texttt{<NextChoice>}  \\
ELSE IF \texttt{<NextChoice>} is null THEN  \\
\hspace{1em} \texttt{DO} \hspace{1em} \texttt{pop stack}  \\
\hspace{1em} \texttt{UNTIL} \texttt{NextChoice} is non-null or stack is empty  \\
\hspace{1em} IF \texttt{NextChoice} is not null THEN  \\
\hspace{2em} continue execution from \texttt{NextChoice}  \\
\hspace{1em} ELSE IF \texttt{stack} is empty THEN  \\
\hspace{2em} return \texttt{TERMINATE -FAIL}  \\
\hspace{1em} ENDIF  \\
ENDIF

POP  \\
Reset registers N, C, M and args to saved values  \\
set \texttt{StackFrame} to previous frame  \\
\texttt{QueueOnRematch (QOR)}  \\
\hspace{1em} \texttt{// copy saved args[0…n-1] to rematch queue }

PUSH  \\
Allocate \texttt{StackFrame} and save registers N, C, M plus \( n \) arguments

QOR \textit{QueueOnRematch}  \\
Allocate space for argument registers 0…\( n-1 \) at the back of the rematch queue and copy content of argument registers to the newly allocated space

RIF \texttt{<userMethod(typed arguments)> RejectIfFailure}  \\
Executes \texttt{userMethod} on the specified arguments. If the result of \texttt{userMethod} is \( \leq \text{Threshold} \), the thread suspends and returns the value \texttt{SUSPENDED-FAIL}

SLN \textit{SaveLiveNode}  \\
IF rematch queue is not empty THEN  \\
\hspace{1em} \texttt{DFR}  \\
\hspace{1em} \texttt{EXEC Contin}  \\
ELSE If rematch queue is empty,  \\
\hspace{1em} \texttt{PUSH}  \\
\hspace{2em} \texttt{IF} \texttt{Contin} register is \(<E>\) THEN  \\
\hspace{3em} terminate execution  \\
\hspace{3em} return \texttt{TERMINATE-SUCCESS}  \\
\hspace{2em} ELSE  \\
\hspace{3em} suspend execution  \\
\hspace{3em} return \texttt{SUSPEND-SUCCESS}  \\
\hspace{1em} ENDIF  \\
ENDIF

TNA \textit{ThereIs No Alternative}  \\
Writes NULL into \texttt{NextChoice} register

XOF \texttt{<userMethod(typed arguments)> ExtendOrFail}  \\
Executes user method with specified arguments (from arg registers and nodeArg registers).  \\
Return value is a number in the range \([0,1]\) representing data match.  \\
If return value is \( \leq \text{Threshold} \), execute \texttt{<FAIL>}  \\
Otherwise set \textit{MatchDegree} = \texttt{min(return value, MatchDegree)} and continue with next instruction.

XWA \texttt{<L1> <L2> ExecuteWithAlternative}  \\
Writes \texttt{<L2>} into \texttt{NextChoice} register and passes control to \texttt{<L1>}

Fig. 1. Virtual machine instructions (listed alphabetically by abbreviated code), with abbreviated code, longer descriptive name if appropriate, arguments, and a brief description.
A. Worked Example

We represent the sequences in Table 1 as a minimal DASG with edges labelled by event categorisations (see Fig 3).

This corresponds to the following virtual machine code. Labels (e.g. L1) correspond to graph nodes and comments are delimited by // and end of line.

L0: // Thread Acceptance Code
    RIF equalityCheck(a[3], nodeArgs[3])
    // accept if day and emp-id match
    RIF equalityCheck(a[1], nodeArgs[1])
    a[7] = elapsedTime(a[2], n[2])
    RIF lessEqCheck(a[7], 8)
    // Time threshold = 8
    RIF allowedAction(a[4], a[5], nodeArgs[4], nodeArgs[5])
    EXEC Contin
    // accepted - match to next edge

LS1: XOF equivCheck (a[2], 7)
     C = L1
     SaveLiveNode

LS2:
    TNA
    XOF equivCheck(a[2], 9)
    C = L5
    SaveLiveNode

L1:
    TNA
    XOF equivCheck(a[2], 13)
    XOF equalityCheck(a[4], b)
    XOF equalityCheck(a[5], in)
    C = L2
    SaveLiveNode

L2: etc

L3:
    TNA
    XOF equivCheck(a[7], 0)
    XOF equalityCheck(a[4], c)
    XOF equalityCheck(a[5], out)
    Contin = L6
    SaveLiveNode

L4: etc

L5:
    TNA
    XOF equivCheck(a[2], 10)
    XOF equalityCheck(a[4], c)
    XOF equalityCheck(a[5], in)
    Contin = L6
    SaveLiveNode

L6:
    XOF equalityCheck(a[4], c)
    XOF equalityCheck(a[5], out)
    a[7] = elapsedTime(a[2], n[2])
    XWA L6A, L6B

L6A:
    XOF equivCheck(a[7], 2)
    Contin = L7
    SaveLiveNode

L6B:
    TNA
    XOF equivCheck(a[7], 3)
    contin = L7
    SaveLiveNode
    etc

After three rows of data have been read, the virtual machine state is shown in Fig 4. The top section of the figure shows the content of the sequence lists OSL, USL, ISL (respectively,
open, unidentified and identified sequence lists. The status of registers for each thread is shown below, labelled T1, T2, ...
(for first thread, second thread etc).

OSL  T3
USL  T1
ISL  T2

Thread T1
Stack : Empty

RematchQueue : empty

Thread T2
Stack

RematchQueue : empty

Thread T3
Stack

RematchQueue : empty

READ data for next event into argument registers a[0 … n-1]
SET ThreadStatus to SUSPENDED-FAIL
WHILE (ThreadStatus == SUSPENDED-FAIL)
   IF (OSL contains untried threads) THEN
      select an untried thread and remove it from the OSL
      ThreadStatus = execute thread from L0
   ELSE
      // i.e. no threads accepted data
      create new thread
      ThreadStatus = execute thread starting from LS
   ENDIF
   IF ThreadStatus == TERMINATE-SUCCESS THEN
      add thread to ISL
   ELSE IF ThreadStatus == TERMINATE-FAIL THEN
      add thread to USL
   ELSE IF ThreadStatus == SUSPENDED-SUCCESS THEN
      add thread to OSL
   ELSE IF ThreadStatus == SUSPENDED-FAIL THEN
      add thread to OSL
   ENDIF
ENDWHILE

Figs 5 and 6 show the machine state after 10 and 20 rows (respectively) have been read. After 10 rows (Fig 5), thread 2
has terminated - this corresponds to the path S-5-6-12-13-14-E
in the graph (Fig 2). This will be recognised on reading
subsequent data when the threshold time will be exceeded. At
this stage, thread 2 can be moved to a record of completed
threads (sequences), processed to extract relevant data, or
simply discarded according to the task requirements.

Thread T1
Stack : Empty

RematchQueue

Thread T2
Stack

RematchQueue : empty

Thread T3
Stack

RematchQueue : empty

Fig 3  Execution steps for each row of data

Fig 4  Machine State after reading three rows of data

Fig 5  Machine State after reading ten rows of data
### Thread T1
(as before)

### Thread T2
(as before)

### Thread T3

**Stack**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>&lt;E&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>L3</td>
<td>1</td>
<td>18</td>
<td>Jan-3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>L6B</td>
<td>1</td>
<td>14</td>
<td>Jan-3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>L6</td>
<td>1</td>
<td>12</td>
<td>Jan-3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>L5</td>
<td>1</td>
<td>10</td>
<td>Jan-3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**RematchQueue**: empty

### Thread T4

**Stack**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>&lt;E&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>L13</td>
<td>1</td>
<td>19</td>
<td>Jan-3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>L12</td>
<td>1</td>
<td>17</td>
<td>Jan-3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>L6</td>
<td>1</td>
<td>13</td>
<td>Jan-3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>L5</td>
<td>1</td>
<td>11</td>
<td>Jan-3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**RematchQueue**: empty

### Thread T5

**Stack**: Empty

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**RematchQueue**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>Jan-4</td>
<td>06:00</td>
<td>12</td>
<td>b</td>
<td>in</td>
</tr>
</tbody>
</table>

![Fig 6](image-url) Machine state after reading 20 rows

### V. Summary

Defining and recognising meaningful event sequences is a complex task which often requires human expertise to group attributes and events into related categories, which tend to be fuzzy in nature. It is a key task in analysing many data sources, including activity logs of users interacting with web applications - and hence, it is a key enabler for web intelligence. Our previous work has described a way of storing event sequences in a compact directed graph format, providing an efficient incremental algorithm to update the graph with an unseen sequence. A human expert can easily add sequence patterns, even if these have not been seen in the data yet. This aspect particularly distinguishes our work from statistical machine learning. The work described in this paper illustrates how a virtual machine can be defined from the directed graph representation, enabling event streams to be filtered and classified according to the sequences identified. The virtual machine can be implemented in software or by means of configurable hardware.

### References


