Analysis of Temporal Data

Muhammad Afan Shah
LRI & INRIA Futurs
Bâtiment 490, Université Paris-Sud
91405 Orsay Cedex, France
shah@lri.fr

ABSTRACT

The main objective of this internship is to design and implement a software model permitting to do analysis on stream of events which describe daily activities of a user. The interest of a complete interaction history between a user and its data is to permit a user to use his own episodic memory (and non semantic) and use it as a search criteria for elements of contextual nature. Therefore, the objective of this internship consists of extending the work on multimedia streams initiated by M. Beaudouin-Lafon and W. Mackay to adapt to the system proposed, and to analyse the data collected by the observers of Micromegas. Including other classic operations on streams (for example temporal filtering, insertion and deletion of events), along with the operations permitting aggregation of events with an objective of being able to work on different scales are also included in this internship.

We present an activity algebra model which is based on the hierarchical relationships between activities and events. The model permits to resolve different activity overlapping issues and provides event-based and time-based filtering methods for data aggregation and multiscale viewing. We also provide details of its implementation and we believe that this model should prove useful for a wide variety of applications.

Categories and Subject Descriptors

H.3.4 [**Information Search and Retrieval**]: Information filtering

General Terms

Algorithms, Design, Documentation

Keywords

Familiar data, Interaction Histories, Temporal data, data streams, activity graphs, activity algebra, context.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Permission may be requested from Université de Paris-Sud, Orsay, France. © 2005

1. INTRODUCTION

As users use their computers, they produce and acquire lots of data in the form of documents, web page histories, emails, video, audio, images, etc. Given the personal and professional diversity of these data, we call them "familiar data", exposing the idea that they are familiar to a user. According to certain studies, soon it will be useless to delete files "to make more space" [11]: so we must be able to save whatever comes in our hands rather than judging what is worthy of interest.

Micromegas[1] is a research project sponsored by the French ACI Masses de Données, where different efforts have been made to monitor user interaction with his familiar data on daily basis (more details about Micromegas will be described later). Reading this type of raw data or events to analyze a user's interaction is a cumbersome process. One solution is to load data in the form of a linear data streams and then apply procedures to find events based on user queries. However, this process of searching for events is a daunting task considering the diversity of the data and it becomes inadequate to integrate context within the search query. Consider an image received by the author on 20 august, which also served in the creation of another document on 25 august, that document was then modified on 29 august. This type of history permits to use whatever form of memories to retrieve that image (e.g. that he manipulated in the last week of august) with respect to all other data manipulated in the close context (e.g. when he submitted his final report). Thus a context is any information that can be used to characterize the situation of an entity. An entity is a person, a computing device or a non computing physical object such as an activity that is considered relevant to the interaction between a user and an application

The raw data can be grouped based upon their activities; for example, the activity of writing a report should contain all the events which took place to fulfill that task. Many existing activity based systems rely on the traditional view of activities as a linear temporal medium. They do not take full advantage of either the logical structure of the activities or of hierarchical relationships between activity segments. We take the advantages of both worlds and present an algebraic activity data model that has,

[1] http://insitu.lri.fr/micromegas/

- The nested user defined activity hierarchies containing data items as events
- A temporal order based on the captured data
- Fast data retrieval algorithms based on keywords
- Multiple views of the same data based on temporal filtering
- Associative access based on the content and temporal information

For example, consider a user that wants to do some analysis on his usage data. First, the user will specify activities as inputs which can be created manually or loaded from some calendar based application. Or else the user can create activities in advance. Then, the system will automatically arrange these activities based upon their temporal ordering and fill them with events that occur during their temporal intervals. Finally, the user can manipulate these activities by insertion, deletion or edition and do analysis based on the time-based and event-based filtering.

The algebraic activity model uses the same stream metaphor as in DIVA [21], which was created by Wendy E. Mackay and Michel Beaudouin-Lafon. The multimedia streams introduced in DIVA enables users to visualize, explore, analyze and evaluate patterns of data that change over time.

This new model captures the stream as well as the hierarchical model to represent data. We enhance the operators of DIVA for our algebra to effectively handle activities. The data model is a hierarchical composition of algebraic activity nodes and pointers to events in the data stream. Therefore each event in the stream will be identified by at least one activity node. We introduce activity algebra as a means for manipulating activity nodes, for retaining temporal ordering, and for associating descriptive information with these nodes. Each of the activity nodes preserves the correspondence between stream events so that all relevant events and their neighbors can be efficiently found.

For example, consider an activity A that was initially inserted in the root node. The activity A already holds many events of the stream and has a definitive start and end positions. Now suppose another activity B comes in whose start position is less then the ending position of A, then in this case A and B both share some portion and all the events that are present during that particular portion. We call this portion A ∩ B because it contains all the common elements between the two activities. If A shares something with B then it also has something which belongs only to itself which is the part A - A ∩ B. Same goes with B as well. Finally we come to the graph structure of nodes where each event is identified precisely by one activity node as depicted in figure 1.

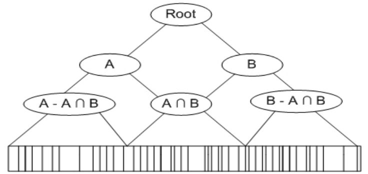

Figure 1 Activities and events

The events representing familiar data have a temporal nature, which allows us to formulate a timeline in which these data follow sequentially one after another. The earlier systems based on timelines as in [6, 21], create separate streams of data for each data type and observe their values as a function of time. The problem with this type of approach is that each stream behaves independently of the others, thus it becomes very difficult to identify relationships between them. For example, a file stream will show all the interactions related to a file system, like file opened, closed, deleted, etc and a network stream will show network based activities like urls. Now if we filter both streams at any time interval then we can find all the files and urls visited during that interval but we can't possibly argue about the purpose or need for which they were used.

This type of problem is encountered usually when a user remembers things based on their context or relationship with other things. For example, Adam, a regular book reader, might read fifty books in a year and he may not remember about the names and authors of each of his book. Therefore to find a book he would recall it by its context and activities close to it. Like the book given to him on his birthday or one of the books that he got from Joe or the book which he read in March. Now each of these contexts provide a clue to Adam from where he can proceed with his search and which would restrict him to only a small set of books rather than all of the fifty.

This context based searching is possible in the algebraic activity model because of the nested stratification provided in terms of the hierarchical relations between the activity nodes. The activity overlapping allows multiple coexisting views on the same data allowing users to reach to it through different means.

The model has been implemented as a part of the Micromegas project. It uses the data which was collected during the interaction between a user and its familiar data. Our algorithms prove efficient enough to quickly search for data as described by the user.

The following sections explain in some detail about the Micromegas project and the DIVA system.

1.1 The Micromegas project:

The project involves four research groups: The LMP (Laboratoire Mouvement et Perception) at Marseille supports its expertise in movement psychology and perception applied for human-computer interaction, the In-Situ (LRI & INRIA Futurs) deals

with the information visualization, interface engineering and participative designing, the MErLIn (INRIA Rocquencourt & LORIA) works on cognitive ergonomics for human computer interaction, and finally Pasteur Institute works on the computer tools for the use of biologist.

In the context of this project, In Situ is investigating the recording and analysis of user interactions with a system: mails received and sent, files manipulated, documents printed, web pages consulted, applications used, organization of windows on the screen and desktop icons etc. As a part of this project we are building a general-purpose prototype for analyzing such data based on their temporal orderings.

The data regarding to the daily usage of a system is captured by using various observers: fwatch, wmwatch, netwatch and mozilla.

- The first observer is used to observe different activities related to files and can be used for recording different attributes (for example, file name, path, accessed time, size, etc) of a file when it is accessed.
- All of the information related with the interaction of a user and the window system is captured by the second observer. This observer captures the information of the processes running in a window with a detailed description of window state (e.g. window operations like minimizing, maximizing, etc), selection and copying of text in a window, and different attributes pertaining to a window. The data generated by this observer will help to analyze what the user was doing at a particular instance.
- With the help of the third observer, the input and output at different ports can be observed permitting to deduce a detailed history of web pages visited, downloaded files using the ftp protocol, chatting sessions, and files printed. This observer can also provide useful data regarding different virus attacks or reading of audio/ video streams.
- Many web browsers don't allow reading of the content of a visited page and just capture its URL, however with the modification of the source code of mozilla browser these activities are now monitored by the fourth observer. This observer also captures detailed information about emails for example their subject, author, download time, view time and action performed after any particular email.

All of these observers record user's activities and store them in a database. The transfer of data from these observers is facilitated by an activity manager, which is an application created in C++. The activity manager loads the data from the data base into memory and allows the execution and visualization of interactive queries in an efficient way.

Although, Micromegas provide the necessary tools and methods to visualize past data on a timeline however it lacks the notion of solving queries based on context and episodic memories as discussed in the previous section.

The events captured by these observers are stored in a database, which are then extracted into log files and analyzed by our system. Although our model has been created based on the need of the temporal analysis for such, we have kept it general enough to be used in other applications for example calendars.

1.2 DIVA:

DIVA supports exploratory data analysis of multimedia streams, enabling users to visualize, explore and evaluate patterns in data that change over time. The underlying stream algebra provides the mathematical basis for operating on diverse kinds of streams. The streamer visualization technique provides a smooth transition between spatial and temporal views of the data. Mapping source and presentation streams into a two-dimensional space provides users with a direct manipulation, non-temporal interface for viewing and editing streams.

It consisted of some predefined streams of type boolean or text each having a variable that allows the users to realize what they were doing at a particular instance. So each stream is controlled by only one variable and the value of that variable at any time determined its status in the timeline. User could also explicitly create more streams if they want to observe other activities. Thus, each new stream extended the vertical space creating visualization problems and making it impractical when there are plenty of activities to be monitored. For example, if we were to monitor file system activities of a user interaction then we have to create a new boolean stream for each file and define true if it was open or false otherwise. Thus the main problem with DIVA in the context of Micromegas is that it would require a huge number of boolean streams to describe the user interactions.

The stream model of DIVA provides the groundwork for constructing a stream based method for the temporal analysis. By extending the stream concept, we present a novel approach for the temporal analysis of familiar data that have been stored by the user themselves, who not only save their own productions but collect external data. Our approach is based on the idea that there is a temporal relationship between the activities that users perform on their systems and these activities can overlap each other producing complex relationships. Each activity can contain different events that took place during its time period. We define an activity as a collection of heterogeneous events spanning over a time interval. Our classification between an activity and an event is that the former has different start and end time while the later has both the same.

By limiting an event to have the same start and end times we effectively view it as an instantaneous fact i.e. something occurring at an instant, where an instant is a time point on an underlying time axis.

Each activity can contain other nested activities and share all those events that are defined in them. This structuring of activities allows us to define relationships between them and it makes the searching process much easier. Finally based upon their time durations, all activities form a logical mapping over a unique stream thereby giving a timeline representation of all the activities and events that the user has performed. A query on any given time interval will filter those activities or events that happened during that interval.

Our model uses stream algebra for the manipulation of data within streams and we provide operations like insertion, deletion, updates, event and time filtering.

The rest of this report is organized as follows. Section two presents a brief overview of related works. Section three describes the structure of our model and its basic ideas. Section four presents the algorithms for the underlying activity algebra by giving detailed explanation of each of its operations. Section five presents our conclusions and future directions for this work.

2. RELATED WORK

The research context of our work is related to stream based temporal systems, multiscale interfaces, activity based systems, graph structures and data mining.

The concept of data streams has been employed in different domains for example data mining. [13], for example discusses techniques for the analysis of weblogs, queries to web search engines, and usage data at high traffic websites. Their analysis techniques are based on the extraction of abstract structures present in a data streams. The difference between there approach and ours is that we use exploratory data analysis while they employ data mining techniques. [14] presents a formal approach for defining digital libraries based on streams, structures, spaces, scenarios and societies (5S). They define a digital object which has a hierarchical tree like structure and provide direct mappings from its nodes to other stream segments. Different digital objects can be grouped together to form one digital library and then mapped over many data streams. This is different from our approach where we use user defined activities, determine their overlapping, dynamically update the internal data structure to incorporate future changes and finally map these activities over a single data stream.

The Stanford Data Stream Management System (STREAM) [15] gives a prototype based on solving continuous queries over a data stream. The model that they present is based on data streams and relations. There are three basic types of operators i.e. relation to relation, stream to relation and relation to stream. Different operators are then defined on top of them such as select, project, join, union, etc. The difference between stream and relation is that the former takes only inserted elements while the later takes deleted as well. In our work, we are not concerned with continuous queries, although we can simulate this effect by executing queries in cycles with user intervention. The data stream in our model contains events that are associated with actions taken by a user, which are mapped directly to the documents. Therefore there is no need to store the event insertion and deletion in the data stream at the same time. The operators that we define are extended from DIVA [21] and are directly applicable to the data stream whereas STREAM defines its operators semantically in terms of relational operators and applies them over a combination of streams and relations.

Defining a multiscale interface was not our objective but getting information at different scales was the area we focused. To further facilitate the designing of a multiscaler interface obeying the theoretical models presented by [16, 17, and 18]; we create different views of our data stream based on its temporal structure. For example, by filtering activities in terms of years, months and weeks would be helpful to design a zoomable interface obeying the model presented by [16].

[19] deals with the problem of recognizing frequent episodes in sequences of events based on data collected for describing the behaviour and actions of a user or systems. This identification of pattern is performed by constructing a graph from one a particular action and then matching it with other actions. We on the other hand search for events based on their parent activities and we don't employ the notion of similar events. Pattern matching could easily be added to our event filtering implementation. Yet, we chose to restraine ourselves from going into those complexities.

[20] presents an index structure called Temporal R-Tree that deals with spatiotemporal data in GIS applications. There approach is based on R-Trees [10] and they allow the retrieval of present and past states of data. The focus of [20] is on the dynamic updating of an internal structure that is very similar to R-Trees but includes temporal nature. TR-Tree is basically an acyclic graph like structure defined by a root node; all the leaves in this graph are limited to fixed number of indices. After reaching this limit, new nodes are created and previous data is copied into these nodes. The TR-Tree maintains a temporal ordering of the nodes throughout its lifetime. The similarity between our model and TR-Trees is that of dynamic updating of the internal structure. We create a new node from an already existing node when the old node is wider than the new one. After creating a new node, the parts excluded of it are also split as sub nodes of the old node and the data of the old node is copied into all of its sub nodes. Apart from that TR-Trees are specifically created to model graphic objects in a Geographical Information System while we are confined to the analysis of usage data.

Activity graphs as defined in [22] serve the purpose of an intermediate layer between skeletal program compilations. Activities are regarded as a contiguously indexed group of processes and different processes can handle different tasks as defined by the source program but their grouping is based on their main tasks. The objective is to write parallel programs without worrying about process or coordination management, the

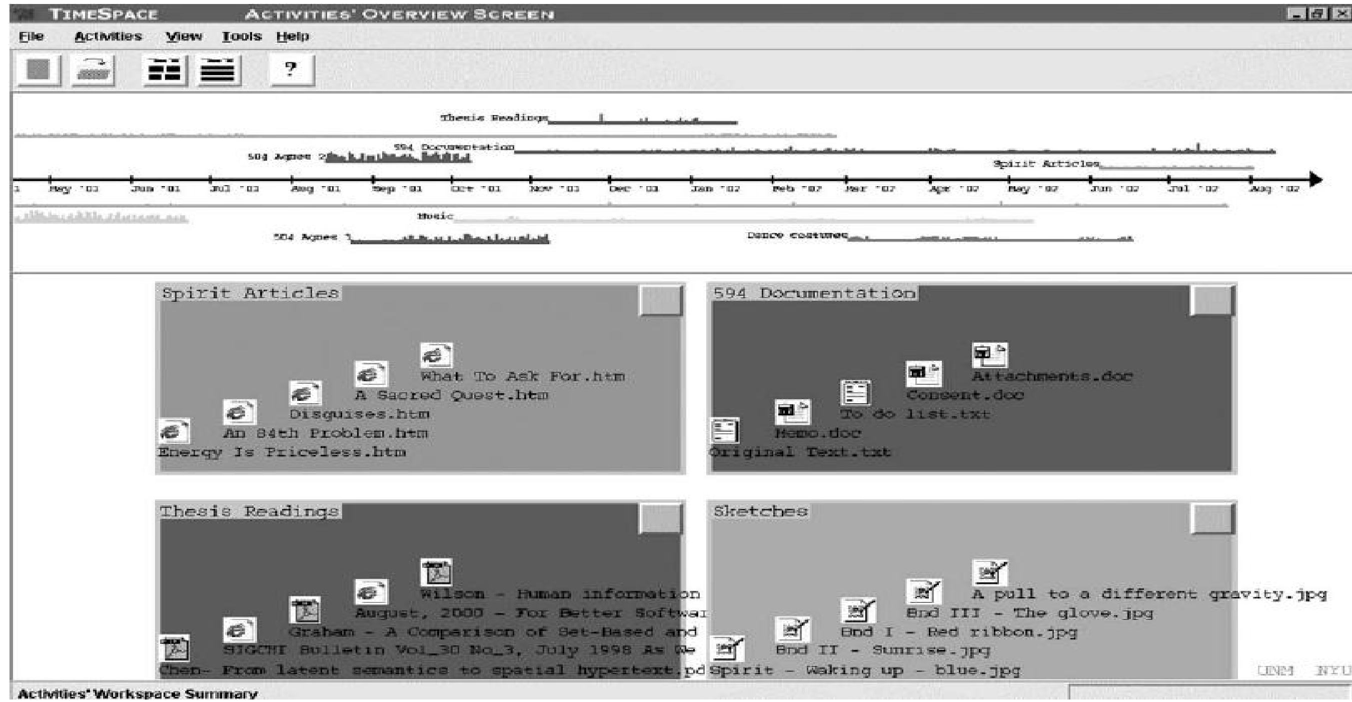

Figure 2: TimeSpace activity workspace window, showing the activity timeline and workspace segment detail.

program is first translated into an activity graph which is then converted into a MPI program. [22] term these intermediate graphs as activities, which has nothing to do with its user . They only group processes and manage their coordination. In our case, our activities encapsulate the events which took place by a user, so they serve as direct entities rather than intermediate layers. Just like activity graphs, our activity algebra provides automatic coordination management between the activities. Users can create or delete activities anytime they want.

Management of personal information space is also an area that is closely associated to our work. According to [9], information can be organized in five different models,

- Hierarchical, as seen in normative folders or directory systems on personal computers.
- Network, as exemplified by the World Wide Web, and other hypermedia systems.
- Spatial, as seen in icon-based presentation supported by desktop metaphor-based systems and 3D interfaces.
- Activity-based, as exemplified by the ROOMS [1, 2] system.
- Temporal, as exemplified by Micromegas system, and Web-browsing history mechanisms and recent file lists.

In the past many interactive systems have been created based on these models such as ROOMS [1, 2], Interlocus [3], NaviQue [4], LifeStreams [5], LifeLines [6], TimeScape [7] and TimeSpace [9]. Only TimSpace [9] is the system that provides both activity-based and temporal based views of the information space.

TimeSace [9] is a tool that monitors user specified file system subsets (like folders) and stores all of their file activities. These activities are then transferred to another module for their visualization, where each activity is displayed on a timeline. The timeline is displayed as a horizontal axis and all the activities that took place at the same time are displayed above or below the timeline. Although TimeSpace provides a chronological view of all the activities that took place at any time, but it becomes difficult to find activities during a particular time instance. However as shown in figure 2, TimeSpace provides an activity workspace screen where there is a static rectangle representing a time interval placed on the activity timeline, upon moving the scrollbar towards right or left the activities pass through this rectangle, a zoomed view is then displayed at the bottom of the screen where these activities are projected at a much higher resolution. The problem in TimeSpace is that one can't directly go to a particular time interval to find out all the activities that took place during that interval and this problem becomes more complex when that interval is shared by several activities, which introduces the problem of activity overlapping. The model that we present is partly related with TimeSpace particularly with the notion of activities and timeline. However, our model stores detailed description of each activity called events along with their overlaps with other activities and provides a powerful searching mechanism based on the context.

3. ACTIVITY ALGEBRA

As discussed in the preceding section, Activity algebra creates a graph like structure of activities and maps them over a data

stream of events. The set of operations derived from stream algebra are similar to DIVA [21]. The operators defined in activity algebra makes it easier to do the analysis of temporal data. There have been other systems in the past, which have used algebra to deal with time-based data but most of them have been used in the production of video presentation. For example, Algebraic video [23] defines an algebra to describe the spatial and temporal composition of video segments. The rest of this section is based on some preliminary concepts whose definitions can be found in the appendix A.

There is no doubt that the essence of our activity algebra lies in the data stream. And it uses operations like create, insert, remove and update on the stream. So let us first describe what a stream is.

Streams have been defined differently by many authors in the literature, like for example [15] define a stream as;

> A Stream S is an unbounded bag (i.e. multiset) of pairs <s, τ >, where s is a tuple and $\tau \in \Gamma$ is the timestamp that denotes the logical arrival time of tuple s on Stream S.

Similarly [21] defines a Stream as,

> A Stream *s* of type *T* is defined as a sequence (t_i) of *n+1* clock times and a sequence (v_i) of *n* values

Also according to [14],

> A stream is a sequence whose codomain is a nonempty set

Almost all of the authors have agreed upon the first definition of stream. DIVA also uses this definition but restricts a stream to a single type. They all embrace the idea that items in a stream coming at a certain time are independent of the others. And the only relationship between them is the stream. However we believe that the items (or events) in the stream are grouped under a certain activity. For example, visiting web pages and opening word documents might be grouped under a report preparation activity. Therefore, it is essential for us to know the start and end of an activity to determine the time interval in which its associated events occur. This leads to the following definition of a stream,

DEFINITION 1: A Stream S *is a set of heterogeneous events E_i with time τ_i*

$$S = \{E_i:\ \tau_i \mid \tau_i \in \Gamma\}$$

Or

$$S = \{E_1:\ \tau_1,\ E_2:\ \tau_2,\ \ldots,\ E_n,\ \tau_n \mid \tau_i \in \Gamma\}$$

Where E is an event and $\tau \in \Gamma$ is the timestamp that denotes the logical arrival time of event E on stream

Since each event comes with one timestamp, so it's difficult to extract activities out of a set of events. In activity algebra, we start off with a global activity where all of the events are grouped together with an increasing order of their timestamps. For every new event E, it's starting and ending times are equal to its arrival time τ , which brings us to the formal definition of an event.

DEFINITION 2: An event E *is an atomic element in Stream defined over a three tuple with V and τ_s and τ_e are both equal to the arrival time τ .*

$$E = \{\tau_s,\ V,\ \tau_e \mid \tau_s = \tau_e = \tau\}$$

where, V is data element, τ_s is the starting time and τ_e is the ending time.

Stream can be queried at any timestamp to determine the event stored at that time. If the inquired time falls outside the range of the stream then its corresponding value is undefined (denoted by $\perp$) or else an event is returned. The value of a stream *s* at a time τ is noted $s\ @\ \tau$ and is defined as follows:

- If $\tau < \tau_1$ or $\tau > \tau_n$ then $s\ @\ \tau = \perp$
- If j such that $\tau_j \leq \tau < \tau_{j+1}$ then $s\ @\ \tau = E_j$

E_j can further be queried to determine its value, starting time and ending time.

Example 1:

A typical example of usage data in a stream can be viewed as in figure 3,

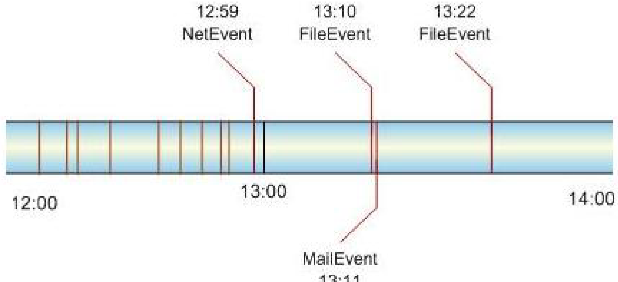

Figure 3 Events in a Stream

The sequence of events can be written as,

S = {NetEvent:12:59:15, FileEvent:13:10:35,
MailEvent:13:11:05,,…, FileEvent:13:22:01}

In this example FileEvent, NetEvent and MainEvent are event objects and represent different actions taken by a user during the course of an activity. Querying the stream at any timestamp would reveal these objects, which encapsulates the value and other time related information.

As said earlier, all the events are grouped under one global activity called root activity and each event contains information

about its starting and ending times which are usually equal. However if a user wants to define a new activity, all he has to do is specify the ending time which should be greater than the starting time of the event. An event of this type is called an Activity and becomes a parent activity for all the events that fall within its range. Apart from the starting and ending times, an activity is recognized by its name V. Formally we define an activity as;

DEFINITION 3: An activity A *is a tuple with* τ_s *, V and* τ_e *and its ending time is strictly greater than its starting time.*

$$A = \{ \tau_s , V, \tau_e \mid \tau_s < \tau_e \}$$

where, V is the name of the activity, τ_s is the starting time and τ_e is the ending time of the activity.

We say A contains events E_i if E_i is within the range of A;

$$A_{contains}E_i \text{ if } E_i(\tau_s) \geq A(\tau_s) \text{ and } E_i(\tau_e) \leq A(\tau_e)$$

Continuing from example 1, a simple activity A can be defined as,

A = {13:01:00,"Meeting in room 101",14:10:00}

Then A will contain all of those events described in *S* which fall within the range of A,

$A_{contains}$FileEvent @ 13:10:35

$A_{contains}$MailEvent @13:11:05

…

$A_{contains}$FileEvent @ 13:22:01

Each of these events can only be referenced from the activity A, the remaining stream will now be modified because all the events before and after activity A cannot be further referenced simultaneously from the root activity. Therefore after the inclusion of this new activity, two more activities will be created the first one will contain all the events before the activity A and the events after A will be contained by the second one as seen in figure 4. In this way a restructuring will take place when ever new activities are encountered.

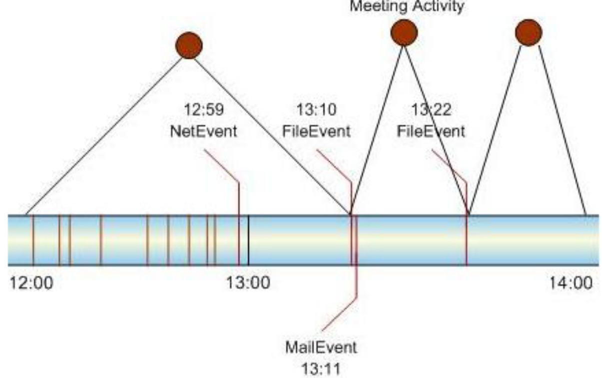

Figure 4 Restructuring of activities

So far we have only worked with user defined activities i.e. when activities are explicitly created by users with the help of a GUI for example. What about implicit activities? As discussed above activities are a special kind of events, therefore we can expect the start and end of an activity by two separate events. For example, consider an activity object with the value "search image file X" that first arrives in the stream at 13:25:15 and another activity object with the same value arrive at 13:30:20. So we must be able to identify those implicit or hidden activities.

In the activity manager of Micromegas, these implicit activities are defined with the help of ActivityEvent object. To distinguish this object from other events, the system only stores its τ_s value and marks its ending time τ_e as zero. So whenever an object of the type ActivityEvent is encountered and which holds the same value then the activity manager reassigns the τ_e of the previous object with the τ_s of this object. Once the bounds of the activities are determined then the activity hierarchy is updated.

3.1 Algebraic activity operations

As seen in the previous section, streams are merely a collection of heterogeneous events but it is the power of activities that make a stream more meaningful. Similar to DIVA, we also provide different operators to do temporal analysis of data in the stream.

One such common operation is editing, where a source stream *s* is combined with an edit stream *e*.

$$\text{edit}(s, e) = s \text{ if } (e = \perp) \text{ else } e$$

The result of this operation is a stream which is the same as the source stream except that it is replaced with the edit stream where the edit stream bounds coincide with the source stream. Figure 5 depicts this concept.

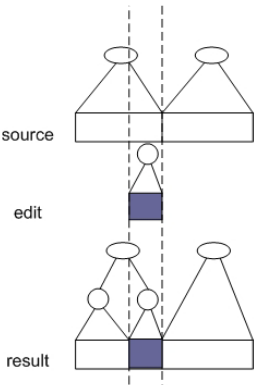

Figure 5 Editing a stream. Resulting stream is the modification of the source stream during the interval that is equal to the edit stream.

Editing a stream in this way may seem a straightforward action but internally it calls insert and delete operations and requires updating of the activity graph structure. If found within the bounds of the source stream, space is created for the edit stream

by deleting all those activities that fall within its range. Then activities of the edit stream are inserted into the source stream and the activity graph structure is updated.

Editing is an important operation when usage data obtained from different users of a group are combined to form a single stream and discuss final analysis on them.

3.1.1 *Insertion and Activity Overlapping*

Insertion of events into stream is based on their temporal value and they get accumulated at the end of the stream. Their mapping in the corresponding activity graph takes place at the deepest activity node which can safely handle the event within its interval.

Things become more complicated when activities are inserted into the activity graph. Because that not only changes the mapping between the graph and the stream but it also modifies the graph structure. There are also some special cases to handle activity overlapping as shown in table1. The cases are always checked whenever a new activity node say *N* is inserted in the activity graph.

Table 1 Cases for inserting activity nodes

Case 1	Case 2
$node_i$ $node_{i+1}$ N	$node_i$ N
N greater then $node_i$	N partially inside $node_i$
Case 3	**Case 4**
$node_i$ N	$node_i$ N
N entirely inside $node_i$	N and $node_i$ have same start
Case 5	**Case 6**
$node_i$ N	$node_i$ N
N and $node_i$ have same end	N and $node_i$ are equal

Case1: when N is greater than the current node then move forward:

$$\text{If } N(\tau_s) > node_i(\tau_e) \text{ then compare to } node_{i+1}$$

This condition is checked for every first level child of the root node. If the start time of N is greater than the end time of the current node $node_i$ with which it is compared than N is compared with the next node. Here we start from the first node and move forward by comparing N with each node to find its appropriate place however depending upon an application's need where nodes are required to be inserted backward then this case can be modified as,

Case 1.1: when N is smaller than the current node then move backward:

$$\text{If } N(\tau_e) < node_i(\tau_s) \text{ then compare to } node_{i-1}$$

In this case Node N starts from the ith node of the root and is compared backwards if N is found less then the current node $node_i$.

Case 2: insert N when it is partially inside the current node.

$$\text{If } N(\tau_s) < node_i(\tau_e) \text{ AND } N(\tau_e) > node_i(\tau_e) \text{ then insert N}$$

If the starting time of N is less than the $node_i$ encountered then N is broken into two parts as follows;

$$P1 = \text{from } N(\tau_s) \text{ to } node_i(\tau_e)$$

$$P2 = \text{from } node_i(\tau_e) \text{ to } N(\tau_e)$$

After breaking N into two parts, the first part P1 is compared with all the descendents of $node_i$ and the second part P2 is compared with all the nodes starting from $node_{i+1}$. These parts are inserted in only those nodes which are greater then them.

This breaking of a node is called Node split, where an activity node is split and distributed across different nodes.

Case 3: when N is entirely inside $node_i$.

$$\text{If } N(\tau_s) > node_i(\tau_s) \text{ AND } N(\tau_e) < node_i(\tau_e) \text{ then insert N}$$

This condition applies when N can be completed inserted in $node_i$. In this case N is first compared with the descendents of $node_i$ and if they are smaller then N than two other nodes are created for $node_i$, i.e. left and right nodes, and they are inserted along with N.

$$node_{ileft} = \text{from } node_i(\tau_s) \text{ to } N(\tau_s) \quad \text{eq.1}$$

$$node_{iright} = \text{from } N(\tau_e) \text{ to } node_i(\tau_e) \quad \text{eq.2}$$

After the insertion of left, right and N nodes in $node_i$, the events which are found within the ranges of these three nodes are moved from $node_i$ to them.

Case 4: when N and $node_i$ have the same start.

$$\text{If } N(\tau_s) = node_i(\tau_s) \text{ then insert N}$$

In this case only one extra node is created for nodei i.e. right node as in eg.2. Similarly events are copied from $node_i$ to N and $node_{iright}$.

Case 5: when N and $node_i$ have the same end.

If N (τ_e) = $node_i$ (τ_e) then insert N

This case is also similar to the previous one, here only a left node i.e. $node_{ileft}$ is created for $node_i$ as in eq.1 and all the events from $node_i$ are copied to $node_{ileft}$ and node N.

Case 6: when N is equal to $node_i$.

If N (τ_s) = $node_i$ (τ_s) AND N (τ_e) = $node_i$ (τ_e) then ignore N

This is the case when the starting and ending times of N correspond exactly with that of the $node_i$. In such a case it's useless to retain an extra node therefore node N is ignored and never compared with the subsequent nodes. There can be various situations in which a user might want to retain the new activity N, for example when a football match happening at the same time as another one. As mentioned before an event is something that occurs at an instant of time which means we can't have multiple events in a stream occurring at the same time. Thus in order to retain N we have to explicitly remove $node_i$ and then insert N or we can simply remove the events of $node_i$ which occur at similar time as in N and then insert the events of N and finally modify the label to reflect these changes.

The cases described for activity node insertion are always considered whenever a node is compared with another node in the structure. They allow us to detect overlapping between two nodes. Since these overlapping require a restructuring of the activity graph and copying of the events therefore they involve a very complicated procedure. The starting step of node insertion is to compare with first level child nodes of the root node. If any of the above cases are met, apart from the case 6, which means a node of interest is found then the search process is applied over the child nodes of that node. This process goes on recursively until an appropriate position of the requested node is found in the activity graph.

3.1.2 Deletion

Deleting few segments of a stream means deleting all those nodes in the activity graph which map on these segments. However, deleting an event demands no structural change in the graph.

When a segment in the stream is selected for deletion then the activity graph is searched for the node corresponding to that segment. Upon finding that node, it is directly removed thereby deleting its child nodes with their events as well and the activity graph is restructured.

The restructuring takes place by modifying τ_s and/or τ_e of the parent nodes of the just deleted node as in figure 6. The parents can modify their τ_s by checking with the τ_s of their first child, similarly their τ_e by the τ_e of their last child. If these bounds have changed after the deletion then the parent node modifies them based on the information available in its child nodes.

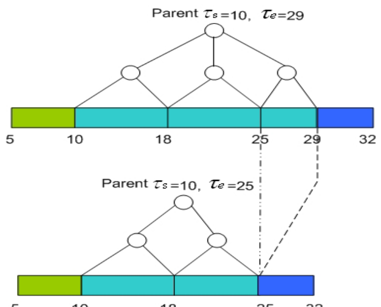

Figure 6 Restructuring based on deletion

3.1.3 Filtering

Analysis involves the separation of an intellectual or material whole into its constituent parts for individual study [24]. Since creation of such parts as activities in our case makes it difficult for an analyst to focus on his area of interest. Therefore, we provide filtering mechanisms based on the conditions provided by the user. The users can either search for events of their interest through event based filtering or find specific time periods based on time based filtering.

3.1.3.1 Event-based filtering

Events represent a basic searchable entity that most users would be focusing on, for example, searching all the file events related to word documents. In order to search for events, we postulate a hypothesis that the most recent events bear more importance. This hypothesis is closely related to the idea that the most recent activities might be unfinished. Therefore instead of starting our search procedure for event filtering from the start of the activity graph, we start from the current time and move backwards in the graph. This approach has a main objective of providing efficient event searching.

As we mentioned previously that each node in the activity graph has the knowledge about its parent and child nodes but for a backward search it needs to know more. The current node structure would work fine if every two distinct nodes are connected together by at least one common node, which would ensure that the common node would serve as a bridge between the two. But this situation can't be guaranteed to occur, because not all activities share events between them. Therefore, we modify the activity structure to accommodate knowledge about its siblings. With this knowledge, we can search on left or right of the node at the same level in the activity graph.

A resulting graph mapped on a logical stream is created, which contains copies of all the events occurring in the original stream which are placed with their temporal ordering as shown in figure 7.

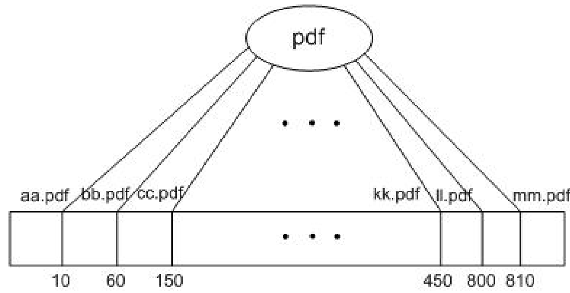

Figure 7 Resulting graph for event-based filtering of all pdf documents

3.1.3.2 Time-based filtering

The conditions required for time-based filtering are very different then the event-based filtering mainly due to the absence of any concrete information. The objective of this type of filtering is to narrow down the focus of the users. Since the data available at the higher level nodes is larger then those at the lower level, therefore it is possible that the users may only want to focus on those low level nodes. This granularity of the data leads us to different possible views of the data, which are categorized into hours, days, weeks, months and years.

Typically a user would like to filter data from one particular starting time called τ_{start} and then specify duration in term of hours, days; etc. Starting from the initial τ_{start} , all nodes that fall within that duration are copied to a new stream where they are kept under one generalized node of the same duration.

For example, if the user starts the time-based filtering process with the starting time of τ_{start} = 15/07/2003 and gives a duration of one year then a new activity graph is generated with its first node having its τ_s = 15/07/2003 and τ_e = 15/07/2004 and all the nodes in the original graph that fall into this range are copied in to this new node. If there exits activities greater than 15/07/2004 then another year node is created and those activities are also copied in that node. This process goes on until no more activities are left in the original graph beyond the τ_{start} . The resulting graph generated by time–based filtering based on the duration of one year is shown in figure 8.

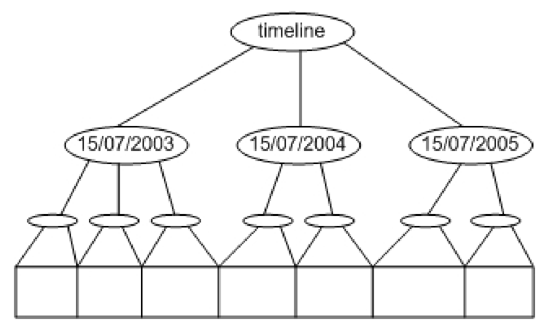

Figure 8 Time-based filtering with a duration of one year with it's τ_{start} = 15/07/2003

4. IMPLEMENTATION

4.1 Introduction

In this section we describe the prototype system created to analyse temporal data based on the usage data of a user. The data on which we operate were taken from the log files generated by the observers in Micromegas (sec 1.1) and sample calendar files from [25]. The objective of testing this prototype across two sources was to justify its generality. In the current Micromegas implementation; the observers that monitor users' activities store their data in a central database. This database is then queried to generate different log files for a single day or many.

As described earlier, the activity manager of Micromegas helps to relate heterogeneous events into some activities but has the drawback of not extracting more inherent relations that our system does. Therefore, the activity manager just behaves as a utility that doesn't ease the burden of analysing temporal data stored in the database.

Our prototype analyser alleviates these problems by implementing the stream based algebraic activity model presented in the previous section.

4.2 Analyser

4.2.1 Principle

The analyser is a Java application which helps the user to analyse temporal data stored in the log files by the observers. The input to the analyser is these log files from which it constructs an internal algebraic activity graph. Initially the analyser uses the activity information stored in a log file called "Activity_Events" to construct its graph structure but once this initialization of the system is finished, users can themselves explicitly modify, insert, or deleted modify new activities.

This initial setup is necessary because the analyser is of no use if there is no data available. The log files of Micromegas are not the only source of data which can be used by the analyser; users can provide data from other sources like shareable calendars on the web [25]. So far our analyser is not compatible with the file formats of these calendars; therefore we have to parse them to produce text based files and then use them for analyses.

A simple activity which has some duration is represented by a Node object. This object contains all the information pertaining to a node, for example its parents, children and pointers to previous and next nodes. A Node can have both multiple parents and children. Since an activity is another form of Event differing only in terms of duration therefore the Node object also encapsulates a mEvent object. This mEvent object comes from a generalized class of Events and store just three attributes namely information, start time and end time. Figure 9 shows these two classes in more detail.

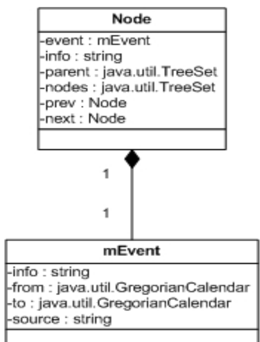

Figure 9 Node and mEvent classes

Once activities are encountered then a mEvent object is constructed with the attributes info, from and to and it is then passed into the constructor of a Node object.

Since activities and events need more interaction then just simple abstraction therefore we need to define another class that manages this interaction. This new class is called TimeLine. It handle all the management of nodes for example their insertion, deletion and editing and also filtering mechanisms which includes both event-based and time-based.

A TimeLine object is constructed from a single string object that helps in giving descriptive information about the timeline. Nodes are then inserted into the TimeLine object. It also uses the java.util.TreeSet class as an internal data structure to hold these Node objects. TreeSet class has a property that it doesn't hold the reference of same object more than once, this is done by comparing the newly arrived object with those already present.

This property of TreeSet helps us to restructure our activity graph whenever new input arrives. The TreeSet always calls the compareTo method of the new object; therefore we implement our node structuring logic in the compareTo method of the Node class. Apart from TreeSet, TimeLine also contains attributes for start and end. These attributes contain the times calculated from the start of the first node and the end of the last node present in the TimeLine.

Figure 10 describes the basic class diagram of a TimeLine class and shows its relationships with other classes.

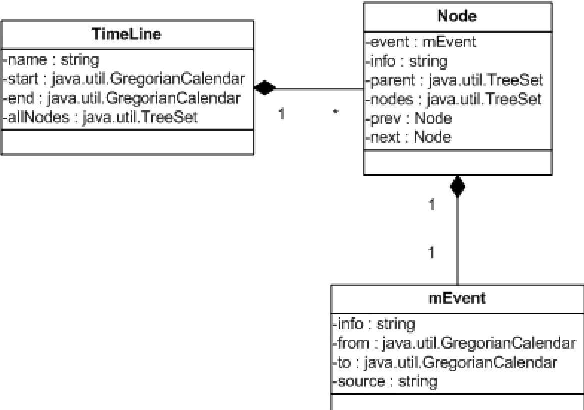

Figure 10 TimeLine and its relationship with other classes

All other events for example network events, file events; window events, etc extend from the mEvent class and add attributes and methods specific to them. For example a File event will have attributes like names, path, access, modified and creation times, data type and size. Therefore a collection of Nodes in a TimeLine can contain events of different types. A Node with its ending time greater than its starting time becomes effectively an activity, while the one with same starting and ending times is called an event.

After looking at how the main classes are defined and how they are related with each other, we now embark on the algorithms which implement the model presented in the previous section.

4.2.2 Algorithms

In section 3, we discussed the formal models of operating with an algebraic activity graph. Those models included insertion and activity overlapping, editing, deletion and filtering methods. Now we describe in more details the algorithms for each of these models.

4.2.2.1 Insertion and Activity Overlapping

The insertion in the TimeLine receives as argument a Node object N, which itself contains its starting and ending times. Initially after making sure that the object N is not null, it is inserted in the TimeLine. If the entry is already inserted in the TimeLine, the insertion must be aborted, else the entry is inserted into the TimeLine, and it triggers all the structural changes necessary in response to activity overlapping if any of the cases described in section 3.1.1 are encountered.

Algorithm insert
Description: Insert a new node N in a TimeLine at time t.

```
Begin
        if N is null then return
        else
         N is already present in the TimeLine then return
```

else
invoke add method of the TreeSet in which to place N. This method internally invokes compareTo on each of its nodes.
end

Algorithm compareTo
Description: Compare N with all the previous nodes and find an appropriate place where N can be inserted. This comparison is based on the starting times of every node.

Begin
Let B be N and the current Node element be A
Case 0: when B has same start and end then
This means B is an event and not an activity, so insert B without altering the structure.
Case1: when B > A then insert B after A
This is not a straightforward condition, if B starts after A then we have to check for all possible conditions and modify the structure after Bs insertion.
Case 1.0: B is totally outside A then
B can't be inserted within this A therefore proceed to another node.
Case 1.1: B is partly inside A then
Break B into two parts AB and B'
AB is common in both A and B and has its start=As end and its end=Bs start
B' has its start=ABs end and its end=Bs end
Invoke Insert AB on B
Invoke insert B' on B
Now A might be having other nodes, so its wise to search within those nodes and then invoke insert AB on A.
B' may also have a range that might also fall within other nodes. So we go at the top most parent of A and try to compare B' with its leafs. If such a node is found then B' is also inserted there.

Case 1.2: B is entirely inside A
Case 1.2.1: B and A have same end
Case 1.2.2: B and A have same start
Case 1.2.3: Bs start and end are inferior to A
Case 2: when B < A then insert B before A
Case 3: when B==A

End

4.2.2.2 Deletion

The deletion in the TimeLine receives as argument a Node object N. Initially a search is made to find if the entry is contained in the TreeSet of the TimeLine. Once the node is found representing the object N to be deleted, it is deleted straightaway if it has no child nodes. But if it has some child nodes then those child nodes are copied in the parent node of N and other adjustments are performed. The node with no child can be either an event or an activity node, in either case the deletion wont need any structural changes.

Algorithm Delete
Description: Remove node N from the TimeLine

Begin
for each node temp in TimeLine
if temp is equal to N then remove N
else temp falls within the range of N then invoke deleteNodes on temp and pass N as a parameter.
if N not found then return false
End

Algorithm deletNodes
Description: Remove node N stored in this node

Begin
if N is a direct descendent of this node then remove N
Adjust the time bounds of this node
else
Recursively invoke deleteNodes on child nodes
End

4.2.2.3 Editing

The edit operation involves the usage of both insertion and deletion method. The edit in the TimeLine receives as argument a Node object N. Like the deletion algorithm, it searches for the parent node P which could accommodate N, copies all P's child nodes within the range of N into N. Finally P is then updated.

Algorithm: Edit
Description: Edit a TimeLine by inserting a Node N somewhere in between the graph structure.

Begin
Let P be the parent of N
Invoke locateParent on Node N and store the result in P
Copy P's child nodes to N which are in the range of N
Delete those child nodes from P
Insert N into P
Update TimeLine
End

Algorithm: locateParent
Description: Locate Parent P that can accommodate N

Begin
for each node temp in TimeLine
If N is in range of Temp then invoke locate on temp and pass N as a parameter and store result in P
return P
End

Algorithm: locate
Description: Recursively locate Parent P for node N in each node of TimeLine

Begin
If current node P is the smallest node which can accommodate N then return P
else
for each temp in P
invoke locate on temp and pass N as parameter
End

4.2.2.4 Filtering

Analysis of a data can't be made effective until the users can narrow down their search and for this purpose we provide the filtering methods. The filtering process as described in the earlier sections is decomposed into two categories namely;

- Event-based Filtering
- Time-based Filtering

4.2.2.4.1 Event-based Filtering

The event searching is performed on the descriptive information provided by each mEvent object. Since the information stored in a Node is in fact the same stored in the mEvent object. Therefore this filtering method searches for those nodes or events which contain the keywords provided in the search query. To provide more efficiency the search procedure starts from the last node 'n' of the TimeLine, searches it till its depth then goes to the previous node 'n-1' and does the same thing till the first node of the TimeLine. This filter operation also calls an internal method of findEvent to recursively find events in each node. The method returns a new TimeLine which contains nodes containing those events.

Algorithm: filter_Event
Description: Filter a TimeLine by searching for the input string S

Begin
Create a new TimeLine T
Store the last node of TimeLine in N
while N is not null
Invoke findEvent and pass as parameters S, N and T
return T
End

Algorithm: findEvent
Description: Recursively search for events in the Node N which match with the input String S.

Begin
if N contains S then insert N in T
else
for each node Temp in T
Invoke findEvent
End

4.2.2.4.2 Time-Based Filtering

The time-based filtering over a TimeLine receives as arguments the starting time S and the duration object D. The duration object determines the time intervals in which all nodes are filtered. A new TimeLine T is created and based on the first node encountered after S, a new node is constructed with a range of S+D and that first node is inserted. If the second node falls in the same range then its inserted straightaway otherwise another node is created from (S+D) +D and is inserted. The duration object is defined in the class Duration and it contains attributes like hour, day, week, month and year, selecting a value for any of them will result in a time line reflecting those changes. For example, by selecting duration.month=2 we effectively tell the analyser to filter the timeline after every two months by starting from the start value. Figure 10 contains the class diagram of the class Duration.

Duration
+hour : int +day : int +week : int +month : int +year : int

Figure 10 Class diagram of Duration

The Duration specified by the duration object helps to provide a multiscale view of the nodes preset in the TimeLine. A user can get values to any of its attributes to achieve better results. For scaling, the algorithm gives priority in the following order; year, month, week, day, hour. For example a user wanting to group all nodes of two years in six months samples starting from 15/07/2005 then he can just do the following;

Start=15/07/2005, Duration.year=2, Duration.month=6

Algorithm: Filter_Time
Description: Filters the TimeLine from the starting time S to the end of the TimeLine with the duration given in the duration object D.

Begin
If TimeLine has no elements then return null
Store non-zero attributes of D into local variables and set all the local flags to true corresponding to them

Create a new TimeLine

For each node temp in TimeLine
If temp is greater than or equal to S then based on the combination of the flags take action

End

4.2.3 Example

We will illustrate the idea of the algorithms by using an example. In this example we will insert all the activities of the figure 11 in temporal order and then delete some of them.

Consider some activities which were involved in the version 0.1 of Micromegas as illustrated by the timeline in figure 11.

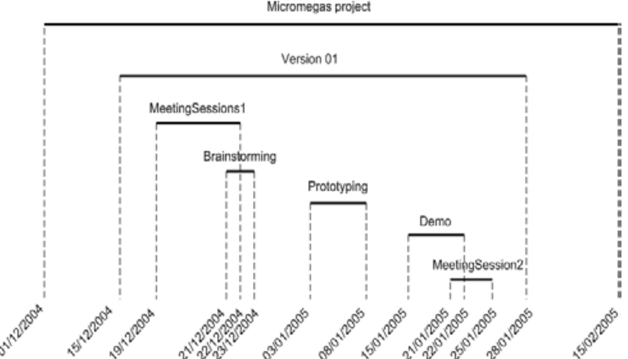

Figure 11 Timeline for Micromegas project

The diagram shows the sequence of activities carried on during a hypothetical version 0.1 of Micromegas. We have illustrated the starting and ending times of each of the activities. Now this information will be entered into micromegas log files with the help of the activity manager. The log files will then be processed by our analyzer, which will generate their corresponding activity graph. The information about this graph is stored in a dot file format which is used for visualization by a graph visualization tool such as Graphviz [26].

Initially, the internal activity graph is empty. The insertion of the activity "Micromegas Project" creates the first activity node which is inserted into the TimeLine. The next insertion of the activity "Version01" triggers a structural change in the activity graph. The first activity is broken into three sub activities because the new activity is entirely in the old one (case 3). All of the events of the parent node are copied to their respective child nodes and the structure looks similar to figure 12.

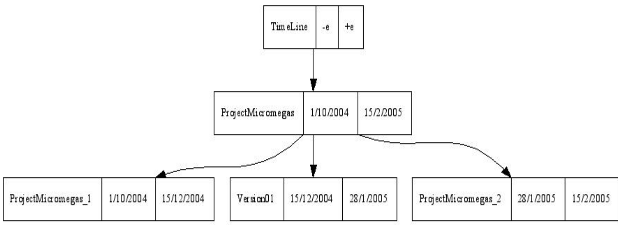

Figure 12 Activity graph after inserting second node

Next, the activity "Meeting1" comes in and it is compared with other nodes in the timeline. Since its duration starts from 19/12/2004 to 22/12/2004, therefore it is inserted under the middle child node of Micromegas project node satisfying case 3. This is shown in figure 13.

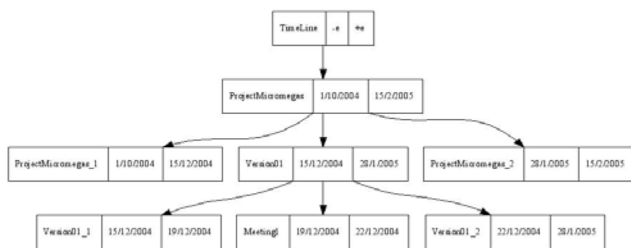

Figure 13 Activity graph after the insertion of the third node

The activity graph grows after the insertion of every new activity node, cases are checked and nodes are placed at appropriate places. Finally the graph structure becomes as shown in figure 14.

Now, we try to delete the activity node "Meeting1" from the graph. When the delete occurs the graph is updated automatically by adjusting the time bounds of the parent nodes which included this node. The resulting graph is depicted in figure 15.

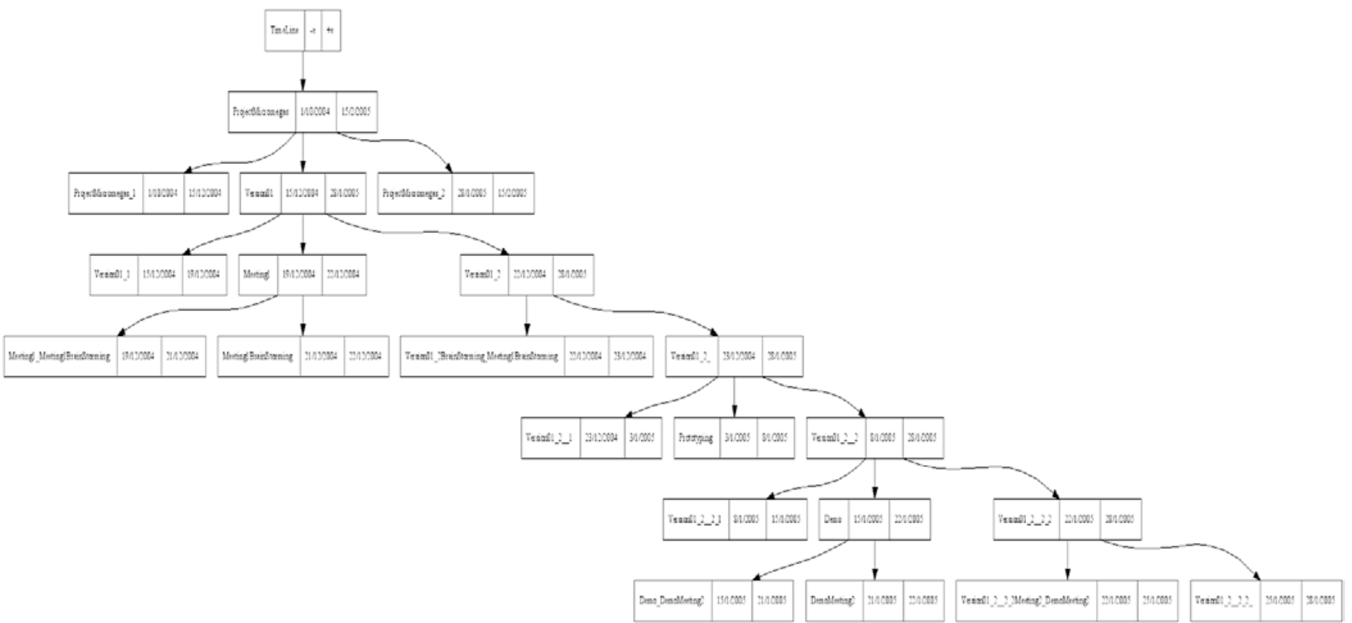

Figure 14 Activity graph after the insertion of all nodes

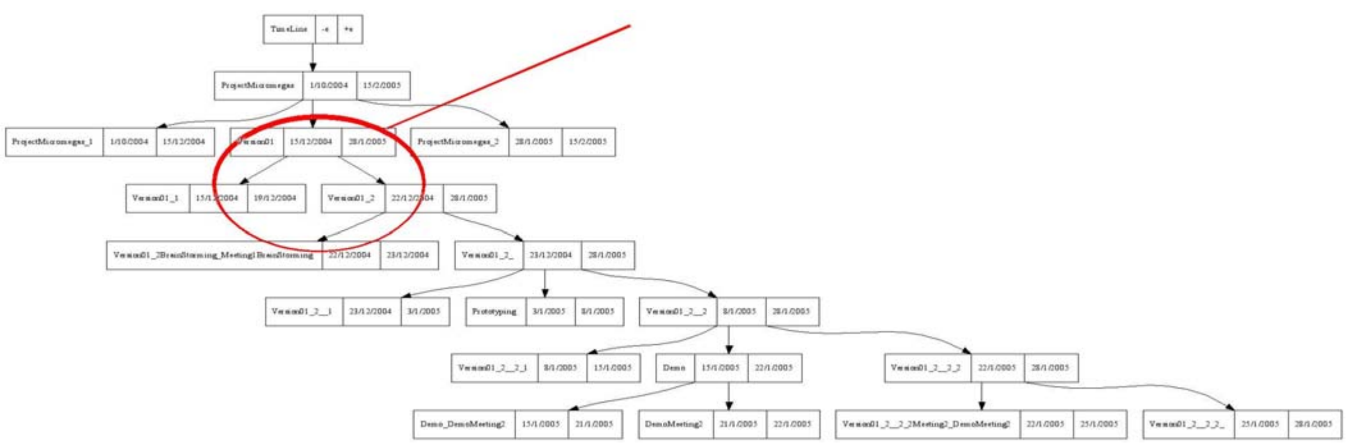

Figure 15 Activity graph after the deletion of node Meeting1

5. CONCLUSIONS

Algebraic activity graphs provide a model for analyzing temporal data present in the form of events captured during a users' interaction with its familiar data. The models' contributions at the architecture level include:

1. An Activity algebra based on stream algebra that provides a simple but mathematical powerful model of activities and operations upon them.
2. Ability to filter event and time based data from activities and manipulate activities with editing functions
3. Provides the capability to search for events in terms of context

The Java implementation satisfies the model by implementing algorithms efficiently and providing filtering methods for event-based and time-based searching.

The implementation is tested on the log files generated by the current Micromegas system and also on the data from shareable calendars.

Future research is needed in several directions, such as:

- interface designing for visualizing multiple activities without much distractions
- using activities for pattern matching
- Replacing the current offline analysis to online analysis
- Inspecting the suitability of the system by performing evaluation

6. ACKNOWLEDGMENTS

Thanks to Nicolas Roussel for providing all the necessary support and guidance as an advisor of my project. Also thanks to Wendy Mackay and Michel Beaudouin-Lafon for allowing me to extend their DIVA stream model into algebraic activity model. Thanks also go to Matthieu Langet for providing Micromegas log data and all the necessary documentation for the existing Micromegas system. I am indebted to Emmanuel Pietriga for helping to understand his ZVTM toolkit and all the members of the InSitu project of INRIA Futurs for their cooperation.

7. REFERENCES

[1] Card SK, and Henderson A., A multiple, virtual-workspace interface to support user task switching. In Proceedings of human factors in computing systems (CHI 1987), Toronto 5-9 April 1987

[2] Henderson A., and Card S., ROOMS: The use of multiple virtual workspace to reduce space contention in a window based graphical user interface. ACM Trans Graph 5(3): 211-243

[3] Nomura T, Hayashi K, Hazama T, and Gudmundson S, Interlocus. In Proceedings of computer supported cooperative work (CSCW 1998), Seattle, 14-18 November 1998

[4] Furnas GW, and Rauch SJ, Considerations for information environments and the NaviQue workspace. In Proceedings of digital libraries (DL 1998) the third ACM conference on digital libraries, Pittsburgh, 23-26 June 1998

[5] Freeman E., and Gelernter D, Lifestreams: a storage model for personal data. ACM SIGMOD (1996) Bull 25(1):80-86

[6] Plaisant C, Milash B., Rose A, Widoff S, and Shneiderman B Lifelines: visualizing personal histories. In. Proceedings of human factors in computing systems (CHI 1996), Vancouver 13-18 April 1998

[7] Rekimoto J., Time-machine computing. In Proceedings of 12th annual symposium on user interface software and technology (UIST 1999), Asheville, 7-10 November 1999

[8] Whittaker S, and Hirschberg J, The character, value & management of personal paper archives. ACM Trans Comput Hum Interact (2001) 8(2):150-170

[9] Krishnan A, and Steve J., TimeSpace: activity-based temporal visualisation of personal information spaces. Pers Ubiquit Comput (2005) 9: 46-65

[10] Guttman A, R-trees: A dynamic index structure for spatial searching. ACM 1984

[11] Fekete, J.D., The InfoVis Toolkit. Proceedings of InfoVis, pp. 167-174. IEEE Press 2004.

[12] Johnson, J., Roberts, T., Verplank, W., Smith, D., Irby, C., Beard, M. and Mackay, K. The Xerox Star: a retrospective. IEEE Computer, 22(9): 11-29, 1989.

[13] Jon K., Temporal dynamics of On-line Information Streams, Department of Computer Science, Cornell University, 2004.

[14] Gonçalves M. A. et al , Streams, Structures, Spaces, Scenarios, Societies (5S): A Formal Model for Digital Libraries. ACM Transactions on Information Systems, Vol. 22, No.2, April 2004.

[15] Arvind A. et al, STREAM: The Stanford Data Stream Management System. In proceedings of SIGMOD pp 665-665. ACM 2003.

[16] George W. and Benjamin B., Space-Scale Diagrams: Understanding Multiscale Interfaces. In proceedings of SIGCHI. ACM 1995.

[17] Yves G. and Beaudouin-Lafon M., Target acquisition in mutiscale electronic worlds. International Journal of Human-Computer Studies, 61, 875-905. 2004.

[18] Ken P. and David F., Pad: An alternative approach to the computer interface. In proceedings of Computer Graphics and Interactive Techniques pp 57-64. ACM 1993.

[19] Heikki M., Hannu T. and A. Inkeri V., Discovery of frequent episodes in event sequences. Data Mining and Knowledge Discovery pp 259-289. 1997.

[20] Geraldo Z. et al, The Temporal R-Tree. Federal University of Rio de Janeiro, Brazil. 1998.

[21] Mackay W. and Beaudouin-Lafon M., DIVA: Exploratory Data Analysis with Multimedia Streams, Proc Human Factors in Computing Science, CHI'98

[22] Murray C. and Andrea Z., Activity Graphs: A model independent intermediate layer for skeletal coordination. In Proceedings of Applied Computing pp. 255-261. ACM 2000.

[23] Andrzej D., Ron W. and David G., Content-Based Access to Algebraic Video. IEEE Multimedia (1996).

[24] http://www.answers.com

[25] http://www.icalshare.com/

[26] http://www.graphviz.org

APPENDIX A

1. Preliminaries

Here, we briefly review the concepts necessary for the development of the preceding discussion.

Definition A1. An *event* is an instantaneous fact, i.e., something occurring at an instant. An event is said to occur at some granule *t* if it occurs at any *instant* during *t*.

Definition A2. An *instant* is a time point on an underlying time axis.

Definition A3. In a data model, a one-dimensional *chronon* is a non-decomposable time interval of some fixed, minimal duration. An *n*-dimensional chronon is a non-decomposable region in *n*-dimensional time.

Definition A4. An *interval* is a directed duration of time. A duration is an amount of time with known length, but no specific starting or ending instants. For example, the duration "one week" is known to have a length of seven days, but can refer to any block of seven consecutive days. An interval is either positive, denoting forward motion of time, or negative, denoting backwards motion in time.

Definition A5. A timestamp is a time value associated with some object,

Definition A6. The *Gregorian calendar* is composed of 12 months, named in order, January, February, March, April, May, June, July, August, September, October, November, and December. The 12 months form a year. A year is either 365 or 366 days in length, where the extra day is used on "leap years." Leap years are defined as years evenly divisible by 4, with years evenly divisible by 100 being excluded, unless that year is evenly divisible by 400. Each month has a fixed number of days, except for February, the length of which varies by a day depending on whether or not the particular year is a leap year.

Definition A7: A tuple is a finite sequence that is often denoted by listing a range values of the function as $\{f(1), f(2),\ldots, f(n)\}$.

Definition A8. A *graph* G is a pair (V, E), where V is a nonempty set (whose elements are called *vertices*) and E is a set of two-item sets of vertices, $\{u, v\}$, $u, v \in$ V, called edges. A *directed graph* (or *digraph*) G is a pair (V, E), where V is a nonempty set of vertices (or nodes) and E is a set of edges (or arcs) where each edge is an ordered pair of distinct vertices (v_i, v_j), with v_i, $v_j \in$ V and $v_i \neq v_j$. The edge (v_i, v_j) is said to be *incident* on vertices v_i and v_j, in which case v_i is adjacent to v_j, and v_j is adjacent from v_i.

Several additional concepts are associated with graphs. A *walk* in graph *G* is a sequence of not-necessarily distinct vertices such that for every adjacent pair v_i, v_{i+1}, $1 \leq i < n$, in the sequence, $(v_i, v_{i+1}) \in E$. We call v_i the origin of the walk and v_n the terminus. The *length* of the walk is the number of edges that it contains. If the edges of the walk are distinct, the walk is a *trail*. If the vertices are distinct, the walk is a *path*. A walk is *closed* if $v_1 = v_n$ and the walk has positive length. A *cycle* is a closed walk where the origin and non-terminal vertices are distinct. A graph is *acyclic* if it has no cycles. A graph is *connected* if there is a path from any vertex to any other vertex in the graph. A *tree* is a connected, acyclic graph. A *directed tree* or (DAG) is a connected, directed graph where one vertex (called the root) is adjacent from no vertices and all other vertices are adjacent from exactly one vertex. A graph $G' = (V', E')$ is a *subgraph* of G = (V, E), if $V' \subseteq V$ and $E' \subseteq E$.