TimeWalker, a tool to visualize eventdata

Theo de Ridder
Pim Buurman
E-mail: Theo.de.Ridder@wxs.nl

1. Introduction

Many systems produce huge amounts of timestamped data (events) like logs from systemcalls, time-series from network monitoring or transactions from database-applications.

In practice eventdata is often thrown away without any inspection. Some of the main reasons are: waste of resources, poor dataformats, non-scalability of traditional tools, lack of an adequate visual instrument.

However, throwing away eventdata unseen implies losing essential information needed to discover cause-effect relations within (un)wanted or (un)expected systembehaviour. *TimeWalker* is a GPL-tool that makes preservation and disclosure of historical details contained in eventdata attractive and feasible.

The architecture and user-interface are made very flexible and portable. *TimeWalker* is implemented in Python (wxPython) and C. The first release will become available in november 2001 for Win32 and Linux. In this release *TimeWalker* will work smoothly for about 500000 records in memory that represent individual events or aggregated events collected from much larger (GB) datasets.

Information about the actual state can be found at www.NLnet.nl/projects/Timewalker and sourceforge.net/projects/timewalker. The development of *TimeWalker* is a project funded by the Stichting NLnet.

2. Philosophy

TimeWalker is based on the following principles:

- **Domain independence**

  It is impossible to optimize the visualization of a multidimensional space for all dimensions without domain-knowledge. *TimeWalker* is concentrated on the time-dimension, assuming that the human perception of time has common characteristics for quite different domains. Optimizing only the time-dimension and handling the other dimensions in a general purpose way is already far from trivial but worthwhile because it enables a first look at unseen worlds behind the mysterious mountains of eventdata.

- **Information visualization**

  The central mantra of information visualization states that one should always show context and detail together. The following properties are due to applying this mantra rather strictly:

  - The information density is very high.
  - All graphical views are visible in parallel.
  - There is no scrolling in graphical views.
  - Each tree-node indicates the number of its children.

- **Universal Pythonized datamodel**

  Every element in *TimeWalker* is represented in Python sources or Python objects. The pickle mechanism is used to make persistency of objects transparent, compact and fast. An important benefit is that the deepest internal data-representation is always rich and object-oriented. Conventional representations (database records, HTML or XML) are only used in conversions from/to external formats.

Part of the unification is that all documentation is also represented within Python and only transformed on the fly into a viewable html format.

- **Rich defaults**

  Defaults for scaling and coloring are derived automatically from the data. The effect is similar to what is offered by digital cameras: a quite reasonable contrast and sharpness for the average case without the need for any specific user-action.

- **Non-modal frames**

  The usage of modal popup-frames is avoided as much as possible. Changes and errors are indicated at the actual spot until the user decides to cancel, commit or correct.

3. Architecture

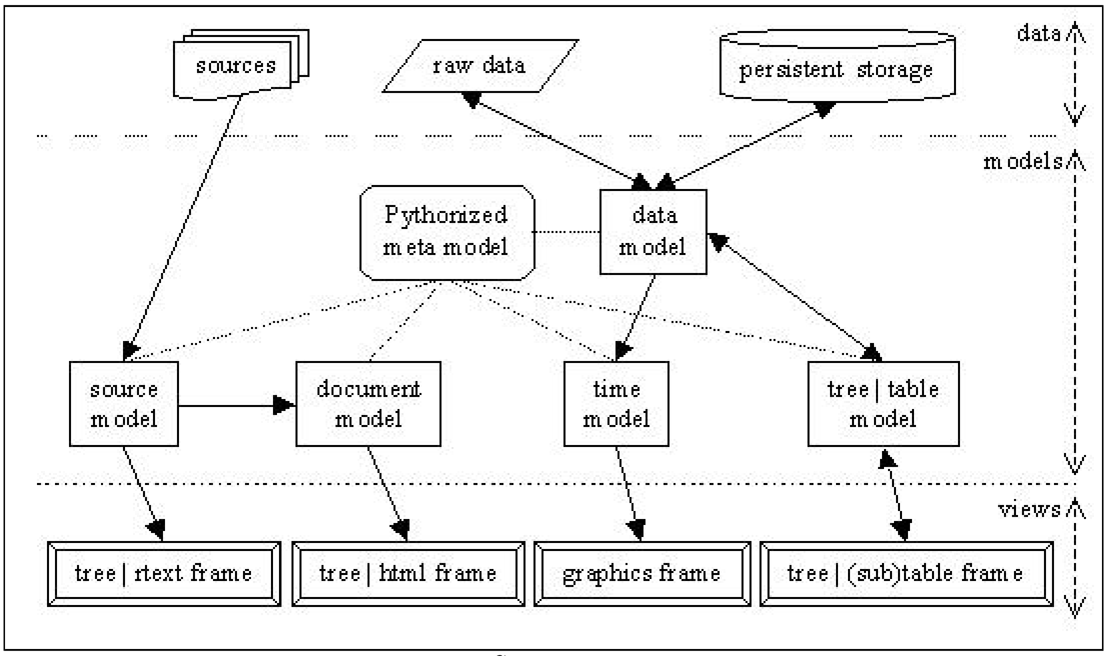


figure 1.

The architecture has 3 main layers (see Fig 1):

- **Data**

  Any data within a running *TimeWalker* is originating from one of the following places:

  1. Python sources:

     All on-line documentation is derived from these sources.

  2. Raw data:

     Raw data in any external can be imported or exported by using an existing convertor or by creating a new one. Import conversion is yet implemented as a simple but rather slow separate batch-process.

  3. Persistent storage:

     All eventdata is made persistent in `.twd` files in a unified pickled format. Each file contains a generated header with a complete self-description about content and context. The same format is also used for saving at any moment a complete running state as a snapshot.

4. User input

   User-input is enabled through editing (or drag-drop) cells of writable tables.

- **Models**

  There 6 types of models:

  1. Meta model

     Any record-like data is modelled with a universal MetaRecord as run-time descriptor for typing, accessibility and layout. MetaRecord is more abstract than a class to enable representations of records that are extremely fast for sequential access.

  2. Data model

     The basic data models are RecordList and EventList. RecordList is used to model arbitrary data. EventList is a specialized RecordList that is immutable and sorted on the required timestamp field. Data models do have an underlying implementation in C to obtain high performance for huge amounts of data.

  3. Time model

     The time model is the working horse of *TimeWalker* for aggregating and transforming eventdata in sequences of equidistant time-intervals. It is based on the concept a TimeShelve on which a pile of TimeBoxes is placed. The shelve is responsible for shifting the pile along the time-line and for controlling which subperiods of the zoomlens are to be calculated on the fly by each of the boxes.

  4. Tree | Table model

     These models enable automatic mapping between arbitrary trees and sequences in Python and the interfaces of the used widgets. This models abstract from the peculiar properties of the used gui-library.

  5. Source model

     The Python sources are to be considered as a complete executable and readable specification of *TimeWalker*. Because Python is a reflective language a model for inspecting the sources comes almost for free.

  6. Document model

     The Python `__doc__` string facility is exploited to represent all the manuals. A simple grammar within a string exploits the Python tokenizer to obtain a readable markup without using nested tags.

- **Views**

  The various views are shown in 4 types of frames

  1. Tree | Table frame

     This type is for inspecting or changing data in tables with dedicated cell-editors (see Figure 4).

  2. Graphics frame

     This type is only used for the graphical mainframe (see Figure 3) of the application.

  3. Tree | Html frame

     This type is for browsing through documentation in a simple html representation with links.

  4. Tree | RText frame

This type is for browsing through the sources, shown with syntax-highlighting.

4. Data handling

TimeWalker unifies arbitrary eventformats into a format that enables a much better performance for persistent storage, aggregation and transformation than can be obtained by using a (traditional) database.

Aggregation is the process of compressing arbitrary eventlists into a fixed-time interval sequences (TimeBoxes) containing a single composite record in each interval. With user-specified expressions important correlations can be preserved during aggregation. Aggregated records are transformed with user-specified expressions into sequences of vectors (ResultBoxes) which are plotted as colored beams.

The clean syntax and semantics of Python is used for all expressions at the user level. Some specific internal techniques are used to improve the performance of the produced byte-code drastically.

There are 3 types of aggregations:

- **simple**

  The result is the tuple (min, max, first, last, sum, count) for each column.
- **freq**

  The result is a frequency distribution of the values in a column with a limit on the number of different values.
- **keyed**

  The result is a combination of simple and freq for the (key, value) pairs returning from a user-defined expression. This type is typical used for keeping some correlations during aggregations. The effect is similar to what can be obtained with GroupBy in SQL.

5. Visualization

TimeWalker uses an innovative technique for information-visualization along the time-axis that enables simultaneous presentation of context and detail of eventdata in a range from 40 years down to 50 milliseconds. The technique is based on a sliding hierarchical *ZoomLens* that shows a bundle of multiple beams with predefined (quarter, week, day, hour, 5 min, 15 sec, 1 sec, 50 ms) time-scales. The zoomlens can be shifted by hand or be started as an animation.

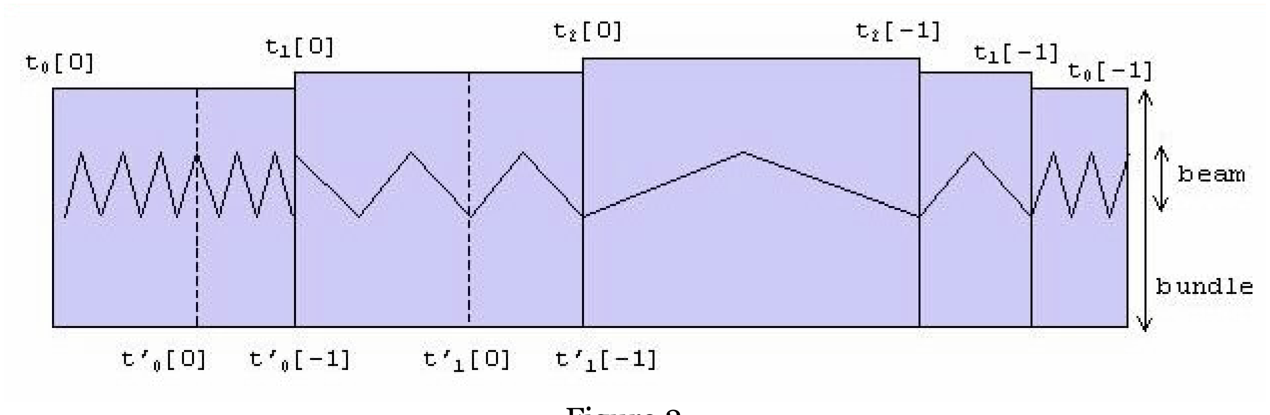


Figure 2.

The functionality of the zoomlens is illustrated in figure 2. Each next level L_{n+1} splits the underlying level L_n in a left and a right part, without hiding anything. The total interval of each L_n is indicated as $t_n[0]..t_n[-1]$. The subinterval of each L_n that is magnified in L_{n+1} is indicated as $t'_n[0]..t'_n[-1]$. The rule holds that $t'_n[-1] = t_{n+1}[0]$ for each n>0. In other words, one has to look just left of each L_n to see which part it magnifies.

Parallel with the zoomlens 4 other widgets are used to visualize time-correlations:

1. Calendar

   This widget shows all days and weeks as small rectangles, each containing a compressed image of 1 vertical vector selected in a Beam.

2. Epoch

   This widget shows all quarters within 40 years, grouped in decades, in the same way as Calendar shows days.

3. Clock

   This widget has 2 functions:

   - Show indicated time

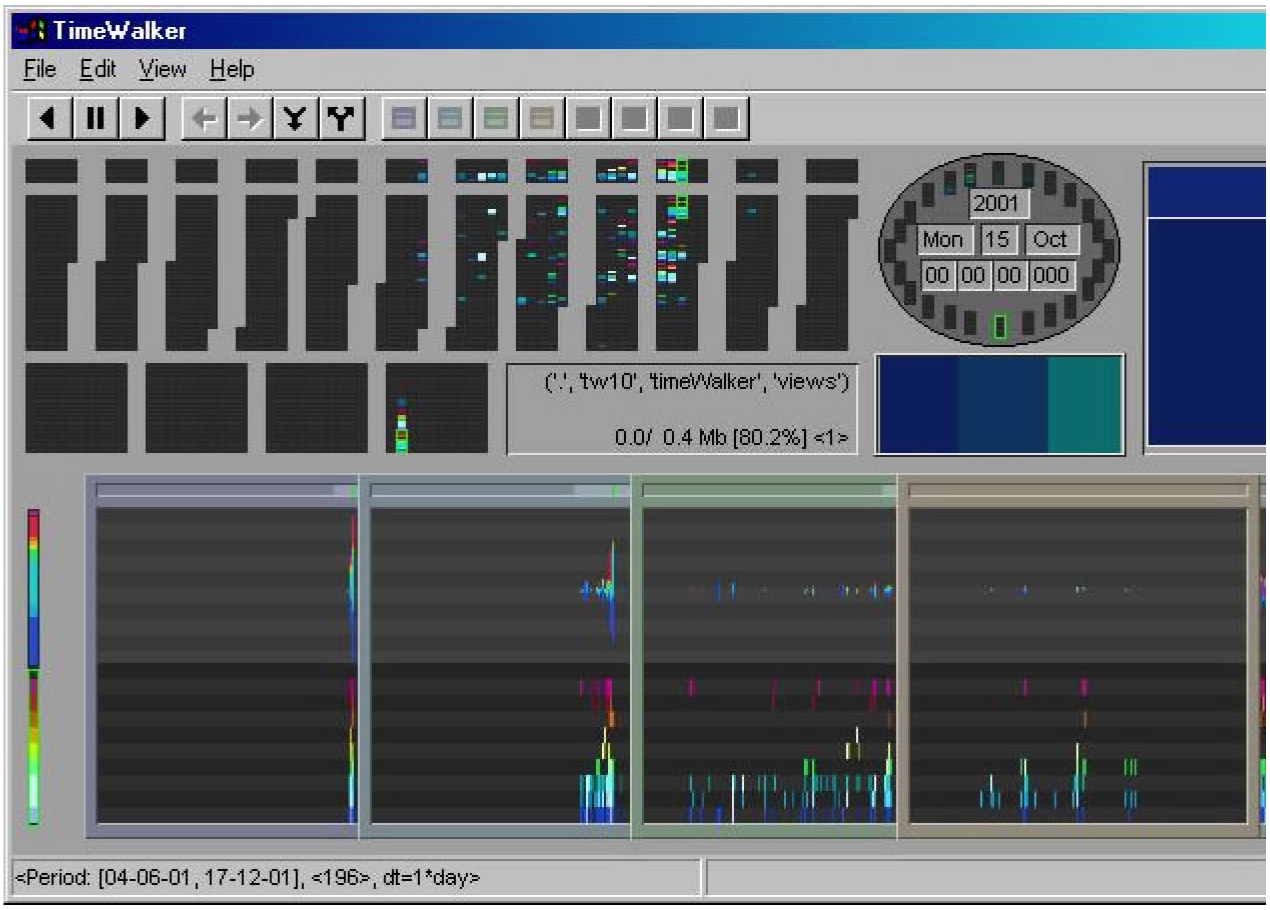
     The time corresponding with a mouse-position is shown continuously as text within the clock at a fixed place. This is used in stead of static and space-consuming textual scales.

   - Show a 24 hours distribution

     A vector selected in a Beam is aggregated and showed over the 24 daily hours.

4. Treemap

   When an aggregation is keyed it is possible to show the aggregation as a treemap. A treemap can be seen as a hierarchical pie-chart. The hierarchy is described by the key (being a tuple), the value to be accumulated is the sum field of the aggregated values.



('.', 'tw10', 'timeWalker', 'views')

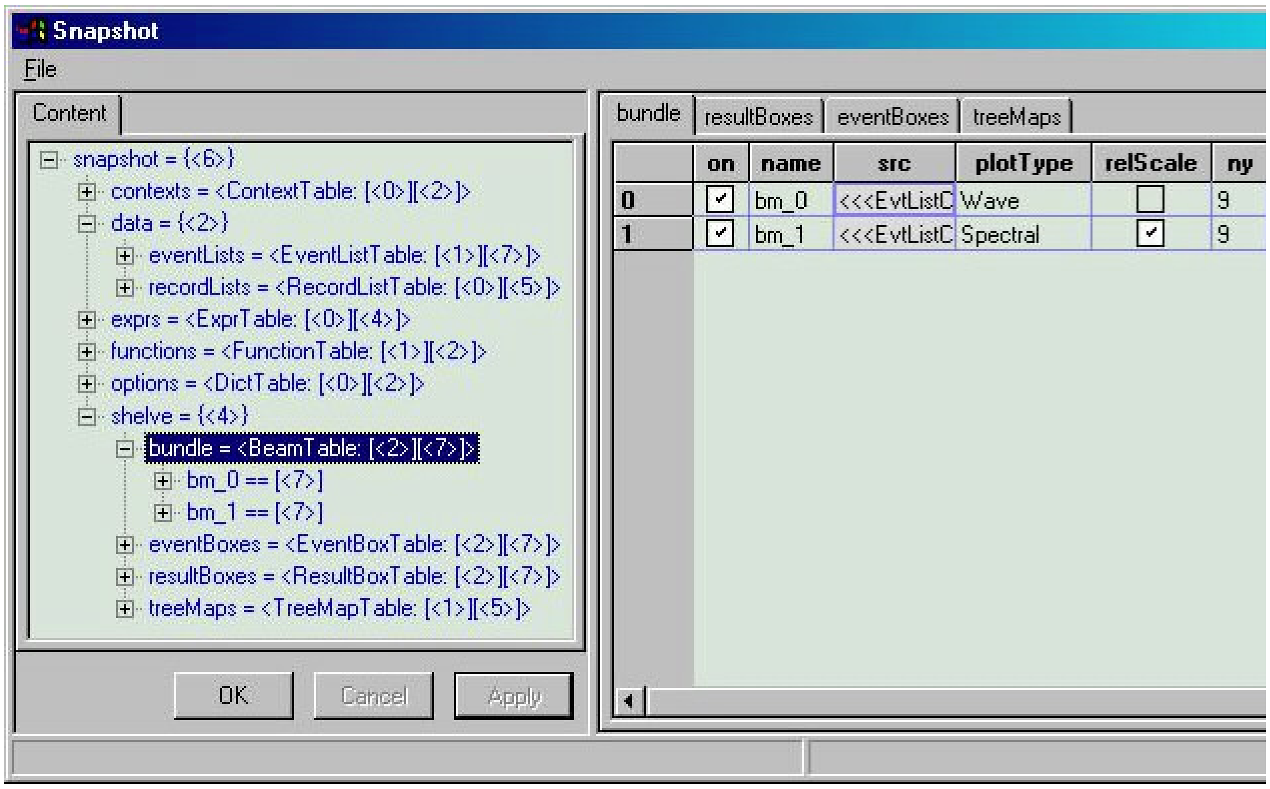
0.0/ 0.4 Mb [80.2%] <1>

<Period: [04-06-01, 17-12-01], <196>, dt=1*day>

Figure 3.

The graphical user-interface as a whole (see figure 3) is carefully designed for quick pattern-recognition by a regular user. Each part has a fixed place, there is no scrolling, the information density is high, scaling and coloring is automatic, and there is (almost) no static and redundant (textual) information.

Apart from the graphical mainwindow there are also frames for textual browsing and manipulating snaphots, data, documentation and sources. All textual navigation is based on data-driven tree/table-grid (see figure 4.) combinations.



```
⊟ snapshot = {<6>}
   ⊞ contexts = <ContextTable: [<0>][<2>]>
   ⊟ data = {<2>}
      ⊞ eventLists = <EventListTable: [<1>][<7>]>
      ⊞ recordLists = <RecordListTable: [<0>][<5>]>
   ⊞ exprs = <ExprTable: [<0>][<4>]>
   ⊞ functions = <FunctionTable: [<1>][<2>]>
   ⊞ options = <DictTable: [<0>][<2>]>
   ⊟ shelve = {<4>}
      ⊟ bundle = <BeamTable: [<2>][<7>]>
         ⊞ bm_0 == [<7>]
         ⊞ bm_1 == [<7>]
      ⊞ eventBoxes = <EventBoxTable: [<2>][<7>]>
      ⊞ resultBoxes = <ResultBoxTable: [<2>][<7>]>
      ⊞ treeMaps = <TreeMapTable: [<1>][<5>]>
```

	on	name	src	plotType	relScale	ny
0	✓	bm_0	<<<EvtListC	Wave	☐	9
1	✓	bm_1	<<<EvtListC	Spectral	✓	9

Figure 4.

Graphical and textual visualizations both scale up for realtime interactive manipulations and animations of very large datasets.

6. Usage

TimeWalker introduces a new way of looking at eventdata of which the benefits are yet to be validated in practice.

The learning curve will be short by starting given templates for domains that are well known. Those templates are basically snapshots. A snapshot is a persistent representation of a running state that enables continuation, exchange and reuse of any intermediate result. In the first release there will be templates for logdata like syslog and wtmp. There will be also a template that handles all files in a filesystem as events with their `mtime` as timestamp. Applying this template on huge file-systems is intended to be an eye-opener for the unexpected capabilities and added value of *TimeWalker* in a very familiar domain.

For external dataformats there is a general import/export dataconvertor class. Subclasses for common formats that are line-based or XML-based are given as examples. First experiences showed that creating and testing a new convertor can be realized within one day by a Python programmer.

All parametrizing is table-driven, but can also be done by writing scripts. Table-modifications are completely supported in the user-interface. Flexibility is obtained by using Python-expressions in some of the table-entries. Those expressions are not complicated as such, but making the right choices requires domain knowledge as well as some experience with the resulting visual effects.

The aggregation-mechanism used for visualization can also be used for storing the original data into a new persistent format that is compressed within one of the 8 available time-scales. The trade-off is between the acceptability of a certain loss of information and the ability to scale up into a GB range of events.

6. Some implementation aspects

The gui-library used is that of wxPython, but there is only a weak coupling with the peculiar features of this library. All graphics that is crucial for performance is painted within caches by simple bitblitting and line-drawing. Standard widgets are just encapsulated in a Python-style. A more specific requirement is the availability of a tree and a table widget that scale, enable drag/drop and have cells with in-place facilities for editing. The behavior and portability of standard wimp-aspects is inherited from the used library. The look-and-feel of the main graphical-window however is an invariant quality of *TimeWalker*.

All persistent storage is based on the pickle-mechanism offered in Python. It was one of the big surprises in using Python for huge amounts of data to discover the generality, robustness, speed and compactness of dumping and loading data directly as semantic rich objects.

A simple grammar, based on indentation and supported by any Python-aware editor, is used for writing all documentation within docstrings. The format is very gentle for eyes that do not like redundant brackets or tags. This way of integrating documentation can be seen as a new incarnation of the old idea of literate programming. The on-line documentation is generated from the docstrings in a html-widget on the fly without the need to start up a separate browser.

References

[1] Christopher Alexander, "The Timeless Way of Building", Oxford University Press, 1979.

[2] S.K. Card, "Reading in Information Visualization, Using Vision to Think", Morgen Kaufmann, 1999.

[3] J. Raskin, "The Humane Interface", Addison Wesley, 2000.

[4] D.M. Beazley, "Python Essential Reference", New Riders, 2000.

[5] M. Hammond, A. Robinson, "Python Programming on Win32", O'Reilly, 2000.

[6] Th.F. de Ridder, "Informatie Visualisatie als Beheerinstrument", IT beheer jaarboek 1999.