Project Report

EVolve: An Extensible Software Visualization Framework

Student : Wei Wang
ID : 110148252

Table of Content

Abstract
This report presents EVolve – an extensible software visualization framework, which is used to generate visual representation of program traces. In the report we describe all available visualizations in EVolve's built-in visualization library and explain how to use these visualizations. We also explain some new features implemented on EVolve and the modified infrastructure.

1. Introduction
The objective of developing EVolve is to create a tool to help us understanding program running behaviors. The tool should be easy to use and easy to extend. To serve this purpose, EVolve is designed to be an open and extensible software framework. As an extensible framework, it is easy to implement new data sources and new types of visualizations and integrate them into EVolve. As an open framework, EVolve is publicly-available and defines interfaces for new data sources and visualization in the forms of Java APIs.

1.1 Design
In order to achieve extensibility, we define a data protocol and visualization protocol. The data protocol provides definitions of data records (**elements**). These data records are classified as either **entities** (static information) or **events** (dynamically occurring events). More specifically, each field in an element is associated with a specific **property**, which allows EVolve to convey information to the visualizations and to automatically create appropriate menus and input fields for visualization creation and configuration. Although EVolve offers a set of visualizations in its built-in library, it is always possible to extend the library and create new visualizations whenever necessary. To create a new visualization, a user implements the interfaces defined in the visualization protocol. The visualization protocol provides interfaces for visualizations to implement. These interfaces enable a visualization to extract appropriate data from the data stream and to present these data in a visual representation.

1.2 Motivation of the project
We have been using the old version of EVolve for quite a long time in our class and research. Although it is very helpful, we found that there still exists space for improvement. For example,

- The hierarchy of the visualization library is not defined well. Legacy codes can only be reused by copy and paste, which makes code maintenance error-prone. A more object-oriented design needs to replace the current one.
- EVolve is not very user-friendly:
  - The user has to remember a lot of settings in order to reproduce an experiment exactly
  - The visualizations created by EVolve are coarse grained when trace files are large, and there is no way in EVolve to help the user get a fine grained view. This makes EVolve less useful when it is used on large trace
  - Too may steps are required to complete a relatively simple operation
- Not enough tools are offered to help the user analyze traces.
- Only one data trace is allowed at a time, which is inconvenient for the user to compare different traces.

This project aims to make EVolve a more useful tool.

2. Architecture
As shown in Figure 1, EVolve consists of three parts: The leftmost part is the data protocol: all data sources should use this protocol to translate the input trace into EVolve's abstract representation. The middle part is the core platform of EVolve: this part is fixed and can not be extended or modified. The main task of this part is passing information between the left and the right part. It serves to de-couple input data formats from output visualization, enabling new visualization to use existing data traces and new data sources to be visualized using existing visualizations. On the right side of the architecture is the visualization protocol. All visualizations obey this protocol to obtain data

records and present these records in visual representations. This protocol also defines default behaviors for visualizations.

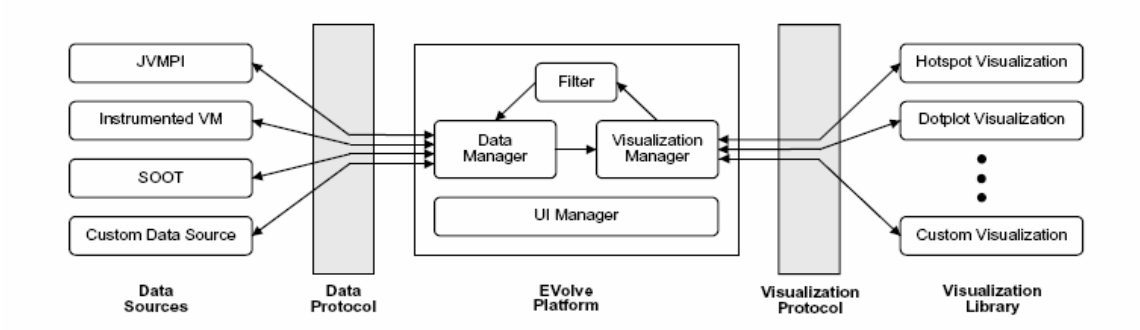

Figure 1 Architecture

2.1 Data Representation

EVolve allows multiple data sources and defines an internal abstract data representation. EVolve can visualize any data source if it is able to translate its own data into this abstract representation.

There are two main data elements in the data representation. The first is **entities.** Entities are named, unordered data elements that remain unchanged in the visualization process. They are cached in the memory by EVolve. **Events** are anonymous, ordered and dynamic data elements. The program's behavior is denoted by these events. Typically 99% percent of elements in a data trace are events stored on disk, while entities occupy memory, so using this abstraction will save EVolve a lot of memory.

Each element contains a set of fields. These fields are either entity references (which will be referred to as a reference) or values. Additionally, each field (value or reference field) is associated with a property. There are several built-in properties:

- **time**: this property means that a data value contained in the field is monotonically increasing, and it is used as a definition of time
- **coordinate**: this property means that the data contained in the field is non-numeric or not summable.
- **amount**: this property means that the data value contained in the field is numeric and summable.
- **reference:** this property means that the data value contained in the field is simply a reference referring to an element. By default, all reference fields will have this property.

Besides these properties, EVolve also allows arbitrary custom properties. We will talk about properties in 2.3.1 in more detail.

2.2 Visualization

Visualizations are required to provide an abstract representation of their capabilities in order to make them as flexible as possible. Visualization's capabilities are defined in terms of **dimensions**. Each visualization defines its dimensions that associated with **properties**. Dimensions can either be values or references. For example, the Bar Chart visualization in Figure 7 declares a reference dimension and a value dimension, where the value dimension is used for the length of the bars. In EVolve, an event is encapsulated as an array of type long. Each element in the array has a **property** associated with it. And for every property, EVolve will create a **data filter**, which simply contains the element index of that property. According to the user's configuration on axes (refer to Figure 5), EVolve will get the corresponding **data filter** and plug it into the dimension. Then by using *dimension.dataFilter.*

getData(), the visualization can extract correct field from the event.

2.3 Core Platform

In EVolve, the core platform is fixed and not modifiable. This part provides functions to manage the user interface and handles communication between data sources and visualizations.

2.3.1 Generating the Appropriate UI

As an extensible frame work, EVolve may have several visualizations and data sources. Both data source and visualization could have their own requirement on the user interface, e.g., a stack visualization makes no sense on a data source unless the data source involves method calls, and we should prevent

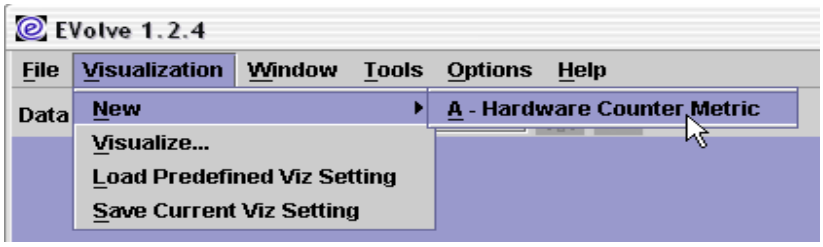

Figure 2 Menu Screenshot 1

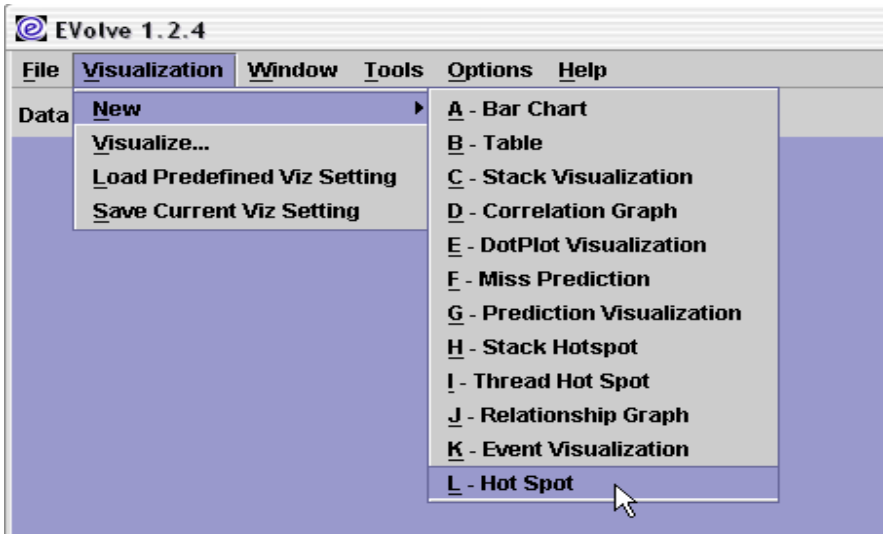

Figure 3 Menu Screenshot 2

showing these visualizations in the menu. Since the core platform is the only one who knows both sides, it is responsible to generate the appropriate user interface correctly.

A data source translates the data stream of the trace file into elements and assigns properties to every field in elements. Remember that every visualization in EVolve has two or more dimensions and these dimensions also have properties. Therefore by matching the properties provided by the data source with the properties attached on dimensions, the core platform is able to decide which visualization is appropriate for a certain data source. Furthermore, the core platform also uses dimensions' properties to generate correct configuration user interface. For example, imagine we have two different data sources and following properties are defined in these data sources:

Data source 1 : PMCPentiumSource	
Properties :	"bytecode", "eventvalue"

Data source 2 : DemoSource	
Properties :	"time", "count", "amount", "coordinate", "thread", "reference"

And in our visualization library we have two visualizations:

Visualization 1: Hardware Counter Metric	
Dimension	Attached Property
X Axis :	“bytecode”
Y Axis :	“eventvalue”

Visualization 2: Hotspot	
Dimension	Attached Property
X Axis :	“time”
Y Axis :	“reference”

In Figure 2 and Figure 3 we load a PMCPentiumSource trace and a DemoSource trace accordingly. As we can see, the Hardware Counter Metric Visualization only appears in Figure 2, because the core platform can not find the “bytecode” and “eventvalue” property (attached on the dimensions of visualization 1) in data source 2’s property definition. Thus the core platform prevents “Hardware Counter Metric” from appearing in the menu of Figure 3. Similar thing happens for visualization 2 in Figure 2.

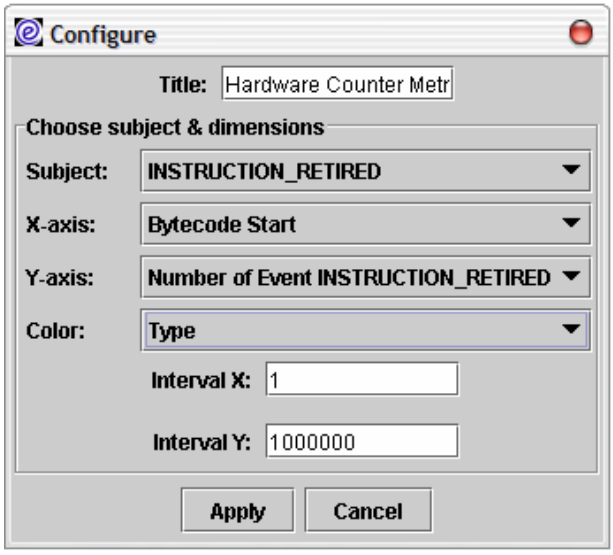

Figure 4 Configuration Screenshot 1

Figure 4 and Figure 5 are configuration boxes corresponding to visualization one and two. The subject of the visualization is simply the type of event that is visualized. Whenever the type of event is decided, the core platform matches the properties attached on dimensions with the properties of fields of the element and creates the drop-down lists. Again, since dimensions of visualization one and two have different properties, the core platform automatically generate correct configuration boxes.

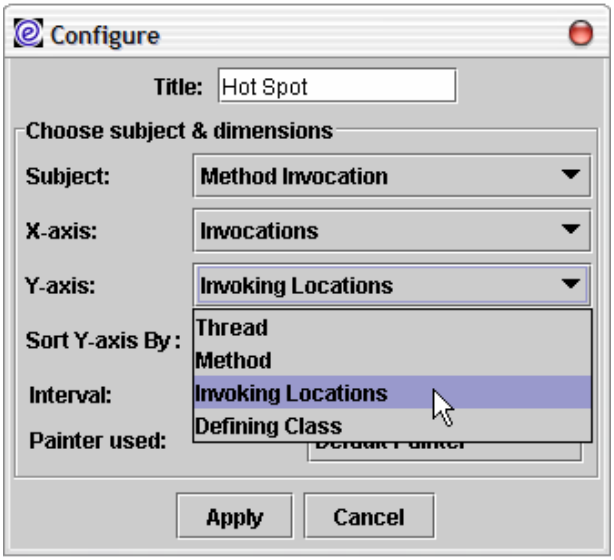

Figure 5 Configuration Screenshot 2

2.3.2 Handling Communication

Besides creating a correct user interface, the core platform also handles communication between the data source and the visualization. As we know, in order to prompt extensibility, EVolve does not allow a visualization to talk with a data source directly. When a visualization wants to communicate with its data source, it must depend on the core platform. A good example of this mechanism is filtering (as shown in Figure 1). Imagine the following scenario: The end user makes a selection in the visualization in order to concentrate on a certain part of data. The visualization accepts this request and passes it to the core platform with necessary data. As a response, the core platform will create a filter based on the information sent by the visualization and re-connect the data source with the visualization through the filter. And next time when the visualization is re-visualized, only filtered events will be passed to it.

3. New Visualization Library

An end-user can use EVolve as a stand-alone tool and interactively create or modify multiple visualizations from an existing library. The ability to view a particular data source in a variety of ways greatly adds to the benefit of visualization. In this section, we will introduce all built-in visualizations.

3.1 Hierarchy

Before we start, let's first take a look at the hierarchy of the visualization library. The old version of EVolve did not define its hierarchy very well and code reuse is not efficient. In this project, we re-defined the hierarchy. According to dimensions of visualizations, the new hierarchy looks like this:

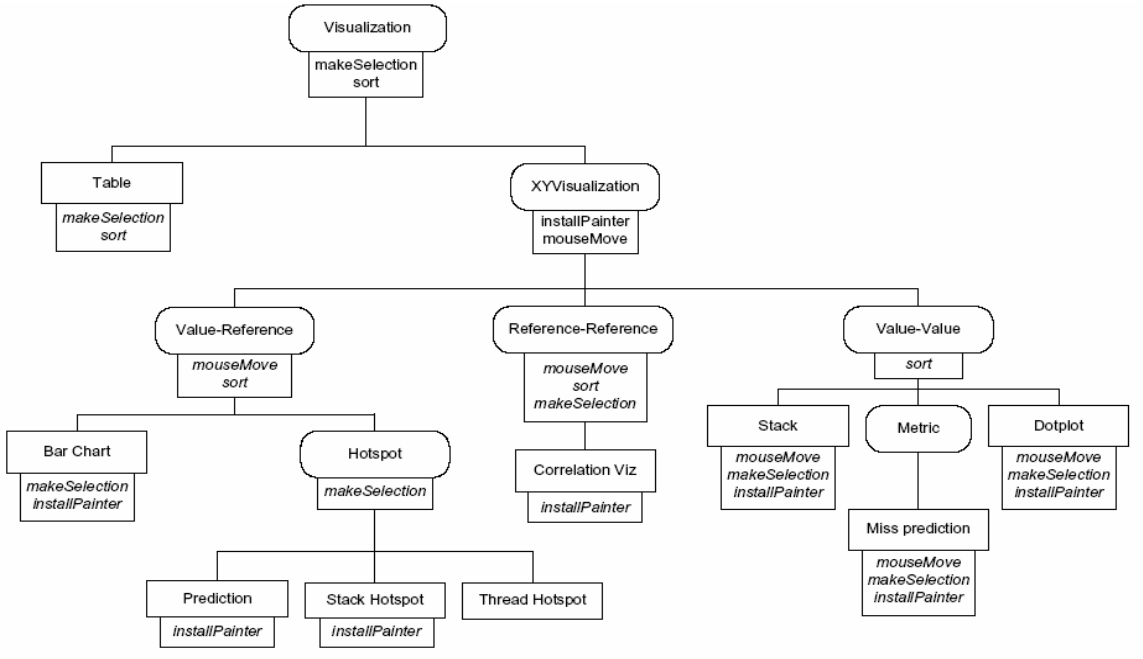

Figure 6 Hierarchy

Abstract classes are shown in rounded boxes, concrete classes in plain boxes. Overridden methods are shown in italics, concrete methods in plain text. Abstract classes provide functionality that all visualizations in the sub-tree have in common.

Visualization declares two methods which all visualizations share: *sort* (allows data to be sorted) and *makeSelection* (allows the user to select a subset of data to be re-colored or re-visualized)

XYVisualization declares two methods: *installPainter* (defines the color scheme used in the visualization) and *mouseMove* (determines the text that appears when the user moves mouse over a particular part of the visualization). Depending on the way that coordinates on x- and y-axis are treated, **XYVisualization** has three variations:

- **Value-Reference** visualizations contain a reference on one axis. They share *mouseMove* and *sort*
- **Reference-Reference** visualizations contain a reference on both x and y axes. They share *mouseMove, sort* and *makeSelection*
- **Value-Value** visualizations contain a value on both x and y axes. This variation is more open-ended, only *sort* is shared and all other methods are overridden.

3.2 Barchart

A bar chart (Figure 7) is a **Value-Reference** visualization. The reference axis (y-axis) shows 882 method invocation locations, each location has its own unique color. The x-axis shows the

total number of invocations occurring at each location. You may notice that some bars are "broken" and with number labels on them, this is a new feature offered in this project, we will talk about it in 4.1 in more detail. The user can find out the name of a particular invocation location by pointing the mouse on it (the name is displayed on the status bar).

3.3 Hotspot

Figure 8 illustrates four different hotspot visualizations. Hotspot visualizations are value-reference visualizations. The reference axis (y-axis) is same as the reference axis of bar chart in Figure 7, showing all different invocation locations in lexical order. The only difference is that hotspot visualizations use x-axis to show the passing of time. Here we use bytecode executed as time (Totally there are 69,735 bytecodes executed, they are grouped in samples of 5000 each).

3.3.1 Ordinary Hotspot

The top left window of Figure 8 is an ordinary hotspot visualization. This visualization is the super type of all hotspot visualizations. Hotspot visualization illustrates when and for how long certain parts of a program become active. There are two painters available for this visualization: **Default Painter** and **Random Color Painter**. The first one is offered by the core platform. It outputs blue color if there is no filter available in the core platform and outputs user selected colors if filters are colored. The **Random Color Painter** gives every entity a unique color and overrides the **Default Painter**, in another word, even if the end user generates some colored filters, the **Random Color Painter** will still generate its own color.

3.3.2 Stack Hotspot

The top right window in Figure 8 is stack hotspot. This visualization extends the original visualization and all it does is just creating a new painter and install it through *installPainter.* The new painter uses three different colors to show within a given time sample all methods that are called, on the stack but inactive, on the stack and active.

3.3.3 Thread Hotspot

This is another variation of hotspot visualization, showed in the bottom left window of Figure 8. Similar to **Stack Hotspot**, this visualization also create a new painter and applies it using *installPainter.* This painter colors entities according to the thread they belong to, e.g., entities of the same thread will have same color.

3.3.4 Prediction Hotspot

The final hotspot variation, **Prediction Hotspot**, is displayed in the bottom left window. In this visualization we generate colors by "predicting" the next entity using a simple last-value predictor: a blue color indicates perfect prediction accuracy, a red color appears when the predictor guesses the wrong target entity at least once in the time sample. By sub classing *Predictor* the end user can import a more sophisticated and accurate predictor to generate different prediction hotspots.

3.4 Stack Visualization

Stack visualization (Figure 9) is a new visualization added into our library. This visualization is a value-value visualization. The y-axis shows the runtime stack and the x-axis measures time as method invocations. The color scheme used in stack visualization assigns each method a unique color. Stack visualization is very memory consuming, so normally we only select a segment of data in the configuration box (as shown in the figure).

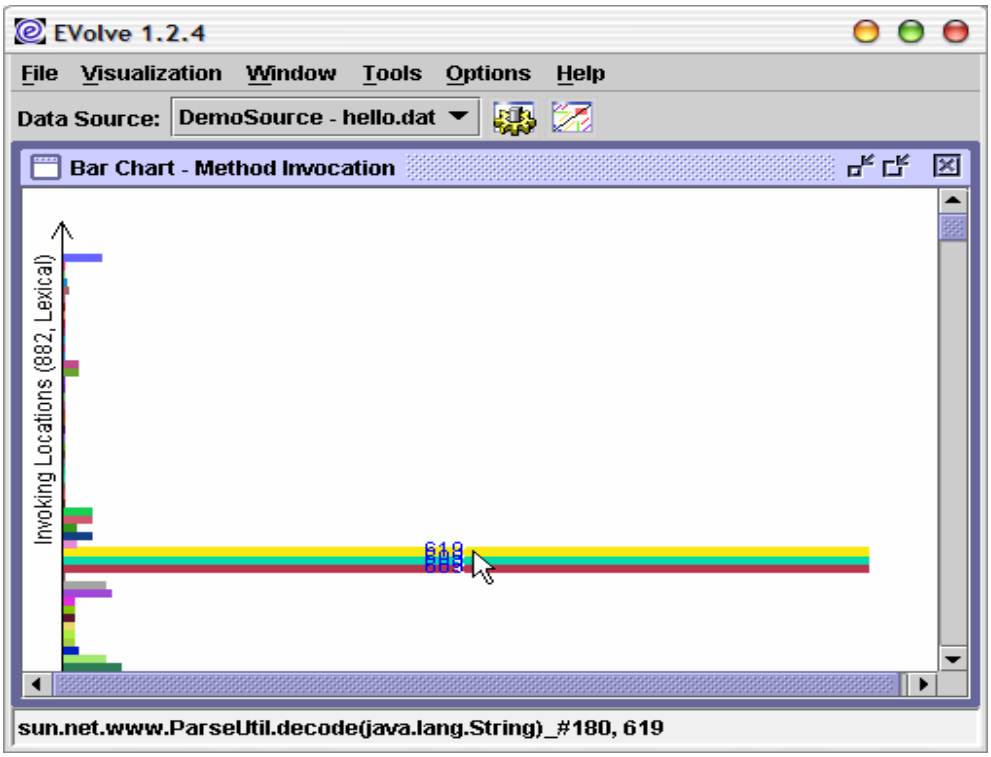

Figure 7 Barchart

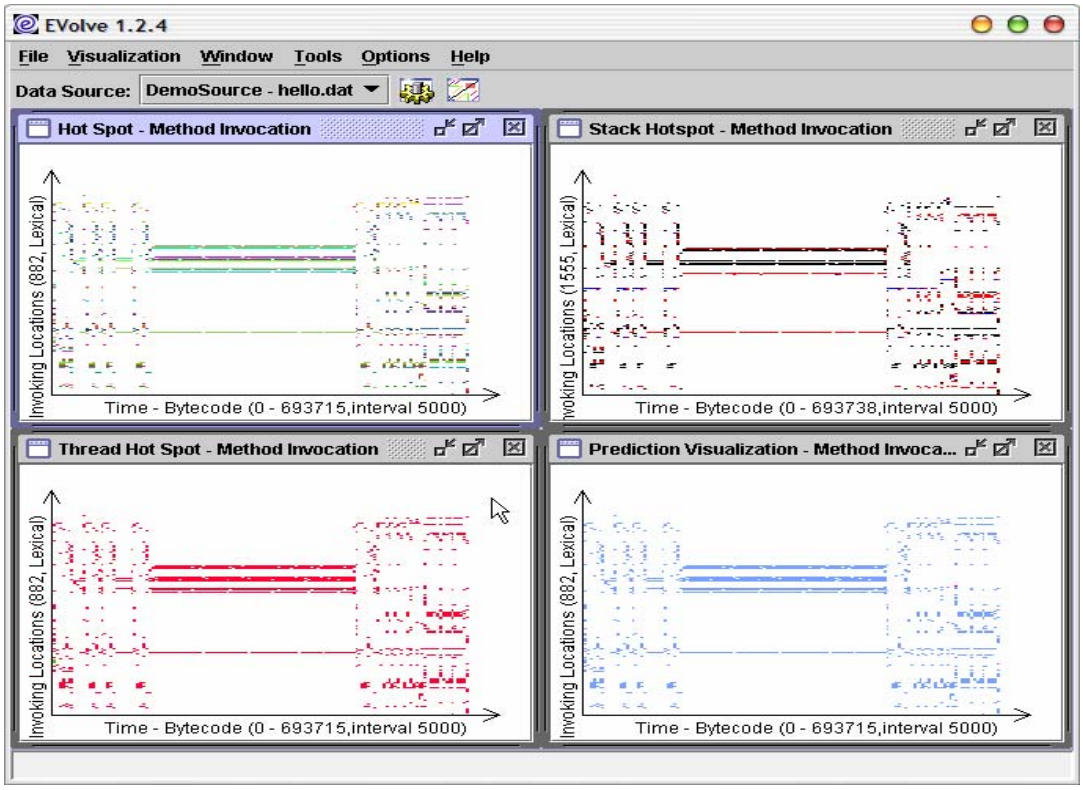

Figure 8 Hotspot

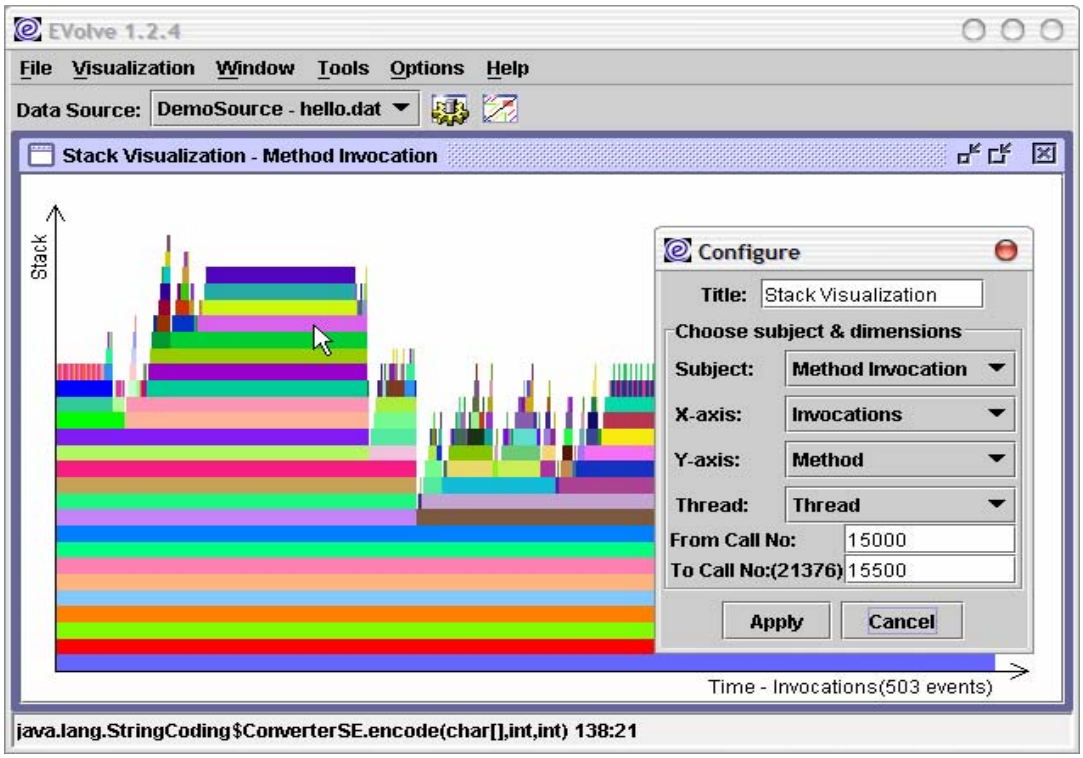

Figure 9 Stack Visualization

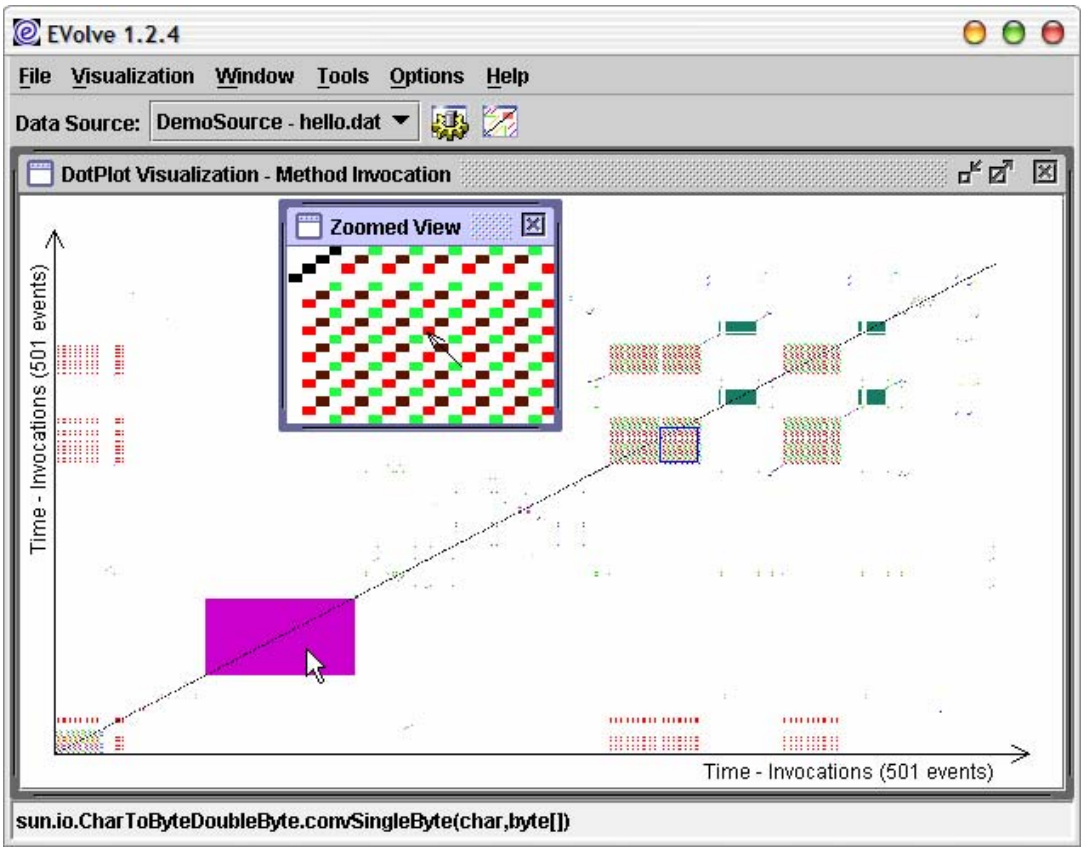

Figure 10 Dotplot Visualization

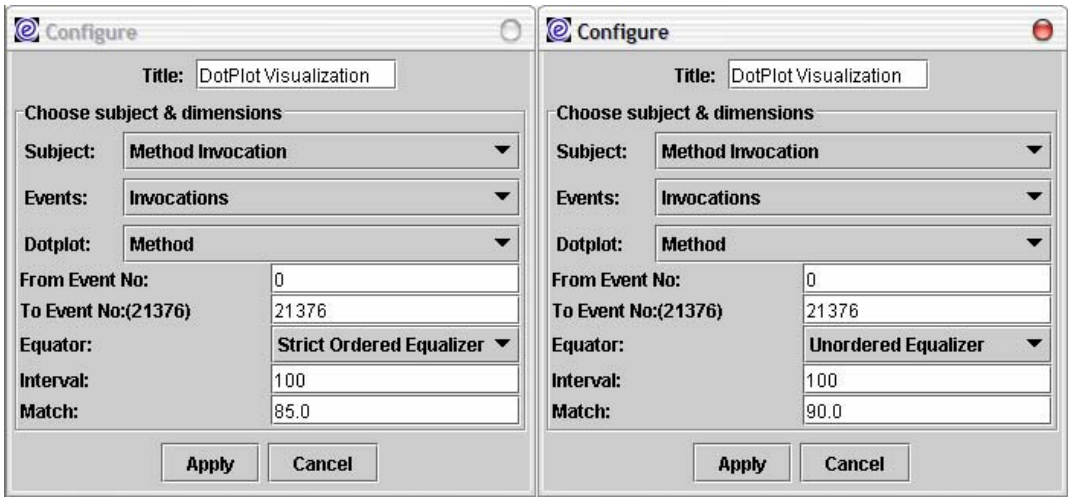

Figure 11 Weaker Equality

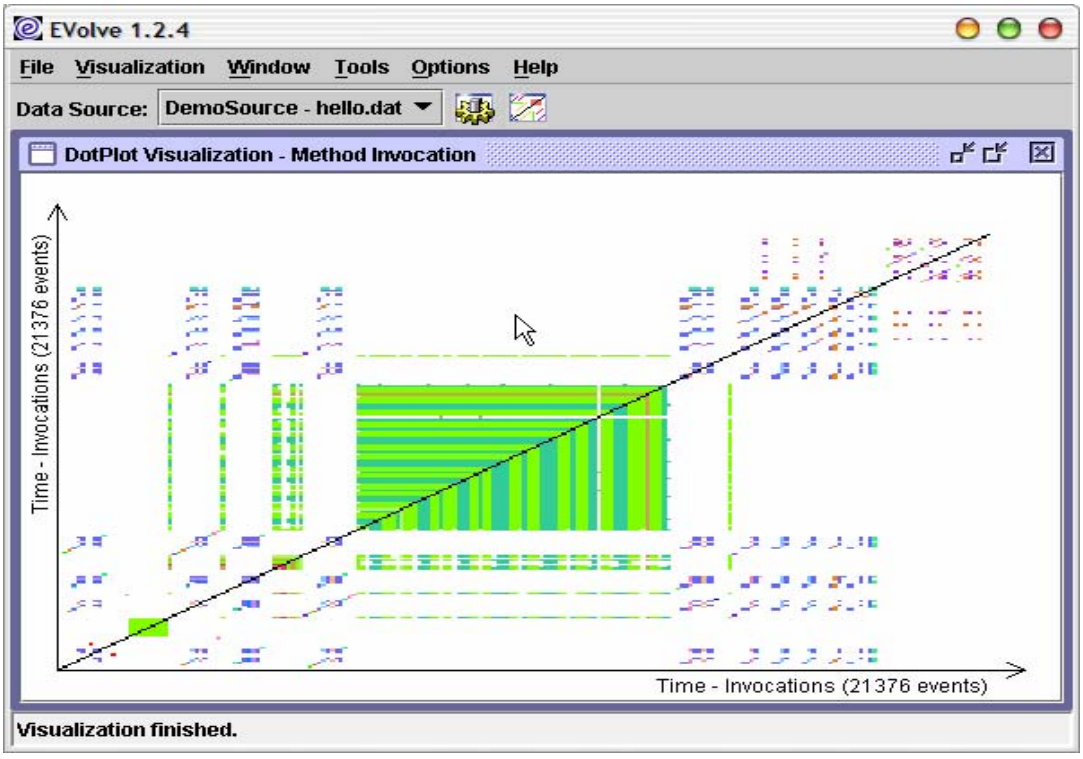

Figure 12 Dotplot Visualization with Weaker Equalizer

3.5 Dotplot Visualization

Dotplot visualization is also a value-value visualization. The purpose of this new visualization is to highlight the repetition in any sequence of values. X-axis and y-axis here is identical. If a value in the sequence at index x is identical to a value at index y (the value repeats at time x and time y), a dot will be put at position (x, y) and (y, x).

Figure 10 illustrates a dotplot for method invocation. The solid purple block indicates method *sun.io.CharToByteDoubleByte.convSingleByte(char, byte[])* is invoked repeatedly. Striped blocks in the graph also represent repetitive behavior, but of a sequence of methods instead of a single method (see zoomed view).

In Figure 10 we used a perfect matching strategy, which means we match invoked methods one by one. This matching scheme is very precise but has a big limitation: it is very memory consuming and can not be applied to the whole trace file. To avoid this problem, we also defined two weaker definitions of equality.

Figure 11 illustrates two configuration boxes for dotplot visualization. In these two configuration boxes, we used interval and match threshold. Interval is used to group certain number of events together and compare set with set instead of event with event. Since 100 percent matching of two groups will be less possible when interval grows ever larger, we hence introduce a matching threshold. Furthermore, we also have two equalizers available:

- Unordered equalizer (the right one).

This equalizer compares two sets and finds out the number of elements shared between these two sets. The number is divided by the total number of elements in two sets in order to get the matching percent. EVolve decides whether two sets are matched by comparing the matching percent with the threshold.

- Strict ordered equalizer (the left one)

This equalizer compares elements in two sets according to the order they were added into. Then the number of identical elements is divided by the size of set. EVolve decides whether two sets is matched by comparing the matching percent with the threshold.

Figure 12 shows the result of a dotplot visualization using an unordered equalizer. Its configuration is shown in right configuration box of Figure 11.

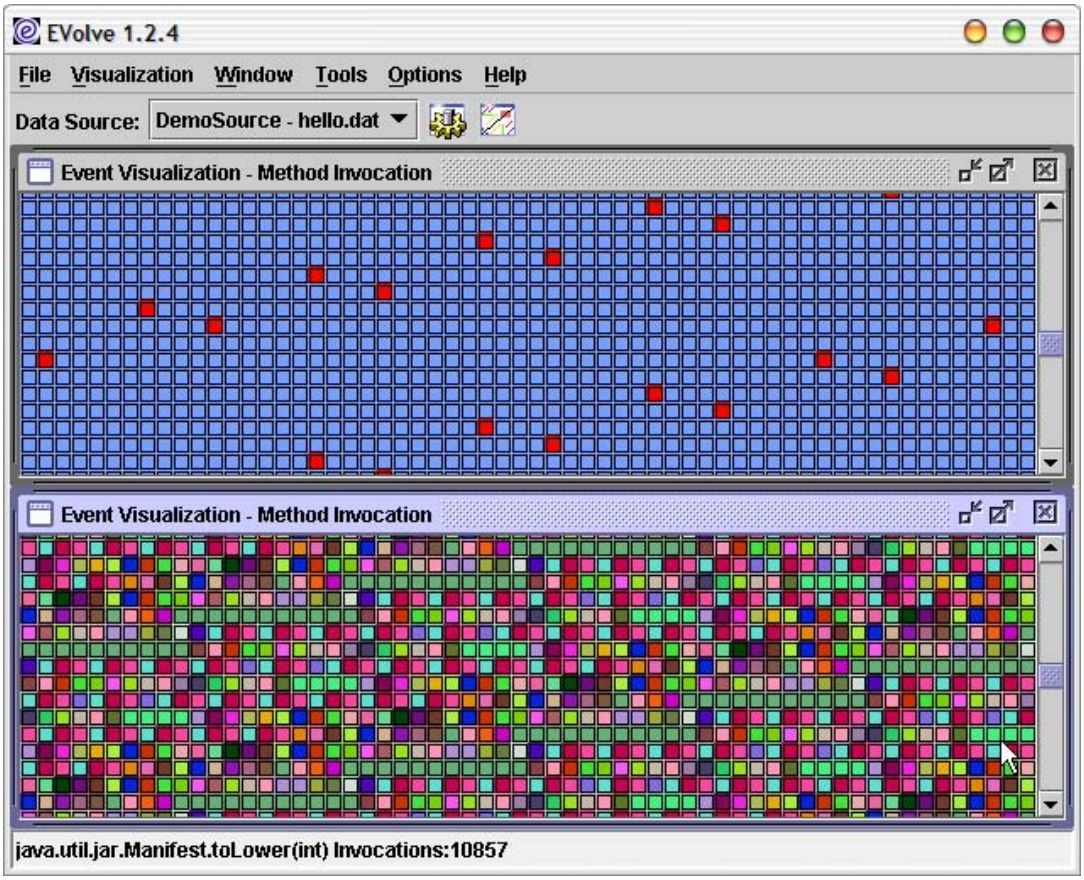

Figure 13 Event Visualization

3.6 Event Visualization

Figure 13 is a demonstration of an event visualization. In an event visualization, each event is displayed as a block. If a random painter is used, it is easy to find out patterns with this visualization. This visualization has three painters available: *RandomPainter, DefaultPainter* and *Predictor Painter.* The top window uses *PredictorPainter* and the bottom one uses *RandomPainter.* All these three painters are shared by this visualization and other visualizations. This visualization is also a value-value visualization.

3.7 Correlation Visualization

The correlation visualization, which is shown in Figure 14, is a reference-reference visualization. A correlation visualization shows a dot on coordinate (x, y) when a reference x occurs in the same event as reference y. Figure 14 illustrates the correlation between methods and invocation locations. Horizontal rows of dots indicate that several methods are invoked at the same location, i.e., the location is a polymorphic location.

3.8 Relationship Visualization

The relationship visualization (Figure 15) is a variation of the correlation visualization. There is no axis in this visualization, two references are placed in a circle and connected if they occur in the same event. The number of occurrence is also printed above the line. In this example we only displayed pairs of entities which have more than 100 correlations.

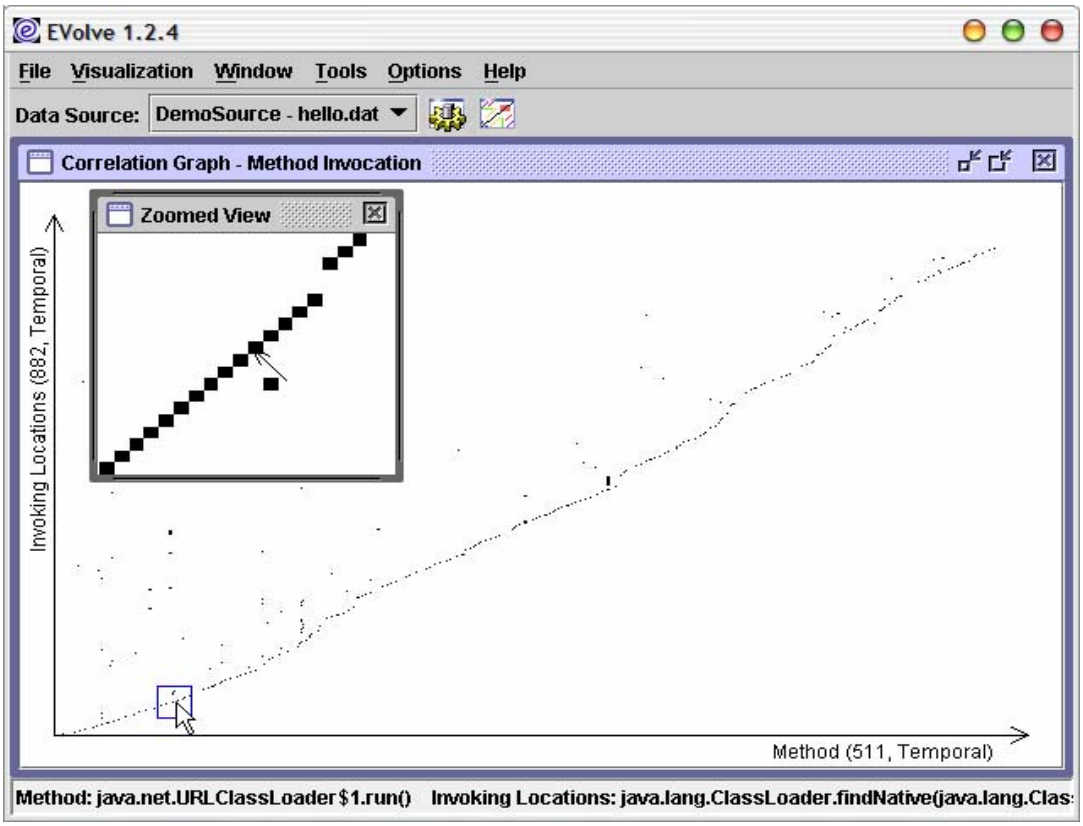

Figure 14 Correlation Visualization

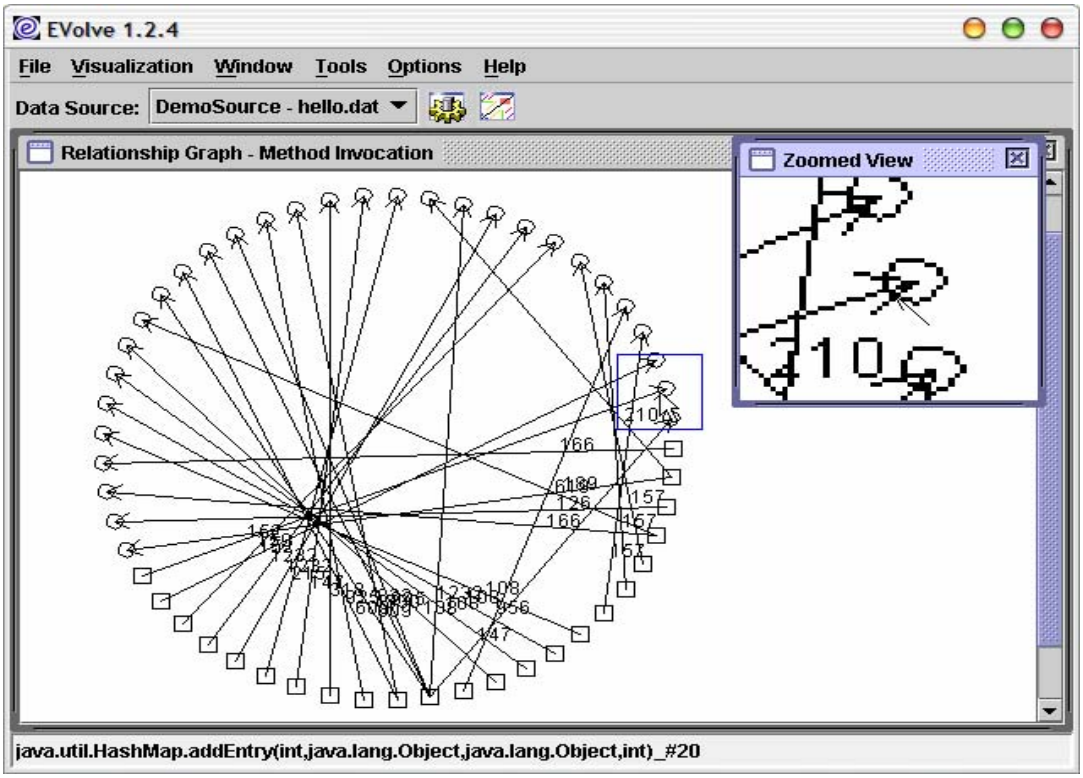

Figure 15 Relationship Visualization

4. New Features

In this project, we put lots of efforts in making EVolve more user-friendly. This section will introduce all new features implemented and explain how they work.

4.1 Enhanced Bar Chart

The old version of bar chart draws a pixel every time an entity occurs. The bar is actually a line of pixels. For a large trace this is very memory consuming. To release pressure on memory, we revised the bar chart visualization. In the new scheme, each entity has a real bar and the bar will be "broken" if it is too long. Number labels will be put on these broken bars (Figure 16). During the process of parsing the data trace, EVolve keeps modifying the length of bars without actually drawing them on the canvas.

4.2 Multiple Data Sources

One important usage of EVolve is comparing a program's behaviors when it is executed with different parameters. The old version of EVolve can only load one data trace at a time, although the user can launch two instances of EVolve, it is still not very convenient to compare two data traces. In this project, we added a new feature that allows EVolve to manipulate not only multiple traces but also multiple types of data source.

Figure 17 shows how to manipulate multiple data sources. There are two menu items under **File** menu (left part in Figure 17): **Add Data Source...** and **Remove Data Source...**. The first menu command allows the user to add a new data source (without loading trace file) into EVolve; the second one allows the user to remove a data source (with or without loaded trace file) from EVolve. If a new data source is added successfully, EVolve will automatically create a **Load a xxx Trace** menu item (e.g., **Load a EVolve.DemoSource Trace**), which allows the user add a new **xxx** data source and attach a trace file with it. Also, if a new data source is added correctly, an entry will be created in the drop-down list on the toolbar (see right part of Figure 17). This drop-down list allows the user to switch from one data source to another.

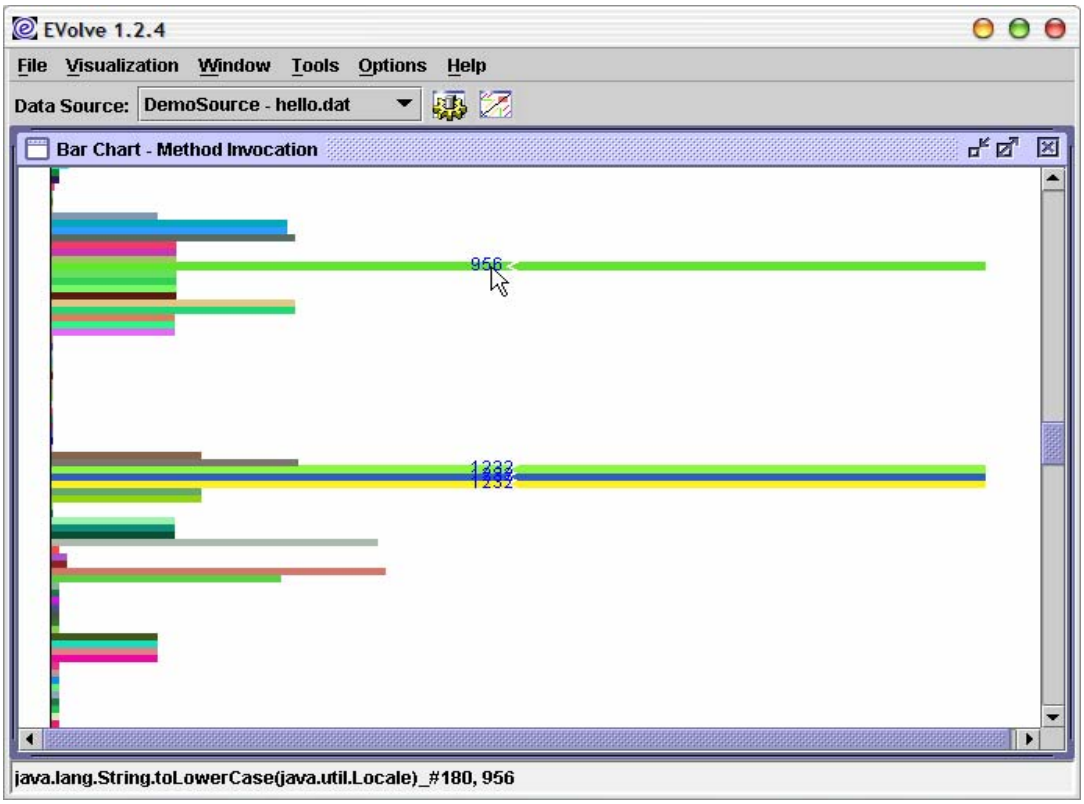

Figure 16 Revised Bar Chart

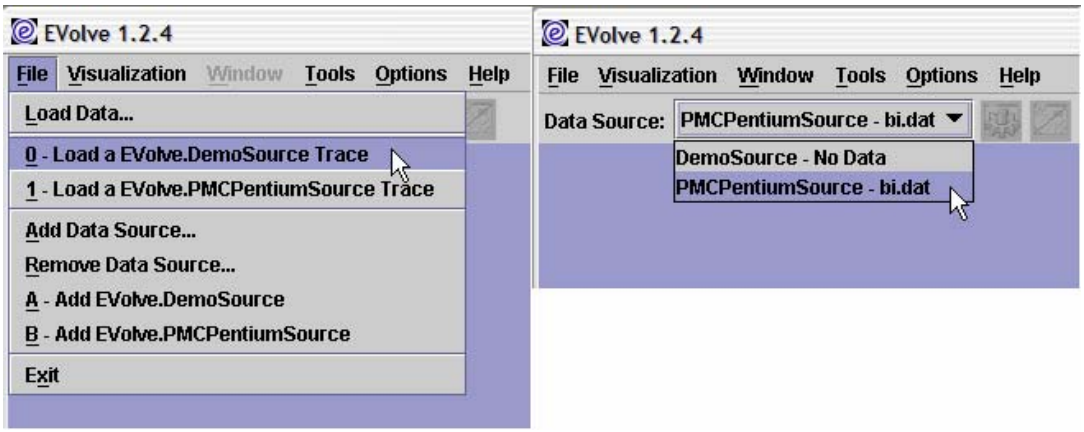

Figure 17 Multiple Data Sources

4.3 Predefined Visualization

When we finish analyzing a data trace with a set of visualizations, we may also want to use these visualizations to analyze other data traces. It is very painful to do this with the old version of EVolve: you have to remember configurations for every visualization you created and set them back when a new trace file is loaded. Now the new feature, **Predefined Visualization**, helps the user to remember and restore these configurations. With this feature, the user can save everything on the screen (configurations, window size, window title etc.) into a XML file. Figure 18 illustrates a sample configuration file segment.

Besides saving visualization configurations, the user can also choose whether or not to save filters (refer to 4.8) created, as shown in Figure 19. There is one thing to be noted: if filters are saved, the configuration file can only be applied

on the current trace file. This is because a filter is only valid in the data source on which it is created. A configuration file can be divided into two categories: *self-loadable* or *not self-loadable*. A *self-loadable* configuration file records a trace file name in it. If EVolve loads a *self-loadable* file, it will also try to load the trace file.

When EVolve starts up, it will look in to the *Default Viz Config Path* (refer to section 4.12), parse all configuration files in that path and create menu items for each configuration under **Visualization** menu. All menu items corresponding to *not self-loadable* configurations will be disabled. Clicking a *self-loadable* menu will append a new data source and load a trace on that data source.

```
    <PathForResult />
    <NumberOfVisualizations>2</NumberOfVisualizations>
    <NumberOfSelections>0</NumberOfSelections>
    <SelectedSelection>-1</SelectedSelection>
  - <SerializedVisualization>
      <VisualizationName>Hot Spot</VisualizationName>
      <FactoryName>Hot Spot</FactoryName>
      <SubjectName>Method Invocation</SubjectName>
      <xAxis>Bytecode</xAxis>
      <xAxisSortScheme />
      <yAxis>Method</yAxis>
      <yAxisSortScheme>Lexical</yAxisSortScheme>
      <zAxis />
      <zAxisSortScheme />
      <PredictorName />
      <PainterName>Default Painter</PainterName>
      <EqualizerName />
      <PlacementName />
      <WindowTitle>Hot Spot - Method Invocation</WindowTitle>
      <Interval>5000</Interval>
      <BeginEvent />
```

Figure 18 Configuration File Segment

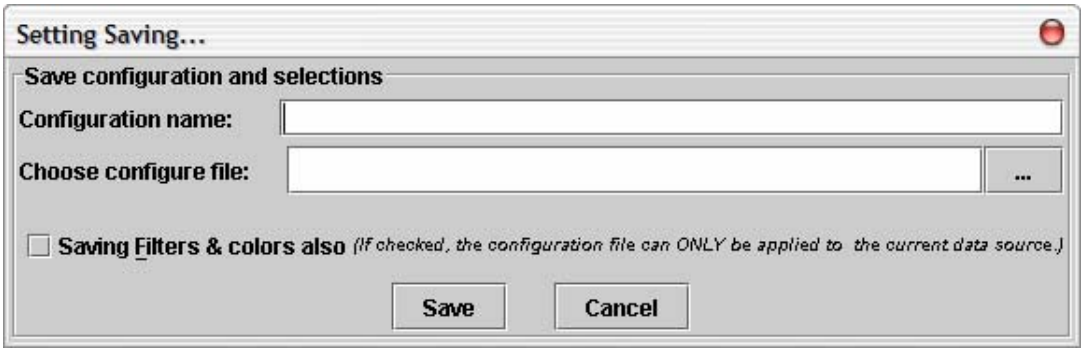

Figure 19 Configuration Saving

4.4 Overlapping

It is always a difficult task to find out the similarity shared by two visualizations. Correlations between two visualizations are most obvious when visualizations are overlapped. Common information then appears in the same physical location, and distinct information appears in different locations; this can be quite informative for quick comparisons. It, for example, allows for rather easy identification of related "phases" in execution – the startup phase common to each program is certainly quite obvious.

EVolve provides a tool to help the user overlap visualizations. In order to make overlapping to be meaningful, x-axis and y-axis of both visualizations must be unified. Two references axes are unified by building a new reference axis that contains the union of the two overlapping visualizations. Figure 20 is a sample overlapped visualization.

The top left window is created from the *HelloWorld* trace, the top right one is created from the *javac* trace. The bottom left window shows the overlapped result. The color scheme of an overlap is determined by each participating visualization: *HelloWorld* in red, *javac* in blue, with overlap in predefined purple. All Value-Reference visualizations can be overlapped. Note that the reference axis is sorted in lexical order, since the temporal ordering of a union from two different programs is ill-defined.

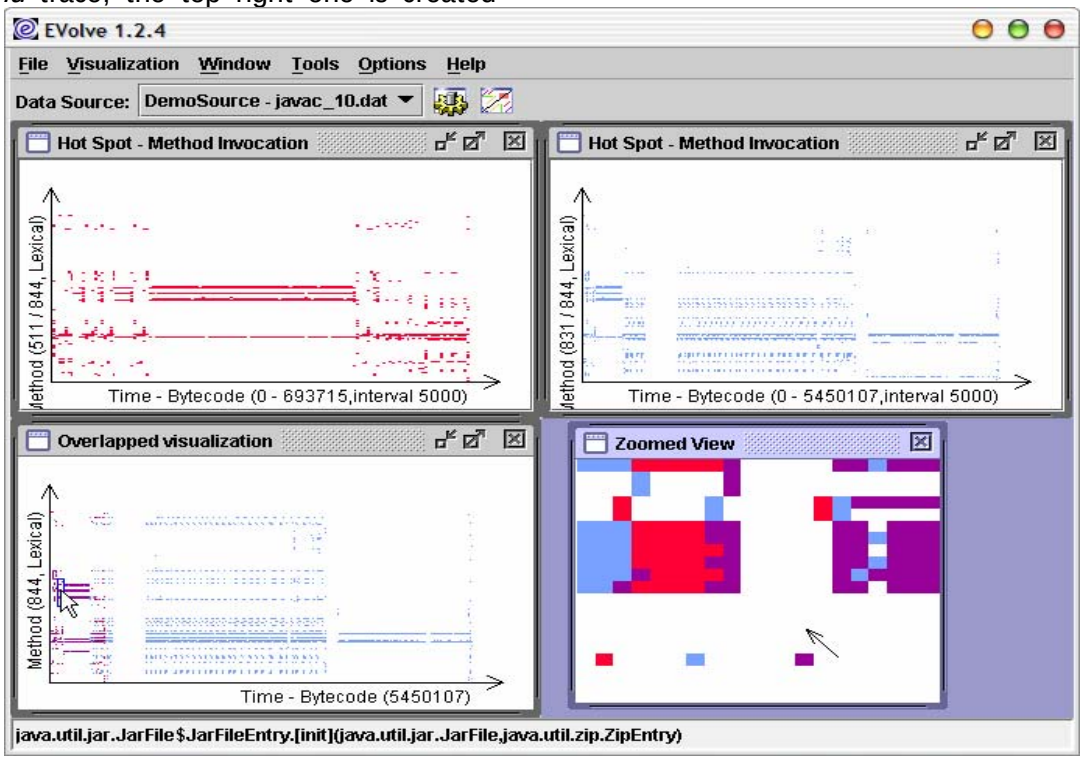

Figure 20 Overlapping Visualization

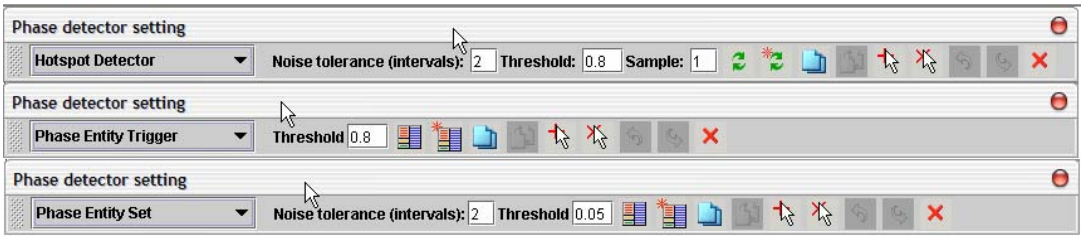

Figure 21 Phase Detector Toolbar

4.5 Generating Visualization Automatically

Generating visualizations for a set of data traces is very time consuming, EVolve has a tool for the user to create visualizations in batch. To use this tool, a configuration file must be prepared and field *PathForResult* (refer to 4.12) must be set.

4.6 Phase Detector

The Phase detector is a new tool we added to EVolve in this project. With this tool we can detect phases in all value-reference visualizations. Three different phase detectors are available (shown in Figure 21):

- **Hotspot Detector**. This detector has three parameters: **noise tolerance, threshold** and **sample**. The detector compares two succeeding n intervals, if the matching percent is higher than the threshold then the detector considers we are in a steady phase, otherwise in a changing phase. If the detecting result changes from a steady phase into a changing phase, or vice versa, a phase line is created. **Noise tolerance** is used to remove noise in the detecting process. For example, if the detector finds the following sequence: steady phase – changing phase – steady phase, and the **noise tolerance** is set to 2, then no phase line will be drawn.
- **Phase Entity Trigger**. To use this detector, the user must first make a selection on the visualization and then define a phase trigger entity, for example, a call to the method main(). The detector then checks every interval, if the trigger entity ever happens, a phase line is created.
- **Phase Entity Set**. Similar to **Phase Entity Trigger**, the user must make a selection first to create an entity set. The detector compares every interval with the entity set, for example, any calls to methods for unzipping class files, to designate class loading phases. If matching percent is higher than the *threshold,* a phase line is created.

Besides the phase detector, EVolve also has a phase clipboard. With the clipboard, the user can copy a phase detection result from one visualization and paste it to another, even if two visualizations have a different time measurement unit (as long as they apply to the same data source). Figure 22 shows an example of the phase clipboard. Numbers above phase lines are interval indices. This is useful, for example, to watch intense object allocation phases to their method calling counterparts. For example in Figure 22, the upper visualization plots method invocation (x-axis: bytecode, y-axis: methods invoked) and lower visualization plots object allocation (x-axis: memory allocated, y-axis: allocating methods). The phases in memory allocation hotspot are copied from the method invocation hotspot. As shown in the figure, one third memory is allocated in **phase 0**, while 80 percents methods are invoked during this phase. This means only a few methods are memory intensive (methods that are invoked in phase 3).

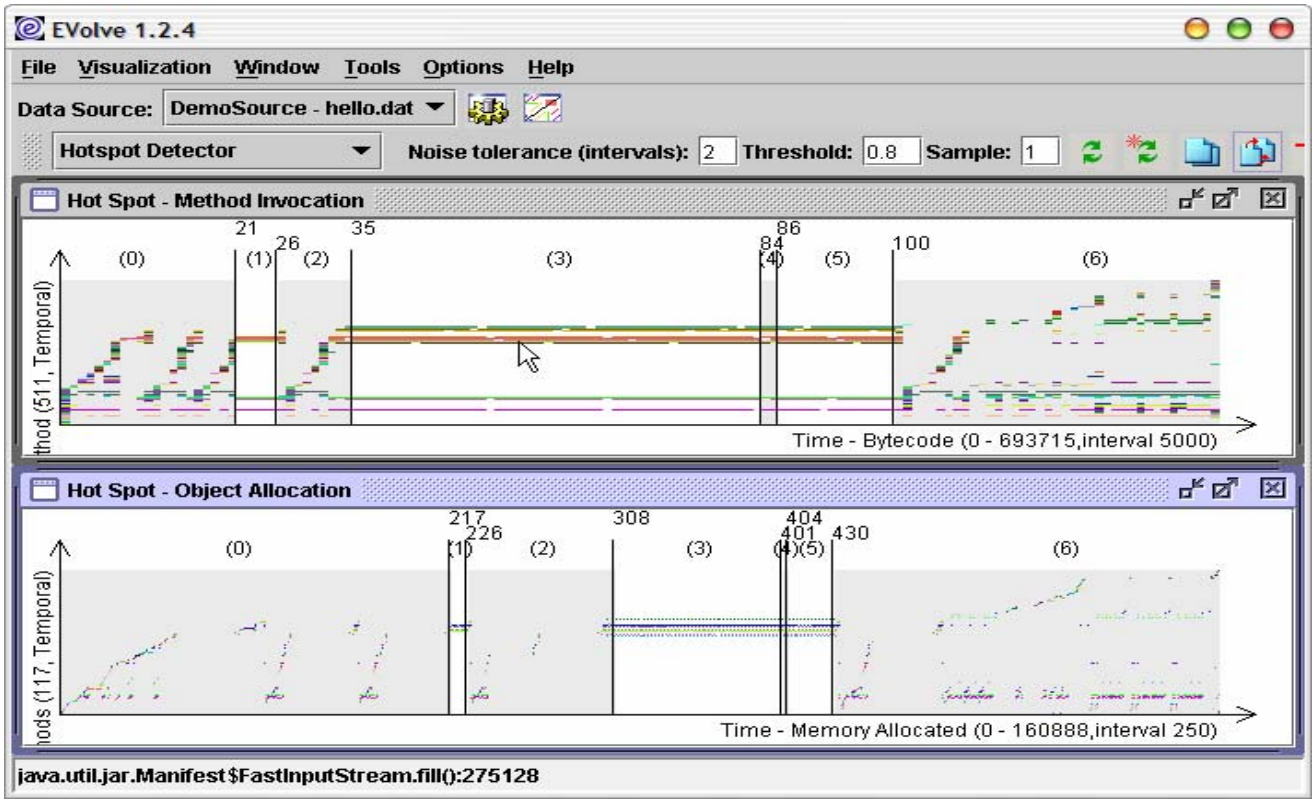

Figure 22 Phase Detection Result

4.7 Visualization Cloning

While practicing with EVolve, we found that duplicating a visualization and then changing its configuration a little is very useful. To serve this purpose, we add the Cloning feature into EVolve. Right clicking on the visualization to be cloned and choosing *Clone* will duplicate this visualization (shown in Figure 23).

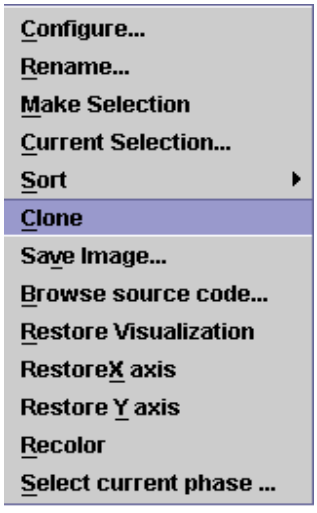

Figure 23 Cloning Visualization

4.8 Manipulating Selections

EVolve enables the user to make selections, these selections can then be used to create filters. In this project, making selections is enhanced.

First, options are offered to help the user define selection more precisely. For example, in Figure 24 two options are available: **Time Frame** and **Occurred Entities**. The first option defines the time duration of selection; if not checked, the whole trace is selected. The second option tells EVolve only select entities occurred in the selection box; if not checked all entities are selected. Note that different types of visualization could have different selection options.

After a selection is made, we can further refine or manipulate the selection. For example, using regular expressions as criteria to remove or keep some entities, changing time frame, cloning selection and so on (Figure 25).

4.9 Zooming

When a trace file grows larger, it becomes difficult to examine detail information. We create a zooming tool (fisheye view) to help the user. Holding down ALT key and moving mouse around can activate the zooming tool. Zooming window enlarges the 20 x 20 area under the mouse pointer to allow the user to see the fine-grained structure of the graph and to point to a specific dot in order to see the reference name. This feature is available to all types of visualizations. Figure 26 demonstrates a zooming example.

4.10 Visualization Restoring

By default, all visualization is resized to fit the window size. Although zooming tool helps the user a lot, it does not work well when the trace becomes very huge. The reason is that in order to fit the visualization to the window, AWT has to map multiple pixels into one pixel when the real size of visualization is large. When this is the case, restoring visualizations becomes more helpful. Restoring visualizations allows the user to restore the visualization on x-axis or y-axis or both and scroll the restored result, mapping every data point to one pixel. Figure 27 illustrates an example of restoring x and y axis. The visualization on the top left is resized to fit the window's size; the y-axis of the top right visualization is restored to display one entity per pixel; the x-axis of the bottom left visualization is restored to display one interval per pixel.

4.11 Source Code Browsing

Although EVolve offers *mouseMove* to help the user find out entity's name, it is still not enough to understand programs sometimes. Browsing source code might become a solution.

Figure 28 is an example of browsing source code. Right clicking on the entity and choosing *Browse Source Code* will activate this function. Note that in order to make this function work correctly, the user must set appropriate values to *source code path* and *additional class path.* Section 4.12 will talk about these in more detail.

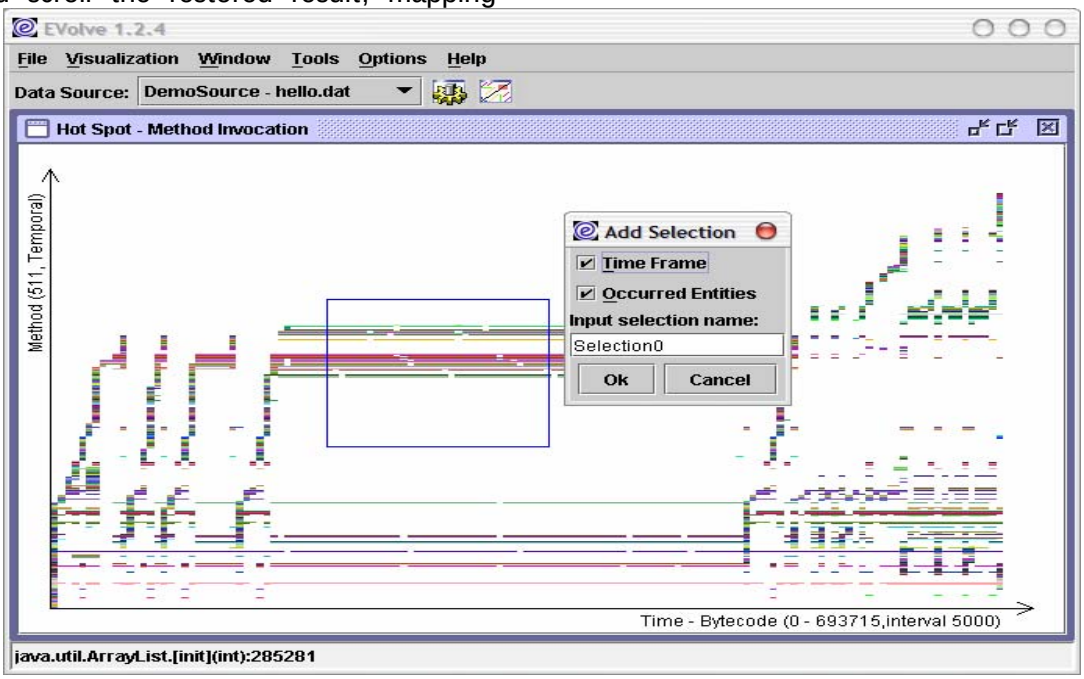

Figure 24 Options for Selection

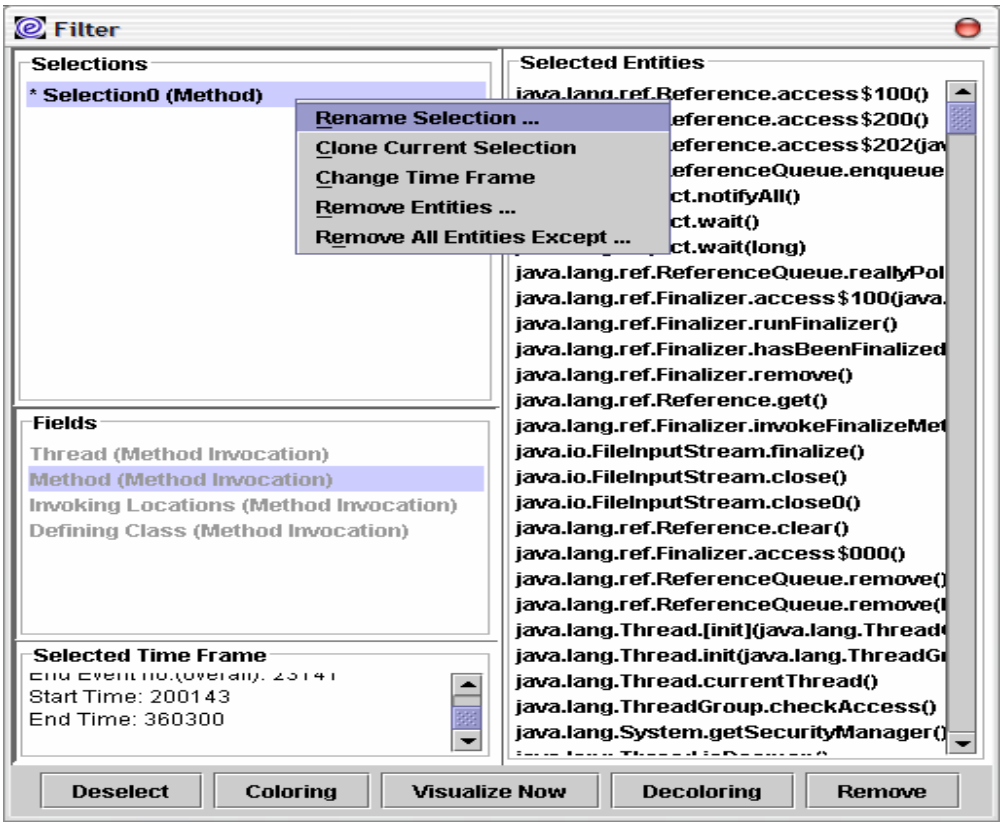

Figure 25 Manipulating Selection

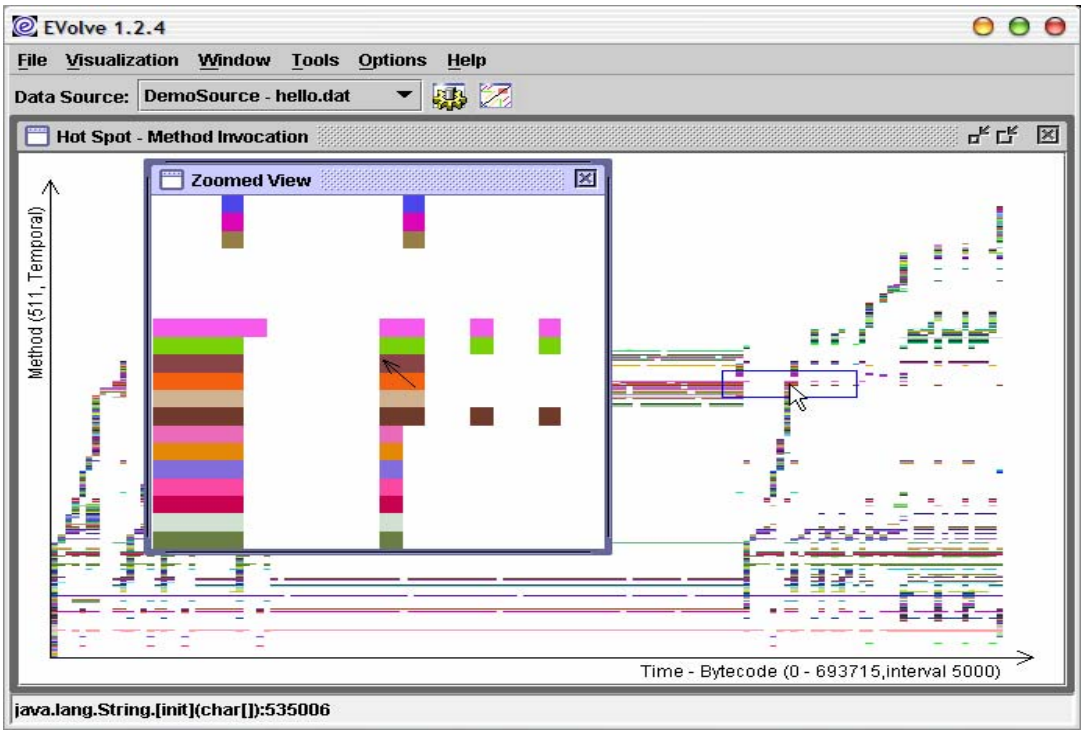

Figure 26 Zoomed View

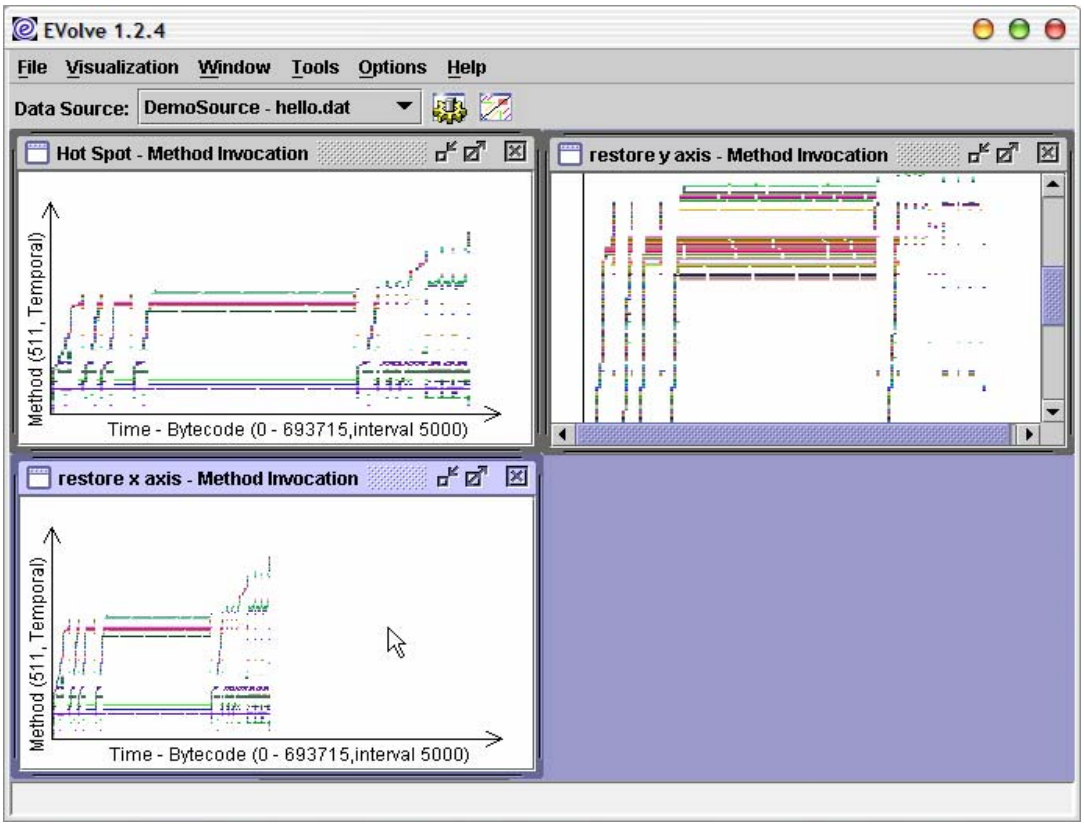

Figure 27 Restoring Axes

Figure 28 Source Code Browsing

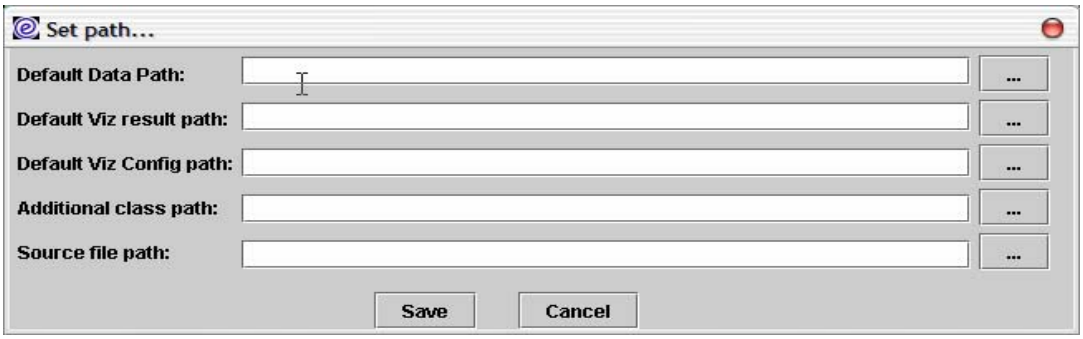

Figure 29 Saving Settings

4.12 Saving EVolve Settings

Figure 29 shows a setting saving dialog, in this dialog the user is allows to set frequently used paths in EVolve. Totally there are five paths:

- Default Data Path: By default, EVolve searches this path for trace files.
- Default Viz Result Path: EVolve uses this path to store visualization files created in batch (Section 4.5)
- Default Viz Config Path: By default, EVolve searches this path for configuration files.
- Additional Class Path: When browsing source code, EVolve searches these paths for class files and parses them in order to get source file names.
- Source File Path: After EVolve gets source file names, it will search these paths and try to open those source files.

5. Related Work

Quite a lot dynamic data visualization tools, such as Jinsight[1,6,7], JProbe[2] and OptimizeIt[3], are designed to help programmers optimize their programs by visualizing the runtime usage of system resources (CPU time, memory usage, etc). Some dynamic visualization tools, such as Jinsight and TARANTULA, are even used on program debugging. These tools, however, are not designed to be extensible. Normally they use specific trace collecting techniques and have a fixed visualization library.

An exception is BLOOM[8], it is designed to be extensible. Unlike EVolve, which provides extensibility through simplified framework, BLOOM's extensibility is achieved by providing a visualization back-end supporting a variety of visualization strategies.

Visualization tools are also used on reverse engineering and algorithm understanding. For example, Dotplot[4] is used to explore self-similarity of code, SHriMP[10] is used on understanding software architecture and hierarchy by parsing and visualizing source code.

Although in software visualization, extensibility is not often seen, quite a lot of information visualization systems are designed to be extensible since they are used to solve general-purpose problems. Visage[18], for example, supports and coordinates multiple visualizations and analysis tools in data-intensive domain.

6. Contributions

We redesigned and extended the EVolve program, originally implemented of Qin Wang[17] in the following ways:

- In order to make EVolve more extensible and more efficient in code reuse, we revised the hierarchy of visualization library. In the new hierarchy, all 2-D visualizations are divided into three categories according to how x and y axis are treated.
- We also created a set of painters and apply them on old visualizations. Applying painters enables these old visualizations to convey more information. We also generated a set of new visualizations (Thread hotspot, Stack hotspot, Dotplot, Stack, Event and Relationship) and added them to the library.
- The user interface was enhanced greatly. We generated a couple of new features to

help the user study the trace file more closely and more clearly, for example, zooming tool and axes restoring give the user a fine-grained view of the visualization, predefined visualization enables the user to reproduce experiment settings easily.

- We added a brand-new phase detector, which is used to detect phases in hotspot visualizations. In addition, we provided several little tools:
  - ✓ Entity trigger, which helps the user to locate specific entity.
  - ✓ Entity set, which helps user to find execution patterns that consists of a set of entities
  - ✓ Phase clipboard, which enables the user to copy and paste phases between visualizations. This kind of operation enables user to set up cross references between visualizations.
- When I worked as an intern at Nokia, I enabled EVolve to parse Nokia's data traces by implementing a new data source. I also created several visualizations by extending Hotspot and Value-Value visualization to help the user at Nokia understand their data trace. Unfortunately, I cannot give more details, but this demonstrates the extendibility of EVolve.

7. Conclusion

In this report, we present an extensible software visualization framework for visualizing the runtime behavior of Java programs. EVolve is divided into three parts: the data source, the core platform and the visualization. Except the core platform, both data source and visualization can be added independently and easily.

To help the user analyze a trace file, we provided a set of visualizations and integrated these into EVolve platform. A new visualization hierarchy is developed to categorize these visualizations more reasonably.

We also describe a set of new features in this report. These features include some user interface enhancements (zooming window, load/save predefined visualization, etc.) and small tools (phase detector, batch runner, etc.).

8. Future Work

Although the current version of EVolve is more powerful than its predecessor, it still has a long way to go. Here are some possible future works:

- ✓ The current phase detector is pretty naive, a more sophisticate phase detection algorithm is required to get better results.
- ✓ Now the phase detector can only be applied on hotspot visualizations, it may be useful to enable phase detector for other type of visualizations.
- ✓ EVolve has memory leak problem. For example, after generating a couple of visualizations for a large data trace, EVolve may throw OutOfMemory exception.
- ✓ Relationship visualization also needs to be revised, a possible scheme may be introducing "gravity" between two references according to the correlation between them.
- ✓ A new user of EVolve may have trouble in choosing correct visualizations to analyze a data trace. It may be necessary to create a "wizard" to guide the user.
- ✓ An on-line help system may be very helpful for a new user. For example, when a new user is using a hotspot visualization, telling him how to extract useful information (i.e. where is the hotspot and what does the hotspot area signify) will be very nice.
- ✓ All current built-in visualizations are 2-D, implementing 3-D visualizations might be useful.

9. References

[1] Jinsight. http://www.research.ibm.com/jinsight

[2] JProbe. http://www.sitraka.com/software/jprobe

[3] Optimizeit. http://www.optimizeit.com

[4] Kenneth Ward Curch and Jonathan Issac Helfman. Dotplot: a program for exploring self-similarity in millions of lines of text and code. In *Porceedings of Journal of Computational and gGraphical Statistics*, pages 2:153-174, 1993

[5] Michele Lanza and Stephane Ducasse. A categorization of classes based on the visualization of their internal structure: the class blueprint. In *Proceedings of Conference on Object-Oriented Programming Systems, Languages and Applications(OOPSLA'01)*, PAGES 300-311,2001

[6] Wim De Pauw, Erik Jensen, Nick Mitchell, Gary Sevitsky, John Vlissides, and Jeaha Yang. Visualizing the execution of Java programs. International Seminar, Dagstuhl Castle, Germany, May 20-25, 2001. In *Lecture Notes in Computer Science Vol. 2269,* pages 151-162. Springer Verlag, 2002

[7] Wim De Pauw, Richard Helm, Doug Kimelman and John Vlissides. Visualizing the behavior of object-oriented systems. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'93)*, pages 326-337, 1993

[8] Steven P.Reiss. An overview of BLOOM. In *Proceedings of the 2001 ACM SIGPLAN – SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, pages 2-5, 2001

[9] Steven F. Roth, Peter Lucas, Jeffrey A. Senn, Cristina C. Gomberg, Michael B. Burks, Philip J. Stroffolino, John A. Kolojejchick, and Carolyn Dunmire. Visage: Auser interface environment for exploring information. In *Proceeding of Information Visualization, IEEE*, pages 3-12, 1996

[10] Margaret-Anne D. Storey and Hausi A. Muller. Manipulating and documenting software Structures using SHriMP views. In *Proceedings of International Conference on Software Maintenance,* pages 275-285, 1995

[11] Margaret-Anne D. Storey, Kenny Wong, an Hausi A. Muller. Rigi: A visualization environment for reverse engineering. In *Proceedings of the International Conference on Software Engineering (ICSE'97),* PAGES 606-607, 1997

[12] Margaret–Anne D. Storey, K.Wong, F.D. Fracchia, H.A. MÜller. On Integrating Visualization Techniques for Effective Software Exploration

[13] Wim De Pauw, Nick Mitchell, Martin Robillard, Gary Sevitsky, Harini Srinivasan. Drive-by Analysis of Running Programs

[14] Win De Pauw, David Lorenz, John Vlissides, Mark Wegman**.** Execution Patterns in Object-Oriented Visualization

[15] Steven P. Reiss. Bee/Hive: A Software Visualization Back End

[16] Bruno Dufour, Karel Driesen, Laurie Hendren, and Clark Verbrugge. Dynamic metrics for compiler developers. Sable Technical Report SABLE-TR-2002-11, McGill University, School of Computer Science

[17] Qin Wang, Rhodes Brown, Karel Driesen, Laurie Hendren, and Clark Verbrugge. EVolve: An Extensible Software Visualization Framework. Sable Technical Report SABLE-TR-2002-06, McGill University, School of Computer Science

[18] Steven F. Roth, Peter Lucas, Jeffrey A. Senn, Cristina C. Gomberg, Michael B. Burks, Philip J. Stroffolino, John A. Kolojejchick, and Carolyn Dunmire. Visage: A user interface environment for exploring information. In *Proceedings of Information Visualization, IEEE,* pages 3-12, 1996.