McGill University
School of Computer Science
Sable Research Group

EVOLVE: An Open Extensible Software Visualization Framework

Sable Technical Report No. 2002-12

Qin Wang, Wei Wang, Rhodes Brown, Karel Driesen, Bruno Dufour,
Laurie Hendren and Clark Verbrugge

December 19, 2002

Contents

1 Introduction **3**
1.1 Design and Features 3
1.2 Visualizations 4
1.3 Paper Organization 4

2 Architecture **4**
2.1 Data Representation 5
2.2 The Data Protocol 5
2.3 The Visualization Protocol 6
2.4 The EVolve Core Platform 6

3 Visualization Library **8**
3.1 Hierarchy 8
3.2 Bar Chart 9
3.3 Hotspot 9
3.3.1 Thread Hotspot 10
3.3.2 Stack Hotspot 11
3.3.3 Prediction Hotspot 11
3.4 Correlation 12
3.5 Stack 13
3.6 Dotplot 13
3.7 Metric 13

4 Features **13**
4.1 Comparing 13
4.1.1 Colouring 13
4.1.2 Overlap 14
4.1.3 Aligning 14
4.1.4 Orientation 14
4.2 User Interface 14

5 Related Work **16**

6 Conclusion and Future Work **16**

List of Figures

1 Architecture of EVolve. 4

2 Barchart visualization process: selecting a registered visualization (top), configuring the Barchart with fields from the data source (middle), and visualization (bottom). 7

3 Visualization Hierarchy. 8

4 Four aligned visualizations: a barchart and three hotspots. 10

5 Polymorphism viewed in prediction hotspot and correlation visualizations. 11

6 Method invocations as dotplot and stack visualization. 12

7 Overlapping: `life` hotspot (top), `qsort` hotspot (middle), and overlapped hotspots (bottom). 15

Abstract

Existing visualization tools typically do not allow easy extension by new visualization techniques, and are often coupled with inflexible data input mechanisms. This paper presents EVolve, a flexible and extensible framework for visualizing program characteristics and behaviour. The framework is flexible in the sense that it can visualize many kinds of data, and it is extensible in the sense that it is quite straightforward to add new kinds of visualizations.

The overall architecture of the framework consists of the core EVolve platform that communicates with data sources via a well defined data protocol and which communicates with visualization methods via a visualization protocol.

Given a data source, an end-user can use EVolve as a stand-alone tool by interactively creating, configuring and modifying visualizations. A variety of visualizations are provided in the current EVolve library, with features that facilitate the comparison of multiple views on the same execution data. We demonstrate EVolve in the context of visualizing execution behaviour of Java programs.

1 Introduction

This paper presents a software visualization framework, EVolve, which has been designed to be both open and extensible. EVolve is *extensible* in the sense that it is very easy to integrate new data sources and new kinds of visualizations. EVolve is *open* in the sense that EVolve framework is publicly-available and the interfaces to new data sources and new visualizations are clearly defined via Java APIs.

The development of EVolve started from our need to visualize the run-time behavior of Java programs in ways that help us develop new compiler optimizations and new run-time systems. In our case, we had trace data from many diverse data sources, including several JVMPI agents, instrumented Java virtual machines and instrumented bytecode. Thus, we needed a system that could be easily adapted to new sources of data. Note that our data is usually collected offline, so we are most interested in offline, and not real-time, visualizations. On the visualization side we wanted to be able to develop custom visualizations that allowed us to view specific program behaviours, such as the predictability of polymorphic virtual method calls. Thus, we needed a system where new visualizations could be easily added, as we discovered new program behaviours to study.

Our final EVolve system can be used in two ways: (1) as a stand-alone tool using our pre-defined data sources and visualizations and (2) as a toolkit for developing new custom visualizers (by adding new data sources and/or new visualizations).

This paper has two major areas of contributions. The first area is the design and features provided by the EVolve platform, and the second is the development of a collection of software visualizations that are suited to the study of the run-time behaviour of Java programs. We expand upon these two major areas in the following subsections.

1.1 Design and Features

In designing the EVolve platform we considered both extensibility and usability. Extensibility was achieved via a clean definition of the data and visualization protocols; in particular, the data protocol provides a well-defined method for defining data records (*elements*) and classifying these as either *entities* (static information) and *events* (dynamically-occurring events). Furthermore, fields in elements are defined with specific *properties* which allows the EVolve system to convey information to the visualizations and to automatically create appropriate menus for visualization creation and configuration.

The EVolve platform provides many useful features for selecting and instantiating visualizations, manipulating those visualizations and comparing visualizations. For example, we found that it was very important to be able to align different visualizations along the same axes, in order to facilitate understanding the interaction of run-time behaviours. Going one step further in this direction, we also provide a method for overlapping visualizations. Other features include a zooming tool which allows one to zoom in on visualizations, a selection tool which allows one to select subsets of the data and color them appropriately, and the ability to sort along any dimension of the visualization that is sortable.

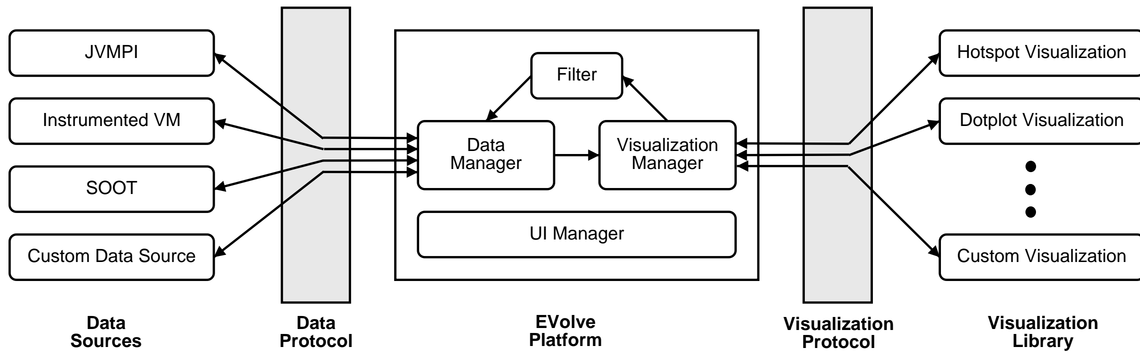

Figure 1: Architecture of EVolve.

1.2 Visualizations

In order to make a useful tool for our research, we have defined a collection of visualizations which include both standard visualizations, as well as new visualizations that are specific to our particular interest in understanding the run-time behaviour of Java programs.

Each visualization implements the EVolve visualization protocol by providing an implementation for the visualization API. EVolve currently supports eight different types of visualizations, implemented using a visualization hierarchy which uses subclasses to group common behaviour together. For example, we have defined the notion of a *hotspot* graph which shows when and for how long various parts of the program become active, and then we define three subclasses that specialize the behaviour to display information by thread (*thread hotspot*), to give an abstract view of the stack (*stack hotspot*), and to view predictive behaviour of the data (*predictive hotspot*).

This initial set can be easily expanded. A user wishing to add a new kind of visualization similar to one already existing can implement a subclass, and only define the new behaviour. Completely new visualizations can be added by defining a new class higher up in the hierarchy. In this case more functionality must be implemented, but we have found that the visualization API is quite clear and the implementations required are relatively small.

1.3 Paper Organization

The paper is organized as follows. In Section 2 we discuss the overall architecture of the system and describe the data and visualizations protocols in more depth. In Section 3 we describe our existing visualizations along with examples taken from real benchmark programs. In Section 4 we give an overview of the most important features provided by EVolve. Finally, we give related work and conclusions in Sections 5 and 6.

2 Architecture

From an external point of view, the EVolve platform can be divided into three major components (see Figure 1). The first (leftmost) of these components is the *data source*, which is responsible for systematically translating input data into EVolve's own abstract representation so that it can be manipulated and visualized. At the opposite end of EVolve is the visualization library, which is responsible for presenting the data to the user in a graphical form. This latter component contains the standard EVolve visualization library and any custom visualizations provided by the end-users. The third (middle) component is the core of the EVolve platform, and is the only component which is fixed in the architecture. The core of the EVolve platform takes care of all communication between the components at either end, and encapsulates the complex machinery that is required in order to manipulate the data. This allows the data sources and the visualizations to focus on their specific tasks.

Although EVolve is distributed with a default data source and a default set of visualizations, end-users can — and are encouraged to — develop custom ones in order to better suit their particular needs. Implementing new data sources and/or new visualizations can be done by extending the provided ones or by creating entirely new ones.

2.1 Data Representation

In order to make the EVolve platform extensible, it is important to limit the imposition of constraints on the input format of the source data. An abstract and flexible internal representation of the data is required that is independent from the source from which it was obtained or the format in which it is stored.

We distinguish between two major classes of data elements. *Entities* are named, unordered data elements that remain constant during the visualization process. Examples of entities include data types and class information. *Events* are anonymous, ordered data elements that are dynamic in nature. They represent the behaviour of a program. Examples of events include object allocations and method invocations. Execution traces usually contain few entities and numerous events, therefore EVolve caches entities in memory. In a typical run 99% of the elements in an execution trace are events.

Elements are composed of fields which can be of two types: *entity reference* (which will be simply referred to as *reference*) or *value*. A reference field refers to an entity whereas a value field holds scalar data.

We additionally define the concept of *properties* for value fields. Properties are used to communicate generic data characteristics so visualizations can operate on appropriate data. Three properties are built-in with the following meanings:

amount: The `amount` property indicates that the data value is numeric and summable. For example, the "allocation size" field of an object allocation event is an amount.

coordinate: The `coordinate` property indicates that the data is non-numeric or not summable. For example, object addresses are coordinates because they are numbers that cannot be meaningfully summed.

time: The `time` property is used in conjunction with either `amount` or `coordinate` and indicates that a data value is monotonically increasing, and thus can be interpreted as a definition of time.

EVolve allows the definition of arbitrary custom properties in order to provide support for new visualization requirements.

2.2 The Data Protocol

The data source component of the EVolve platform is responsible for converting the input data into a format that can be manipulated within the framework. The data source communicates with the core of EVolve through the data protocol. A new data source supports this data protocol by implementing 7 simple methods, each taking care of one specific part of the process. These methods are called by EVolve, so the data source designer does not have to be concerned with their relative ordering.

The `init()` method is responsible for initializing the data source. This typically involves opening the input file for reading and instantiating global objects and data structures. The remaining 6 methods are grouped in pairs: there are three kinds of objects that a data source has to send to EVolve. Each pair provides the start and the delivery of the next object for one of these object types.

EVolve uses *element definitions* to represent the kind of data elements (entities and events) which are present in the execution trace. Element definitions encode information about the relationships that exist between the various fields of the available elements, as well as their respective properties. The data source sends this information to EVolve by implementing a method pair: `startBuildDefinition()` and `getNextDefinition()`. EVolve calls `startBuildDefinition()` to let the data source know that it is ready to accept definitions, and calls `getNextDefinition()` repeatedly to obtain definitions until a `null` value is returned. A second pair of methods, `startBuildEntity()` and `getNextEntity()`, deliver entities to the platform. Finally, a third pair, `startBuildEvent()` and `getNextEvent()`, deliver events. Events constitute the visualizable data in the execution trace.

To assist in the process of creating these three kinds of objects, EVolve provides two convenience classes, `EntityBuilder` and `EventBuilder`. They are used to generate a definition of an element or an instance of it. The process is similar for both tasks. In essence, the builder class is instructed to start building a new defection (instance). Fields are repeatedly added to the item being built. Once every field is provided, the builder class is instructed to return the newly built

item. In the case of element instances, builder classes verify that all previously-defined fields have been properly set and throw an exception if any are missing, thus avoiding potential problems associated with malformed input.

2.3 The Visualization Protocol

In order to make visualizations as flexible as possible, they must be able to display data from a variety of different data sources. Therefore any specific requirement of visualization must be expressed in a systematic, data-independent way. Similarly as for data sources, EVolve requires visualizations to provide an abstract representation of their visualization capabilities.

Visualization capabilities are defined in terms of *dimensions*. Every visualization defines its dimensions, which can either be values or references. For example, the horizontal bar chart visualization in Figure 2 declares both a reference dimension and a value dimension, where the value dimension is used for the length of the bars.

Every dimension is associated with a property. The property constrains which fields from a data source can be mapped to a dimension. Dimensions are also associated with *data filters*. Data filters are provided by EVolve and are responsible for extracting the field of interest for the selected type of event.

A new visualization in EVolve extends `Visualization`, which defines 9 abstract methods that have to be implemented. These methods collectively form the visualization protocol. The first task supported by the protocal is the *creation* of the visualization. Two of the 9 methods belong to the creation phase. `createDimension()` provides EVolve with the dimensions of the visualization. `createPanel()` creates the canvas on which the visualization will be drawn. Two methods make the visualization configurable via the configuration dialog. `createConfigurationPanel()` creates a visualization-specific configuration panel to be inserted in the generic configuration dialog. A notification that a user has configured a visualization is sent to the visualization by EVolve calling its `updateConfiguration()` method. Three additional methods support the visualization process itself. The start of a new visualization phase is signalled by `preVisualize()`, allowing EVolve to perform an initialization sequence. Elements are then passed one by one to the visualization using `receiveElement()`. Once all elements have been sent, EVolve calls `visualize()`, asking a visualization to display the resulting representation of the data. After the visual representation is produced, the user can manipulate visualization entities by making selections and sorting dimensions, two visualization-specific tasks provided by the `makeSelection()` and `sort()` methods.

2.4 The EVolve Core Platform

In order to allow other components to focus on their specific function, the core of the EVolve platform is responsible for many tasks, including managing the user interface and handling communication between the data source and different visualizations.

Figure 2 illustrates some of the services the EVolve provides. The topmost picture in the figure shows the list of visualizations that are available. The `UIManager`, which is a part of the EVolve core (see Figure 1), automatically generates this menu from the list of visualizations that are registered with EVolve. The user selected a simple `Bar Chart`.

In order to promote extensibility, direct interactions between data source and visualizations are prohibited. EVolve's data manager is responsible for sending only appropriate data elements to a visualization, while the visualization manager manages visualization-specific tasks, such as creating new instances of visualizations. Data sources provide an abstract description of their input format, and visualizations provide a description of requirements and capabilities. The EVolve core is responsible for the communication between them.

The configuration dialog in the middle screen shot of Figure 2 illustrates the link between data source and visualization: the EVolve core generates the items in the drop-down lists automatically. The *subject* of the visualization is simply the type of event that is visualized, in this case a `Method Invocation`. This information is extracted from the abstract element definitions generated by the data source. Once a subject has been selected, EVolve extracts from the element definition the fields which have properties appropriate to each dimension of the visualization. A field can be mapped to a dimension if it is the proper kind (value or reference) and possesses the property associated with the definition. In this example, the y-axis of a (horizontal) bar chart must be a reference, and therefore EVolve shows the list of references available in a `Method Invocation` event. The user selected `Invoking Locations`.

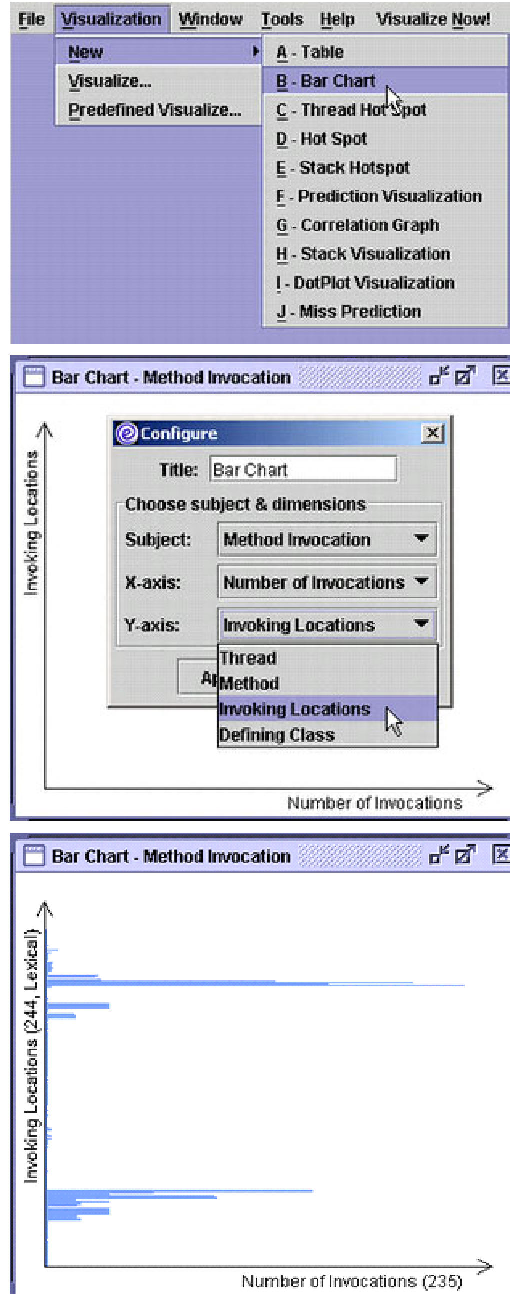

Figure 2: Barchart visualization process: selecting a registered visualization (top), configuring the Barchart with fields from the data source (middle), and visualization (bottom).

The bottom screen shot in Figure 2 shows the resulting bar chart visualization. At this point the user can manipulate the bar chart, for example by changing the order of references on the y-axis. Reference dimensions in EVolve can be ordered according to two schemes: *temporal* and *lexical*. References refer to named entities, and a a lexicographic sort of the names provides lexical ordering (the default as shown). Alternatively, a temporal ordering sorts entities in the order they appear an execution trace. A user can customize EVolve by plugging in different sorting schemes for data fields with particular properties.

Users can also make *selections* on the graph, defining particular subsets of the data; operations can then be applied to a selection as a unit. Colouring, for instance, can be managed in this way. Once a colouring scheme has been

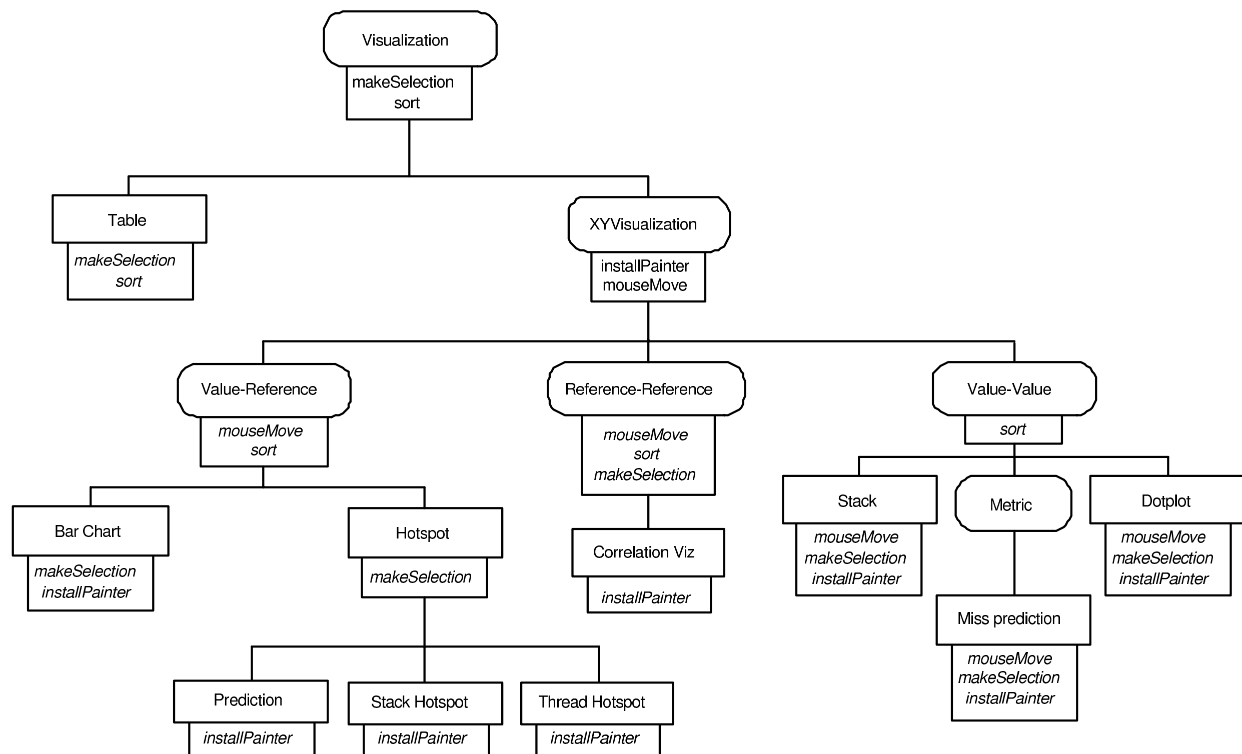

Figure 3: Visualization Hierarchy.

specified for a given selection within a visualization, it is shared among the visualizations that do not provide their own colouring scheme (see Section 3 for examples). This allows one to identify and keep track of a subset of interest.

3 Visualization Library

An end-user can use EVolve as a stand-alone tool and interactively create and modify multiple visualizations from an existing library (the right part of Figure 1). The ability to view a particular data source in a variety of ways greatly adds to the benefit of visualization, so we provide examples in this section in which multiple visualizations are combined.

EVolve was designed to be extensible. The core platform (the central part of Figure 1) provides the glue between visualizations and data sources, ensuring that each new visualization can be applied to any data source which has the appropriate properties. The visualization interface already ensures seamless integration of a new visualization into the existing library. However, if the interface was the only part that visualizations had in common, then every new visualization would have to be defined from scratch. Instead we provide the existing library as an implementation hierarchy. Extending the library with new visualizations is now just a matter of finding the right super class to inherit from, and overriding the appropriate methods.

3.1 Hierarchy

Figure 3 shows the visualization 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 subtree have in common. The topmost abstract class is `Visualization`, which declares all methods necessary to communicate with the visualization platform, as discussed in Section 2. `Visualization` declares two methods which all visualizations share: `sort` allows data to be sorted by the user and `makeSelection` allows the user to select a particular subset of data to be coloured or visualized in a new win-

dow. `Visualization has` two subclasses: `Table`, a concrete class which shows a set of references/values in table format, and `XYVisualization`, an abstract class which serves as root for all visualizations which show data in two-dimensional form (*x* and *y*-axis).

`XYVisualization` declares two methods: `installPainter` determines the colouring of a visualization (determined by the user or by the semantics of the visualization itself), and `mouseMove` determines the text that appears when the user moves the mouse over a particular part of the visualization (e.g. showing the name of an invoked method). `XYVisualizations` come in three kinds, distinguished by the way that coordinates on *x* and *y*-axis are treated: each axis can be either a reference or a value. References refer to entities in data sources (see Section 2). They are named and sorted in predefined ways, and can be coloured by the user to facilitate comparisons between different views on the same set of references. A value axis shows computed values. Nothing is known a priori about values: they can be extracted directly from the data source (e.g. object size) or be computed (e.g. a metric such as average inline cache miss rate).

`Value-Reference` visualizations contain a reference on one axis, and therefore share `mouseMove` and `sort`: `mouseMove` shows the name of the reference corresponding to the mouse position on the reference axis [1]. `Reference-Reference` visualizations contain a reference on both *x* and *y*-axis, and therefore have predefined `mouseMove`, `sort` and `makeSelection`. `Value-Value` visualizations, which are more open-ended, share `sort` but each overrides `mouseMove`, `makeSelection` and `installPainter`.

In the next sections, we show examples of concrete classes and discuss how they fit into the hierarchy.

3.2 Bar Chart

Figure 4 shows four aligned visualizations of method invocations from a fragment of the Volano benchmark [4] run. We start with a simple visualization: the top left window shows a horizontally oriented bar chart. A `Bar Chart` is a `Value-Reference` visualization. This is the same bar chart as shown at the bottom of Figure 2, after the user has selected colours for particular subsets of the data: the reference axis (*y*-axis) shows 244 method invocation locations in lexical order. Volano invocations are drawn in yellow at the bottom, Java libraries are shown in red (frequent invocations) and blue (infrequent invocations). This colouring is provided by the default painter, which allows the user to manually select and colour subsets of references. The value axis (*x*-axis) shows the total number of invocations occurring at each location. Users can find the name of the invoking location by placing their mouse over a bar in the chart. This functionality is provided by `mouseMove` for all `Value-Reference` visualizations.

3.3 Hotspot

The top right window of Figure 4 shows a lexical hotspot. A `Hotspot` is a `Value-Reference` visualization. The picture is similar to the horizontal bar chart, but now method invocations are shown as they occur in time, instead of summed together per location. The reference axis (*y*-axis) is identical to the bar chart reference axis, showing the same 244 method invocation locations sorted by lexical order and coloured by the user. The only difference is the value axis (*x*-axis), which shows the passing of time as number of method invocations (2439 invocations in this program phase, grouped in samples of 20 invocations each [2]).

Both windows are aligned according to the reference axis using EVolve's window alignment feature. The hotspot visualization shows when and for how long particular parts of a program become active. For example, the visualized program phase starts with class loading, executing exclusively Java library code in red, and then switches to `com.volano` invocations, which also call some library code in blue and red.

The bottom right window of Figure 4 shows the same hotspot graph, but with a temporal ordering of the reference axis. This is also a `Hotspot` visualization, sharing the same colouring and *x*-axis. The only difference is the sorting scheme of the reference axis (*y*-axis), which is now sorted in temporal order, by the time stamp of the first invocation at each location. A temporal hotspot clusters together invocations that start together, and therefore emphasizes program phases. The yellow `com.volano` invocations therefore appear clustered together with the Java library code that they

[1] all `Value-Reference` visualizations can be oriented horizontally or vertically

[2] If an invoking location occurs once in the time sample then the appropriate location is coloured. Therefore the surface area of a hotspot is an approximation of the number of invocations. The bar chart shows this number precisely.

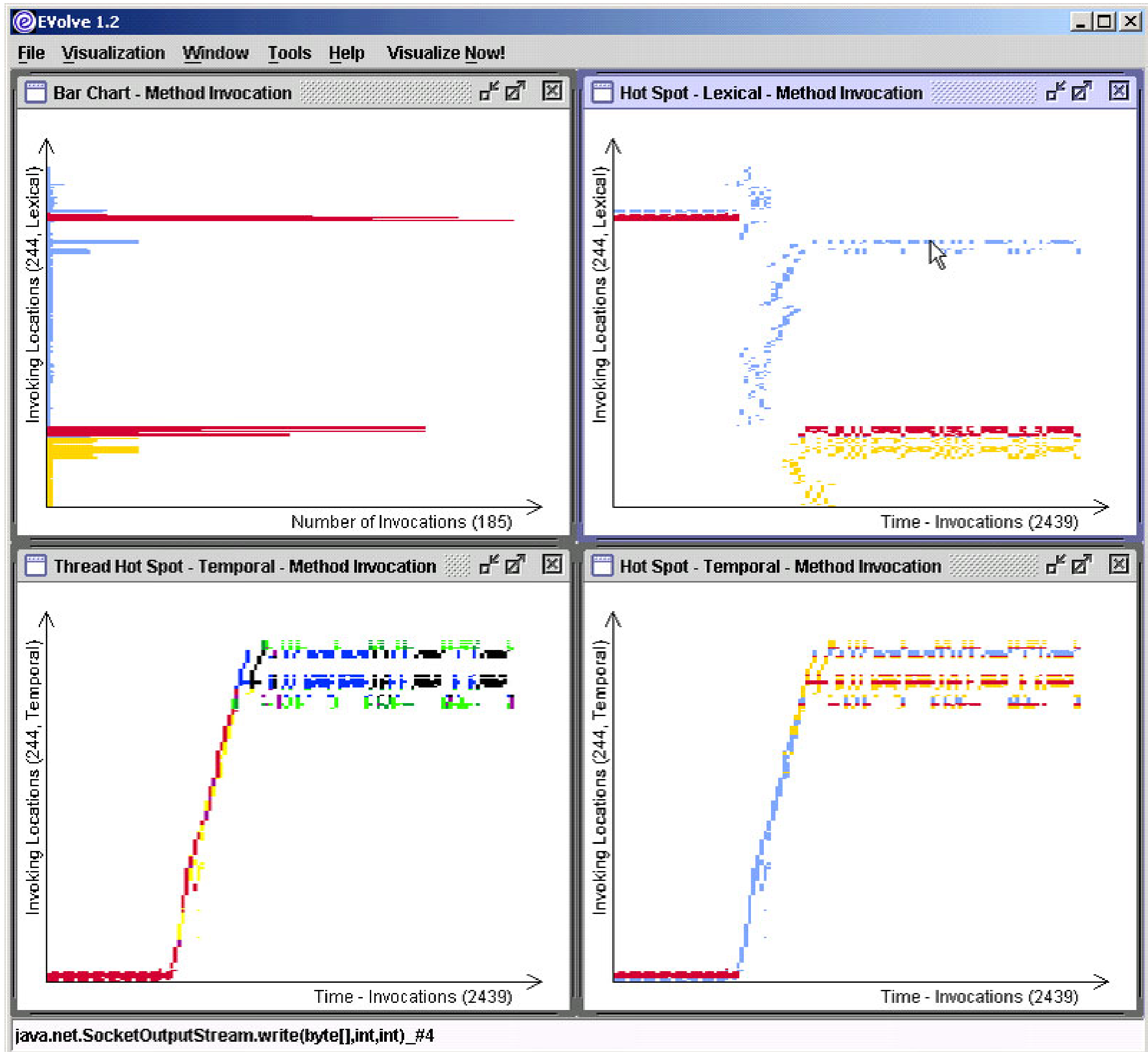

Figure 4: Four aligned visualizations: a barchart and three hotspots.

call.

3.3.1 Thread Hotspot

The bottom left window of Figure 4 shows a thread hotspot graph. `Thread Hotspot` is a subclass of `Hotspot`, and shares almost all of its code with `Hotspot`. This example shares identical axis orderings with the temporal hotspot to its right. However, `Thread Hotspots` define their own colouring scheme by overriding (`installPainter`). Each different Java execution thread is assigned a colour by the user. Volano begins in single thread execution and quickly moves into a more colourful multiple thread execution when it reaches the actual Volano code.

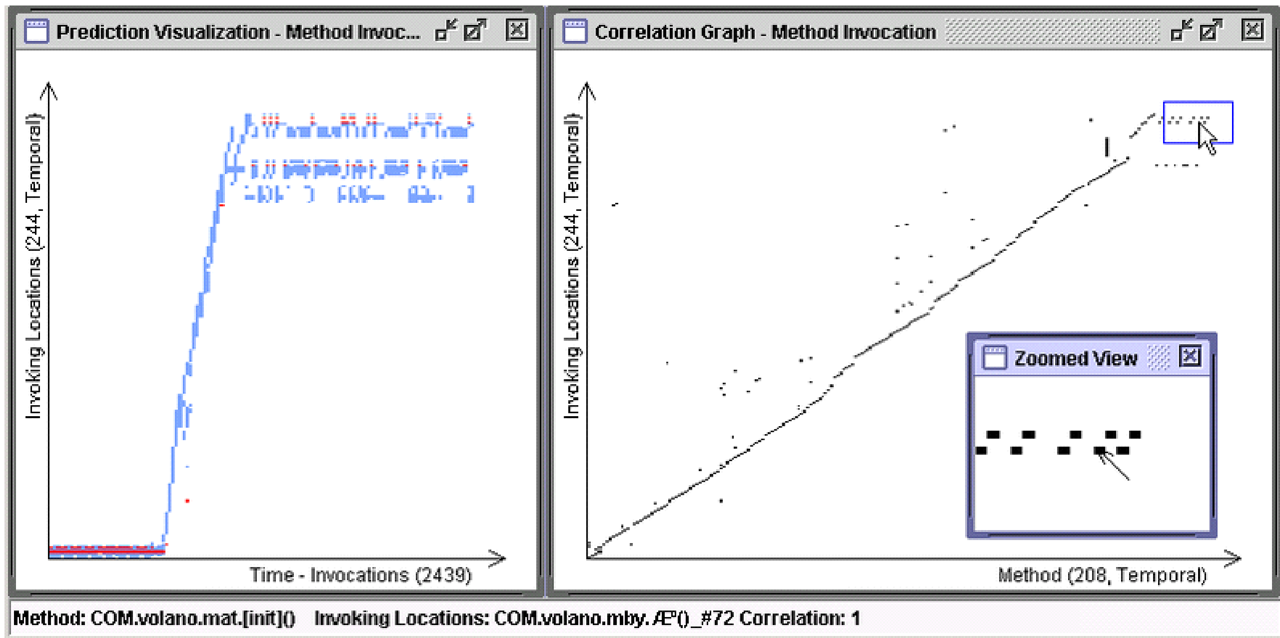

Figure 5: Polymorphism viewed in prediction hotspot and correlation visualizations.

3.3.2 Stack Hotspot

Adding new hotspots with different colouring schemes takes minimal effort: only `installPainter` needs to be defined. The `Stack Hotspot` visualization uses three different colours to show within a given time sample all methods that are *called*, on the stack but *inactive*, on the stack and *active*, by overriding `installPainter`. We do not show an example because of space limitations.

3.3.3 Prediction Hotspot

The final `Hotspot` class, `Prediction Hotspot`, is illustrated in the left window of the next figure: Figure 5. This hotspot shares identical value reference axis definitions as the previous thread and temporal hotspot. Only the colouring scheme is customized. Method invocations appear in blue when the invoked target method does not change within a time sample—i.e., the invocation is not polymorphic (in these cases some sort of inline cache for virtual method calls would be expected to work well). This visualization shows that `com.volano` exhibits some polymorphism in both phases.

This visualization is called `Prediction Hotspot` because it displays the predictability of events. The name "prediction" stems from the technique used to generate colours: a simple last-value predictor guesses that an invocation location will invoke the exact same method as the last time it was executed. The blue areas indicate perfect prediction accuracy, the red areas show when the predictor guesses the wrong target method at least once in the time sample. Different predictors can be visualized by plugging them into the framework. Sub classes of `Predictor` can implement more sophisticated and accurate predictors to generate different prediction hotspots.

Note that this visualization is not restricted to method target prediction. It displays a measurement of polymorphism only because the user selected method invocations events and invoked methods. In general, the prediction of any field in an event by any other field can be visualized. For example one can show whether array allocation size is stable (blue) or unpredictable (red) by selecting the size field in an array object allocation event.

The implementation effort required to build the prediction visualization was very small. About 120 lines of code needed to be added to plug in the visualization, 40 of which implemented a new colouring scheme including the simple last-value predictor. The actual programming took only a few hours.

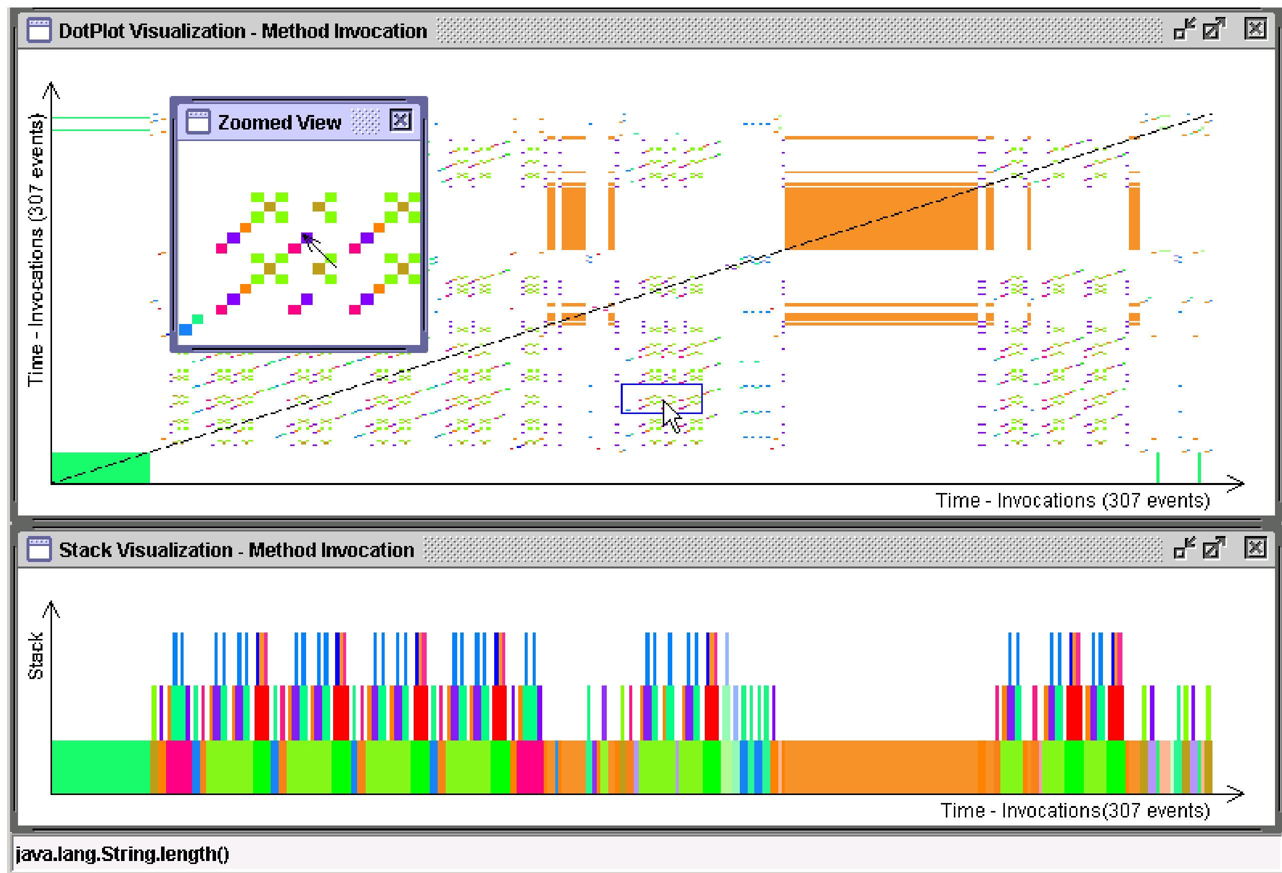

Figure 6: Method invocations as dotplot and stack visualization.

3.4 Correlation

The `Correlation` class, a subclass of `Reference-Reference`, is illustrated in the right part of Figure 5. Its y-axis is identical to the y-axis of the aligned prediction hotspot to its left, showing method invocation locations. Its x-axis refers to invoked methods. A correlation visualization shows a dot on coordinate (x, y) when a reference x occurs in the same event as reference y. For invoking location / method, the graph displays the method targets of all invocation locations.

A horizontal row of dots therefore indicates an invoking location with more than one target method (a polymorphic location). A vertical row of dots indicates a method that is called from multiple locations. Most of the dots are close to the diagonal, indicating that most methods are monomorphic and only called from one location. Polymorphic locations (red) from the prediction visualization thus show as horizontal rows of dots.

Figure 5 also demonstrates the use of a `Zoomed View`, which is a feature available in all visualizations. The 20 x 20 area under the mouse pointer is enlarged 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, a functionality implemented by the `moveMouse` method which all visualizations define.

Note that this example visualizes polymorphism because the user selected invoking locations and method targets from the data source. A correlation visualization is more versatile, allowing for instance to correlate an object's type with its invoking location's method.

3.5 Stack

The `Stack` class is a sub class of `Value-Value`, defining its own `mouseMove`, `makeSelection` and `installPainter`. Figure 6 shows a stack visualization in the bottom window for a fragment of execution in the SPEC JVM98 `javac` benchmark. The *y*-axis measures time as method invocations. The *x*-axis shows the runtime stack. `Stack` uses a colouring scheme that assigns a random colour to each invoked method. This visualization is similar to stack visualization in Jinsight [9]. Random colouring at method granularity shows various execution phases.

3.6 Dotplot

The `Dotplot` class, shown in the top window of Figure 6, is also a sub class of `Value-Value`. A dotplot is a general visualization technique used to highlight repetition in any sequence of values [5]. Often, x and y-axis are identical. The dotplot graph has a dot at position (x,y) 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). Figure 6, shows a dotplot for the same method invocations as the aligned stack visualization at the bottom. They both use identical colouring schemes (the top of the stack has the same colour as a dotplot dot at position x).

Dotplots show repetitive calls as a solidly coloured block. In Figure 6, the orange block represents one method called 50 times. The block has some smaller echoes to its left and right, below and above (the dotplot is symmetric with respect to the diagonal). Striped blocks in the graph also represent repetitive behaviour, but of a sequence of methods instead of a single method (see zoomed view). Dot plots seem particularly good for highlighting similarities between program phases. We plan to extend dotplots with the ability to select non-identical x and y-axis. Weaker definitions of equality may also be useful, for example to define equality as "defined in the same class", showing the repetition of classes instead of the methods themselves.

3.7 Metric

The final `Value-Value` subclass is the abstract `Metric` class, which visualizes the value of a dynamic metric as it changes over time. Dynamic metrics are values that reflect particular aspects of program behaviour useful to compiler developers , such as inline cache misprediction rate, byte codes touched, average object lifetimes and many more [6], which are computed for an entire program run. A metric visualization shows the metric value on the *y*-axis per time sample, visualizing its evolution as execution proceeds. `Metric` can be customized by creating a subclass defining the appropriate metric (see `Miss prediction` in Figure 3).

4 Features

In order to make our visualizations as useful as possible, certain important features are shared among all visualizations. These include aspects of an enhanced user-interface and features that allow one to combine multiple visualizations in order to improve understanding of the inspected program.

4.1 Comparing

Comparisons between visualizations can be quite informative; information gathered and presented for one purpose can be correlated with another visualization, and thereby give a more complete picture of the program activity. We provide 4 distinct methods for facilitating such comparisons: colouring, overlapping, aligning and orientation.

4.1.1 Colouring

In many visualizations, colour is used somewhat arbitrarily to provide contrast between adjacent elements (e.g. bars in a bar graph). Other visualizations may use colour as a third dimension (implying a colour ordering). While these are useful applications of colour, a third possibility is to use colour to relate visualizations together.

EVolve allows reference colours in one visualization to be shared with another visualization. For example, the colours used to identify the most frequent methods executed within a bar chart can be maintained when viewing method execution using a hotspot graph (see Figure 4). This allows information presented in two views to be easily correlated.

4.1.2 Overlap

Correlation between two visualizations are most obvious when the visualizations are overlayed. 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. Figure 7 illustrates this feature on the startup phase of `life` and `qsort`, two small benchmarks. These two programs are very different but clearly have almost identical startup phases.

In order for overlapping to be meaningful, *x* and *y*-axis of both visualizations must be unified. EVolve unifies two reference axes by building a new reference axis that contains the union of the two overlapping visualizations. The overlap visualization (bottom window of Figure 7), contains all methods invoked by `life` and `qsort`. The *x*-axis measures time as bytecodes executed since program start, is and is unified by scaling down the shorter run (`life`). The colouring scheme of an overlap is determined by each participating visualization: `life` in blue, `qsort` (in red), 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.

4.1.3 Aligning

For "busier" or more detailed visualizations, overlaying can obscure visualization data. Moreover, it is most meaningful when all axes of the visualizations are shared, or unifiable. A related approach is then to simply align or tile the visualizations vertically and/or horizontally; this suggests sharing of just one axis, but permits features to be correlated by simple straight lines (or eye movements) between visualizations. EVolve will align visualizations by axis to facilitate this sort of comparison; for example, object allocation behaviour aligned with method invocation behaviour allows one to correlate these two activities to find the methods that are most likely to allocate objects at each point. Aligned visualizations are illustrated by figure 4.

4.1.4 Orientation

A simple change that can aid in the application aligning or overlaying is to allow permutation of axes. EVolve allows orientation of axes to be arbitrarily assigned, and trivially changed.

4.2 User Interface

User interface features are particularly valuable in visualization systems; simple but important features can greatly amplify the ability to interpret data and draw conclusions. EVolve has several UI features designed to help in the interpretation or meaningful presentation of visualizations.

Sorting of reference axes is supported. These may be visualized in temporal order as identified by time stamps, or lexical order (by name). Figure 4 demonstrates both temporal and lexical order.

Zooming is provided to inspect details of a particular visualization. Event traces of program behaviour can be rather large, and so it is usually not possible to show the entire trace at the finest granularity. A magnification window allows the user to check specific data results without losing track of the context of the details shown. Figures 5 and 6 demonstrate this feature.

Mouse-overs are used to show the exact reference under the cursor as an identifier string. This simple feature allows one to very quickly identify program elements responsible for behaviour of interest. Figures 4, 5, and 6 show mouse-overs.

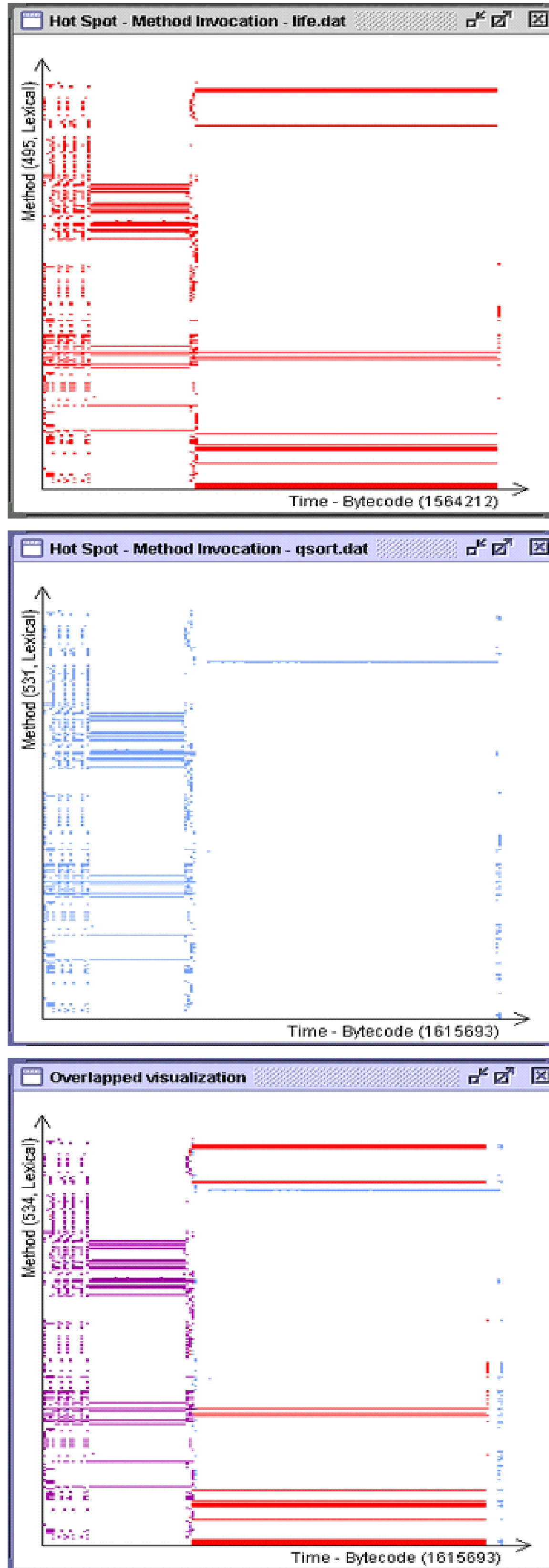

Figure 7: Overlapping: `life` hotspot (top), `qsort` hotspot (middle), and overlapped hotspots (bottom).

Selections of a subset of data can be separately visualized. A user may not find all aspects of their program worthy of inspection; reducing the visualization size allows one to focus on the interesting parts, and also permits faster processing of the visualization. All figures show selections of an entire program run.

5 Related Work

Visualization tools are often used for performance tuning. Programs such as Jinsight [1,8,9], JProbe [2] and OptimizeIt [3] are designed to help programmers optimize their programs by visualizing the runtime usage of system resources (CPU time, memory, etc). Unlike EVolve, these tools tend not to be extensible—they use built-in or specific profiling front-ends to generate trace data and use a fixed set of visualizations to interpret the data.

Visualization has also been applied to the fields of software understanding and reverse engineering. These tools, such as Dotplot [5] helps users explore self-similarity of code, Rigi [13] (using SHriMP views [12]) and Moose [7] help developers understand the hierarchy and structure of systems by visualizing static information (classes, methods, fields, etc.), usually obtained from parsing the source code,

Most software visualization tools are designed to visualize particular aspects of software systems, and provide aid in the development of new visualizations. One of the exceptions is BLOOM [10], which provides extensibility by using a visualization back-end supporting a variety of visualization strategies. Our approach to extensibility is to provide a framework that simplifies the task of both connecting new data sources and designing new visualizations.

Extensibility is more often seen in information visualization (vs software visualization) systems. These systems tend to concentrate on extensibility because they are designed to solve general-purpose problems. Visage [11], for example, is an information visualization environment for data-intensive domains that supports and coordinates multiple visualizations and analysis tools. Furthermore, Visage provides an interactive tool to facilitate creating new visualizations. EVolve is designed with a more specific domain in mind: Java program execution.

6 Conclusion and Future Work

In this paper we have presented the EVolve platform, an open and extensible framework for visualizing the runtime behaviour of Java programs. The architecture of EVolve is designed to facilitate the addition of new data sources as well as new kinds of visualizations. Both can be added independently, enabling a data provider to examine a new data source immediately using a wide range of visualizations, and allowing a visualization provider to test a new visualization technique on a variety of existing sources.

In order to study various aspects of the runtime behaviour of Java programs we have developed a collection of visualizations and integrated these into the EVolve platform. These visualizations are implemented as a hierarchy designed to be easily extended, and encoding features such as colouring, alignment and overlapping, which facilitate the comparison of multiple visualizations.

We have illustrated some visualizations from EVolve's built-in library. Hot spot visualizations highlight different program phases, showing when and for how long particular events occur. Specialized hot spots use colouring to visualize the occurrence of particular aspects of execution such as thread occurrence and polymorphism. Correlations visualize the co-occurrence of entities such as method invocation location and target. Stack visualization provides a detailed view of the run-time stack. Dotplot visualization highlights recurring phases as blocks of similar color or pattern.

Extensibility was demonstrated by adding new visualizations as the library developed. This was a straightforward process, requiring relatively little coding and minimal time (a few hours). For example, the predictability hotspot visualization was implemented with about 120 lines of Java code. Adding new data sources is similarly easy; we have used EVolve with traces generated from JVMPI, a customized Java virtual machine, and several other internal formats.

We plan to continue to extend EVolve's repertoire of visualization techniques, and test these on more data sources. Since extensibility is built-in, the core of the EVolve platform does not need to change. We are very welcome to suggestions of other kinds of visualizations to add to the framework and other users of EVolve are encouraged to contribute their new visualizations and/or data sources to the project. We have setup a website for downloading of

EVolve at `http://www.sable.mcgill.ca/evolve`. We are actively using EVolve in our research and graduate courses. This version of EVolve has already benefited greatly from the feedback of the students using the system. We plan to continue investigating other user-interface and comparison techniques that may improve comprehension of the resulting visualizations.

Acknowledgments

This work was supported, in part, by NSERC, FCAR and McGill FGSR.

References

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

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

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

[4] *Volano Benchmark*. http://www.volano.com/benchmarks.html.

[5] Kenneth Ward Church and Jonathan Issac Helfman. Dotplot: a program for exploring self-similarity in millions of lines of text and code. In *Proceedings of Journal of Computational and Graphical Statistics*, pages 2:153–174, 1993.

[6] 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.

[7] Michele Lanza and Stéphane 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.

[8] 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.

[9] 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.

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

[11] 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.

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

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