EVolve: An Open Extensible Software Visualization Framework*

Qin Wang, Wei Wang, Rhodes Brown, Karel Driesen, Bruno Dufour,
Laurie Hendren and Clark Verbrugge
School of Computer Science
McGill University
Montréal, Québec, CANADA H3A 2A7
[qwang21,wwang22,rhodesb,karel,bdufou1,hendren,clump]@cs.mcgill.ca

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.

CR Categories: D.3.5 [Software]: Programming Languages—Language Classifications - Object-oriented Languages; D.2.5 [Software]: Software Engineering—Testing and Debugging - Tracing

Keywords: Software Visualization, Trace-based Visualization, Object-oriented Systems, Java, JVMPI

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. 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. On the visualization side we wanted to be able to develop new 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 also be easily added.

Our final EVolve system can be used in two ways: (1) as a stand-alone tool using pre-defined data sources and visualizations and (2) as a toolkit for developing new visualizers (adding new data sources and 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 stored in a trace file). Furthermore, fields in elements are defined with specific *properties* which allows the EVolve system to automatically create appropriate menus for visualization creation and configuration.

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

1.2 Visualizations

In order to make a useful tool for our research, we have defined a collection of standard as well as new visualizations,some specific to our particular interest in 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. This initial set can be easily

*Supported in part by NSERC, FCAR and McGill FGSR.

ACM Symposium on Software Visualization, San Diego, CA

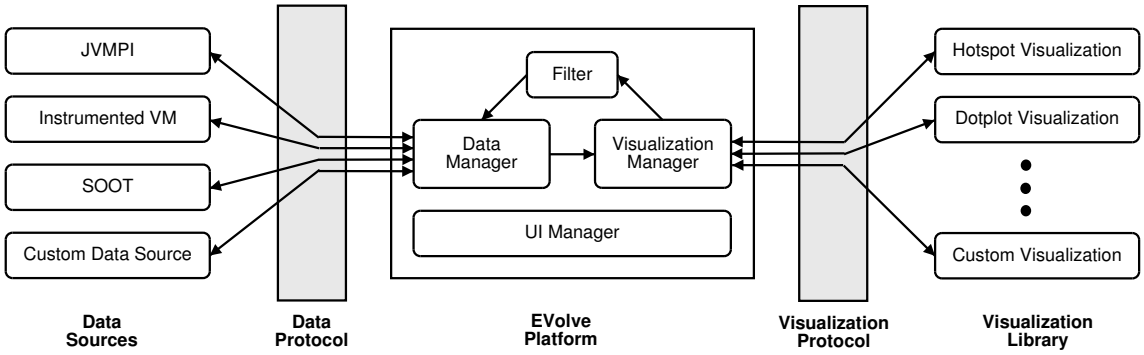

Figure 1: Architecture of EVolve.

expanded. A user wishing to add a new kind of visualization similar to an existing one can implement a subclass, and only define new behaviour. Completely new visualizations can be added by defining a new class higher up in the hierarchy. 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. In Section 5 we give an example of a new visualization and discuss the implementation effort involved. Finally, we give related work and conclusions in Sections 6 and 7.

2 Architecture

The EVolve platform consists of three components (see Figure 1). The leftmost *data source* component translates input data into EVolve's abstract representation so that it can be manipulated and visualized. The *visualization library* component at the opposite end presents this data using standard and newly built custom visualizations. The third (middle) component forms the fixed core of the EVolve platform. The core takes care of all communication between data source and visualization library, encapsulating the complex machinery that manipulates the data. This allows data sources and 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 suit their particular needs.

2.1 Data Representation

In order to make the EVolve platform extensible we limit the constraints imposed on the input format of the source data. An abstract and flexible internal representation is independent from the source from which data 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 are 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 are object allocations and method invocations. Execution traces typically contain few entities and numerous events, therefore EVolve caches entities in memory and keeps events on disk. In a typical run 99% of the elements in an execution trace are events.

Elements consist of fields of two types: *entity reference* (which we will refer to as *reference*) and *value*. A reference field refers to an entity whereas a value field holds scalar data.

In addition we define *properties* for value fields. Properties communicate generic data characteristics so visualizations can operate on appropriate data. Three properties are built-in:

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

coordinate: The `coordinate` property indicates a non-numeric, non-summable value. For example, an object address is a coordinate because addresses cannot be meaningfully summed.

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

EVolve allows the definition of custom data properties in order to support new visualization requirements.

2.2 The Data Protocol

The data source component converts 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 the data protocol by implementing 7 simple methods which are called by EVolve.

The `init()` method initializes 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: one for each of three three kinds of objects that a data source sends to EVolve. Each pair provides the start and the delivery of the next object for one object type.

EVolve uses *element definitions* to represent the structure of data elements (entities and events) in the execution trace. Element definitions encode properties of data fields and relationships between fields of data elements. The data source implements a method pair to send this information to EVolve: `startBuildDefinition()` and `getNextDefinition()`. EVolve first calls `startBuildDefinition()` to let the data source know that it is ready to accept definitions, and then calls `getNextDefinition()` repeatedly until a `null` value is returned. The second pair of methods, `startBuildEntity()` and `getNextEntity()`, deliver entities. The third pair, `startBuildEvent()` and `getNextEvent()`, deliver events. .

EVolve provides two convenience classes to assist in the creation of these three kinds of objects: `EntityBuilder` and `EventBuilder`. They generate a definition or an instance of a data element. The process is similar for both tasks. The builder class is instructed to start building a new definition (instance). Fields are repeatedly added. Once every field is provided, the builder class is instructed to return a 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 flexible, they must be able to display data from a variety of different data sources. Therefore any specific requirement must be expressed in a systematic, data-independent way. Similarly to 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 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 data property. The property constrains the fields from a data source which can be mapped to a dimension. For example, the bar chart in Figure 2 requires that its value dimension is an amount, i.e. a summable value. Dimensions are also associated with *data filters* which extract the field of interest for the selected type of event.

A new visualization in EVolve extends `Visualization`, which defines 9 abstract methods. These methods collectively form the visualization protocol. The first task is the *creation* of the visualization, supported by two methods: `createDimension()` creates the dimensions of the visualization, `createPanel()` creates the canvas on which the visualization will be drawn. The next two methods make the visualization configurable via the configuration dialog: `createConfigurationPanel()` creates a visualization-specific panel to be inserted in the generic configuration dialog (see middle panel of figure 2), `updateConfiguration()` sends a notification that a user has configured a visualization. Three additional methods support the visualization process itself: `preVisualize()` starts a new visualization phase, `receiveElement()` then passes elements one by one to the visualization, `visualize()` produces the visual representation once all elements have been sent. Two more methods then allow the user to manipulate visualization entities by making selections and sorting dimensions: `makeSelection()` and `sort()`.

2.4 The EVolve Core Platform

The core of the EVolve platform manages the user interface and handles communication between the data source and different visualizations, in order to allow both these components to work as independently as possible. The topmost picture in figure 2 shows the list of available visualizations. The `UIManager`, part of the EVolve core, automatically generates this menu from a list of registered visualizations with EVolve. The user selected a simple `Bar Chart`.

In order to promote extensibility, direct interactions between data source and visualizations are prohibited. Data sources provide an abstract description of their input format, and visualizations provide a description of requirements and capabilities. EVolve's data manager makes sure that only data elements with appropriate properties are sent to the visualization manager.

The configuration dialog in the middle screen shot of Figure 2 illustrates the link between data source and visualization: the EVolve

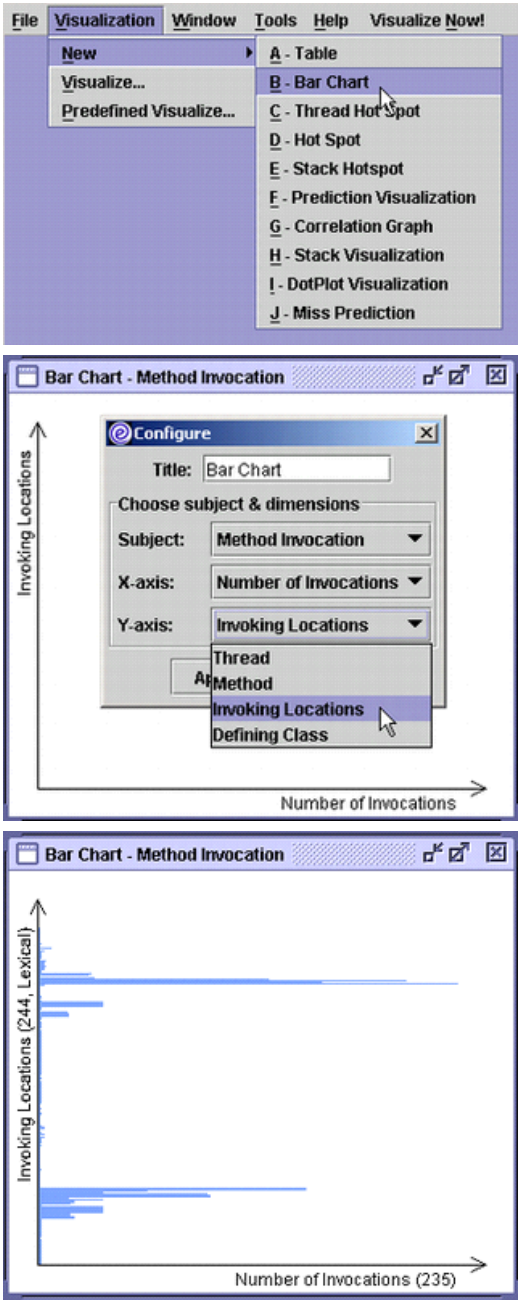

Figure 2: Barchart visualization process: selecting a registered visualization (top), configuring the Barchart with fields from the data source (middle), and visualization (bottom).

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 appropriate property. 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`.

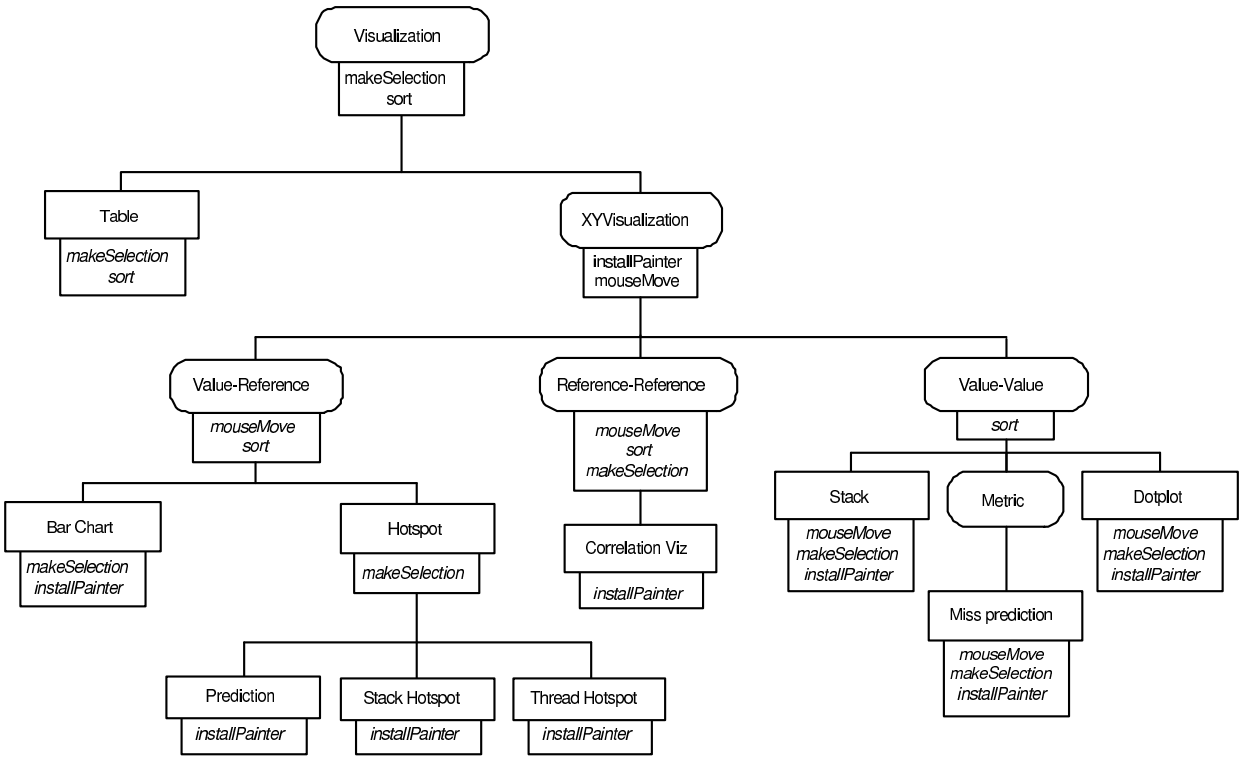

Figure 3: Visualization Hierarchy.

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 (*temporal* or *lexical*: see Section 3). 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 and apply operations on them, for instance colouring a particular data subset. A colouring scheme can be shared among different visualizations allowing one to keep track of a subset of interest across different views (see Section 3).

3 Visualization Library

EVolve can be used as a stand-alone tool to interactively create and modify multiple visualizations from an existing library. In this section we demonstrate these built-in visualizations, while emphasizing the ability to view one data source simultaneously in a variety of different views.

EVolve is also designed to be extensible. The core platform ensures that each new visualization can be applied to any data source which has the appropriate properties. The visualization interface described in Section 2 ensures seamless integration of any new visualization into the existing tool. In addition, we provide the existing library as an implementation hierarchy, so that new visualizations can reuse code from old visualizations. 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. We show abstract classes in rounded boxes, concrete classes in plain boxes, and overridden methods in italics. 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. Two methods can be inherited or overridden for visualization-specific behavior: `sort` provides different sorting schemes and `makeSelection` encodes how the user selects subsets of data. `Visualization` has a concrete subclass `Table`, which shows a set of references/values in table format, and an abstract subclass `XYVisualization`, which serves as root for all two-dimensional visualizations showing an *x* and *y*-axis.

`XYVisualization` declares `installPainter`, which determines the colouring of a visualization, and `mouseMove` which generates text when the user moves the mouse over a particular spot (e.g. showing the name of an invoked method). `XYVisualizations` differ in the way they treat coordinates on *x* and *y*-axis: each axis can be either a reference or a value. References refer to entities (see Section 2), are named and can be sorted and coloured to facilitate comparisons between different views. 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 one reference axis, and therefore share code for `mouseMove`, showing the name of the reference corresponding to the mouse position[1], and `sort`, sorting the reference axis temporally or lexically. `Reference-Reference` visualizations contain a reference on both *x* and *y*-axis. `Value-Value` visualizations, which are more open-ended, share `sort` but override `mouseMove`, `makeSelection` and `installPainter`.

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

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

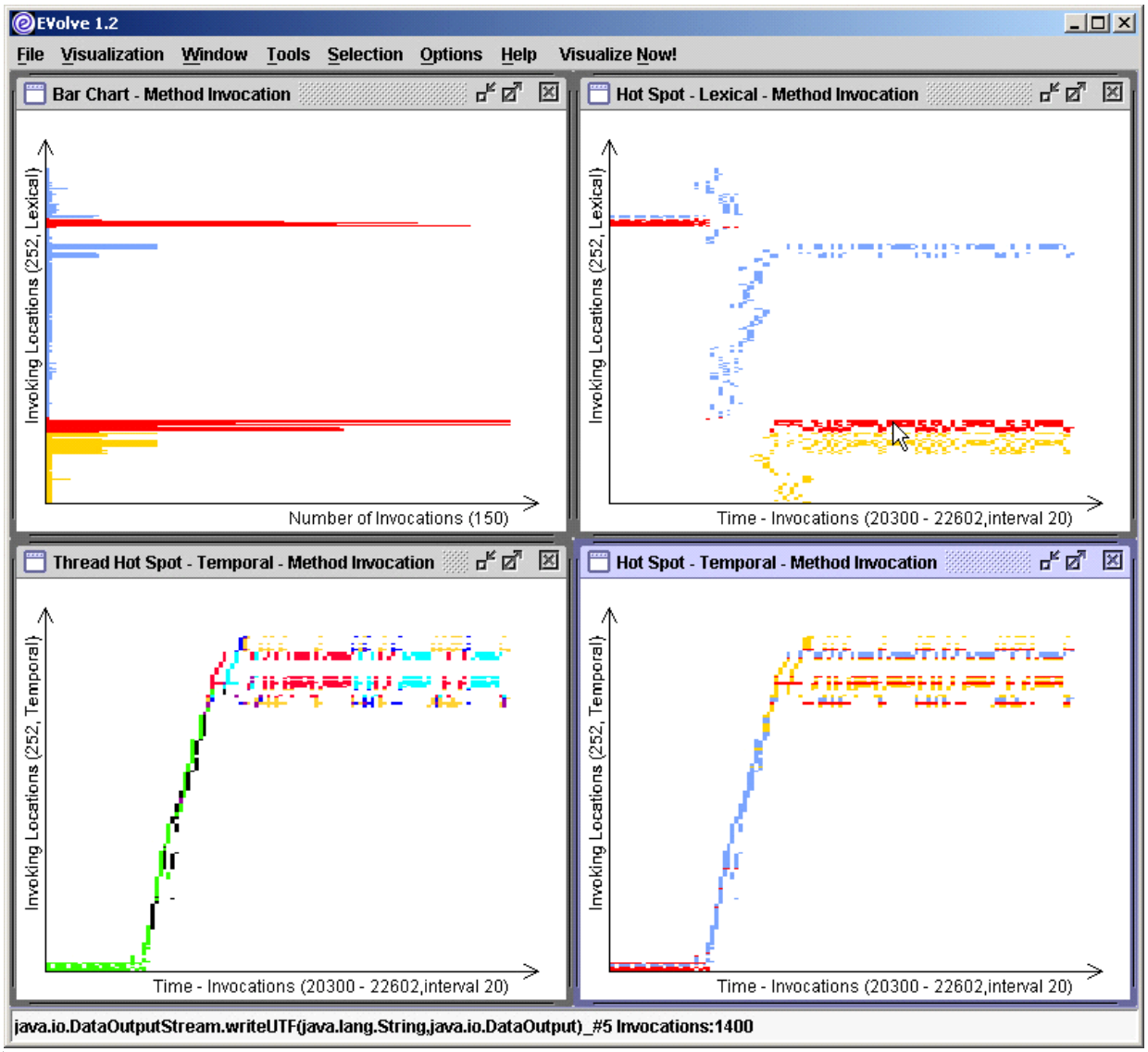

Figure 4: Four aligned visualizations: a barchart and three hotspots.

3.2 Bar Chart

Figure 4 shows four aligned visualizations of method invocations from a fragment of the Volano benchmark[VOLANO 2001]. The top left window shows a simple visualization: a horizontally oriented bar chart. This is the bar chart shown in Figure 2, after the user selected colours for particular subsets: the method invocation locations on the *y*-axis are coloured yellow (Volano), red (frequently called Java libraries) and blue (infrequently called Java libraries). The *x*-axis shows the total number of invocations for each of the 252 locations.

3.3 Hotspot

The top right window of Figure 4 shows a lexically sorted hotspot. The picture is similar to the bar chart, but now method invocations are shown as they occur in time, instead of summed together. The *y*-axis is identical to the bar chart *y*-axis, but the value axis (*x*-axis), shows the passing of time as method invocations: invocations 20300 to 22602, grouped in intervals 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 `mouseMove` method, implemented for all `ValueReference` visualizations, shows the name of the invoking location pointed at by the mouse (the name shows at the bottom of the screen).

The bottom right window of Figure 4 shows the same hotspot graph as the top right window, but with a temporal sorting of the

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

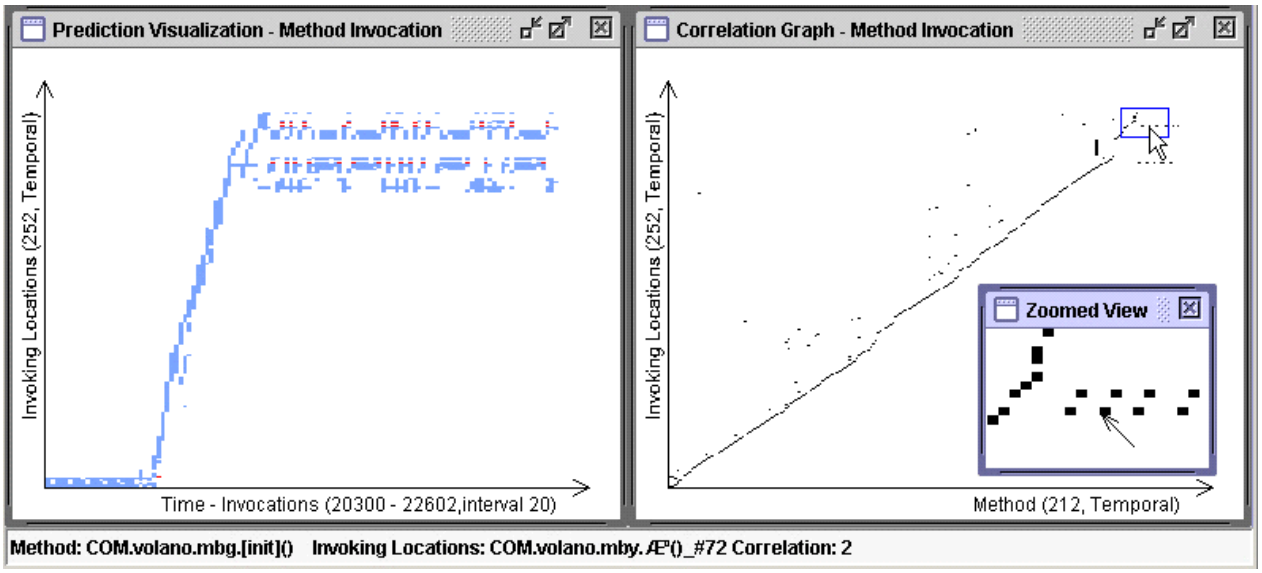

Figure 5: Polymorphism viewed in prediction hotspot and correlation visualizations.

reference axis. A temporal sorting orders entities by the time stamp of their first occurrence in the data source, while the default lexical sorting sorts them in alphabetical order. A temporal hotspot clusters together invocations that start together, and emphasizing program phases. The yellow `com.volano` invocations now appear clustered together with the Java library code that they 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.

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 *x* and *y*-axis definitions with the previous hotspots. Only the colouring scheme is customized: method invocations appear in blue when the invoked target method does not change within an interval, in red when it does change —i.e., the invocation is polymorphic. Apparently `com.volano` exhibits some polymorphism in the later phase.

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 wrong at least once in the time interval. Different predictors can be visualized by plugging them into the framework.

Note that this visualization is not restricted to method target prediction. It displays a measurement of polymorphism only because the user selected invoked method fields from method invocations events. In general, the prediction of any event field 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.

3.4 Correlation

The `Correlation` class, a subclass of `Reference-Reference`, is illustrated in the right part of Figure 5. Its *y*-axis shows method invocation locations, identical to the *y*-axis hotspot to its left. Its *x*-axis shows invoked methods. A correlation visualization shows when a reference *x* occurs in the same event as reference y, therefore this graph displays the method targets of all invocation locations.

A horizontal row of dots indicates an invoking location with more than one target method (a polymorphic location). A vertical row 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 thus show as horizontal rows of dots.

Figure 5 also demonstrates the use of a `Zoomed View`, which is 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 (implemented by the `moveMouse` method.

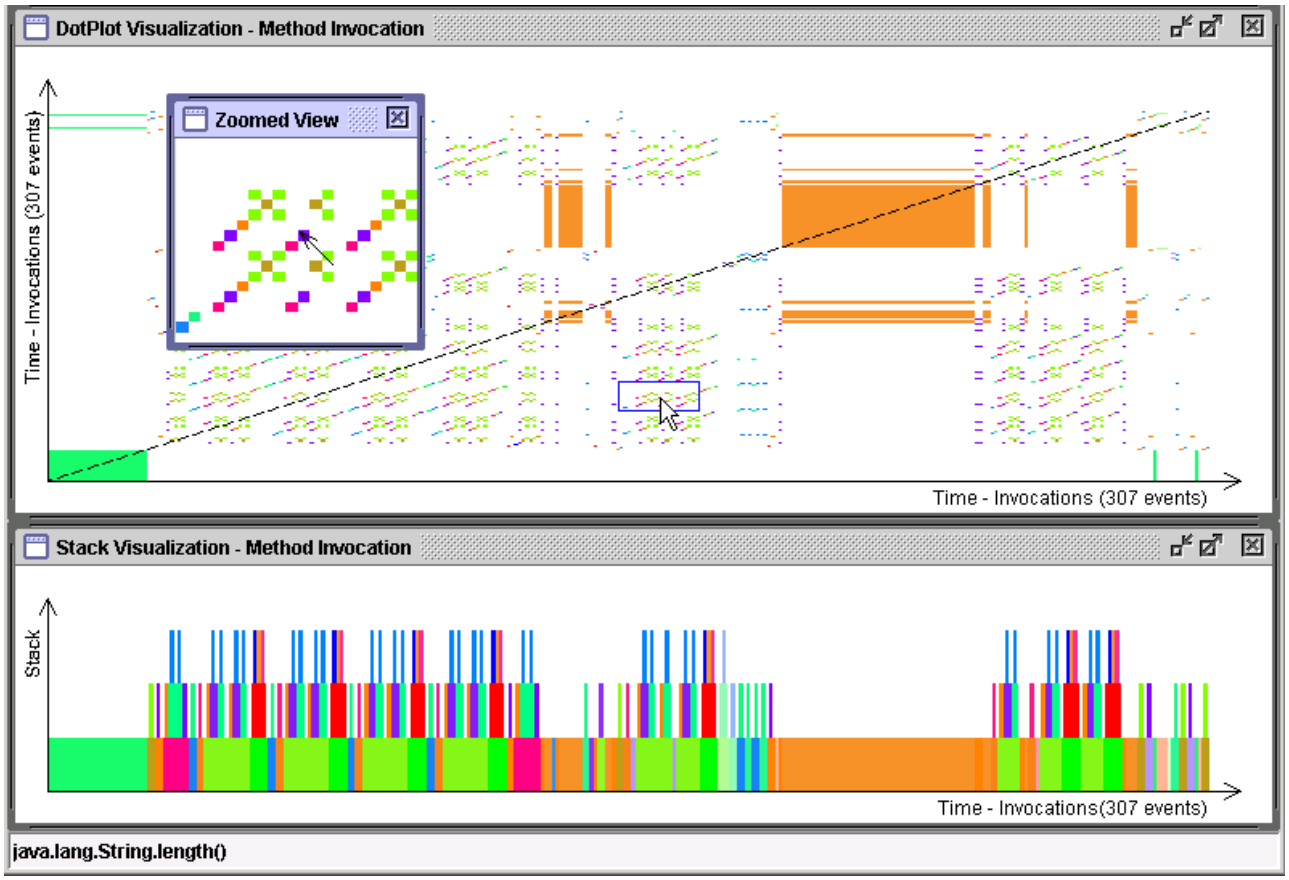

Figure 6: Method invocations as dotplot and stack visualization.

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 [Pauw et al. 2002]. 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 [Church and Helfman 1993]. 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 [Dufour et al. 2002], 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 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 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), allowing information presented in two views to be easily correlated.

4.1.2 Overlap

Correlations between two visualizations are most obvious when the visualizations are overlapped. Common information then appears in the same location, and distinct information appears in different locations; this can be quite informative. For example, identification of related "phases" in execution becomes easy—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, 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.

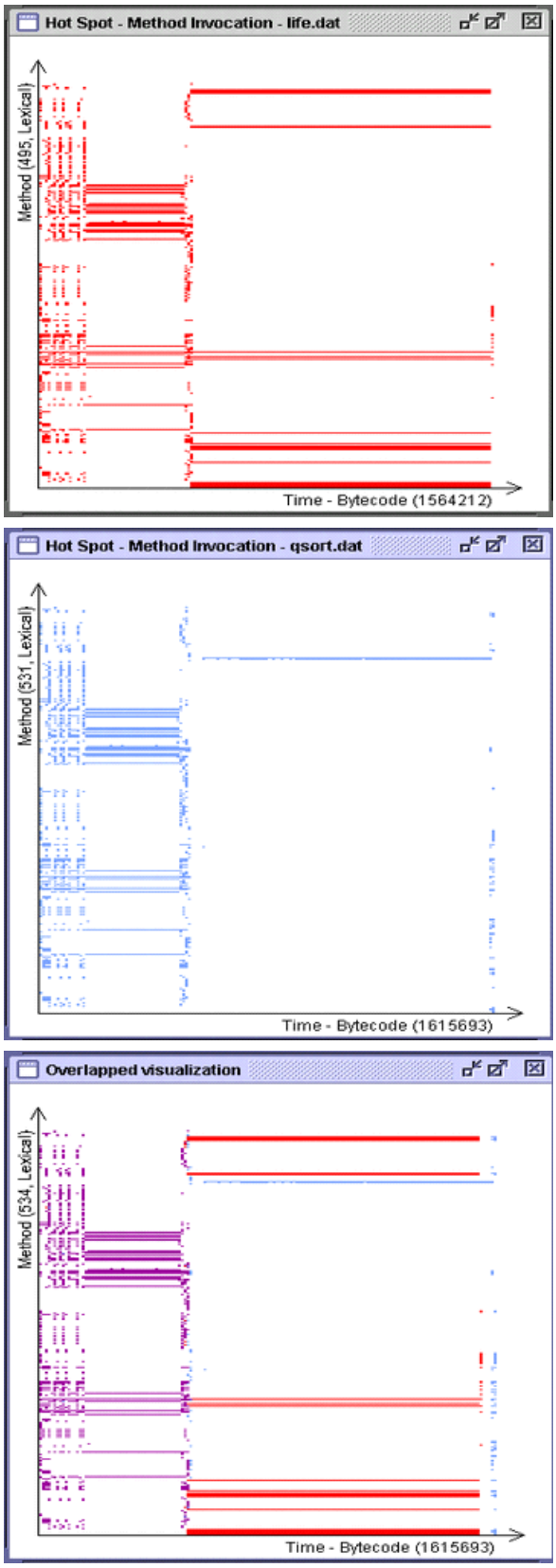

Figure 7: Overlapping: `life` hotspot (top), `qsort` hotspot (middle), and overlapped hotspots (bottom).

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.

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 Extending EVolve

We recently added the Event visualization (see Figure8), in which each event is denoted by a colored rectangle. Events are presented from left to right and top to bottom as in the order that they appear in the data trace. Adding this visualization was simplified by the implementation hierarchy. The three different coloring schemes (painters) were re-used from other visualizations: default painter (user-determined coloring as in Figure4, with Volano calls in yellow, Java library calls in red and blue), prediction painter (polymorphic calls in red, others in blue, as in Figure5) and random painter (a unique color for each method). These are the same painters as used by Hotspot, Prediction Hotspot and Dotplot, respectively. Besides some codes for the UI interface,the following methods had to be overridden:

preVisualize: prepare data structures used in this visualization

receiveElement: extract data from events and feed them to painters in correct formats

sort: show the sorted visual representation

mouseMove: show entity information according to mouse pointer

makeSelection: collect data selected by the end-user

This was accomplished in a few days, demonstrating the ease with which EVolve can be extended and a new visualization can be integrated into the existing framework.

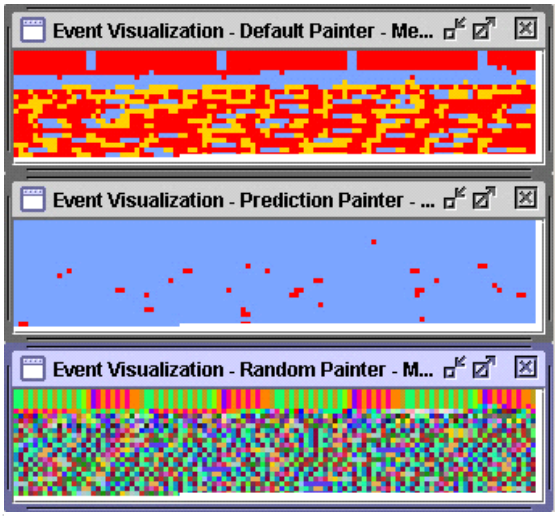

Figure 8: Event Visualization.

6 Related Work

Visualization tools are often used for performance tuning. Programs such as Jinsight[JINSIGHT 2002; Pauw et al. 2002], JProbe[JPROBE 2003] and OptimizeIt[OPTIMIZEIT 2003] are designed to help programmers optimize their programs by visualizing the runtime usage of system resources (CPU time, memory, etc). An earlier visualization system originating in the IBM T.J. Watson Research Center is PV [Kimelman et al. 1994][Kimelman et al. 1998], which provides trace-based visualization of different system layers, including the hardware, and the juxtaposition of different Views. Jerding, Stasko and Ball visualize message patterns in object-oriented languages using information murals[Jerding et al. 1997]. These resemble the hotspot visualizations generated by EVolve. An other trace-based visualization tool which bears some resemblance to EVolve is ParaGraph[Heath and Etheridge 1991], which focuses on parallel program visualization, 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[Church and Helfman 1993] helps users explore self-similarity of code, Rigi[Storey et al. 1997] (using SHriMP views[Storey and Müller 1995]) and Moose[Lanza and Ducasse 2001] 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 no aid in the development of new visualizations. One of the exceptions is BLOOM[Reiss 2001], 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. Also, EVolve differs from BLOOM in providing features to easily compare visualizations (overlapping and sharing of color schemes), and to refer back to entities from the data that generates a visualization (mouse-over).

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. ADVIZOR[Eick 2000] is a commercial tool with a rich set of visualization mechanisms that supports the linking of different views and can be extended. Visage[Roth et al. 1996], 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: to visualize program execution behavior.

7 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.

References

CHURCH, K. W., AND HELFMAN, J. I. 1993. Dotplot: a program for exploring self-similarity in millions of lines of text and code. In *Proceedings of Journal of Computational and Graphical Statistics*, 2:153–174.

DUFOUR, B., DRIESEN, K., HENDREN, L., AND VERBRUGGE, C. 2002. Dynamic metrics for compiler developers. Sable Technical Report SABLE-TR-2002-11, McGill University, School of Computer Science.

EICK, S. 2000. Visual discovery and analysis. *IEEE Transactions on Visualization and Computer Graphics 6*, 1 (Jan.), 44 –58.

HEATH, M. T., AND ETHERIDGE, J. A. 1991. Visualizing the performance of parallel programs. *IEEE Software 8*, 5 (Sept.), 29 –39.

JERDING, D., STASKO, J., AND BALL, T. 1997. Visualizing message patterns in object-oriented program executions. In *Proceedings of the Nineteenth International Conference on Software Engineering (ICSE'97)*, 360–370.

JINSIGHT. 2002. Jinsight. *http://www.research.ibm.com/jinsight/*.

JPROBE. 2003. Jprobe. *http://www.sitraka.com/software/jprobe/*.

KIMELMAN, D., ROSENBURG, B., AND ROTH, T. 1994. Strata-various: Multi-layer visualization of dynamics in software system behavior. In *Proceedings of the IEEE Visualization '94 Conference*, 172–178.

KIMELMAN, D., ROSENBURG, B., AND ROTH, T. 1998. Visualization of dynamics in real world software systems. In *Software Visualization: Programming as a Multimedia Experience*, J. Stasko, J. Domingue, M. H. Brown, and B. A. Price, Eds. MIT Press, Cambridge, MA, 293–314.

LANZA, M., AND DUCASSE, S. 2001. 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)*, 300–311.

OPTIMIZEIT. 2003. Optimizeit. *http://www.optimizeit.com/*.

PAUW, W. D., JENSEN, E., MITCHELL, N., SEVITSKY, G., VLISSIDES, J., AND YANG, J. 2002. Visualizing the execution of Java programs. International Seminar, Dagstuhl Castle, Germany, May 20-25, 2001. In *Lecture Notes in Computer Science Vol. 2269*, Springer Verlag, 151–162.

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

ROTH, S. F., LUCAS, P., SENN, J. A., GOMBERG, C. C., BURKS, M. B., STROFFOLINO, P. J., KOLOJEJCHICK, J. A., AND DUNMIRE, C. 1996. Visage: A user interface environment for exploring information. In *Proceedings of Information Visualization, IEEE*, 3–12.

STOREY, M.-A. D., AND MÜLLER, H. A. 1995. Manipulating and documenting software structures using SHriMP views. In *Proceedings of International Conference on Software Maintenance*, 275–285.

STOREY, M.-A. D., WONG, K., AND MÜLLER, H. A. 1997. Rigi: A visualization environment for reverse engineering. In *Proceedings of the International Conference on Software Engineering (ICSE'97)*, 606–607.

VOLANO. 2001. Volano benchmark. *http://www.volano.com/report/index.html*.

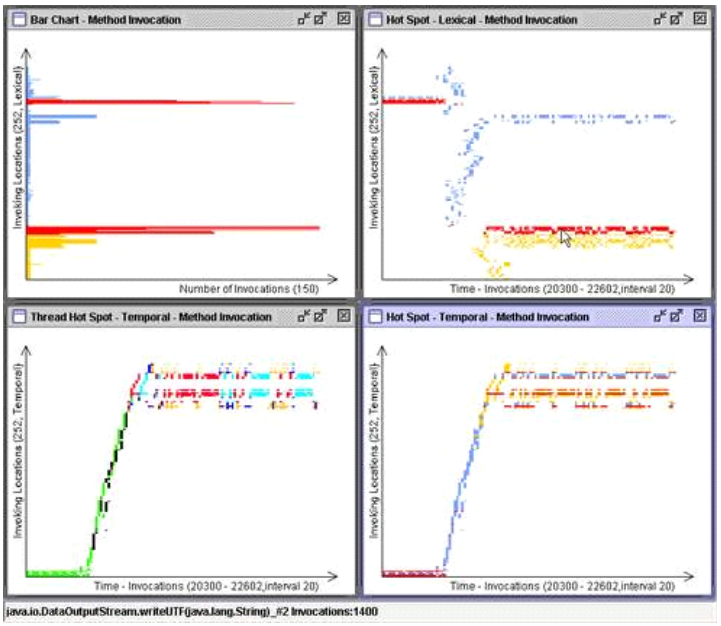

Figure 4: Four aligned visualizations: a barchart and three hotspots.

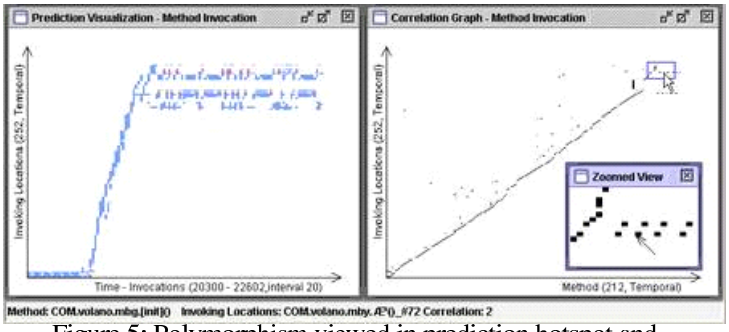

Figure 5: Polymorphism viewed in prediction hotspot and correlation visualizations.

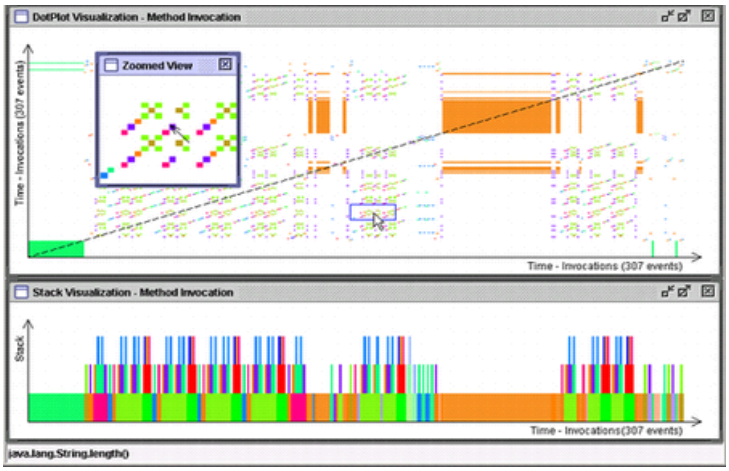

Figure 6: Method invocations as dotplot and stack visualization.

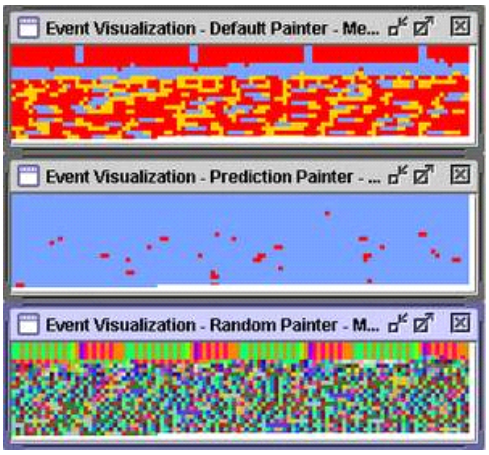

Figure 8: Event Visualization.

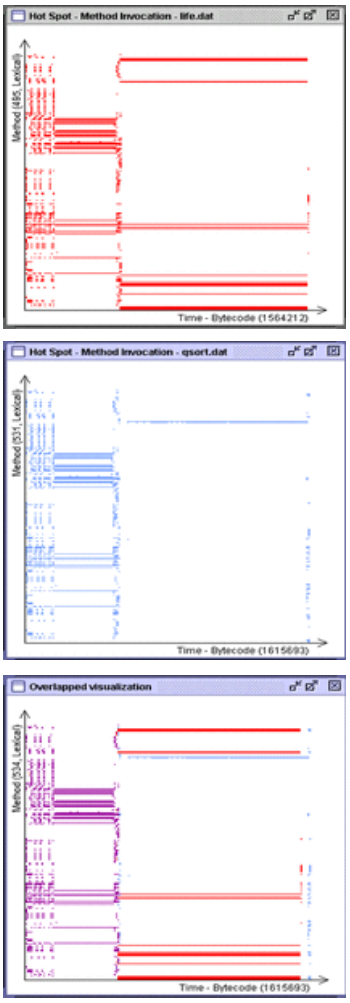

Figure 7: Overlapping: **life** hotspot (top), **qsort** hotspot (middle), and **overlapped** hotspots (bottom)