# Visualization of Dynamic Program Aspects

Pieter Deelen, Frank van Ham, Cornelis Huizing, Huub van de Wetering
Mathematics and Computer Science Department
Technische Universiteit Eindhoven

## Abstract

Object-oriented software is designed by introducing classes and their relationships. When software is being executed, however, a developer loses sight of the classes he created during the design and coding of the software. This paper describes a tool that visualizes the program behaviour during execution. For Java programs this tool, called TraceVis, can collect relevant program execution events, like object creation, method entries and method exits. It uses bytecode instrumentation techniques to extract this information. The resulting trace of events is visualized using a dynamic call graph and a timeline representation. These views can be customized by selecting a time range in the execution, filtering the events, and manipulating the graphic representation. Some use cases for TraceVis are presented.

## 1 Introduction

Nowadays, much software is developed in an object-oriented fashion. The idea behind object-orientation is that, during execution, a program is comprised of a collection of objects, which contain data and interact by sending each other messages. The form of the data and the behavior of an object is described in a class. When working in an object-oriented way, a software developer first designs a structure for his program by introducing classes and their relationships. He then implements these classes by writing source code in one of the many object-oriented programming languages. To construct this software, the developer reasons about classes and objects using a mental model. To see whether this reasoning is correct, the software is executed. During execution, however, the concept of classes and objects is hidden. This is reasonable, because the end user, the person the developer creates the software for, is not interested in how a program works, but rather in what it does. However, the developer might want to know what the software does internally.

The study of program executions can be interesting, because through execution many facts can be found which are not (immediately) apparent from the source code. Such facts include the number of objects which are created from a class, or the amount of time spent in a method.

Several kinds of tools already exist to inspect the inner workings of an executing program. The most prominent of these are the debugger and the profiler. A debugger gives the developer a fine-grained control of the program, but it does not provide any overview of the execution. A profiler is meant to find performance problems in software. To this purpose, a profiler gathers certain statistics about the execution of a program, such as the time spent in a certain class or the memory usage over time. This gives a high-level overview of the execution, but not many details.

One method to provide insight into executions is to create a visualization. The tools mentioned above use visualization only in a limited fashion, if at all. Visualization is the process of using computer imaging techniques to provide insight into abstract data, like executions. A visualization can be used to make complex information presentable. Two kinds of software visualization can be discerned, viz.:

- visualization of static software aspects, and
- visualization of dynamic software aspects.

Static software aspects are software aspects which can be determined before the execution of the software. Source code and UML class diagrams are examples of static software aspects. Dynamic software aspects are software aspects which can only be determined by executing the software. The number of object creations and the number of sent messages are examples of dynamic software aspects. A dynamic software visualization can still use static aspects of the software, such as the classes which are defined in the source code. In fact, it is beneficial to do so, because this creates a link between the artifacts the developer has created and the execution of the program.

The visualization of the program execution can be run either during execution (*online*) or after the execution (*offline*). An online visualization tool updates its views continuously to show the program's state. It allows the user to suspend and resume the executing program. In case of an offline visualization, the executing program stores information about its execution in a trace file. After the program

has terminated, the visualization tool can read this trace file and visualize it. With an online visualization the user can more easily associate external behavior (e.g., the response to clicking on a button) with internal behavior (e.g., a method call). An offline visualization, on the other hand, can provide an overview of the whole execution. The trace can be studied in its entirety.

This paper describes the tool TraceVis which uses offline visualization to show aspects of program executions. We chose it to be an offline tool to avoid a performace penalty due to resource sharing with the running program and to allow the handling of larger datasets. The amount of information in a program execution and the number of different aspects are so large that a choice has to be made as what to visualize. After some experimentation, we chose to focus on class interaction. So, the main purpose of TraceVis is to gain insight into the dynamic interaction between classes. This information should be presented using visualization. In theory, such a tool could visualize programs programmed in any object-oriented programming language. However, for practical reasons, we restrict to Java. Java provides interfaces which facilitate the collection of execution data without changing the source code of the executed program. TraceVis is implemented in Java which makes it easy to use these interfaces.

We targeted a group of users interested in program executions, asked them what questions they would like to have answered by TraceVis and compiled the following list of questions: Which classes interact, i.e., call each other? How frequently do these classes interact? Which methods are called? How many instances does each class have? Which classes are high-level (mainly send calls) and which are low-level (mainly receive calls)? The first four questions can also be applied to subranges of the execution. For instance, a programmer might ask which classes interact during the initialization of a program.

In the next section we discuss related work. In sections 3 and 4 we discuss the data and its collection. Section 5 explains the visual solutions, section 6 introduces the interaction, followed by an evaluation in section 7 and conclusions and future work in section 8. For a more detailed discussion we refer to the master's thesis [3] of othe first author and a website for the tool [11].

## 2 Related Work

There are many tools that visualize program executions in one way or the other. We shortly discuss some of them.

Jive [10] is an online visualization tool. It provides two views: a class and package view, and a thread view. The class and package view is divided into a grid in which each cell visualizes runtime aspects about a class or package. These runtime aspects include the number of calls a class has received, the number of instances, or how often a thread has synchronized on instances of a class. These aspects can be mapped to various visual properties of a cell, such as width, height, brightness, or hue. Jive is equipped with a "rewind" function, that is, the user can temporarily suspend the program and scroll to a previous state of the program. An interesting aspect of Jive is that it has an efficient data collection. This allows it to be run alongside the program under study without too much of a slow down. However, as described in [10], the authors had to restrict the amount of collected data, which limits its usefulness. Furthermore, Jive does not visualize class interactions directly.

*Inter-class call clusters* was designed by De Pauw et al. [9] to provide a dynamic overview of communication patterns between classes. It is an online visualization tool. Classes are drawn as floating labels containing their names. The amount of communication between classes determines the distance between them. As the execution progresses, classes which communicate more often will be placed closer together. The view does not show lines between a caller class and a callee, but it does show the stack.

The *communication interaction view* is one of the views presented by Bertuli et al. in [2]. It is an offline visualization. During the execution of the program under study, statistics about classes are gathered, such as the number of method invocations, and the number of called methods. After the execution these statistics are displayed in the communication interaction view. The creation interaction view is similar, but instead of statistics related to calls, statistics related to creation are displayed. With these views it is not possible to select a subrange of the execution.

## 3 Data model

From a Java program execution we obtain a list $\Omega$ of time-stamped events. Several different types of events and associated attributes are collected:

- Java virtual machine (JVM) start/end
- thread start/end: thread id
- class loads: class name
- object (de)allocation: class name, object id
- method entry/exit: class name, method name, thread id.

From the event list $\Omega$ and a time interval $[t_1, t_2]$ we build a *dynamic call graph* $G(\Omega, t_1, t_2)$. $G$ is a directed graph $(V, E)$ where the set of vertices $V$ is the set of all classes that appear in any of the events in $\Omega$, and $E$ is a collection of edges $AB$ such that a method call from class $A$ to class $B$ exists in the events of $\Omega$ with timestamp in $[t_1, t_2]$. From

$\Omega$ we can deduce several attributes for the vertices and the edges. A vertex $C$ has the following attributes:

- the number of calls class $C$ has made,
- the number of calls $C$ has received, and
- the number of instances of exactly class $C$.

An edge $AB$ has attributes concerned with calls from class $A$ to class $B$:

- the total number of calls from $A$ to $B$, and
- the number of times each method in $B$ has been called from $A$.

So, the data model consists of a set of time-stamped events $\Omega$ and a dynamic call graph $G(\Omega, t_1, t_2)$. We now enhance it using call assignment and event filtering.

### 3.1 Call Assignment

The phrase "a call from class $A$ to class $B$" can have several meanings. When a method $m$ is called on an instance of class $C$, the call could be associated to the following, not necessarily different, classes:

1. The *declaring* class (or interface): the highest supertype of $C$ containing a definition or declaration of $m$.
2. The *defining* class of $m$: the lowest superclass of $C$ containing a definition of $m$.
3. The *object* class: $C$ itself.

Assigning calls to the declaring class provides a high-level view, since classes high in the inheritance hierarchy are usually more abstract. This option is mainly relevant from the point of view of class design. It can be used to view dependencies between classes. If calls are assigned to the defining class, the call graph gives a more implementation-oriented view. Therefore, this option is particularly interesting if the user wants to see which parts of the code are related and which parts are being used most frequently. If calls are assigned to the object class, the call graph emphasizes the interaction between objects, which is central to the object-oriented paradigm. The view this option provides is that a class is the collection of its instances.

### 3.2 Filters

To reduce the amount of information which is presented to the user, the data model allows the user to specify filters. A filter $f$ is applied on the set of events $\Omega$ to obtain a new set of events $f(\Omega)$ and a new call graph $G(f(\Omega), t_1, t_2)$. So, the result of applying a filter is that the attributes of all vertices and edges are modified to reflect the new situation. An example of a filter is: Filtering out a class, which means that all events directly or indirectly related to this class are removed from the list of events and consequently from the dynamic call graph. A second example is a filter that leaves only constructor calls. This can be used to show which classes allocate objects of other classes.

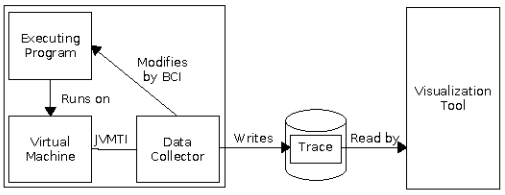


**Figure 1. The data collection process.**

## 4 Data Collection

The program under study runs on a Java Virtual Machine (JVM). Data about its execution is collected by the data collector (see figure 1). The data collector is a library which implements a JVM Tool Interface (JVM TI) agent and performs byte code instrumentation (BCI) on the executing program. The data collector writes a trace file containing the events. The file is read and visualized by the visualization tool. The current data collector can collect all required information. Nonetheless, it has some limitations: It may not be feasible to instrument all classes of a program due to the overhead BCI imposes, and parts of the program, like the standard libraries, can be excluded from instrumentation; the current data collector cannot detect calls from instrumented classes to uninstrumented classes, and vice versa.

## 5 Visualization

We visualize the collected data according to the data model in a *structural* view for the dynamic call graph and a *time line view* for the events.

### 5.1 Structural View

The Structural View visualizes the dynamic call graph during execution. One way to present this graph is to use a dynamic layout that is continually modified during the execution. This has the effect, however, that the position of vertices changes continuously and we found that a static layout is more preferable: the graph layout is computed in advance, and never changed during the use of the visualization. A static layout does not provide an ideal layout for every moment of the execution, but it allows the user to

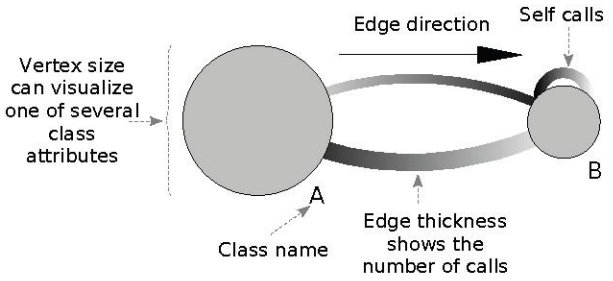


**Figure 2. Representation of the call graph.**

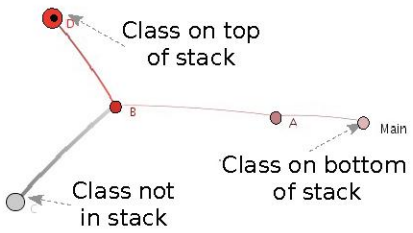


**Figure 3. Stack based color mode.**

build up a so-called "mental map", which links the identities of vertices to their positions. This mental map can be facilitated even further by drawing a "background graph", i.e., by drawing elements which are not part of the current graph vaguely in the background.

The layout of a graph is critical to its readability. We use the following quite general criteria for the layout: Vertices should be uniformly distributed, edges should have uniform lengths, and vertices should not overlap. The first criterion reduces clutter caused by too many vertices packed too close together, the second criterion implies that adjacent vertices should be close, and the last criterion prevents the cluttering of vertex circles. A force-directed layout algorithm in the style of the spring-embedder algorithm of Eades [4] can generate a layout while optimizing the above criteria. Regarding the second criterion, we cannot use the distance between the vertex centers, as usual, since we represent vertices as circles of varying size (see below). Instead, we use the distance between the vertex circle borders. To improve convergence we follow Harel and Koren [6] in starting with iterations with vertices of size zero and during subsequent iterations we gradually increase the vertex sizes to their intended values. The spring force has also been modified in two other aspects. First, it was observed that high-degree vertices are subject to many different forces. This causes those vertices to jump around wildly. To dampen these movements, the force a spring exerts on a vertex is divided by the degree of the vertex. Second, the springs are natural (linear) springs, instead of the logarithmic springs used by Eades.

Figure 2 gives an overview of the concepts used in the visual representation of the call graph. Vertices represent classes and are drawn as circles, varying in size and color. The *size* of a vertex can visualize one of the three vertex attributes. The *color* of a vertex can visualize different aspects, by means of two different color maps: one map containing a user-specified color per class for class/package identification, or another map based on the position of this class in one of the call stacks, with classes lower on the stack being darker. Furthermore, the vertex on top of the stack is marked by a filled black circle in its interior. Figure 3 shows an example of this color mode.

Edges are drawn from the center of its source to the center of its destination. To prevent overlap between edges $AB$ and $BA$, edges are slightly curved. To indicate direction, gradients are used: An edge is light at its source and dark at its destination. The main attribute of an edge is the number of calls from its source class to its destination class. The logarithm of this attribute is visualized by the thickness of the curve. If two vertices are colored according to their position in the stack, edges between these vertices interpolate these colors.

## 5.2 Time Line View

The time line view visualizes time-related aspects of the trace. It can be used to pinpoint interesting points in the execution. TraceVis, offers two different time line views: The activity view that visualizes the activity of classes, and the instance view that visualizes the number of instances of classes during the execution.

### 5.2.1 Activity View

The activity view provides a compact view on the activity of classes. A class is considered to be active when a method from this class is executing.

The activity view has been inspired by UML sequence diagrams [5, Chapter 4]. A sequence diagram (Figure 4) is a two-dimensional diagram in which the horizontal axis displays the objects which participate in the collaboration and the vertical axis visualizes time. Sequence diagrams were designed to model specific behavior of a small set of objects. For larger sets of objects and messages, the diagram will become too large and cluttered. To make the sequence diagram more suitable for large traces, we made some modifications: The activity view displays classes, instead of individual objects. Calls and returns are only visualized after brushing. A different notion of activation is used: in sequence diagrams, a participant is active when it

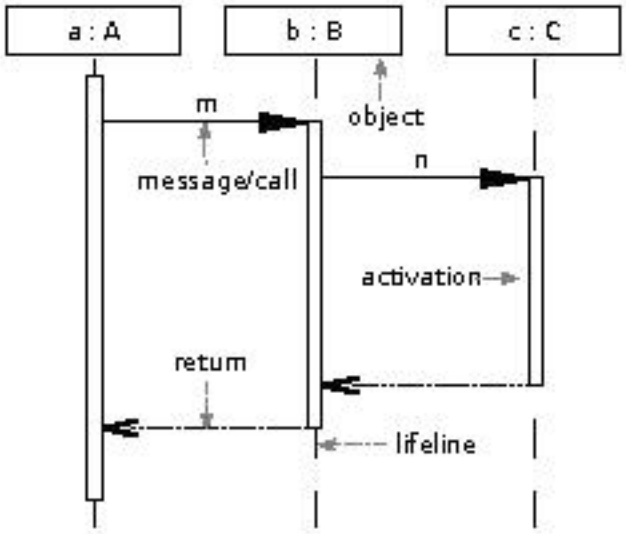


Figure 4. An example UML sequence diagram. Object a of class A calls method m on object b of class B, which in turn calls the method n of object c of class C. Each object has a *lifeline* with so-called *activations*, which indicate when the object is active.

is on the call stack; in the activity view, a participant is active only when it is on top of the call stack. In addition, the functions of the axes are interchanged, in order to make the activity view fit in better with other visualizations. And finally, the notation has been made more compact, primarily by using just a few pixel rows for each class. Each row displays the activations of the corresponding class.

Figure 5 provides a conceptual drawing of this notation. The number of activations in an execution is typically much larger than the number of activations depicted in a sequence diagram. Moreover, the length of activations can vary from a few microseconds spent in, e.g., an accessor method, to minutes spent in a long-running computation or even longer. Therefore, it is impractical to draw each activation separately. On the horizontal axis, each pixel represents a time range. The saturation of the pixel summarizes the activity of the class in this time range. A desaturated color (close to white) means that the class was barely active during this time range. A fully saturated color means that the class has been active during the full time range. This approach allows for the drawing of both short- and long-running activations.

The amount of pixel rows $h_c$ assigned to a class $c$ is given as follows.

$$h_c = \frac{a_c^\beta}{\sum_{c \in C} a_c^\beta} \cdot H, \qquad (1)$$

Where $C$ is the set of classes, $H$ is the height of the view,

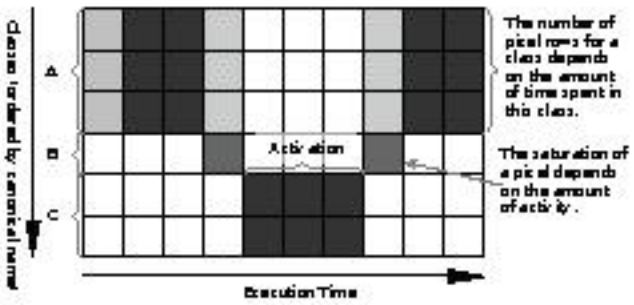


Figure 5. Conceptual drawing of an activity view. First class A is active, then B, and then C. After the activation of C, B becomes active again, and finally A resumes activity.

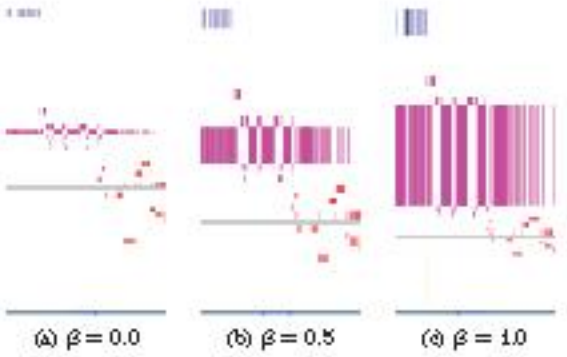


Figure 6. Different activity exponents $\beta$ in activity views showing the same trace.

$a_c$ is the activity ratio given by

$$a_c = \frac{\text{time spent in class } c}{\text{total execution time}}$$

and $\beta \in [0, 1]$ is the activity exponent, which parameterizes the relative weight of classes (see figure 6). If $\beta = 0$, all classes will be assigned the same height. If $\beta = 1$, the height of the class is linearly dependent on its activity ratio.

### 5.2.2 Instance View

The instance view provides a compact view on the amount of instances each class has during the execution. It is similar to the activity view. Time is on the horizontal axis and classes are on the vertical axis. In the instance view, however, the saturation of a pixel does not visualize the activity of the corresponding class in the corresponding time range, but the amount of instances of this class in this time range. Figure 7 gives an example of this view. Unlike the activity view, all classes have the same height in the instance view.


**Figure 7. An instance view showing the metric time lines for selection of a subrange of the execution.**

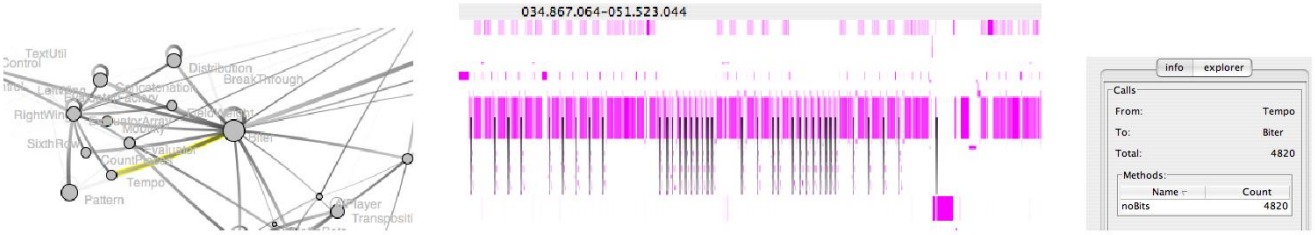


**Figure 8. A selected edge (l), the corresponding calls (m), and an aggregated attribute (r).**

## 6 Interaction

The views described in the previous section are by no means static views. The user can interact with them to retrieve additional information. Vertices (classes) and edges (method calls) (see figure 8) can be selected to show their (aggregated) data. They can also be brushed to show class names and method call statistics in a tooltip. The views can be zoomed and panned to show specific areas of interest. The metric start time $t_1$ and the current time $t_2$ can be changed by dragging the corresponding colored vertical lines in the time line view (see figure 7). The user can replay the events one by one during which method entries and exits are animated in the structural view. Furthermore, classes can selectively be hidden, predefined filters can be applied, and custom colors can be assigned to classes.

## 7 Evaluation

We used TraceVis to examine traces generated by an international draughts playing program Rambo [13], and a software building tool Ant [1].

### 7.1 Rambo

The execution trace of Rambo discussed in this section covers the initialization of the program, and a few moves from a international draughts game between two computer players. A computer player determines its next move by selectively walking the game tree, evaluating the encountered game situations, and picking the move resulting in the best evaluated situation. Classes have been colored in the following way: red for the evaluator classes, pink for an important utility class, Biter, blue for the main class, Mooi, and green for user interface classes. The trace contains 58 active classes and nearly 380.000 events.

Figure 12 shows an overview of the trace. From this picture, and after interaction, several observations can be made. The program spends most of its time in the classes Biter and ConsoleArea. The class Biter receives many calls from the evaluator classes and sends many calls to the class BoardState. The main class Mooi sends calls to many other classes and receives very few calls. Furthermore, a clear separation is visible between the back end, which computes the best move for a given game situation (red, pink, and purple classes in the figure), and the front end, which shows the game situation on the screen (green classes). And finally, two phases in the execution can be discerned: initialization and playing of the game.

TraceVis can be used to gain an insight into object instantiation by assigning calls to the object class when loading the trace, by selecting the number of instances as the attribute which determines the vertex size, and by filtering all but the constructor calls from the event list. The results are shown in Figure 9. Three classes stand out: GameNode, Pattern and Move. By looking at the structural view it can be determined that GameNode objects are allocated by three other classes, Pattern by EvaluatorFactory, and Move mainly by Biter. The instance view tells the programmer that most instances of GameNode are allocated at once. At one point in the execution, the number of instances of Move drops drastically, together with the numbers of instances of other classes. This is due to garbage collection.

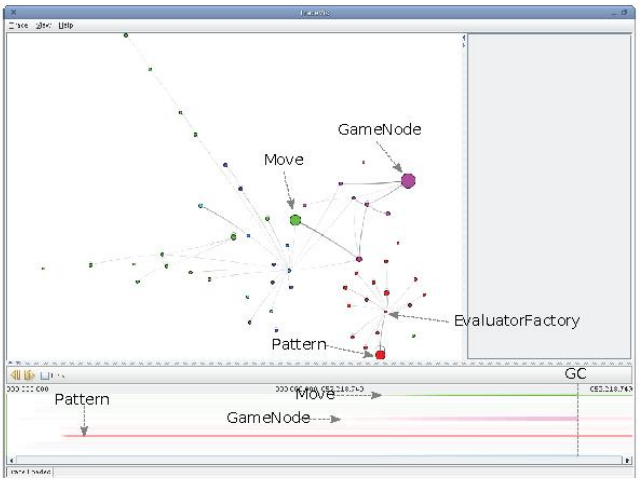


**Figure 9. TraceVis showing allocations.**

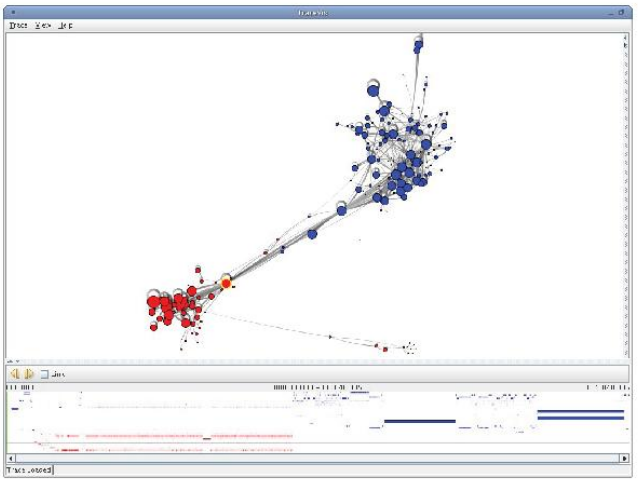

**Figure 10. An execution trace from Ant. The structural view shows two clusters: left the Xerces XML library, right Ant's code. The time line shows Xerces activity at the beginning when Ant is reading its buildfile.**

### 7.2 Ant

Apache Ant is a software build tool. Ant's main input is an XML file containing task definitions. The trace used in this section results from running tasks for compiling a small Java project and creating its documentation. It contains classes from both the Xerces XML library [14] and Ant itself. The trace has 175 classes and over 250.000 events. Ant classes are colored darker (blue) than Xerces classes (red).

Figure 10 shows an overview of the trace. A few observations can be made. The separation between Xerces and Ant itself is strict, as can be determined by the presence of two clusters. A conclusion which can be drawn from this is that both Xerces and Ant are cohesive, and are loosely coupled. The XML library is active in the first half of the execution. This corresponds to the reading of the XML buildfile. Long activations can be seen in the second half of the execution. These correspond to the running of external tools, such as the Java compiler and the Javadoc tool. Another observation is that with 175 classes, the structural view becomes cluttered. This cluttering can be overcome by, for example, selecting a time range, adding color to interesting classes, and zooming in. Figure 11 shows such a configuration where task classes have been colored yellow.

## 8 Conclusions and Future Work

TraceVis can collect execution traces from Java programs. Furthermore, it can be used to visualize these traces and the class interactions they contain, by displaying a dynamic call graph and a timeline which can be customized for showing different aspects of the execution, like number of received calls and number of instances. The data can be filtered and restricted to a time subrange; views can be zoomed and further customized to allow detailed browsing of the trace. This combination of features is not found in other tools as described in [10, 9, 2].

There are still several ways to enhance TraceVis. For large programs some improvements in efficiency are needed. The scalability of the visualizations could be improved by using the package hierarchy to aggregate data to package level and use a tree cut on the hierarchy to selectively show package or class information. For the structural view, two methods that can achieve this are the matrix view of [12] and the hierarchical edge bundles of [8]. Finally, the timeline view can be improved by using a theme river visualization [7]: replacing the height of a class based on its activity over the whole of the execution by a height on a per time unit basis. The height of a class then replaces the saturation as cue for indicating activity and saturation can be used for other purposes.

## References

[1] Apache Ant. `http://ant.apache.org`.

[2] R. Bertuli, S. Ducasse, and M. Lanza. Run-time information visualization for understanding object-oriented systems. In *Proceedings of WOOR 2003 (4th International Workshop on Object-Oriented Reengineering)*, pages 10 – 19, 2003.

[3] P. Deelen. Visualization of dynamic program aspects. Master's thesis, Technische Universiteit Eindhoven, June 2006.

[4] P. Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

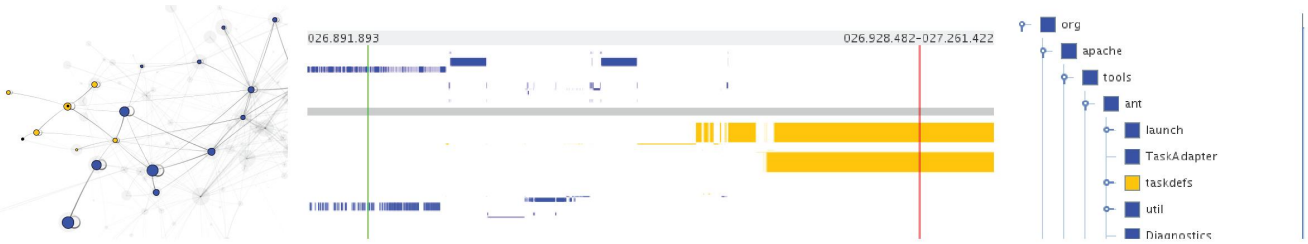


**Figure 11. View of Ant trace customized to show task classes (yellow) during javadoc generation.**

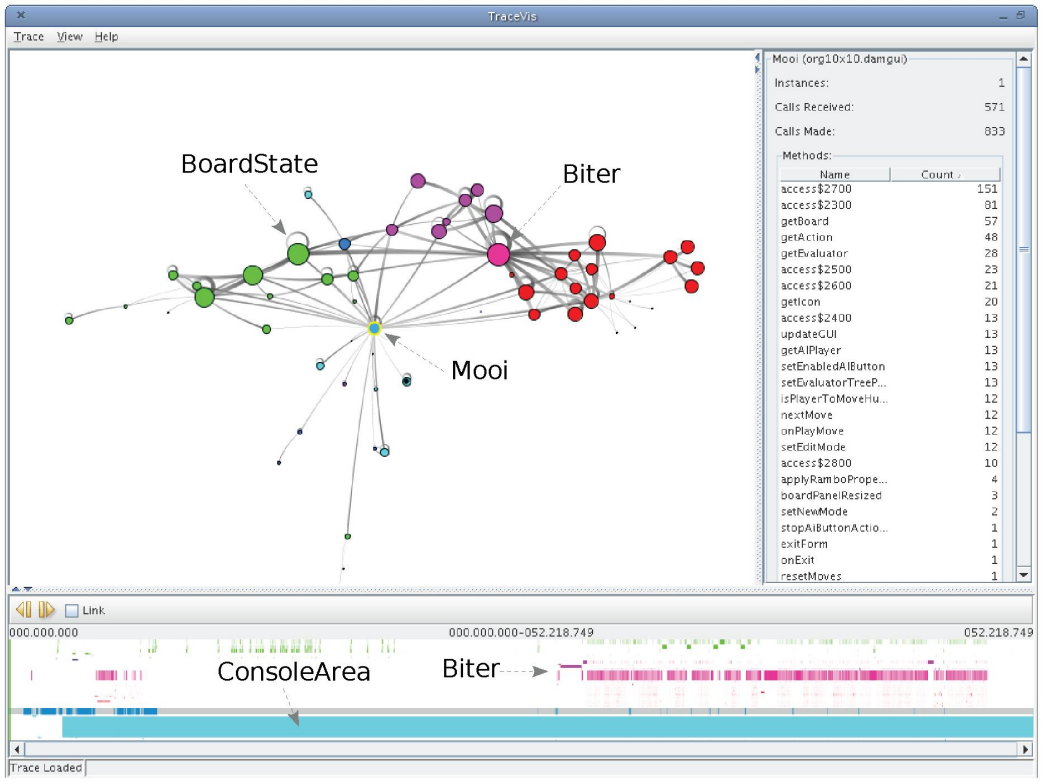


**Figure 12. An overview of an execution trace from Rambo.**


[5] M. Fowler. *UML Distilled, Third Edition: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, Boston, Mass., 2004.

[6] D. Harel and Y. Koren. Drawing graphs with non-uniform vertices. In *Proceedings of Working Conference on Advanced Visual Interfaces (AVI'02)*, pages 157–166. ACM Press, 2002.

[7] S. Havre, E. Hetzler, P. Whitney, and L. Nowell. Themeriver: Visualizing thematic changes in large document collections. *IEEE Transactions on Visualization and Computer Graphics*, 8(1):9–20, 2002.

[8] D. Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE Transactions on visualization and computer graphics*, 12(5):741–748, October 2006.

[9] W. D. Pauw, D. Kimelman, and J. M. Vlissides. Modeling object-oriented program execution. In *ECOOP '94: Proceedings of the 8th European Conference on Object-Oriented Programming*, pages 163–182, London, UK, 1994. Springer-Verlag.

[10] S. P. Reiss. Visualizing java in action. In S. Diehl, J. T. Stasko, and S. N. Spencer, editors, *SOFTVIS*, pages 57–65, 210. ACM, 2003.

[11] TraceVis. `http://www.win.tue.nl/~wstahw/tracevis`.

[12] F. van Ham. Using multilevel call matrices in large software projects. In *INFOVIS*. IEEE Computer Society, 2003.

[13] W. Wesselink and H. van de Wetering. Draughts site. `http://10x10.org`.

[14] Xerces XML parser. `http://xerces.apache.org`.