Automatic Visualization of Program Tasks in Code Bubbles

Steven P. Reiss and Alexander Tarvo
Department of Computer Science
Brown University
Providence, RI. 02912
{spr,alexta}@cs.brown.edu

***Abstract*—We provide a high-level, on-line visualization of the behavior of a complex, reactive Java program in terms that are familiar to the programmer. The visualization is part of the Code Bubbles integrated development environment. It is generated automatically by the environment without any effort by the developer. Code Bubbles performs static and dynamic analysis of the program. The dynamic analysis is done invisibly during debugging runs and is kept up to date as the program evolves. The analysis is used to determine the transactions and tasks describing the major phases of event processing by the program. Finally, Code Bubbles visualizes executions of transactions and tasks by the program's threads in real time as the program executes.**

Keywords—Software visualization, dynamic visualization, debugging, program understanding, integrated development environments.

I. Introduction

Modern software systems are large and complex. They are often reactive, responding to the external or internal events in an asynchronous manner. Responding to an event may involve multiple threads performing various computations, I/O activities, and synchronizations.

Such complex interactions make understanding the behavior of such programs very difficult. To better understand a program's dynamics, its internal activities must be evidenced. Existing approaches attempt to visualize the behavior of low-level events and primitives such as interactions between the program's threads. Such low-level visualizations can be difficult to interpret and the programmer is burdened with the task of mapping this visualization into the higher-level terms in which they conceive the system. To make the visualization understandable and meaningful to the programmer, the program behavior should be visualized using the high-level metaphors the system was designed with and the programmer thinks in terms of.

We visualize a program in terms of transactions and tasks. *Transactions* are major stages of processing a request by the program. They correspond to handling user interface events, network events, etc. *Tasks* are smaller processing steps that occur in a context of a transaction. They are initiated by the transactions and are processed later by the same or other threads. Where programs are purely interactive, tasks and transactions can be one and the same. However, in complex systems, a transaction can result in multiple tasks then being performed asynchronously by separate threads to service the transaction.

We visualize the execution of the transactions and tasks using a two-dimensional graph divided into horizontal rows. Each row represents a single thread of the program and colors within the row correspond to execution of particular transactions and tasks. The horizontal axis represents the flow of time. This is a relatively common notation, sometimes used to explain threading behavior, and akin to UML sequence diagrams and similar notations, that can be readily understood by programmers.

II. Related Work

There has been a lot of work on dynamic program understanding [8,30]. While a large number of tools have been built, few are actually used or incorporated into current integrated development environments. We see several reasons for this: most of the efforts to date involve off-line tools; the tools often provide low-level information rather than the application level information the programmer requires; the tools can require substantial work on the part of the programmer; and the tools have significant overhead that precludes their everyday use.

Off-line tools execute a program in a controlled environment while collecting trace data, analyze the collected data, and then create a visualization from the analysis. This approach has several advantages. First the tool can have access to a large amount of potential information by collecting a broad range of detailed trace data. Second, the displays and the underlying analysis can be more sophisticated since they don't have to be done in real time as the program executes. Third, once the run is complete, programmers can use the resultant visualization at their leisure, spending time appropriate to understanding their problem. An early industrial example of such a system is *Jinsight* [20,21]. Other examples include [9,11,14,15,17,37,38], our own work with Cacti [23] and BLOOM [24-26], and more recent systems such as SynchroViz [36], Verso [3], and SyncTrace [16],

Having worked on and with such tools, we have come to the conclusion that they are not going to work for many of the problems and programs that need to be addressed. There are several reasons for this. First, these tools tend to gener-

ate a significant amount of trace data. The cost of generating and storing this data slows the execution of the program considerably, making it difficult to analyze programs that are interactive, long running, or that need to be run in a production environment. Second, because the trace files are large, the analyses that need to be done can also take a significant amount of time; the cost of just getting to the point of being able to see the analysis is high and is discouraging to potential users. Third, the tools typically model the whole run and show the result after execution is over. This makes it difficult for the user to correlate a particular external event or abnormal behavior with the analysis or even to remember what was going on at a point where the analysis might look interesting. Fourth, the collection of large amounts of trace data tends to perturb the behavior of the program, making problems involving timing, threads, or process interaction difficult to reproduce.

What is required is an on-line dynamic analysis where the analysis runs along with the program and the program runs at close to its normal speed with minimal perturbation. Such tools require compromises since they are limited in terms of the data that can be displayed, the data that can be collected while still running the program at speed, and the types of analysis that can be performed on that data. However, the benefits of being able to correlate the output with what is currently going on, of being able to use the visualization on arbitrary programs, or being able to interact with the program based on the analysis, and of having relatively low analysis overhead, far outweigh the drawbacks. This is especially true if the visualization offers a history mechanism so that the user can go back to look at a particular event or visualized anomaly in more detail.

While such tools are less common, they do exist. We have been developing We developed early on-line tools to provide visualizations of execution in terms of classes, thread behavior, an estimation of the program phase, and line counts [27,28]. Performance visualization tools, such as our that integrated with Eclipse, offer another approach. More recently, tools such as ExplorViz [1], use tracing frameworks that can be made to work in an on-line mode [13].

Most dynamic understanding and visualization tools concentrate on providing the low-level information that characterizes the execution. Such information can include individual thread locks, function calls, allocations and deallocations, etc. Even tools that aim at higher level events, do not really address problems at the programmer's level. For example, recent server-related work such as the web services navigator [19] or streamsight [22] provide visualization of interactions but can't integrate these with other program information or specialize them based on program knowledge. Such tools is that they make it difficult for the programmer to get a high-level understanding of what the program is doing or to answer specific questions about the program [7].

Real program understanding comes from looking at the program at the same high level that the programmer thinks about the program. This lets programmers understand programs in their terms. It also lets them address specific, high-level questions they might have about program behavior.

There have been previous attempts to provide program-centric visualizations. Our original attempt, VELD, was off-line, using static analysis, full program tracing, and a variety of dynamic analyses [26]. A later attempt let the programmer specify behaviors in terms of program events and finite automata over these events [29], Follow-up work along these lines is described in the next section. Other systems that offer high-level views include *EVolve* [37,38] which provided an alternative framework where the programmer defined a specific visualization from different analyses, and the *Tracer* system provides visualizations of UML-based program models [18].

A problem with these tools is that they can require substantial work on the part of the programmer. Having the programmer define the model to visualize as is done in EVolve and VELD is more than one can expect. Alternatively, forcing the programmer to define the significant events as is the basis for tracing systems like Kieker [13] or X-Trace [10] is similarly a lot to ask. This is especially true in that the visualization may or may not solve the particular problem the programmer is interested in and the model will have to evolve as the code changes, so it might only be useful once.

Finally, a problem with most of these dynamic visualizations tools is that they impose significant overhead when running the code. Using the Eclipse profiler, for example, often slows the program down so much that interactive applications become unusable. We have addressed this problem for performance analysis in [31] by using very low overhead techniques combined with dynamic monitoring capabilities. Others have addressed this by using program analysis to minimize monitoring [2] or by turning instrumentation on and off appropriately [12].

III. Prior Work

Our goal was to incorporate a useful dynamic visualization that provided high-level information about program execution into a development environment. The tool would need minimum input from the programmer and would have low enough overhead so that it could be running continuously.

We concentrated on understanding complex, reactive systems. In such systems, the user is mainly interested in what are the primary operations or transactions, how are they broken down into logical tasks, and how are those tasks being handled by the program using threads.

We designed and implemented the system based on our recent work in both dynamic visualizations and on development environments.

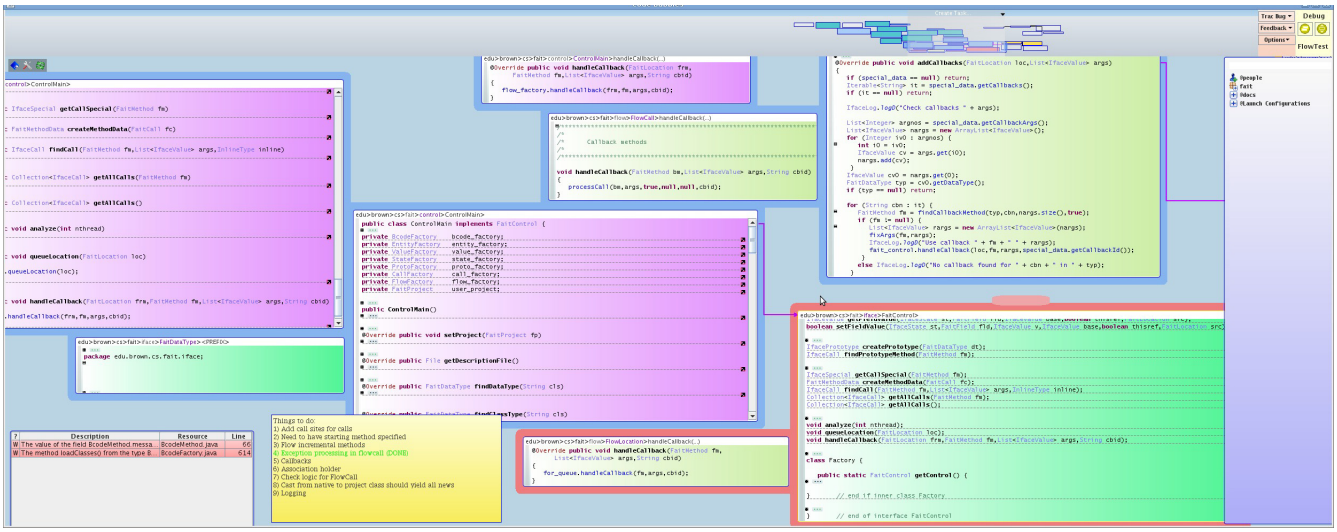

Figure 1. The Code Bubbles Integrated Development Environment.

A. DYVIEW

Our prior work on visualizing threads, tasks and transactions, DYVIEW, required significant effort on the part of the developer [34]. The developer had to specify the project to be analyzed; then he had to initiate or update a static analysis of the project; then he had to run the program in a representative way while the system did a dynamic analysis. The system would then use the results of the static and dynamic analysis to determine candidate classes representing transactions, tasks, and threads. The programmer, however, would have to choose among these candidates. Next he had to associate program values (e.g. the task class, the thread class) with the graphical properties of the visualization (although the system choose appropriate defaults). Finally he had to run the program again and attach the visualization to it dynamically. While visualizations done this way were accurate and highly customizable, the considerable effort was a strong deterrent to the use of the system.

The visualization this system provided was similar to the one used here and was used as a starting point for our new visualizations.

B. Code Bubbles

Code Bubbles is a novel integrated development environment for Java. In contrast to the existing IDEs that represent the project as a collection of source files, Code Bubbles organizes the project as a working set [4,5]. A working set is a collection of *task-relevant fragments*, such as methods, small classes, test cases, notes, documentation pieces, etc. Each fragment is displayed in a separate lightweight window — a *bubble*. The bubbles can be related to each other. For example, a note can be attached to a particular method, or a test case can be associated with the bug report. The user can rearrange the bubbles as needed to provide a logical context for the particular maintenance or development task they are currently working on. A view of Code Bubbles can be seen in Figure 1.

Code Bubbles offers a wide range of features to support different styles of development. To help the programmer to simultaneously work on multiple tasks, Code Bubbles provides a large overview space in which the user can embed multiple working sets. To support creation of the working set, the environment provides a number of navigation aids.

Code Bubbles provides a range of debugging features. In addition to the normal notions of breakpoints and stepping, Code Bubbles lets the user see multiple debugging sessions in parallel (including a history of previous ones), and displays bubbles for each level of the execution stack. The environment provides a history view of the debugging session, a low-overhead performance analysis of the program being debugged, and the ability to delve into Swing/AWT hierarchies by pointing to a pixel in the program's output. A debugging view of Code Bubbles can be seen in Figure 2.

Support for debugging is provided both through Eclipse and by a separate debugging agent, BandAid, that is automatically included by Code Bubbles. This agent communicates with Code Bubbles through a intermediate server, sending XML data back to Code Bubbles periodically, and accepting a small set of commands. It is organized into individual agents, each of which is responsible for a particular feature. One agent, for example, does periodic stack sampling get statistical performance information, accumulates this, and sends back performance reports every few seconds. Another agent tracks the origin of Swing and AWT widgets and accepts commands to determine how a particular pixel is drawn and what widgets it occurs in. A third agent tracks the state of each thread, using stack samples to determine if the thread is running, sleeping, waiting, blocked, or doing I/O. A fourth agent checks for and reports thread deadlocks.

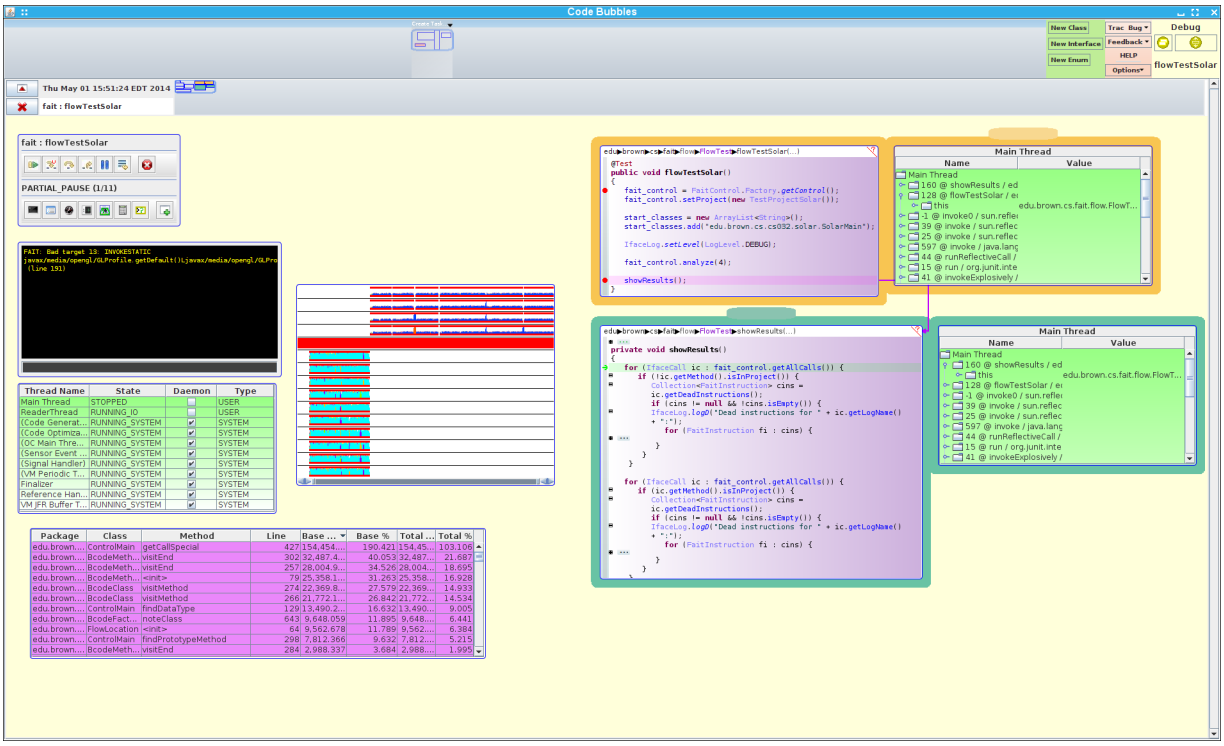

Figure 2. A debugging view of Code Bubbles. The window in the upper left is the debugger control panel. Below that is the console window, a window showing the status of each thread, and a performance table. To the right of the console is the task visualization window. The Windows on the right half of the display are text editors and stack viewers showing the current execution.

Our new visualization was designed to be part of this Code Bubbles debugging facilities, making use of its own agents. It was designed to provide the types of visualizations offered by DYVISE, but without any of the costs. It was also designed to have low enough overhead so it can be used at any time.

The ability to visualize program transactions and tasks is an important addition to this range of intelligent debugging facilities. The visualization tool is designed to provide the user with an understanding of what is happening in the system being debugged in high-level terms, i.e. in terms of the program itself, rather than in terms of low-level events.

IV. Visualizing Program Tasks

A sample visualization is shown in Figure 3. The visualization uses a swim-lane metaphor to show the different threads. Each thread is shown in a separate row (lane) of the visualization. Within the row, the visualization draws a pipe. The outside of the pipe is colored to represent the transaction and the inside of the pipe is colored to represent the task that is part of the transaction.

Time is represented along the X-axis. Since each pixel can represent multiple events, the transaction (outside) coloring represents the transaction that occupies the majority of the corresponding interval, while the task (inside) coloring is a stack of pixels where the height of each color represents the proportion of the time spent by that task. White is used here to indicate the thread was not processing any known task.

The visualization is on-line and is updated as the program runs. This lets the programmer understand what is happening in the program as it happens, easily correlating outside events with program activity.

Figure 3 represents the execution of a single test request by our semantic search tool [33]. The processing can be broken up into eight distinct phases, most of which can be seen in the visualization. The first phase involves setting up the request. This is done by the main thread at the bottom, and is the slight yellow area at the left. The second phase involves retrieving the first page of search results from the search engine. This is done by one of the threads in the thread pool. Once the first search result page is returned, the system uses the thread pool to retrieve the next nine pages of search results and all the files referenced in all ten pages. This is done in parallel using 32 threads as seen in the visualization. The yellowing areas here represent retrieving results pages, while the green areas represent retrieving and converting the retrieved files into candidate solutions.

After all the pages are retrieved, the system takes each candidate solution and attempts to transform it into a new solution that is more likely to be acceptable. The resultant solutions are also transformed. This phase is done using 4 threads in the thread pool and is the blue area with internal purple and red tasks (representing different parts of the transformation process). After no more solutions can be found, the system does a dependency analysis on the remaining solutions to add outside methods and fields and eliminate any that won't compile. This is again done with 4

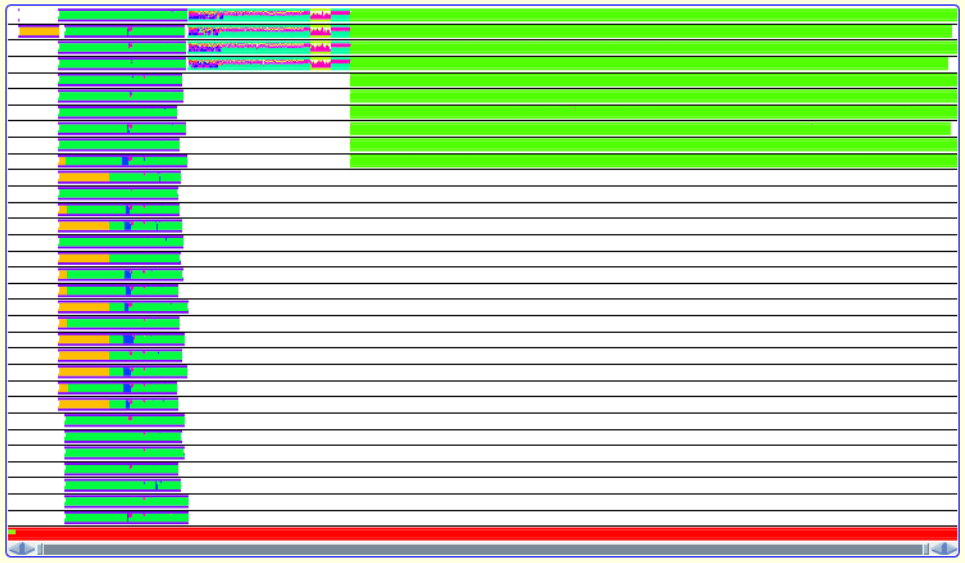

Figure 3. The Transaction-Task Visualization. Each row of the visualization corresponds to a user thread. Colors indicate the transaction and task the thread is working on. Time is represented along the X-axis.

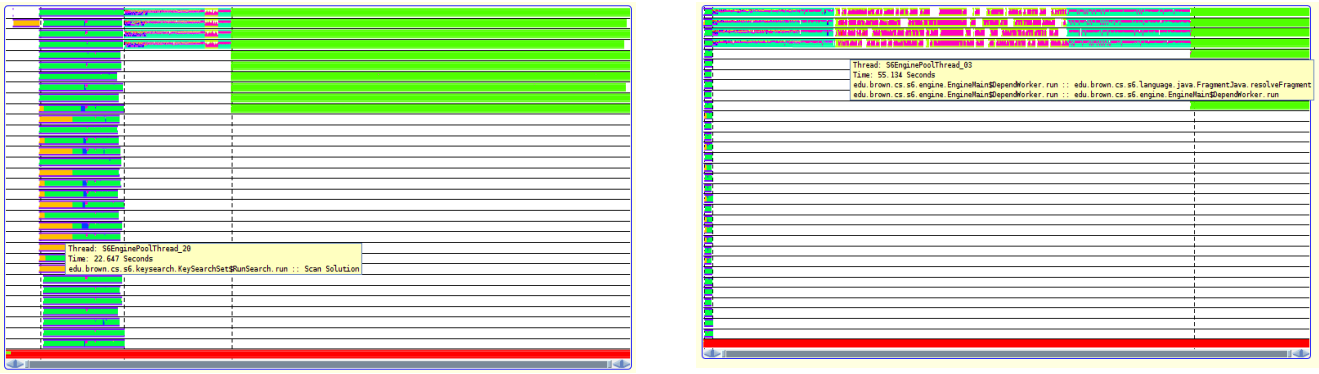

Figure 4. Two additional views of the visualization shown in Figure 3. The left view shows both time marks as vertical dashed lines and an example tool tip. The right hand view shows the result of focusing one part of the visualization using fish-eying.

threads and can be seen in the small yellow/red area. After this is done, the system does a few additional transformations to get the remaining solutions ready to test.

The system next uses 10 threads to run the resultant solutions using junit and ant. This is the part of the process that takes up the most time and is seen in the visualization as the long green rows on the upper right. The last part of the process involves outputting the solutions that passed. This is relatively quick and is done by the main thread so it is difficult to see in the visualization.

This interpretation of the visualization is not directly obvious from the diagram. Indeed, attempting to place all this information on the figure would over complicate the diagram and make it incomprehensible. Instead, we rely on two things. The first is that, since the visualization is part of a development environment, the programmer is probably familiar with the system and its expected behavior, can make a good guess at what the system is doing, and hence has a good chance of interpreting the visualization directly.

Second. we provide tool tips so that the programmer can mouse over the visualization and see exactly what is happening at any time. Such tool tips can be see in Figure 4. The tool tips show the time, the task and the set of transactions that were active during the interval represented by the corresponding pixel. Transactions and tasks are identified by their routine name.

The user can manipulate and interact with the visualization in various ways. First, they can change the tool tip labels, replacing a routine name with their own label for a particular transaction or task. This name is remembered and used in subsequent visualizations.

Second, they can set the color of a particular task or transaction explicitly. Normally the system will assign colors to the transactions or tasks sequentially, picking colors that are as distinct from previous colors as possible.

(It uses hues in the sequence 0, 0.5, 0.25, 0.75, 0.125, 0.375, 0.625, 0.875, …). The user can select a task or transaction and assign it a color. Then when assigning colors to a new task or transaction, the system will skip any hue that is too close to a user selected color.

Third, the user can insert time markers. These are represented in the visualization as vertical dashed lines and can be seen in Figure 4. These can be inserted at a specific point or can be inserted at the current time during a run. This lets the user flag a particular point in the program where the something interesting is happening.

Finally, the user can focus on a particular point in the visualization. This can be done in two ways. First, the two-sided scroll bar at the bottom of the visualization lets the user specify a particular part of the visualization to look at. Second, the system supports fish-eying. By dragging with the left mouse button down, the system will ensure that the point under the mouse represents at most 10 milliseconds of execution and will smoothly compress the rest of the visualization. This can be seen in the right hand visualization of Figure 4 where the user has zoomed in on a point in the short dependency analysis phase of the program.

V. Behind the Scenes

While the visualization is interesting and informative, the important point is that it is generated automatically, with no input from the programmer, and with low-enough overhead so that it can be used all the time. Code Bubbles will automatically collect the data needed for the visualization during every debugging run of the program. The user can ask for the visualization at any point during (or after) the run and see what happened or what is going to happen. If the visualization bubble is already open, a new run will clear it and reuse it.

The visualization is generated by identifying and tracing the entry and exit of a select set of routines in the application. Most of these routines are the program's *event handlers*. An event handler is a routine that is called when an event occurs. What is an event can vary widely depending on the program. It can be a message or request from external source such as a web browser; it can be a task given to a thread queue; it can be a user interface event; it can be a timer event.

The basic visualization is generated from a trace of all the (non-nested) event handlers in a system. Transactions are defined as event handlers that are invoked by the system without a correspondence to a prior transaction. Tasks are defined as event handlers that are invoked in the context of a previously invoked transaction and share some data with that transaction. Whenever an event handler is entered, Code Bubbles records the entering thread, the identity of the event handler, the entry timestamp, and the associated set of objects. When an event handler is exited, the tool records the thread, the handler identity, and the exit timestamp.

In order to associate tasks with prior transactions, the system assumes that the transaction (or a prior task) will create an object representing the work to be done by the new task. This might be a *Runnable* placed on thread queue or a handler associated with an event. The system identifies the types of such objects and records all allocations of them. If the allocation occurs within a transaction context, any task that later uses that object will be associated with that transaction.

More details are added to the visualization by identifying *key routines*. These are routines in the program in which the system spends a significant amount of time (currently defined as above 5% of the total and above a threshold of about 1 second). Tasks that take a long time can then be broken down by what they are doing. Key routines are viewed as tasks in themselves for visualization purposes even though they are actually nested in tasks or other key routines.

A. Computing What to Trace

Generating a visualization this way requires knowledge of a) the set of event handlers; b) the set of relevant object types referenced by these handlers; c) the set of key routines; and d) locations in the program's code that should be instrumented to gather the necessary information with minimal overhead. All this information is determined without programmer intervention. The visualization also requires that the necessary information be acquired efficiently and that it be kept at the proper level of detail so it can be stored and displayed in real time.

This information is gathered automatically using a combination of static and dynamic analysis. First, the system uses the smart debugging facilities of Code Bubbles, by adding a BandAid agent to look at stack samples. This agent builds a call trie that represents a summary of the execution. Each call stack is added to the trie by starting with its start node and working down to the current active routine. Where the stack entry corresponds to an existing trie node, that node is reused. The agent keeps counts for each node of the trie of the number of times that node was actively executing. Leaf nodes of the call stack that represent waits or I/O operations are ignored and separate counts are kept for each trie node of the number of times it was executing, waiting, and doing I/O. Because the system is designed to be used during debugging, stack samples that correspond to routines that are stopped by the debugger are ignored completely. The tries are periodically sent back to Code Bubbles incrementally and the final trie is analyzed when the debugging run is ended. This dynamic analysis has low enough overhead so that it is essentially invisible to the user while debugging and can be used all the time.

The system defines an event handler as a user routine that is called from system code, as in our prior work [32]. These are found by finding trie nodes where the caller is a

system routine and the callee is a user routine. The actual computation is a bit more sophisticated.

First, the systems we are looking for are often complex, using both internal an external libraries as well as their own registration and callback methods. We use the static analysis facilities of Code Bubbles to build a hierarchy of packages, where system routines are at the lowest level of the hierarchy, and routines at a higher level only directly use methods from packages at their level or below. Such hierarchies often exist even within the application's code (which we consider user code), since a) the application is often built on top of user-defined libraries; b) hierarchicalizing the code can yield a cleaner implementation; and c) packages are often set up so they can be separately compiled, which requires that their be a compilation order and hence hierarchy. The static analysis looks at each file to find references to other packages and builds the corresponding hierarchy.

Then we extend to the definition of an event handler to be any invocation of a method in the call trie where the caller is at a lower level of the hierarchy than the callee. One complication is that there are routines that introduce false event handlers, for example *toString*. These are special cased. A second complication is that we only want to consider user routines as event handlers. Here we ensure that the routine is not a special Java-defined routine (such as an accessor), that the routine has associated source within the user's project as known to Code Bubbles, and that the routine has either public or protected access (only public for constructors). Next, the main entry point is automatically considered as a callback if it is part of the user's source. A further complication in defining event handlers, especially with this broader definition, is that on any path from the root to a leaf in the call trie there might involve multiple event handlers. The system currently only considers the first such handler on each path.

Key routines are also identified automatically from the returned call trie when the count totals are significant. We first compute, for each trie node, the total of its local counts and the total counts of each of its children. Then we consider any trie node that is below an event handler node, that has a total count of greater than 5% of the total and where either it has no children satisfying this condition.

The set of relevant object types is found directly from the set of event handlers. In validating an event handler, we find its source code. This allows us to identify its argument types. (Stack sampling within the Java management framework returns routine names and line numbers, not the fully qualified routine with arguments.) We consider all the arguments (the *this* parameter included) and find up to two that are object types defined in user code. The types of these parameters added to the set of relevant object types and the parameter ids are saved along with the event handler. For each relevant object type, we identify the constructors that need to be instrumented.

This approach differs from our original approach in several ways. First we display all transactions and tasks rather than having the programmer define what should be displayed. Second, we use a simplified model of what can be a transaction or task, omitting the levels of indirection that we used originally. This also lets us use a simplified static analysis. Finally, we substitute multiple debugging runs for a single exemplar run.

B. Computing the Trace

To gather the actual trace information, we use another agent as part of the Code Bubbles debugging facility. This agent takes a list of all the routines to be traced from Code Bubbles along with the type of tracing required. This agent registers with the Java Management facility to potentially instrument classes as they are loaded. For event handlers and key routines, it patches the corresponding classes to add a trace entry point at both the start and each exit of that routine. For each constructor of a relevant class, it instruments the constructor with a trace entry point called after any super constructor is called. This is done using *asm* [6], Trace events are handled separately for each thread to avoid introducing false synchronizations. The trace collection code itself is run in its own thread, using multiple buffers to minimize interactions with the program's threads. Each trace entry includes an integer identifying the routine (negative for exits), the time (in nanoseconds) the entry occurred, and any relevant object parameters. The later are either identified when computing the set of relevant object types or are the this parameter in a constructor.

Because the number of top-level event handler calls and key routine calls is relatively small as is the relevant set of object types, the overhead caused by the monitoring is generally not noticeable in running the program. The trace information is periodically (every half-second or so) passed back to Code Bubbles where it is analyzed and converted into a form more suitable for visualization, essentially a time-ordered set of events to visualize where each event represents a task and has a pointer to the corresponding transaction.

This set is constructed by processing the trace events in temporal order and tracking the current tasks associated with each thread. Tasks for a thread are kept on a stack for that thread so that calls and returns can be tracked appropriately. Nested event handlers generally do not generate new tasks. However, a nested key routine will end the current task (generally one associated with an event handler) and start a new one when it is entered. When it is exited, this task ends and another instance of the prior task is started. To prevent the set of events from becoming overwhelming, which would make it costly to store and to visualize, very short tasks are either ignored (if they are short because of nesting), or are merged (if they are short but consecutive). Tasks that have started but not yet finished are considered to continue up until the current execution time.

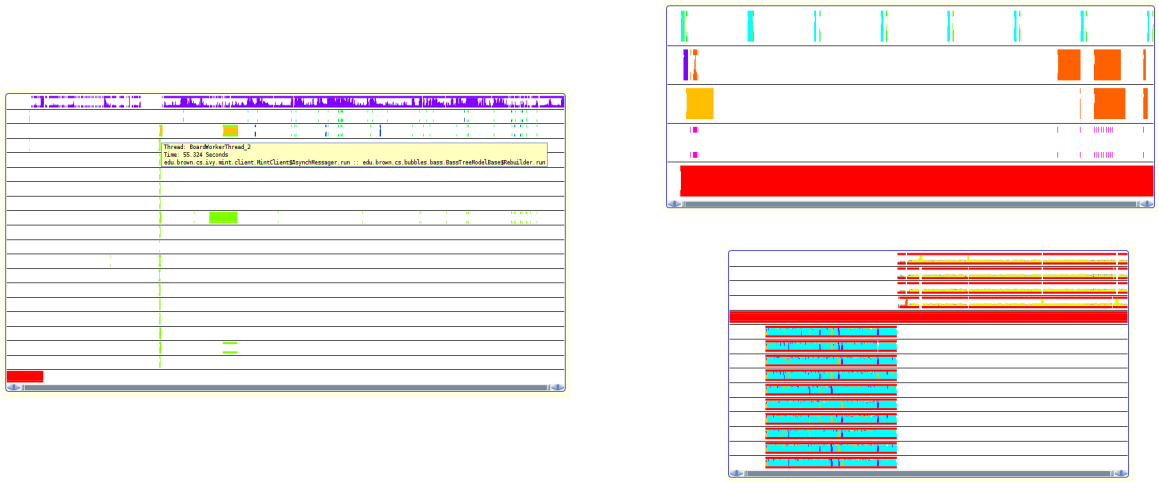

Figure 5. Three different task visualizations. The one on the left Code Bubbles running as a front end to code search. The top thread is the user interface thread. The bottom line is the main program setting up and initializing the system. The middle lines represent the various utility tasks done in background such as reformatting, rebuilding the search tree, and handling edits. The visualization at the upper right is a web-server/home automation engine. The top thread is the timer which periodically checks the state of various sensors. The bottom thread is the main program which is waiting to exit. The middle threads represent the thread pool. The yellow area on the left is updating a device. The orange areas on the left are handling web requests. Finally, the visualization at the bottom right is a multithreaded Java flow analyzer. The blue area on the left is a thread pool used for loading the various classes. The yellow/red area on the upper right is a separate thread pool doing the actual flow analysis.

After the set of events has been updated based on information obtained by the trace agent, the corresponding visualization windows will be notified so they can update their display as necessary.

While the system works completely without programmer intervention, we do provide a small set of options to let the programmer customize the display based on their requirements. These options affect how the trace is generated and processed, so they generally only take effect on the next run. The first option is whether to include key routines as tasks or not. The default is to include them. The second option is whether the main routine should be considered as a task. This is optional since for some reactive systems, the main program just handles initialization and can be ignored. The default is to consider it. The final option currently is whether the main program should be considered a transaction. If it is, then almost all callbacks are going to be associated with it since initialization typically is used to define the callbacks. The default is to not consider it a transaction.

C. *Validating the Set of Events*

Our system uses information gleamed from all prior debugging runs to determine what to instrument in order to generate the trace in the next debugging run. This is somewhat complicated in that the system being traced is being worked on within a development environment, and hence should be considered as under active development. This means that the information needs to be maintained as the system evolves. Such evolution can occur both within the environment and outside of it (as programmers can and sometimes do change code outside the development environment).

To accommodate this, Code Bubbles does several validations. First, whenever the set of callbacks (event handlers, key routines, and constructors for relevant classes) changes, the system will write these out in a file within the project. When Code Bubbles starts up, it reads this file and then validates each routine, ensuring that it still exists within the system with the same parameters, access, etc. Any routine that does not validate correctly is discarded. Second, it only considers routines as callbacks if it can actually find that routine in the current user source. Third, if does not assume that each routine has to exist on every run. The instrumentation package is designed to be error tolerant. The system also has the capability to validate all the routines each time the system is updated (i.e. files are saved), but this has proven to be too expensive with the current implementation of Eclipse.

The system also gives the programmer the option of discarding all the previously selected event handlers, key routines, etc. and starting the computation over based on future debugging runs.

VI. Experience

The task visualization functionality is part of the running and distributed Code Bubbles environment. As such, we have used it to look at (and to some extent understand) the behavior of several of the systems we have been working on. Examples of three systems (beside S^6 which was covered previously) are shown in Figure 5.

The first example involves Code Bubbles itself (although run as a front end for code search rather than as a Java environment [35]). Here one can clearly see the initial-

ization phase, and the fact that the user interface thread (purple on top), dominates. Moreover, one can estimate how busy that thread is by the amount of white space (versus colored space) within its display. The thread pool used by Code Bubbles as seen here is used occasionally, mainly when loading the files that come back from code search, when putting up new code bubbles on the files (the yellow tasks), and when formatting after editing (the purple tasks).

The second example (upper right) is from a system that manages an automatic sign outside Dr. Reiss's office. This system takes input from various sensors to determine if he is in the office, if he has a visitor, or if he is on the phone, and also consults his electronic calendar. There is a program that determines, based on the time and these conditions, what should be displayed on the sign.The system also supports a web interface to define, view, and edit this program. The visualization clearly shows the periodic, time-based updates at the top, the of time required to update the sign (yellow on the middle left), and the time needed to handle web requests for programming (orange on the right).

The third example is from a computationally intense application that computes control flow of Java systems. The visualization clearly shows that two distinct thread pools are used, one to start with which is charged with loading all the necessary classes into an internal representation; and one that is used to actually compute the control flow. From the visualization, it appears that the actual computation could be more efficient in that the threads are idle a considerable part of the time.

These examples show that the visualization is applicable and practical for a wide variety of applications, ranging from simple interactive ones to highly computational ones. It can be used to provide an understanding of what is happening in the system and to point out potential problems (such as the idle times in the control flow threads) or unexpected behavior (such as the fact that the testing phase of S^6 is the longest while the transformation phase that we were worried about, is generally short). It also shows that the visualization finds the relevant event handlers and does a reasonable job of identifying key routines and associating tasks with transactions.

They also demonstrate that our solution is practical. We have been using Code Bubbles extensively both for code development, testing, and debugging. Yet, although the system has been gathering the appropriate data on every debugging run, we have not noticed any slowdown or interference with our development.

While our experience with the system has been mainly positive, we have noted a number of aspects that may require additional work. These include

- Validation of existing methods and classes is somewhat expensive. Right now we only do it when Code Bubbles starts up. However it should be done each time the user saves a file or before each debugging run.
- The set of event handlers and hence task and transaction classes is only used to instrument the application at the start of a run. It should be possible to discover new event handlers and reinstrument the code on the fly.
- The visualization often illustrates thread pools or groups. This is accidental in that threads are sorted by name and thread pools tend to have similar names. There should be a way of making use of this information or of displaying thread pools as one swim lane for very complex applications.
- Similarly, the user might only be interested in a subset of the threads. In this case it would be helpful if the visualization let the user identify which threads were relevant and which should be ignored.
- For many systems, the callbacks are all defined initially as part of initialization and are handled in one thread at one time. For such systems tasks and transactions are the same thing. In this case there might be a better way of displaying the tasks. Moreover, if we know this, we can do less instrumentation and thus incur less overhead.
- Right now we are displaying everything that the system identifies as an event handler or key routine. The programmer may find some of these irrelevant or want to concentrate on a particular set of items. Thus, it might be useful to let the user select routines and indicate that they should be ignored.
- The system currently generates colors dynamically on each individual run, assigning colors to tasks (and hence transactions) in the temporal order they occur. Because the systems are nondeterministic, this can yield different color sets on different runs, which can be somewhat disconcerting. The system could avoid this, for example, by remembering the color of each item from its prior run.
- While we haven't noticed the overhead when using Code Bubbles, we know it is there and there might be ways of reducing it. In particular, we know that getting the time is a costly operation in current Java implementations.
- Some applications can involve a large number of short-lived calls to event handlers or key routines (e.g. if the key routine is expensive because it is called a lot rather than because it does extensive computation). Right now, simplifying the trace by eliminating or merging these is done while processing the trace. It would be more efficient to do this processing as the trace is generated.

VII. Conclusion

In this paper we have described an extension to Code Bubbles that provides a high-level visualization of program behavior without any programmer input using low-overhead techniques so that the visualization can be available at any time.

The Code Bubbles environment, including the task visualization plug-in, is available in both source and binary form from *http://www.cs.brown.edu/people/spr/codebubbles*. A

video of the system is available at *http://www.cs.brown.edu/video/37*.

VIII. Acknowledgement

This work was done with support from the National Science Foundation through grants CCR1012056 and support from Microsoft.

IX. References

1.
2. Thomas Ball and James R. Larus, "Optimally profiling and tracing programs," *19th ACM Symposium on Principles of Programming Languages*, pp. 59-70 (January 1992).
3. Omar Benomar, Houari Sahraoui, and Pierre Poulin, "Visualizing software dynamicities with heat maps," *Proceedings of First IEEE Working Conference of Software Visualization*, (2013).
4. Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr., "Code bubbles: rethinking the user interface paradigm of integrated development environments," *ACM/IEEE International Conference on Software Engineering 2010*, pp. 455-464 (2010).
5. Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr., "Code bubbles: a working set-based interface for code understanding and maintenance," *Proceedings SIGCHI Conference on Human Factors in Computing Systems*, pp. 2503-2512 (2010).
6. E. Bruneton, R. Lenglet, and T. Coupaye, "Asm: a code manipulation tool to implement adaptable systems," *Adaptable and Extensible Component Systems, http://www.objectweb.org/asm/current/asm-eng.pdf*, (November 2002).
7. Stuart M. Charters, Nigel Thomas, and Malcolm Munro, "The end of the line for software visualization?," *Proceedings 2nd VISSOFT*, (September 2003).
8. Bas Cornelissen, Andy Zaidman, Arie van Deursen, Leon Moonen, and Rainer Koschke, "A systematic survey of program comprehension through dynamic analysis.," *Technical Report TUD-SERG-2008-033, Delft University of Technology*, (2008).
9. Mikhali Dmitriev, "Design of JFluid: A profiling technology and tool based on dynamic bytecode instrumentation," *Sun Microsystems Report TR_2003-125*, (November 2003).
10. Rodrigo Fonseca, Goerge Porter, Randy H. Katz, Scott Shenker, and Ion Stoica, "X-Trace: a pervasive network tracing framework," *Proceedings 4th USENIX Symposium on Networked Systems Design and Implementation*, pp. 271-284 (2007).
11. Paul Gestwicki and Bharat Jayaraman, "Methodology and Architecture of JIVE," *SoftVis 2005*, pp. 95-104 (May 2005).
12. Martin Hirzel and Trishul M. Chilimbi, "Bursty tracing: a framework for low-overhead temporal profiling," *4th Workshop on Feedback-Directed and Dynamic Optimization*, (December 2001).
13. Andre van Hoorn, Jan Waller, and Wilhelm Hasselbring, "Kieker: a framework for application performance monitoring and dynamic software analysis," *3rd International Conference on Performance Engineering 2012*, pp. 247-248 (2012).
14. Dean Jerding, John T. Stasko, and Thomas Ball, "Visualizing interactions in program executions," *Proceedings 19th International Conference on Software Engineering*, pp. 360-370 (May 1997).
15. Dean F. Jerding, "Visualizing patterns in the execution of object-oriented programs," pp. 47-48 in *Proceedings of SIGCHI Conference on Human Factors in Computing Systems*, (1996).
16. Benjamin Karran, Jonas Trumper, and Jurgen Dollner, "SyncTrace: visual thread-interplay analysis," *Proceedings of First IEEE Working Conference of Software Visualization*, (2013).
17. Eileen Kraemer, "Visualizing concurrent programs," pp. 237-256 in *Software Visualization: Programming as a Multimedia Experience*, ed. John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, MIT Press (1998).
18. Shahar Maoz, Asaf Kleinbort, and David Harel, "Towards trace visualization and exploration for reactive systems," *IEEE Symposium on Visual Languages and Human- Centric Computing 07*, pp. 153-156 (2007).
19. W. De Pauw, M. Lei, E. Pring, L. Villard, M. Arnold, and J. F. Morar, "Web services navigator: visualizing the execution of web services," *IBM Systems Journal* ***44***(4) pp. 821-845 (2005).
20. Wim De Pauw, Doug Kimelman, and John Vlissides, "Visualizing object-oriented software execution," pp. 329-346 in *Software Visualization: Programming as a Multimedia Experience*, ed. John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, MIT Press (1998).
21. Wim De Pauw, Nick Mitchell, Martin Robillard, Gary Sevitsky, and Harini Srinivasan, "Drive-by analysis of running programs," *Proceedings International Conference on Software Engineering Workshop of Software Visualization*, (May 2001).
22. Wim De Pauw, Henrique Andrade, and Lisa Amini, "Streamsight - a visualization tool for large-scale streaming applications," *Softvis 2008*, pp. 125-134 (September 2008).
23. Steven P. Reiss, "An engine for the 3D visualization of program information," *Journal of Visual Languages*, (December, 1995).
24. Steven P. Reiss and Manos Renieris, "Encoding program executions," *Proceedings International Conference on Software Engineering 2001*, (May 2001).
25. Steven P. Reiss, "Bee/Hive: a software visualization backend," *IEEE Workshop on Software Visualization*, (May 2001).
26. Steven P. Reiss, "An overview of BLOOM," *PASTE ,01*, (June 2001).
27. Steven P. Reiss, "JIVE: visualizing Java in action," *Proceedings International Conference on Software Engineering 2003*, pp. 820-821 (May 2003).
28. Steven P. Reiss and Manos Renieris, "JOVE: Java as it happens," *Proceedings SoftVis ,05*, pp. 115-124 (May 2005).
29. Steven P. Reiss, "Visualizing program execution using user abstractions," *SOFTVIS 06*, pp. 125-134 (September 2006).
30. Steven P. Reiss, "Visual representations of executing programs," *Journal of Visual Languages and Computing* ***18***(2) pp. 126-148 (2007).
31. Steven P. Reiss, "Controlled dynamic performance analysis," *Proceedings 2nd International Workshop on Software and Performance*, (June 2008).
32. Steven P. Reiss, "Dynamic detection of event handlers," *Proceedings WODA 2008*, (July 2008).
33. Steven P. Reiss, "Semantics-based code search," *International Conference on Software Engineering 2009*, pp. 243-253 (May 2009).
34. Steven P. Reiss and Suman Karumuri, "Visualizing threads, transactions, and tasks," *PASTE 2010*, pp. 9-16 (June 2010).
35. Steven P. Reiss, "Browsing software repositories," *Unpublished manuscript submitted for publication; available at http://www.cs.brown.edu/people/spr/rebuspaper.pdf*, (2014).
36. Jan Waller, Christian Wulf, Florian Fittkau, PHilipp Dohring, and Wilhelm Hasselbring, "Synchrovis: 3D visualization of monitoring traces in the city metaphor for analyzing concurrency.," *Proceedings of First IEEE Working Conference of Software Visualization*, (2013).
37. Qin Wang, Wei Wang, Rhodes Brown, Karel Driesen, Bruno Dufour, Laurie Hendren, and Clark Verbrugge, "EVolve: an open extensible software visualization framework," *Proceedings of SoftVis 2003*, (June 2003).
38. Wei Wang, "EVolve: An extensible software visualization framework," *McGill University School of Computer Science*, (2004).