2010 17th Working Conference on Reverse Engineering

Immediacy through Interactivity: Online Analysis of Run-time Behavior

Michael Perscheid, Bastian Steinert, Robert Hirschfeld, Felix Geller, and Michael Haupt
Software Architecture Group
Hasso Plattner Institute
University of Potsdam, Germany
Email: {firstname.lastname}@hpi.uni-potsdam.de

***Abstract*—Visualizations of actual run-time data support the comprehension of programs, like examples support the explanation of abstract concepts and principles. Unfortunately, the required run-time analysis is often associated with an inconvenient overhead that renders current tools impractical for frequent use.**

We propose an interactive approach to collect and present run-time data. An initial shallow analysis provides for immediate access to visualizations of run-time information. As users explore this information, it is incrementally refined on-demand. We present an implementation that realizes our proposed approach and enables developers to instantly explore run-time behavior of selected code entities. We evaluate our interactive approach by measuring time and memory overhead in the context of ten different-sized projects. Our empirical results show that run-time data for an initial overview can be collected in less than 300 milliseconds for 95% of cases.

***Keywords*-Program comprehension; dynamic analysis; object collaboration; test cases; development environments**

I. Introduction

Developers of object-oriented software systems spend a significant amount of time on program comprehension [1]–[3]. They require an in-depth understanding of the code base that they work on; ranging from the intended use of an interface to the collaboration of objects, and the effect of a method activation during this collaboration. Gaining an understanding of a program by reading source code alone is difficult as it is inherently abstract.

The visualization of run-time information supports the comprehension of programs. Collected run-time data reports on the effects of source code and thus helps understanding it. At run-time, the abstract gets concrete; variables refer to concrete objects and messages get bound to concrete methods. For example, profilers and back-in-time debuggers support exploration of a program's run-time to answer questions such as: "What is the value of a particular method argument?" or "How does the value of a variable change?"

Unfortunately, the overhead imposed by current tools renders them impractical for frequent use. We argue that this is mainly due to two issues, which we discuss in more detail in Section II: a) Setting up an analysis tool usually requires a significant configuration effort, as well as a context switch, b) performing the required in-depth analysis is time-consuming. Both issues inhibit immediacy and thus discourage developers from using these tools frequently.

We argue that the overhead imposed by current approaches to dynamic analysis is uncalled-for and that immediate accessibility of run-time information is beneficial to developers. Continuous and effortless access to run-time views on source code supports developers in acquiring and evaluating their understanding. Run-time views are based on actual data. Thus, they arguably encourage the evaluation of assumptions and eliminate space for speculation.

We employ a new approach to dynamic analysis that enables a feeling of immediacy that current tools are missing. To that effect, the central contributions of this work are:

- A novel approach to dynamic analysis based on a shallow analysis and detached in-depth on-demand refinements,
- A realization of this approach by providing an integrated tool for accessing run-time information during program development,
- Empirical results to evaluate our claims with respect to feasibility.

The remainder of this paper is organized as follows: Section II highlights the benefits of dynamic views for program comprehension and discusses desired tool characteristics. In Section III, we present our interactive approach to dynamic analysis that collects data exactly when needed. Section IV describes our implementation of the proposed approach. In Section V, we demonstrate the desired immediacy characteristics by evaluating our corresponding implementation. Section VI discusses related work and Section VII concludes.

II. Background and Motivation

Due to its abstract nature, source code provides a limited perspective on software systems. Conversely, dynamic views support program comprehension as they aid developers in understanding how a system works. In this section, we illustrate, by means of a running example, how developers benefit from visualized run-time information during program comprehension. We continue by discussing requirements that visualization tools should meet to encourage their frequent adoption in practice.

1095-1350/10 $26.00 © 2010 IEEE
DOI 10.1109/WCRE.2010.17

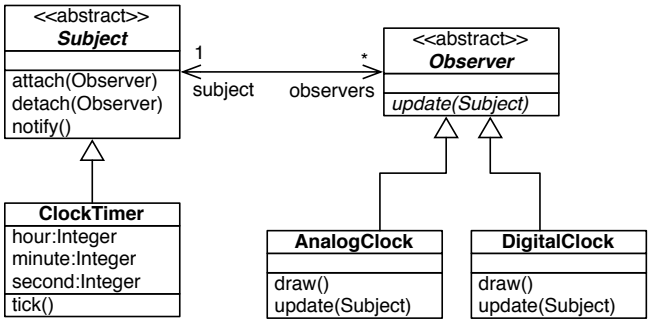

Figure 1. Observer pattern running example.

A. Exploring a Program's Run-time

Visualized run-time information helps developers to better understand program behavior. In our running example, a developer faces the task of understanding a simple clock application, which provides an analog and digital view. The application is based on the description of the Observer design pattern [4]. Figure 1 shows the structure of the sample application. The `ClockTimer` subject represents a ticking clock, whose instances either of the two concrete observers can display. Each `tick` invocation notifies the observers about the change of state.

The developer in our example is unaware of these internals, but can use visualized run-time information to learn about them, and to eventually discover the Observer usage. This process could look as follows.

The visualized information in Figure 2 primarily consists of a call tree that reflects a particular run of the application. A call tree provides comprehensive information of the entire program execution rather than a single execution path. Some of the tree nodes have been expanded to reveal details: for instance, it is evident that `tick` invokes `notify` (index 1).

The figure shows (at index 2) that `notify` sends the `update:` message to two different clocks. From this information, the developer can conclude that there exist two observers, and ascertain this by inspecting the run-time state information attached to the execution of `notify`. The object explorer view at index 3 confirms that the `observers` list contains two clock objects. Moreover, index 4 highlights that a `ClockTimer` participates as the subject in the Observer pattern.

The provided run-time view helps to answer follow-up questions. For instance, call trees and object explorers point out the relationship between observers and subjects. At index 5 in the figure, the developer speculates that `attach:` is responsible for registering observers. In an expanded `attach:` invocation, at index 6, the combined *before* and *after* views of a method node execution show how a `ClockTimer` registers a `DigitalClock` observer. As another example, index 7 marks two views that show how the state of the subject changes after a `tick` execution. If interested, the developer could now further examine the implementation of that method to continue exploring.

In a nutshell, the developer is able to identify the conceptual structure as part of the application: It conforms to the structure of the Observer pattern. In addition to comprehending structural aspects of the application, the developer also gains deep insight about the interactions of the structural elements at run-time.

Visualized run-time information sensibly augments the information available from static views on applications, e. g., their source code. For instance, the authors of the *Gang of Four* book on design patterns [4] aid comprehension of their examples by presenting sequence diagrams alongside class diagrams to visualize collaborations among objects.

In a sense, visualizations of run-time information make the mental model readily available that developers otherwise would have to elaborate manually. There exist valuable approaches to building mental models of software systems from static representations. IDEs support developers in navigating a code base, for example by tracing message sends, in order to gain an understanding of how a system works. However, visualizations such as call trees put application source code and structure into meaningful behavioral contexts, and object explorers provide actual examples of objects rather than their abstract names.

B. The Need for Immediacy

Tools that provide such visualizations of run-time information should allow for a feeling of immediacy to encourage frequent use. To that effect, two essential characteristics should be met. Firstly, visualization tools have to be integral parts of the programming environment. Developers would welcome a tool carrying them from method source code to the visualization of an actual run of the same method by means of one click. Secondly, response times have to be low. Visualized run-time information has to be available within some hundreds of milliseconds rather than minutes [5]. However, immediacy must not hamper the level of detail available from views.

We intend to support program comprehension by reducing the effort of accessing run-time information. We aim to encourage developers to use our tools frequently. Developers shall be able to avoid guesswork and validate assumptions by inspecting actual run-time information instead. The main question that our work addresses is how to make dynamic analysis results available to developers immediately.

III. Immediacy through Interactivity

Our interactive approach to dynamic analysis enables immediacy. Traditional approaches are time-consuming as they capture comprehensive information about the entire execution up-front. Low costs can be achieved by structuring program analysis according to user interaction. More specifically, user interaction allows for dividing the analysis

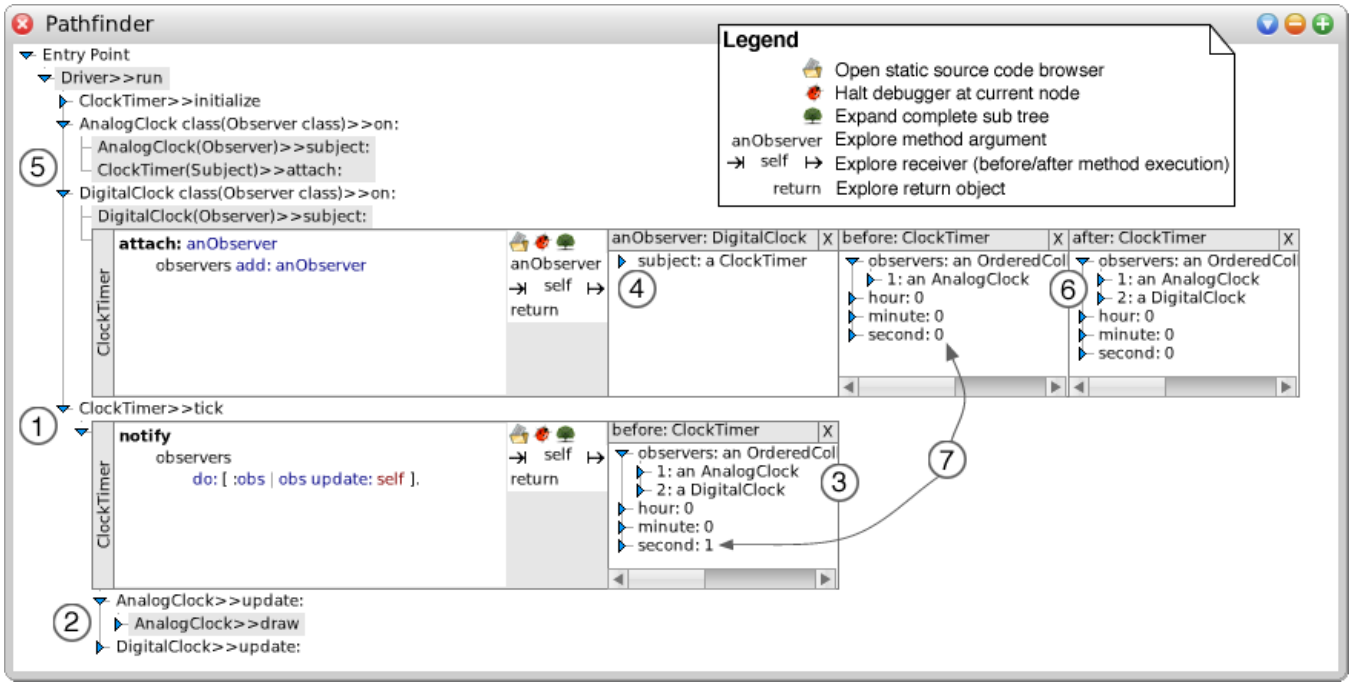

Figure 2. Pathfinder is our interactive dynamic analysis tool for the Squeak IDE.

into multiple steps: A high-level analysis followed by on-demand refinements. This distinction reduces the overhead to provide visualizations of run-time information while preserving instantaneous access to detailed information.

A. Step-wise Run-time Analysis

Splitting the analysis of a program's run-time over multiple runs is meaningful because developers typically follow a systematic approach to understand program behavior. For example, in our scenario (see Section II-A), the developer first uses the presented call tree to gain an initial understanding (1). Later on, the developer identifies execution paths that lead to the population of the list of observers by inspecting relevant state (2). More generally, program comprehension is often tackled by exploring an overview of all run-time information, and continuing to inspect details.

This systematic approach to program comprehension guides our approach to dynamic analysis: Run-time data is captured when needed. (1) A first *shallow analysis* focuses on the information that is required for presenting an overview of a program run. For example, method and receiver names are sufficient to render a call graph as presented in Section II-A. Further information about method arguments or instance variables are not recorded. (2) As the user identifies relevant details, they are recorded on-demand in additional *refinement analysis* runs. In our example, the developer clicks on the `observers` variable to see registered clocks. Information about instances contained in the list are recorded in a separate run triggered by user interaction.

This interactive approach to dynamic analysis requires the ability to reproduce arbitrary points in a program execution. In order to refine run-time information in additional runs, we assume the existence of entry points that specify deterministic program executions. For our implementation, we leverage test cases as such entry points, as they commonly satisfy this requirement [6]. However, our approach is applicable to all entry points that describe reproducible behavior.

B. Less Effort through Step-wise Analysis

Splitting run-time analysis and refining the results on-demand reduces the effort for providing an initial overview, as well as comprehensive details. The amount of required data for generating a run-time visualization to support an inital overview is limited compared to the information that is generated in an entire program run. The data on method activations is sufficient to render the call tree in our example. More specifically, the overhead for collecting method name and receiver information is significantly less than performing a full analysis. A full analysis includes recording exhaustive information before each state change in the execution of a program. In contrast to performing a complete analysis up-front, minimizing the collected data imposes a reduced overhead with respect to the execution of the instrumented program.

User interaction with the initial overview can be leveraged to minimize the overhead of refinement analysis. As the

user identifies interest in individual objects at explicit points of the execution, the information is loaded on-demand in additional analysis steps. Such a subsequent refinement step involves recording of object state at the specified point in the execution. While recording object state might be time-consuming in general, we limit the extent of data collection. More specifically, a refinement step imposes a minimal overhead by focussing on a single object at a particular execution step. This means that refinement analysis is hardly more expensive than execution without instrumentation (see Section V-C).

Our approach divides the effort for dynamic analysis across multiple runs. The information required for program comprehension is arguably a subset of what a full analysis of a program execution can provide. While our approach entails multiple runs, the additional effort is kept to a minimum, especially when compared to a full analysis that has no knowledge of which data is relevant to the user. We reduce the costs by loading information only when the user identifies interest. This provides for quick access to relevant run-time information without collecting needless data.

Our tool Pathfinder (Figure 2) realizes the described interactive approach to dynamic analysis. It is integrated into the the Squeak Smalltalk IDE following our objective of achieving a feeling of immediacy. Pathfinder[1] demonstrates the feasibility of our approach and is the basis of our evaluations in Section V.

IV. Pathfinder Implementation

The realization of step-wise run-time analysis and its associated immediacy is built on flexible and lightweight data structures and algorithms. This section describes selected implementation details of Pathfinder based on our interactive dynamic analysis example described in Section II-A.

Identification and maintenance of suitable analysis entry points are described in Section IV-A. These entry points drive instrumentation of the system under observation with method wrappers, as discussed in Section IV-B. Section IV-C describes how wrappers gather run-time information to build the initial call tree for shallow analysis. In subsequent refinement steps, this call tree is further augmented with deeply copied objects, as Section IV-D illustrates.

A. Analysis Entry Points

As mentioned in Section III-A, principled entry points allow for tracing particular examples of system behavior. Pathfinder leverages test cases as entry points, an approach that we presented in previous work [7]. By maintaining a test coverage relationship, it is possible to identify tests that provide examples for covered methods. We achieve this by extending each method's meta-data by a collection of all tests covering it; the collection is continuously updated during development as tests are selectively executed [8]. That way, methods of interest are reliably embedded in meaningful examples at all times.

Our implementation also includes an extension of the Smalltalk class browser that supports immediate access to method coverage information. From within this view, Pathfinder can directly be invoked. For instance, the example described in Section II-A contains a test case that automatically executes the `run` method and thereby covers `notify`.

Leveraging test cases as entry points is not a requirement for our interactive approach to dynamic analysis. Pathfinder works best if test coverage for the developed application is high, but resorts to manually specified entry points if no covering test is found. This, however, requires more knowledge about the system under observation in the developer than relying on test coverage: it is not always trivial to anticipate control flows leading to methods of interest.

B. Instrumenting Methods

Pathfinder analyzes the execution of entry points by instrumenting the code using method wrappers [9]. Method wrappers are light-weight first-class entities that transparently replace methods with alternative implementations and can delegate to the original (wrapped) implementation. They can be handled like ordinary objects, allowing for simple extensions of method behavior as well as dynamic installation and deinstallation. Pathfinder uses different kinds of method wrappers to decorate methods with additional functionality required during shallow and refinement analysis. *Call wrappers* simply wrap methods and provide fundamental trace data for constructing the call tree (Section IV-C), and *explore wrappers* extract objects for refinement (Section IV-D).

Pathfinder restricts instrumentation of application code to relevant methods. Methods in selected packages that the developer is interested in are wrapped. This process takes test coverage data into account, wrapping only methods covered by a test case used as entry point. Library and framework methods being of no interest are excluded from wrapping, yielding *partial traces*. The selection of relevant packages and exclusion of others avoids unnecessary overhead [10]. In terms of the Observer example, the tool instruments only those classes and methods that actually belong to the method in question—for instance, the `add:` method of Smalltalk's collection library is not relevant for the call tree, and is consequently not instrumented.

Method wrappers are simple and flexible, but bear a certain overhead when compared to immediate bytecode modification. Although they are arguably not the approach to tracing that exhibits the best performance characteristics, the run-time overhead is still acceptable, as we will show in Section V.

In programming languages that do not allow direct access to the meta-level like Smalltalk, other techniques can be employed to achieve the same behavior. In Java, for

[1] A screencast of Pathfinder is available online at http://www.hpi.uni-potsdam.de/swa/projects/pathfinder/

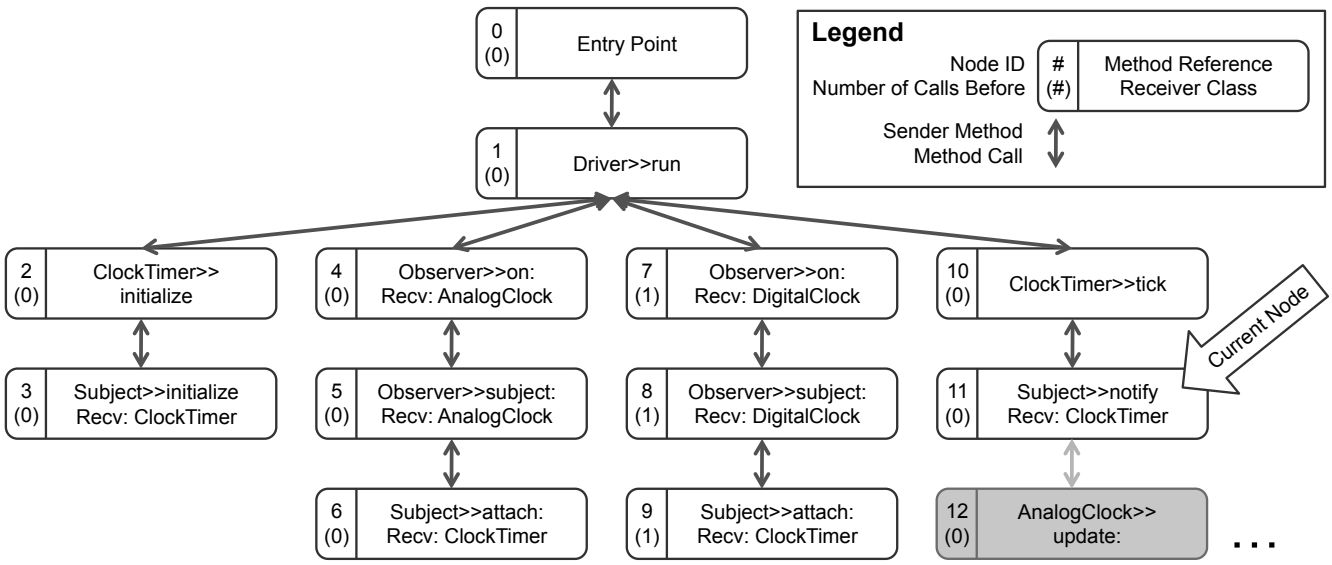

Figure 3. Building the call tree from the perspective of the tracer.

instance, it would be possible to use aspect-oriented programming (AOP) in the form of the AspectJ [11] programming language to instrument all methods with analysis logic. However, since AspectJ does not support dynamic updates, the approach requires auxiliary conditions to be checked at all method invocations. Alternatively, dynamic AOP implementations like CaesarJ [12] could be used, as they allow for dynamically reconfiguring a software system. The underlying technology, however, is the same as used in AspectJ with conditions.

C. Building the Call Tree

Shallow analysis (see Section III-A) ensures low start-up costs for Pathfinder. It builds upon a light-weight call tree that is later on augmented with additional information (see below). This section covers initial call tree construction in detail. The data structure resulting from building a call tree for the example from Section II-A is illustrated in Figure 3.

At first, Pathfinder creates a tracer object for the entry point in question. The tracer object is responsible for constructing and managing the call tree. During execution of entry points, events signaled by call wrappers lead to new nodes being inserted into the call tree. Initially, the call tree consists of a sole root node representing the entry point. All subsequent nodes are attached to the root or its children. In the figure, the root node (with ID 0) can be seen at the top of the tree; it represents a test case calling only the `run` method on an instance of the `Driver` class.

Next, all methods covered by the selected entry point are decorated with method call wrappers. These reference the aforementioned tracer object and report data they collect to it in the form of tracing events, which they signal as soon as each wrapped method is invoked. Now, the entry point is executed, leading to the production of tracing events and their consumption by the tracer object. The information carried by tracing events consists of the current method reference, number of already traced calls to that method, and, optionally, the receiver's type. The latter is only included if it differs from that of the class declaring the method. The number of previous calls is relevant for refinement analysis (see below). In Figure 3, the tracer has already constructed the call tree up to and including the `notify` method, i. e., `AnalogClock`'s `update:` method is currently being run and the corresponding event consumed by the tracer.

From each tracing event, a new call node with a consecutive unique ID is created and inserted in the call tree. The relationship between a node in the tree and its children is bidirectional: downward links denote method calls, while upward links denote senders. To facilitate quick node insertion, the tracer object maintains a reference to the most recently inserted call node, called the "current node". A new node is inserted below the current node, and the current node is updated to reference the newly inserted node after that. This way, call node insertion takes place in constant time. In the figure, a new call node (with ID 12 and grey background) has just been inserted below the current node (ID 11). The new node represents the method `update:` in `AnalogClock`, which has not been called before (the call counter is 0).

When methods terminate, the tracer is notified by the corresponding wrappers and reacts by adjusting the current node reference accordingly. The unique call node IDs are used to correctly drive this adjustment in case of recursive method calls and non-local returns. Once the entry point itself terminates, shallow analysis is completed, and all wrappers related to the given entry point are deinstalled.

D. Refining Nodes

The implementation of *refinement analysis* guarantees fast lookup and re-execution of entry points, during which call tree nodes are augmented with deep copies of relevant objects.

Retrieval of call nodes is very quick, as the Pathfinder GUI's representations of method activations are directly connected to their call node counterparts. Once a call node is known for which details are requested, the method whose activation the node represents is decorated with an *explore wrapper*. Explore wrappers, like call wrappers, signal tracing events to the tracer object, but their payload consists of deep copies of relevant objects, e. g., the receiver and method call arguments. Next, the tracer executes the entry point containing the call node in question.

Since the explore wrapper wraps an entire method, it will be triggered multiple times during call tree re-execution. It must however produce tracing events *only* when the activation corresponding to the call node in question is met. This is realized by means of the count of already traced calls to the respective method (see above): the wrapper maintains an internal activation counter, checking whether its value matches that of the call node's counter. As soon as the desired activation is recognized, the required deep copies are created and attached to the call node.

For copying objects, we rely on Smalltalk's `veryDeepCopy` protocol, which relieves us of the burden of dealing with details of copying. When refinement affects an object for the first time, a deep copy is created right away, even though not all depth levels of the object's structure are of interest at this point in time. While this approach is arguably more memory-consuming than necessary, it is straightforward to implement. Investigation of a more fine-grained solution that copies object elements only as needed is deferred to future work.

Note that the structure of the call tree itself is not changed at all during refinement analysis—it is merely augmented with new data. The wrapper is deinstalled once the requested information has been delivered.

In the example in Figure 3, assume additional information for the node with ID 9 is required. Pathfinder will, in this case, decorate `Subject`'s `attach:` method with an explore wrapper and trigger re-execution at the entry point node (ID 0). Re-execution of the wrapped method at node 6 will not lead to augmentation of that node. Reaching node 9 will trigger deep copying of all relevant objects and addition of these copies to the node.

V. Evaluation

The main argument of this paper is that an interactive approach to dynamic analysis enables immediacy. A stepwise analysis becomes feasible by leveraging user interaction, and allows for instant access to run-time information. We evaluate our approach by measuring the total time and memory needed for collecting the required data for rendering dynamic views.

A. Experimental Setup

We examine our implementation's immediacy characteristics by measuring the time required for shallow and refinement analysis for various projects. While low response times for providing dynamic views are the desired result, our interactive approach aims to reduce the overhead for collecting run-time information. Therefore, we focus our experiments on the intermediary dynamic analysis steps, rather than the time required for selecting entry points or rendering results. Figure 4 summarizes the individual steps and highlights the evaluated parts: Code instrumentation, the collection of data and subsequent de-instrumentation. The experiments were run on a MacBook with a 2.4 GHz Intel Core 2 Duo and 4 GB RAM running Mac OS X 10.6.2, using Squeak version 3.11 alpha on a 4.2.1b1 virtual machine.

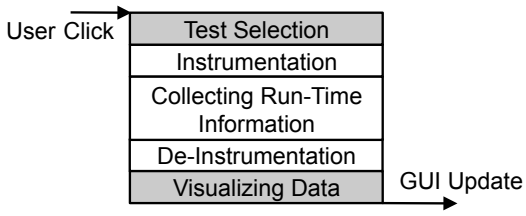

Figure 4. Process for providing dynamic views.

We selected ten different Squeak Smalltalk projects to analyze our approach with respect to various characteristics of the included test suites. Of the ten projects, two (AweSOM and zEmu) are research prototypes developed in our group. The remaining eight are production-quality projects ranging from the Smalltalk compiler over IDE tools to Web application frameworks. All of the latter are in daily use in software development and business activities.

The projects include a total of 4,378 tests, which cover system, acceptance, and unit tests. Measurements for both shallow and refinement analysis entail execution of each test. The individual suites impose different computational costs: The assertions range from checking for simple return values to more demanding I/O operations.

The project properties are summarized in Table I. The last row lists the average number of method calls per test for application code that are instrumented for shallow analysis.

To allow for a more general assessment of both the project characteristics and experiment results, we present the project characteristics of an arguably representative Java project: JHotDraw[2]. The metrics indicate that the selected Smalltalk projects are similar: The average number of method invocations for tests in Squeak Smalltalk projects is 1,416 while JHotDraw's tests entail 1,429 method activations on average.

[2]http://www.jhotdraw.org/, last accessed on June 25, 2010

Table I
SUMMARY OF PROJECT PROPERTIES.

	Compiler	Xml	SUnit	Monticello	Browser	aweSOM	zEmu	Seaside	Magritte	Pier	JHotDraw
Classes	64	25	16	140	51	68	47	410	189	266	551
Methods	1294	367	313	1773	1179	742	1012	4030	1972	2564	5052
Tests	49	4	45	112	59	164	349	242	1645	1709	1216
Calls/Tests	4502	2732	47	30703	218	6664	1178	327	74	474	1429

Table II
AVERAGE EXECUTION TIME, SHALLOW ANALYSIS (SA), REFINEMENT ANALYSIS (RA) AND MEMORY OVERHEAD.

	Compiler	Xml	SUnit	Monticello	Browser	aweSOM	zEmu	Seaside	Magritte	Pier
Execution time (ms)	7.69	1.13	63.81	720.42	28.09	17.33	15.12	0.47	0.38	0.50
Δ SA (ms)	247.23	164.38	133.50	1216.52	164.64	235.79	172.28	288.19	203.38	238.39
Δ RA (ms)	2.15	5.38	1.61	108.40	1.13	5.93	1.81	564.14	3.44	2.13
SA Memory (kbyte)	991	608	12	6860	60	1464	259	93	17	106

B. Results

The empirical results[3] of our experiments are summarized in Table II. It presents average values for each project: The time required for executing tests, the overhead resulting from shallow and refinement analysis, and memory consumption of a populated data structure that is used to generate the initial call tree. The average time for collecting data for shallow analysis is below 246 milliseconds (ms) and the 99th percentile is below 375 ms. The collected data during shallow analysis required less than 320 kilobytes on average. Figure 5 illustrates that the time and memory overhead for shallow analysis grow linearly with respect to the number of methods that are invoked by a test. The average analysis time required for refinement, which entails a single deep copy, is below 37 ms.

C. Discussion

Our empirical results illustrate the feasibility of our interactive approach: The imposed overhead for dynamic analysis is divided across multiple runs and allows for short response times. The time it takes to collect data for generating a call tree is below 300 ms and for a refinement step below 60 ms on average (this includes the time it takes to run the test). While our evaluation focusses on measuring the time required for dynamic analysis, we also conducted independent experiments to consider the response time of our graphical interface. The time required for rendering call trees averaged around 200 ms for the Monticello project, which involves the highest number of calls per test. We argue that this supports our claim of achieving immediacy characteristics by providing a visualization of run-time information in considerably less than a single second in the majority of cases. Schneiderman [5] argues that two seconds is the upper limit for responding to a user request.

The overhead for refinement analysis is relatively high for the Monticello and Seaside projects. This is due to the deep structures of the objects that are being copied: source code repository mock ups for Monticello and web sessions with attached web applications for the Seaside project. However, incremental refinement imposes a minimum of overhead in most cases: the 95th percentile for refinement analysis overhead is 6 ms for all tests.

Our approach focusses on collecting data that is of direct use, as we are able to load additional details on-demand. We rely on the user to identify relevant information. Refinement steps only entail deep copies of single objects, while shallow analysis is limited to message and receiver names. Thus, we are able to keep response times and memory consumption low for both shallow and refinement analysis. This is a major advantage over other tools that collect a vast amount of data to reconstruct a program's runtime.

Threats to Validity: Validity might be impeded by limited scalability to practical applications. As mentioned in Section V-A, most of the studied projects are in fact practical real-world systems. Moreover, it cannot be concluded that the observed linear growth (see Figure 5) continues for tests with much more method activations.

Regarding general applicability of our approach, one might argue that evaluating it only in a Smalltalk context does not allow for such a conclusion. However, as we have pointed out in Section V-A, a particular complex Java framework exhibits characteristics comparable to those of the studied Smalltalk systems. While this insight does not guarantee scalability to arbitrary languages and systems, it provides a worthwhile direction for future studies assessing applicability.

The evaluation setting has two particular characteristics that might limit validity. On the one hand, garbage collection was disabled during measurement runs to be able to gather memory consumption data, and to elide performance influences of garbage collector runs. In a realistic setting with enabled garbage collection, minimal slow-downs would be possible. On the other hand, we rely on tests to obey certain rules of good style: e. g., they should be deterministic.

[3]Raw data is available online at http://www.hpi.uni-potsdam.de/swa/projects/pathfinder/

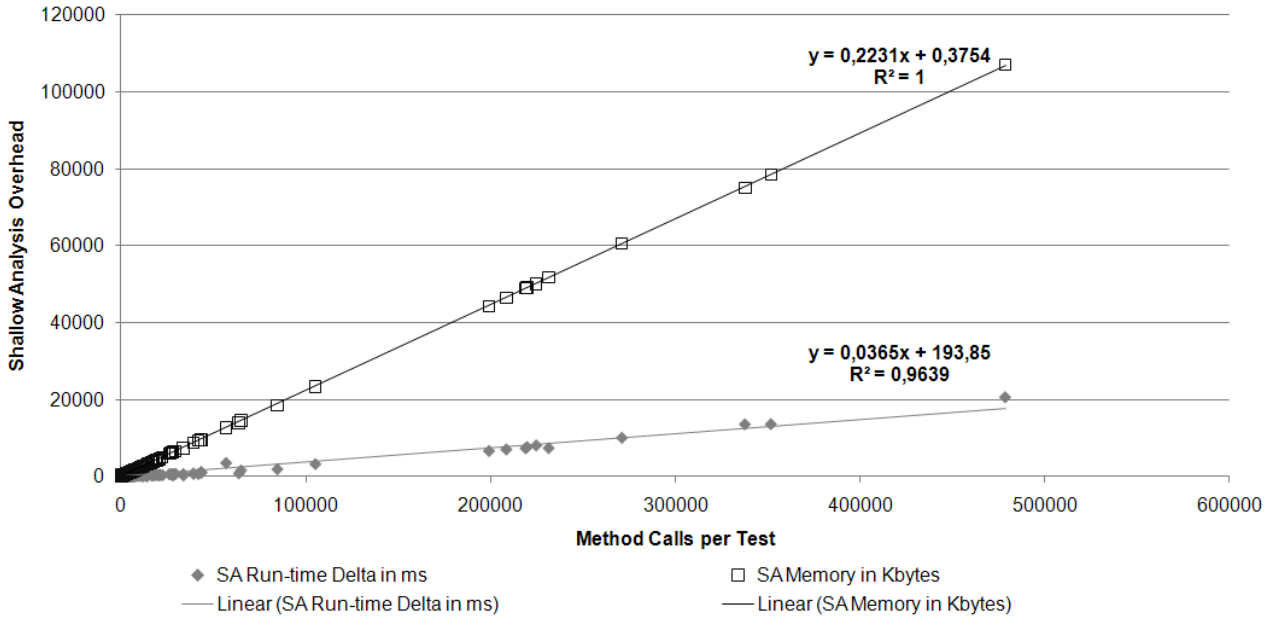

Figure 5. Shallow analysis overhead (run-time and memory consumption).

Tests that do not follow these guidelines might hamper the conclusions we made. The tests that we used in our evaluation were all acceptable in this respect.

VI. Related Work

To highlight the motivation for our work, we present recent studies of developer activities in the context of program comprehension and software maintenance tasks. We continue by presenting several approaches that leverage collected run-time information to support developers. We divide related approaches into three categories: Integration of run-time data into IDEs, debugging tools, and call graph visualizations.

A. Program Comprehension Studies

Sillito et al. [13] focused on source code related questions that arise during maintenance. The studies investigated what kinds of information developers required, which information retrieval paths they followed, and which tools they used. The results indicate that run-time views as described in Section II can help developers to answer 13 of the 29 questions that are currently not well supported.

Latoza et al. [14] presented three studies concerning reachability questions, which are described as a "search across feasible paths through a program for target statements matching search criteria". The authors show that developers often failed to understand program behavior and modified the code relying on false assumptions. Developers were asked to rate the frequency and difficulty of 12 questions concerning program comprehension, such as "In what situations is this method called?". The study revealed that developers ask over nine of these questions in day-to-day work, and that they are often hard to answer. Furthermore, "9 of the 10 longest activities were associated with reachability questions." These questions address program run-time behavior rather than relationships between source code artifacts. The authors concluded that lack of adequate tool support was a reason for the developers' problems with answering reachability questions.

Our proposed approach provides run-time information visualizations immediately. Run-time views support exploring run-time behavior and facilitate answering specific questions such as: In what context is a particular method used? Our approach encourages frequent use of corresponding tools and thus promotes the validation of assumptions rather than relying on guess work.

B. Run-time Information in IDEs

The feature driven browser in [15] and Senseo [16] are both extensions to an IDE that provide additional support by collecting run-time information. The former helps developers to locate and highlight source code that contributes to features of interest [17]. For that purpose, developers have to assign test cases to features. Analyzing the execution of these tests, the feature driven browser provides a constrained set of code entities that are relevant for a certain feature.

The IDE extension Senseo [16] supports the navigation in source code artifacts. IDEs usually trace message sends statically. For instance, they provide lists of all possible callers of a particular method. Due to late-binding, these lists can include much more entries than those that are actually relevant for the program at hand. Senseo collects run-time

information to restrict those lists to entries that are relevant. This approach helps to identify actual types that are used at run-time in the context of polymorphism.

Both tools support program comprehension mainly by improving the navigation in static views. In contrast, our interactive approach to dynamic analysis targets visualizations of actual program executions. It allows for understanding application source code based on sensible examples.

C. Debugging Tools

Debuggers usually allow for examining the effects of a program step by step. They support reasoning about the behavior of programs in general, which might be the reason why developers use debuggers to improve their understanding of source code [18]. However, the use of debugging tools requires appropriate entry points into the execution of program code. Pathfinder applies an approach introduced in [7]: Leveraging test cases as sensible entry points. Appropriate test cases can be determined by maintaining a test coverage relationship during the regular execution of tests.

Back-in-time debuggers [19] enable developers to navigate an entire program execution and help to answer questions about the cause of a particular state [20]. This is achieved by recording run-time data until execution is interrupted, e. g., by defining a break-point. The required dynamic analysis is time-consuming, the performance slows down up to 300 times and up to 100 MB of memory are consumed per second [19]. Several techniques to reduce the costs have been reported [21], [22], in return for a more complicated setup.

Compared to such debugging tools, our approach is designed specifically to facilitate program comprehension tasks. Pathfinder quickly provides established dynamic views. This is achieved by enabling developers to interact with the analysis tool to specify interest in particular parts to load detailed information on-demand.

D. Software Visualization of Call Graphs

There are plenty approaches to software visualization that present behavior using call graphs [23]. One of the first approaches [24] presents several prototypes ranging from a global overview of large traces to lowest system level events. Subsequent approaches primarily focus on presenting large amounts of trace data. The execution pattern view [25] automatically classifies repetitive behavior into high order execution patterns. The VizzAnalyzer [26] merges static and dynamic analysis to generate static call graphs with run-time information. The call graph analyzer [27] visualizes call graphs in a 2.5D environment and extends this information by a number of other analysis approaches. Circular Bundle Views [28] deal with the fact that sequence diagrams do not scale and present a scalable view inside a circle.

Although all approaches are useful in their specific scenario, no approach can be generalized for various tasks in program comprehension [29]. Most such visualizations are rarely used during software development [30]. One reason might be the missing integration into development environments [31] or the neglected aspect of low setup and performance costs that Pathfinder addresses.

VII. Summary

We have proposed an interactive approach to collect and present run-time data. We have described the systematic approach that developers use to understand program behavior, and argued that it can be leveraged to divide dynamic analysis across multiple runs. Connecting dynamic analysis with user interaction enables reduced effort for providing an initial overview. Refine steps impose a minimum of additional cost to provide relevant details on-demand. We have presented an integrated implementation of our approach: Pathfinder. It enables immediate access to run-time views for selected code entities at the push of a button. Our empirical results demonstrate that our approach allows for immediacy characteristics, and thus arguably encourages frequent use of actual run-time information for understanding abstract source code.

Future work is two-fold. Firstly, the implementation is being extended to support multi-threaded applications, and we investigate the performance / memory tradeoffs of a refinement analysis implementation that does not make use of eager deep copies. Secondly, we are planning a user study to assess how developers use the features available in Pathfinder, and how the approach improves on maintenance tasks.

References

[1] T. A. Corbi, “Program Understanding: Challenge for the 1990s,” *IBM Systems Journal*, vol. 28, no. 2, pp. 294–306, 1989.

[2] V. R. Basili, “Evolving and packaging reading technologies,” *Journal of Systems and Software*, vol. 38, no. 1, pp. 3–12, 1997.

[3] A. J. Ko, R. DeLine, and G. Venolia, “Information Needs in Collocated Software Development Teams,” in *ICSE ’07: Proceedings of the 29th International Conference on Software Engineering*. IEEE Computer Society, 2007, pp. 344–353.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[5] B. Schneiderman, *Designing the User Interface*. Addison-Wesley, 1992.

[6] G. Meszaros, *XUnit Test Patterns: Refactoring Test Code*. Prentice Hall, 2006.

[7] B. Steinert, M. Perscheid, M. Beck, J. Lincke, and R. Hirschfeld, “Debugging Into Examples: Leveraging Tests for Program Comprehension,” in *TestCom ’09: Proceedings of the 21st International Conference on Testing of Communicating Systems*. Springer, 2009, pp. 235–240.

[8] B. Steinert, M. Haupt, R. Krahn, and R. Hirschfeld, "Continuous Selective Testing," in *XP '10: Agile Processes in Software Engineering and Extreme Programming*. Springer, 2010, pp. 132–146.

[9] J. Brant, B. Foote, R. Johnson, and D. Roberts, "Wrappers to the Rescue," in *ECOOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*. Springer, 1998, pp. 396–417.

[10] T. Gschwind and J. Oberleitner, "Improving Dynamic Data Analysis with Aspect-Oriented Programming," in *CSMR '03: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*. IEEE Computer Society, 2003, pp. 259–268.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An Overview of AspectJ," in *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*. Springer, 2001, pp. 327–353.

[12] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann, "An Overview of CaesarJ," *Transactions on Aspect-Oriented Software Development I*, vol. 3880/2006, pp. 135–173, 2006.

[13] J. Sillito, G. C. Murphy, and K. De Volder, "Asking and Answering Questions during a Programming Change Task," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 434–451, 2008.

[14] T. LaToza and B. Myers, "Developers Ask Reachability Questions," in *ICSE '10: Proceedings of the 32nd International Conference on Software Engineering*. ACM, 2010, pp. 185–194.

[15] D. Röthlisberger, O. Greevy, and O. Nierstrasz, "Feature Driven Browsing," in *ICDL '07: Proceedings of the 2007 International Conference on Dynamic Languages*. ACM, 2007, pp. 79–100.

[16] D. Röthlisberger, M. Härry, A. Villazón, D. Ansaloni, W. Binder, O. Nierstrasz, and P. Moret, "Augmenting Static Source Views in IDEs with Dynamic Metrics," in *ICSM '09: The 2009 IEEE International Conference on Software Maintenance*. IEEE Computer Society, 2009, pp. 253–262.

[17] O. Greevy, "Enriching Reverse Engineering with Feature Analysis," Ph.D. dissertation, University of Berne, 2007.

[18] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining Mental Models: A Study of Developer Work Habits," in *ICSE '06: Proceedings of the 28th International Conference on Software Engineering*, 2006, pp. 492–501.

[19] B. Lewis, "Debugging Backwards in Time," in *AADEBUG'03: Proceedings of the Fifth International Workshop on Automated Debugging*, 2003, pp. 225–235.

[20] A. J. Ko and B. A. Myers, "Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior," in *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*. ACM, 2008, pp. 301–310.

[21] G. Pothier, E. Tanter, and J. Piquer, "Scalable Omniscient Debugging," in *OOPSLA '07: Proceedings of the 22nd Conference on Object-Oriented Programming Systems and Applications*. ACM, 2007, pp. 535–552.

[22] A. Lienhard, T. Gîrba, and O. Nierstrasz, "Practical Object-Oriented Back-in-Time Debugging," in *ECOOP '08: Proceedings of the 22nd European Conference on Object-Oriented Programming*. Springer, 2008, pp. 592–615.

[23] A. Hamou-Lhadj and T. C. Lethbridge, "A Survey of Trace Exploration Tools and Techniques," in *CASCON '04: Proceedings of the 2004 Conference of the Centre for Advanced Studies on Collaborative Research*. IBM Press, 2004, pp. 42–55.

[24] D. F. Jerding, J. T. Stasko, and T. Ball, "Visualizing Message Patterns in Object-Oriented Program Executions," *Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Technical Report GIT-GVU-96-15*, 1996.

[25] W. D. Pauw, D. Lorenz, J. Vlissides, and M. Wegman, "Execution Patterns in Object-Oriented Visualization," in *COOTS'98: Proceedings of the 4th Conference on Object-Oriented Technologies and Systems*. USENIX Association, 1998, pp. 16–16.

[26] W. Löwe, A. Ludwig, and A. Schwind, "Understanding Software–Static and Dynamic Aspects," in *ICAST '01: 17th International Conference on Advanced Science and Technology*, 2001, pp. 52–57.

[27] J. Bohnet and J. Döllner, "Visual Exploration of Function Call Graphs for Feature Location in Complex Software Systems," in *SoftVis '06: Proceedings of the 2006 ACM Symposium on Software Visualization*. ACM, 2006, pp. 95–104.

[28] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J. J. van Wijk, and A. van Deursen, "Understanding Execution Traces Using Massive Sequence and Circular Bundle Views," in *ICPC '07: Proceedings of the 15th IEEE International Conference on Program Comprehension*. IEEE Computer Society, 2007, pp. 49–58.

[29] M. J. Pacione, M. Roper, and M. Wood, "A Comparative Evaluation of Dynamic Visualisation Tools," in *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering*. IEEE Computer Society, 2003, pp. 80–89.

[30] D. Röthlisberger, O. Greevy, and O. Nierstrasz, "Exploiting Runtime Information in the IDE," in *ICPC '08: Proceedings of the 16th IEEE International Conference on Program Comprehension*. IEEE Computer Society, 2008, pp. 63–72.

[31] S. M. Charters, N. Thomas, and M. Munro, "The End of the Line for Software Visualisation?" in *In Proceedings of the 2nd Workshop on Visualizing Software for Analysis and Understanding*. Society Press, 2003, pp. 110–112.