# A Controlled Experiment for Program Comprehension through Trace Visualization

Bas Cornelissen, Andy Zaidman, *Member, IEEE Computer Society,*
and Arie van Deursen, *Member, IEEE Computer Society*

**Abstract**—Software maintenance activities require a sufficient level of understanding of the software at hand that unfortunately is not always readily available. Execution trace visualization is a common approach in gaining this understanding, and among our own efforts in this context is EXTRAVIS, a tool for the visualization of large traces. While many such tools have been evaluated through case studies, there have been no quantitative evaluations to the present day. This paper reports on the first controlled experiment to quantitatively measure the added value of trace visualization for program comprehension. We designed eight typical tasks aimed at gaining an understanding of a representative subject system, and measured how a control group (using the Eclipse IDE) and an experimental group (using both Eclipse and EXTRAVIS) performed these tasks in terms of time spent and solution correctness. The results are statistically significant in both regards, showing a 22% decrease in time requirements and a 43% increase in correctness for the group using trace visualization.

**Index Terms**—Program comprehension, dynamic analysis, controlled experiment.

✦

## 1 INTRODUCTION

PROGRAM comprehension has become an increasingly important aspect of the software development process. As software systems grow larger and their development becomes more expensive, they are constantly modified rather than built from scratch, which means that a great deal of effort is spent on performing maintenance activities. However, as up to date documentation is often lacking, it is estimated that up to 60% of the maintenance effort is spent on gaining a sufficient *understanding* of the program at hand [1], [2]. It is for this reason that the development of techniques and tools that support the comprehension process can make a significant contribution to the overall efficiency of software development.

With respect to such techniques, the literature offers numerous solutions that can be roughly broken down into static and dynamic approaches (and combinations thereof). Whereas static analysis relies on such artifacts as source code and documentation, dynamic analysis focuses on a system's execution. An important advantage of dynamic analysis is its precision, as it captures the system's actual behavior. Among the drawbacks are its incompleteness, as the gathered data pertains solely to the scenario that was executed; and the well-known scalability issues, due to the often excessive amounts of execution trace data. Particularly this latter aspect is troublesome because of the cognitive overload on the part of the maintainer.

To cope with the issue of scalability, a significant portion of the literature on program comprehension has been dedicated to the reduction [3], [4] and visualization [5], [6] of execution traces. One of these techniques and tools is EXTRAVIS, our tool from prior work [7] that offers two interactive views of large execution traces. Through a series of case studies we illustrated how EXTRAVIS can support different types of common program comprehension activities. However, in spite of these efforts, there is no *quantitative* evidence of the tool's usefulness in practice. As we will show in the next section, no such evidence is offered for any of the trace visualization techniques in the program comprehension literature.

The purpose of this paper, therefore, is a first quantification of the usefulness of trace visualization for program comprehension. Furthermore, to gain a deeper understanding of the nature of its added value, we investigate which types of tasks benefit most from trace visualization and from EXTRAVIS. To fulfill these goals, we design and execute a controlled experiment in which we measure how the tool affects (1) the time that is needed for typical comprehension tasks, and (2) the correctness of the solutions given during those tasks.

This paper extends our previous work [8] with a survey of 21 trace visualization techniques, an additional group of subjects with an industrial background (thus strengthening the statistical significance as well as the external validity), and a discussion on the implications of our EXTRAVIS findings for trace visualization tools in general.

The remainder of the paper is structured as follows. Section 2 extensively reviews existing techniques and

---

- *B. Cornelissen is with the Software Improvement Group, A.J. Ernststraat 595-H, 1082LD Amsterdam, The Netherlands.*
  *E-mail: b.cornelissen@sig.eu.*
- *A. Zaidman and A. van Deursen are with the Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, Mekelweg 4, 2628CD Delft, The Netherlands.*
  *E-mail: {a.e.zaidman, arie.vandeursen}@tudelft.nl.*

tools for trace visualization, and motivates our intent to conduct a controlled experiment. Section 3 offers a detailed description of the experimental design. Section 4 presents the results of our experiment, which are then discussed in Section 5. Section 6 discusses threats to validity, and Section 7 offers conclusions and future directions.

## 2 BACKGROUND

### 2.1 Execution trace analysis

The use of dynamic analysis for program comprehension has been a popular research activity in the last decades. In a large survey that we recently performed, we identified a total of 176 articles on this topic that were published between 1972 and June 2008 [9]. More than 30 of these papers concern *execution trace analysis*, which has often shown to be beneficial to such activities as feature location, behavioral analysis, and architecture recovery.

Understanding a program through its execution traces is not an easy task because traces are typically too large to be comprehended directly. Reiss and Renieris, for example, report on an experiment in which one gigabyte of trace data was generated for every two seconds of executed C/C++ code or every ten seconds of Java code [3]. For this reason, there has been a significant effort in the automatic *reduction* of traces to make them more tractable (e.g., [3], [10], [4]). The reduced traces can then be *visualized* by traditional means: for example, as directed graphs or UML sequence diagrams. On the other hand, the literature also offers several non-traditional trace visualizations that have been designed specifically to address the scalability issues.

In Section 2.2 we present an overview of the current state of the art in trace visualization. Section 2.3 describes EXTRAVIS, our own solution, and Section 2.4 motivates the need for controlled experiments.

### 2.2 Execution trace visualization

There exist three surveys in the area of execution trace visualization that provide overviews of existing techniques. The first survey was published in 2003 by Pacione et al., who compare the performance of five dynamic visualization tools [42]. Another survey was published in 2004 by Hamou-Lhadj and Lethbridge, who describe eight trace visualization tools from the literature [43]. Unfortunately, these two overviews are incomplete because (1) the selection procedures were non-systematic, which means that papers may have been missed; and (2) many more solutions have been proposed in the past five years. A third survey was performed by the authors of this paper in 2008, and was set up as a large-scale systematic literature survey of all dynamic analysis-based approaches for program comprehension [9]. However, its broad perspective prevents subtle differences between trace visualization techniques from being exposed, particularly in terms of evaluation: for example, it does not distinguish between user studies and controlled experiments.

To obtain a complete overview of all existing techniques and to reveal the differences in evaluation, we have used our earlier survey to identify all articles on trace visualization for program comprehension from 1988 onwards, and then reexamined these papers from an evaluation perspective. In particular, we have focused on techniques that *visualize (parts of) execution traces*. We identified the types of validation and the areas in which the techniques were applied. Also of interest is the public availability of the tools involved, which is crucial for fellow researchers seeking to study existing solutions or perform replications of the experiment described in this paper.

Our study has resulted in the identification and characterization of 21 contributions[1] that were published between 1988 and 2008, shown in Table 1. For each contribution, the table shows the appropriate references, associated tools (with asterisks denoting public availability), evaluation types, and areas in which the technique was applied. In what follows, we briefly describe the contents of each paper.

**1988-2000**

Kleyn and Gingrich were among the first to point out the value of visualizing run-time behavior [11]. Their visualization of execution traces is graph-based and aims at better understanding software and identifying programming errors. In particular, their graph visualization is animated, in the sense that the user of the tool can step through the entire execution and observe what part(s) of the program are currently active. A case study illustrates how their views can provide more insight into the inner workings of a system.

De Pauw et al. introduced their interaction diagrams (similar to UML sequence diagrams) in Jinsight, a tool that visualizes running Java programs [5]. Jinsight was later transformed into the publicly available TPTP Eclipse plugin, which brings execution trace visualization to the mainstream Java developer. The authors also noticed that the standard sequence diagram notation was difficult to scale up for large software systems, leading to the development of their *"execution pattern"* notation, a much more condensed view of the typical sequence diagram [12].

Koskimies and Mössenböck proposed Scene, which combines a sequence diagram visualization with hypertext facilities [15]. The hypertext features allow the user to browse related documents such as source code or UML class diagrams. The authors are aware of scalability issues when working with sequence diagrams and therefore proposed a number of abstractions.

Jerding et al. created ISVis, the *"Interaction Scenario Visualizer"* [6], [16]. ISVis combines static and dynamic

---

[1] Of the 36 papers found, Table 1 shows only the 21 unique contributions (i.e., one per first author).

TABLE 1
Overview of existing trace visualization techniques

| References | Tool | Evaluation type | Applications |
|---|---|---|---|
| [11] | GRAPHTRACE | small case study | debugging |
| [5], [12], [13], [14] | JINSIGHT; OVATION; TPTP* | preliminary; user feedback | general understanding |
| [15] | SCENE* | preliminary | software reuse |
| [6], [16] | ISVIS* | case study | architecture reconstruction, feature location |
| [17], [18] | SCED; SHIMBA | case study | debugging; various comprehension tasks |
| [19] | FORM | case study | detailed understanding; distributed systems |
| [20] | JAVAVIS | preliminary; user feedback | educational purposes; detailed understanding |
| [21], [4], [22], [23] | SEAT | small case studies; user feedback | general understanding |
| [24], [25], [26], [27] | SCENARIOGRAPHER | multiple case studies | detailed understanding; distributed systems; feature analysis; large-scale software |
| [28], [29], [30] | – | small case study | quality control; conformance checking |
| [10] | – | multiple case studies | general understanding |
| [31] | – | case study | trace comparison; feature analysis |
| [32] | – | case study | feature analysis |
| [33] | – | case study | architecture reconstruction; conformance checking; behavioral profiles |
| [34] | TRACEGRAPH | industrial case study | feature analysis |
| [35], [36] | SDR; JRET* | multiple case studies | detailed understanding through test cases |
| [37] | FIELD; JIVE; JOVE | multiple case studies | performance monitoring; phase detection |
| [38] | – | – | API understanding |
| [39], [7] | EXTRAVIS* | multiple case studies | fan-in/-out analysis; feature analysis; phase detection |
| [40] | OASIS | user study | various comprehension tasks |
| [41] | – | small case studies | general understanding; wireless sensor networks |

information to accomplish amongst others *feature location*, the establishment of relations between concepts and source code [44]. ISVis' dynamic component visualizes *scenario views*, which bear some resemblance to sequence diagrams. Of particular interest is the *Information Mural* view, which effectively provides an overview of an entire execution scenario, comprising hundreds of thousands of interactions. The authors have applied ISVis to the Mosaic web browser in an attempt to extend it.

Systä et al. presented an integrated reverse engineering environment for Java that uses both static and dynamic analysis [17], [18]. The dynamic analysis component of this environment, SCED, visualizes the execution trace as a sequence diagram. In order to validate their approach, a case study was performed on the Fujaba open source UML tool suite, in which a series of program comprehension and reverse engineering tasks were conducted.

### 2000-2005

Souder et al. were among the first to recognize the importance of understanding distributed applications with the help of dynamic analysis [19]. To this purpose, they use Form, which enables to draw sequence diagrams for distributed systems. The authors validate their approach through a case study.

Oeschle and Schmitt built a tool called JAVAVIS that visualizes running Java software, amongst others through sequence diagrams [20]. The authors' main aim was to use JAVAVIS for educational purposes and their validation comprises informal feedback from students using the tool.

Hamou-Lhadj et al. created the Software Exploration and Analysis Tool (SEAT) that visualizes execution traces as trees. It is integrated in the IDE to enable easy navigation between different views [22]. SEAT should be considered as a research vehicle in which the authors explored some critical features of trace visualization tools. Subsequently, they began exploring such solutions, such as trace compression [4] or removing parts of the trace without affecting its overall information value [23]. While the degree of compression is measured in several case studies, the added value for program comprehension remains unquantified.

Salah and Mancoridis investigate an environment that supports the comprehension of distributed systems, which are typically characterized by the use of multiple programming languages [24]. Their environment visualizes sequence diagrams, with a specific notation for inter-process communication. The authors also report on a small case study. Salah et al. later continued their dynamic analysis work and created the so-called module-interaction view, that shows which modules are involved in the execution of a particular use case [27]. They evaluate their visualization in a case study on Mozilla and report on how their technique enables feature location.

Briand et al. specifically focused on visualizing sequence diagrams from distributed applications [28], [30]. Through a small case study with their prototype tool they have reverse engineered sequence diagrams for checking design conformance, quality, and implementation choices.

Zaidman and Demeyer represented traces as signals in time [10]. More specifically, they count how many times individual methods are executed and using this metric, they visualize the execution of a system throughout time. This allows to identify phases and re-occurring behavior.

They show the benefits of their approach using two case studies.

**2006-2007**

Kuhn and Greevy also represented traces as signals in time with their "dynamic time warping" approach [31]. In contrast to Zaidman and Demeyer, they rely on the stack depth as the underlying metric. The signals are compared to one another to locate features, as illustrated by a case study.

Greevy et al. explored polymetric views to visualize the behavior of features [32]. Their 3D visualization renders run-time events of a feature as towers of instances, in which a tower represents a class and the number of boxes that compose the tower indicates the number of live instances. Message sends between instances are depicted as connectors between the boxes. The authors perform a case study to test their approach.

Koskinen et al. proposed *behavioral profiles* to understand and identify extension points for components [33]. Their technique combines information from execution traces and behavioral rules defined in documentation to generate these profiles, which contain an architectural level view on the behavior of a component or application. Their ideas are illustrated in a case study.

Simmons et al. used TraceGraph to compare execution traces with the aim of locating features [34]. Furthermore, they integrate the results of their feature location technique into a commercial static analysis tool so as to make feature location more accessible to their industrial partner. The authors furthermore report on a case study performed in an industrial context.

**2007-2008**

Cornelissen et al. looked specifically into generating sequence diagrams from test cases, arguing that test scenarios are relatively concise execution scenarios that reveal a great deal about the system's inner workings [35]. They initially applied their SDR tool to a small case study, and later extended their ideas in the publicly available JRET eclipse plugin, which was evaluated on a medium-scale open source application [36].

Over the years, Reiss has developed numerous solutions for visualizing run-time behavior [37]. Among the most notable examples are FIELD, which visualizes dynamic call graphs, and JIVE, which visualizes the execution behavior in terms of classes or packages. JIVE's visualization breaks up time in intervals and for each interval it portrays information such as the number of allocations, the number of calls, and so on.

Jiang et al. concentrated on generating sequence diagrams specifically for studying API usage [38]. The rationale of their approach is that it is often difficult to understand how APIs should be used or can be reused. An evaluation of their approach is as yet not available.

Bennett et al. engineered the Oasis Sequence Explorer [40]. Oasis was created based on a focus group experiment that highlighted some of the most desirable features when exploring execution traces. The authors then performed a user study to validate whether the Oasis features were indeed helpful during a series of typical software maintenance tasks, with quite useful measurements as a result.

Dalton and Hallstrom designed a dynamic analysis visualization toolkit specifically aimed at TinyOS, a component-based operating system mainly used in the realm of wireless sensor networks [41]. They generate annotated call graphs and UML sequence diagrams for studying and understanding TinyOS applications. They illustrate the benefits of their tool through a case study on a TinyOS component.

## 2.3 Extravis

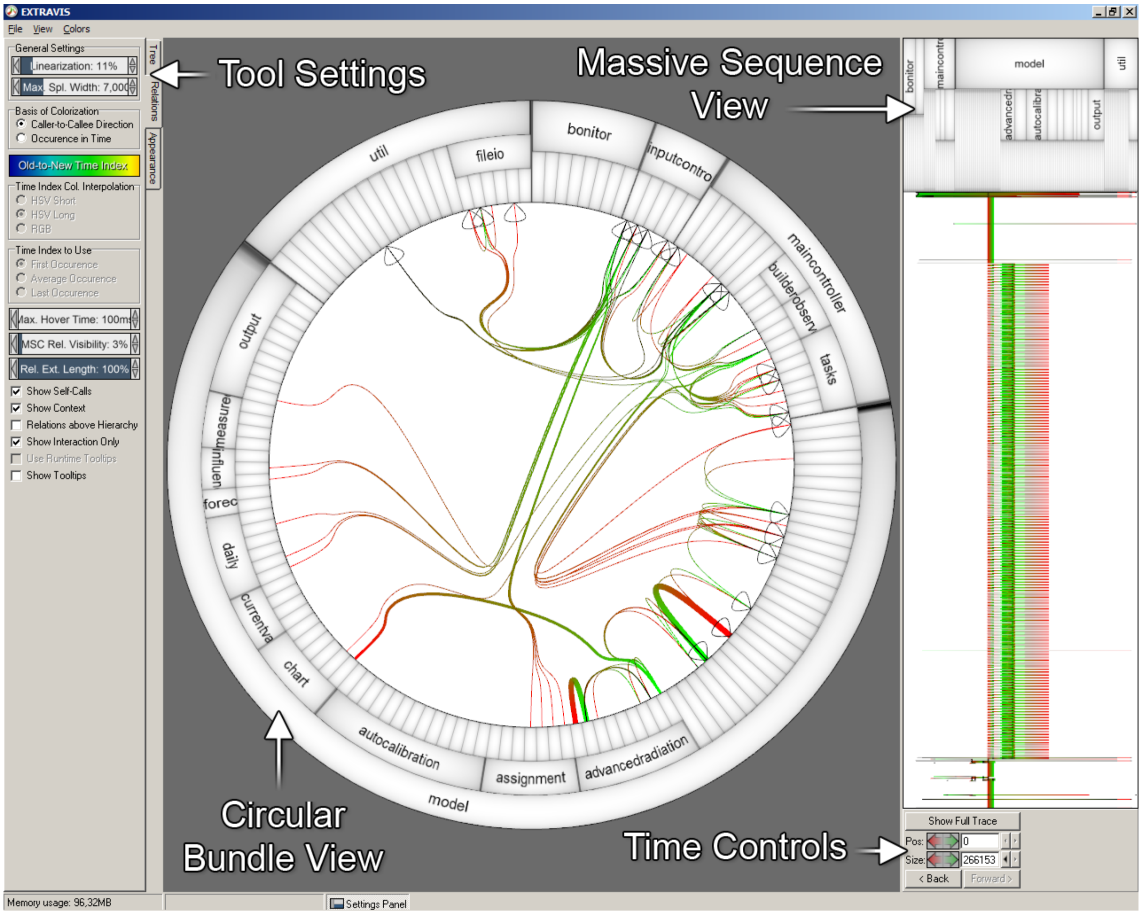
Among our own contributions to the field of trace visualization is EXTRAVIS. This publicly available[2] tool provides two linked, interactive views, shown in Figure 1. The *massive sequence view* is essentially a large-scale UML sequence diagram (similar to Jerding's Information Mural [45]), and offers an overview of the trace and the means to navigate it. The *circular bundle view* hierarchically projects the program's structural entities on a circle and shows their interrelationships in a bundled fashion. A comparison of EXTRAVIS with other tools is provided in our earlier work [7].

We qualitatively evaluated the tool in various program comprehension contexts, including trace exploration, feature location, and top-down program comprehension [7]. The results provided initial evidence of EXTRAVIS' benefits in these contexts, the main probable advantages being its optimal use of screen real estate and the improved insight into a program's structure. However, we hypothesized that the relationships in the circular view may be difficult to grasp.

## 2.4 Validating trace visualizations

The overview in Table 1 shows that trace visualization techniques in the literature have been almost exclusively evaluated using case studies. Indeed, there have been no efforts to *quantitatively* measure the usefulness of trace visualization techniques in practice, e.g., through controlled experiments. Moreover, the evaluations in existing work rarely involve broad spectra of comprehension tasks, making it difficult to judge whether the associated solutions are widely applicable in daily practice. Lastly, most existing approaches involve *traditional* visualizations, i.e., they rely on UML, graph, or tree notations, to which presumably most software engineers are accustomed [9]. By contrast, EXTRAVIS uses non-traditional visualization techniques, and Storey argues [46] that advanced visual interfaces are not often used in development environments because they tend to require complex user interactions.

[2] EXTRAVIS, http://swerl.tudelft.nl/extravis



General Settings
Linearization: 11%
Max. Spl. Width: 7,000
Basis of Colorization
● Caller-to-Callee Direction
○ Occurence in Time
Time Index Col. Interpolation
○ HSV Short
● HSV Long
○ RGB
Time Index to Use
● First Occurence
○ Average Occurence
○ Last Occurence
Max. Hover Time: 100ms
MSC Rel. Visibility: 3%
Rel. Ext. Length: 100%
☑ Show Self-Calls
☑ Show Context
☐ Relations above Hierarchy
☑ Show Interaction Only
☐ Use Runtime Tooltips
☐ Show Tooltips
Memory usage: 96,32MB
Pos: 0
Size: 266153

Fig. 1. EXTRAVIS' circular bundle view and massive sequence view.

These reasons have motivated us to empirically validate EXTRAVIS through a controlled experiment in which we seek to assess its added value in concrete maintenance contexts.

## 3 EXPERIMENTAL DESIGN

The purpose of this paper is to provide a quantitative evaluation of trace visualization for program comprehension. To this end, we define a series of typical comprehension tasks and measure EXTRAVIS' added value to a traditional programming environment: in this case, the Eclipse IDE[3]. Similar to related efforts (e.g., [47], [48]) we maintain a distinction between the *time spent* on the tasks and the *correctness* of the answers given. Furthermore, we seek to identify the types of tasks to which the use of EXTRAVIS, and trace visualization in general, is the most beneficial.

### 3.1 Research Questions and Hypotheses

Based on our earlier case studies, we distinguish the following research questions:

1) Does the availability of EXTRAVIS reduce the *time* that is needed to complete typical comprehension tasks?
2) Does the availability of EXTRAVIS increase the *correctness* of the solutions given during those tasks?
3) Based on the answers to these research questions, which *types* of tasks can we identify that benefit most from the use of EXTRAVIS and from trace visualization in general?

Associated with the first two research questions are two null hypotheses, which we formulate as follows:

- $H1_0$: The availability of EXTRAVIS does not impact the time needed to complete typical comprehension tasks.
- $H2_0$: The availability of EXTRAVIS does not impact the correctness of solutions given during those tasks.

The alternative hypotheses that we use in the experiment are the following:

- $H1$: The availability of EXTRAVIS reduces the time needed to complete typical comprehension tasks.
- $H2$: The availability of EXTRAVIS increases the correctness of solutions given during those tasks.

The rationale behind the first alternative hypothesis is the fact that EXTRAVIS provides a broad overview of the subject system on one single screen, which may guide the user to his or her goal more quickly than if switching between source files were required.

---

[3] Eclipse IDE, http://www.eclipse.org

The second alternative hypothesis is motivated by the inherent precision of dynamic analysis with respect to actual program behavior: for example, the resolution of late binding may result in a more detailed understanding and therefore produce more accurate solutions.

To test hypotheses $H1_0$ and $H2_0$, we define a series of comprehension tasks that are to be addressed by both a control group and an experimental group. The difference in treatment between these groups is that the former group uses a traditional development environment (the "Eclipse" group), whereas the latter group also has access to EXTRAVIS (the "Ecl+Ext" group). We maintain a between-subjects design, meaning that each subject is either in the control group or in the experimental group.

Sections 3.2 through 3.7 provide a detailed description of the experiment.

### 3.2 Object

The system that is to be comprehended by the subject groups is CHECKSTYLE, a tool that employs "checks" to verify if source code adheres to specific coding standards. Our choice for CHECKSTYLE as the object of this experiment is motivated by the following factors:

- CHECKSTYLE is open source, which helps to make the results of our experiments reproducible.
- CHECKSTYLE comprises 310 classes distributed across 21 packages, containing a total of 57 KLOC.[4] This makes it tractable for an experimental session, yet adequately representative of real life programs.
- It is written in Java, with which many potential subjects are sufficiently familiar.
- It addresses an application domain (adherence to coding standards) that will be understandable for most potential subjects.
- The authors of this paper are familiar with its internals as a result of earlier experiments [49], [50], [7]. Furthermore, the lead developer is available for feedback.

To obtain the necessary trace data for EXTRAVIS, we instrument CHECKSTYLE and execute it according to two scenarios. Both involve typical runs with a small input source file, and only differ in terms of the input configuration, which in one case specifies 64 types of checks whereas the other specifies only six. The resulting traces contain 31,260 and 17,126 calls, respectively, which makes them too large to be comprehended in limited time without tool support.

Analyzing the cost of creating these traces is not included in the experiment, as it is our prime objective to analyze whether the availability of trace information is beneficial during the program comprehension process. In practice, we suspect that execution traces will likely be obtained from test cases – a route we also explored in our earlier work [35].

TABLE 2
Pacione's nine principal comprehension activities

| Activity | Description |
|---|---|
| A1 | Investigating the functionality of (a part of) the system |
| A2 | Adding to or changing the system's functionality |
| A3 | Investigating the internal structure of an artifact |
| A4 | Investigating dependencies between artifacts |
| A5 | Investigating run-time interactions in the system |
| A6 | Investigating how much an artifact is used |
| A7 | Investigating patterns in the system's execution |
| A8 | Assessing the quality of the system's design |
| A9 | Understanding the domain of the system |

### 3.3 Task design

With respect to the comprehension tasks that are to be tackled during the experiment, we maintain two important criteria: (1) they should be representative of real maintenance contexts, and (2) they should not be biased towards either Eclipse or EXTRAVIS.

To this end, we apply the comprehension framework from Pacione et al. [51], who argue that *"a set of typical software comprehension tasks should seek to encapsulate the principal activities typically performed during real world software comprehension"*. They have studied several sets of tasks used in software visualization and comprehension evaluation literature and classified them according to nine principal activities, representing both general and specific reverse engineering tasks and covering both static and dynamic information (Table 2). Particularly the latter aspect significantly reduces biases towards either of the two tools used in this experiment.

Using these principal activities as a basis, we propose eight representative tasks that highlight many of CHECKSTYLE's aspects at both high and low abstraction levels. Table 3 provides descriptions of the tasks and shows how each of the nine activities from Pacione et al. is covered by at least one task.[5] For example, activity A1, *"Investigating the functionality of (part of) the system"*, is covered by tasks T1, T3.1, T4.1, and T4.2; and activity A4, *"Investigating dependencies between artifacts"*, is covered by tasks T2.1, T2.2, T3.2, and T3.3.

To render the tasks more representative of real maintenance situations, tasks are given as open rather than multiple-choice questions, making it harder for respondents to resort to guessing. Per answer, 0–4 points can be earned. Points are awarded by the evaluators, in our case the first two authors. A solution model is available [52], which was reviewed by CHECKSTYLE's lead developer. To ensure uniform grading among the two evaluators, the answers from five random subjects are first graded by both evaluators.

### 3.4 Subjects

The subjects in this experiment are fourteen Ph.D. candidates, nine M.Sc. students, three postdocs, two profes-

---

[4] Measured using `sloccount` by David A. Wheeler, http://sourceforge.net/projects/sloccount/.

[5] Table 3 only contains the actual questions; the subjects were also given contextual information (such as definitions of fan-in and coupling) which can be found in the technical report [52].

TABLE 3
Descriptions of the comprehension tasks

| Task | Activities | Description |
|---|---|---|
| | | *Context: Gaining a general understanding.* |
| T1 | A{1,7,9} | Having glanced through the available information for several minutes, which do you think are the main stages in a typical (non-GUI) Checkstyle scenario? Formulate your answer from a high-level perspective: refrain from using identifier names and stick to a maximum of six main stages. |
| | | *Context: Identifying refactoring opportunities.* |
| T2.1 | A{4,8} | Name three classes in Checkstyle that have a high fan-in and (almost) no fan-out. |
| T2.2 | A{4,8} | Name a class in the top-level package that could be a candidate for movement to the `api` package because of its tight coupling with classes therein. |
| | | *Context: Understanding the checking process.* |
| T3.1 | A{1,2,5,6} | In general terms, describe the life cycle of the `checks.whitespace.TabCharacterCheck` during execution: when is it created, what does it do and on whose command, and how does it end up? |
| T3.2 | A{3,4,5} | List the identifiers of all method/constructor calls that typically occur between `TreeWalker` and a `TabCharacterCheck` instance, and the order in which they are called. Make sure you also take inherited methods/constructors into account. |
| T3.3 | A{3,4,5,9} | In comparison to the calls listed in Task T3.2., which additional calls occur between `TreeWalker` and `checks.coding.IllegalInstantiationCheck`? Can you think of a reason for the difference? |
| | | *Context: Understanding the violation reporting process.* |
| T4.1 | A{1,3} | How is the check's warning handled, i.e., where/how does it originate, how is it internally represented, and how is it ultimately communicated to the user? |
| T4.2 | A{1,5} | Given `Simple.java` as the input source and `many_checks.xml` as the configuration, does `checks.whitespace.WhitespaceAfterCheck` report warnings? Specify how your answer was obtained. |

sors, and six participants from industry. The resulting group thus consists of 34 subjects, and is quite heterogeneous in that it represents 8 different nationalities, and M.Sc. degrees from 16 universities. The M.Sc. students are in the final stages of their computer science studies, and the Ph.D. candidates represent different areas of software engineering, ranging from software inspection to model-based fault diagnosis. Our choice of subjects partly mitigates concerns from Di Penta et al., who argue that *"a subject group made up entirely of students might not adequately represent the intended user population"* [53]. All subjects participate on a voluntary basis and can therefore be assumed to be properly motivated. None of them have prior experience with EXTRAVIS.

To partition the subjects, we distinguish five fields of expertise that can strongly influence the individual performance. They represent variables that are to be controlled during the experiment, and concern knowledge of Java, Eclipse, reverse engineering, CHECKSTYLE, and language technology (i.e., CHECKSTYLE's domain). The subjects' levels of expertise in each of these fields are measured through a (subjective) a priori assessment: we use a five-point Likert scale, from 0 (*"no knowledge"*) to 4 (*"expert"*). In particular, we require minimum scores of 1 for Java and Eclipse (*"beginner"*), and a maximum score of 3 for CHECKSTYLE (*"advanced"*). The technical report provides a characterization of the subjects.

The assignments to the control and experimental group are done by hand to evenly distribute the available knowledge. The result is illustrated by Figure 2: in each group, the expertise is chosen to be as similar as possible, resulting in an average expertise of 2.12 in both groups.

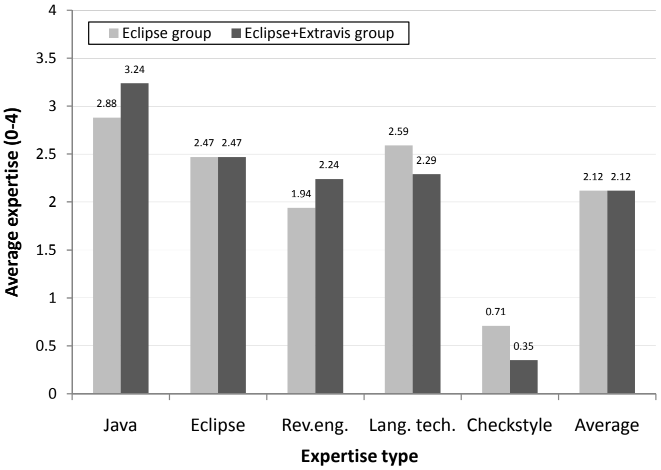


Fig. 2. Average expertise of the subject groups.

### 3.5 Experimental procedure

The experiment is performed through a dozen sessions, most of which take place at the university. Sessions with industrial subjects take place at their premises, in our case the Software Improvement Group,[6] the industrial partner in our project. The sessions are conducted on workstations with characteristics that were as similar as possible, i.e., at least Pentium 4 processors and comparable screen resolutions (1280×1024 or 1600×900). Given the different locations (university and in-house at company) fully equivalent setups were impossible to achieve.

Each session involves at most three subjects and

---

[6] Software Improvement Group, http://www.sig.eu

features a short tutorial on Eclipse, highlighting the most common features. The experimental group is also given a ten minute EXTRAVIS tutorial that involves a JHOTDRAW execution trace used in earlier experiments [7]. All sessions are supervised, enabling the subjects to pose clarification questions, and preventing them from consulting others and from using alternative tools. The subjects are not familiar with the experimental goal.

The subjects are presented with a fully configured Eclipse that is readily usable, and are given access to the example input source file and CHECKSTYLE configurations (see Section 3.2). The Ecl+Ext group is also provided with EXTRAVIS instances for each of the two execution traces mentioned earlier. All subjects receive handouts that provide an introduction, CHECKSTYLE outputs for the two aforementioned scenarios, the assignment, a debriefing questionnaire, and reference charts for both Eclipse and EXTRAVIS. The assignment is to complete the eight comprehension tasks within 90 minutes. The subjects are required to motivate their answers at all times. We purposely refrain from influencing how exactly the subjects should cope with the time limit: only when a subject exceeds the time limit is he or she told that finishing up is, in fact, allowed. Finally, the questionnaire asks for the subjects' opinions on such aspects as time pressure and task difficulty.

### 3.6 Variables & Analysis

The independent variable in our experiment is the availability of EXTRAVIS during the tasks.

The first dependent variable is the *time spent* on each task, and is measured by having the subjects write down the current time when starting a new task. Since going back to earlier tasks is not allowed and the sessions are supervised, the time spent on each task can be easily reconstructed.

The second dependent variable is the *correctness* of the given solutions. This is measured by applying our solution model to the subjects' solutions, which specifies the required elements and the associated scores.

To test our hypotheses, we first test whether the sample distributions are normal (via a Kolmogorov-Smirnov test) and whether they have equal variances (via Levene's test). If these tests pass, we use the parametric Student's t-test to evaluate our hypotheses; otherwise we use the (more robust, but weaker) non-parametric Mann-Whitney test.

Following our alternative hypotheses, we employ the one-tailed variant of each statistical test. For the time as well as the correctness variable we maintain a typical confidence level of 95% ($\alpha$=0.05). The statistical package that we use for our calculations is SPSS.

### 3.7 Pilot studies

Prior to the experimental sessions, we conduct two pilots to optimize several experimental parameters, such as the number of tasks, their clarity, feasibility, and the time limit. The pilot for the control group is performed by an author of this paper who had initially not been involved in the experimental design. The pilot for the experimental group is conducted by an outsider. Both would not take part in the actual experiment later on.

The results of the pilots led to the removal of two tasks because the time limit was too strict. The removed tasks were already taken into account in Section 3.2. Furthermore, the studies led to the refinement of several tasks in order to make the questions clearer. Other than these unclarities, the tasks were found to be sufficiently feasible in both the Eclipse and the Ecl+Ext pilot.

TABLE 4
Descriptive statistics of the experimental results

| | **Time** | | **Correctness** | |
|---|---|---|---|---|
| | *Eclipse* | *Ecl+Ext* | *Eclipse* | *Ecl+Ext* |
| mean | 77.00 | 59.94 | 12.47 | 17.88 |
| difference | | -22.16% | | +43.38% |
| min | 38 | 36 | 5 | 11 |
| max | 102 | 72 | 22 | 22 |
| median | 79 | 66 | 14 | 18 |
| stdev. | 18.08 | 12.78 | 4.54 | 3.24 |
| **one-tailed Student's t-test** | | | | |
| Kolmogorov-Smirnov *Z* | 0.606 | 0.996 | 0.665 | 0.909 |
| Levene *F* | | 1.370 | | 2.630 |
| df | | 32 | | 32 |
| t | | 3.177 | | 4.000 |
| p-value | | 0.002 | | <0.001 |

## 4 RESULTS

Table 4 shows descriptive statistics of the measurements, aggregated over all tasks. The technical report provides a full listing of the measurements and debriefing questionnaire results.

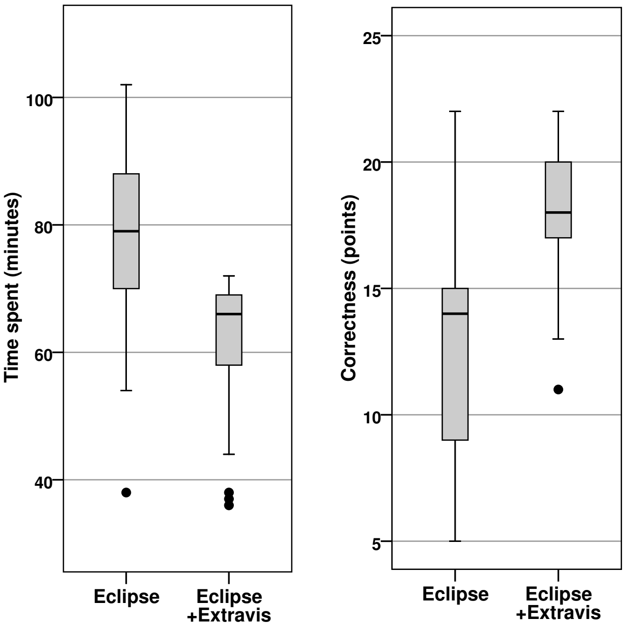
Wohlin et al. [54] suggest the removal of *outliers* in case of extraordinary situations, such as external events that are unlikely to reoccur. We found four outliers in our timing data and one more in the correctness data, but could identify no such circumstances and have therefore opted to retain those data points.

As an important factor for both time and correctness, we note that two subjects decided to stop after 90 minutes with two tasks remaining, and one subject stopped with one task remaining, resulting in ten missing data points in this experiment (i.e., the time spent by three subjects on task T4.2 and by two subjects on task T4.1, as well as the correctness of the solutions involved). Nine others finished all tasks, but only after the 90 minutes had expired: eight subjects from the Eclipse group and one subject from the Ecl+Ext group spent between 95 and 124 minutes. The remaining 22 participants finished all eight tasks on time.[7]

In light of the missing data points, we have chosen to disregard the last two tasks in our quantitative analyses.

---

[7] Related studies point out that it is not uncommon for several tasks to remain unfinished during the actual experiments (e.g., [48] and [40]).



(a) (b)

Fig. 3. Box plots for time spent and correctness.

Not taking tasks T4.1 and T4.2 into account, only three out of the 34 subjects still exceeded the time limit (by 6, 7 and 12 minutes, respectively). This approach also reduces any ceiling effects in our data that may have resulted from the increasing time pressure near the end of the assignment. The remaining six tasks still cover all of Pacione's nine activities (Table 3).

### 4.1 Time results

We start off by testing null hypothesis $H1_0$, which states that the availability of EXTRAVIS does not impact the time needed to complete typical comprehension tasks.

Figure 3(a) shows a box plot for the total time that the subjects spent on the first six tasks. Table 4 indicates that on average the Ecl+Ext group required 22.16% less time.

The Kolmogorov-Smirnov and Levene tests succeeded for the timing data, which means that Student's t-test may be used to test $H1_0$. As shown in Table 4, the t-test yields a statistically significant result. The average time spent by the Ecl+Ext group was clearly lower and the p-value 0.002 is smaller than 0.05, which means that $H1_0$ can be rejected in favor of the alternative hypothesis H1, stating that the availability of EXTRAVIS reduces the time that is needed to complete typical comprehension tasks.

### 4.2 Correctness results

We next test null hypothesis $H2_0$, which states that the availability of EXTRAVIS does not impact the correctness of solutions given during typical comprehension tasks.

Figure 3(b) shows a box plot for the scores that were obtained by the subjects on the first six tasks. Note that we consider overall scores rather than scores per task (which are left to Section 5.3). The box plot shows that the difference in terms of correctness is even more explicit than for the timing aspect. The solutions given by the Ecl+Ext subjects were 43.38% more accurate (Table 4), averaging 17.88 out of 24 points compared to 12.47 points for the Eclipse group.

Similar to the timing data, the requirements for the use of the parametric t-test were met. Table 4 therefore shows the results for Student's t-test. At less than 0.001, the p-value implies statistical significance, meaning that $H2_0$ can be rejected in favor of our alternative hypothesis H2, stating that the availability of EXTRAVIS increases the correctness of solutions given during typical comprehension tasks.

## 5 DISCUSSION

### 5.1 Reasons for different time requirements

The lower time requirements for the EXTRAVIS users can be attributed to several factors. First, all information offered by EXTRAVIS is shown on a single screen, which eliminates the need for scrolling. In particular, the overview of the entire system's structure saves much time in comparison to conventional environments, in which typically multiple files have to be studied at once. Second, the need to imagine how certain functionalities or interactions work at run-time represents a substantial cognitive load on the part of the user. This is alleviated by trace analysis and visualization tools, which show the actual run-time behavior. Examples of these assumptions will be discussed in Section 5.3.

On the other hand, several factors may have had a negative impact on the the time requirements of EXTRAVIS users. For example, the fact that EXTRAVIS is a standalone tool means that context switching is necessary, which may yield a certain amount of overhead on the part of the user. This could be solved by integrating the trace visualization technique into Eclipse (or other IDEs), with the additional benefit that the tool could provide direct links to Eclipse's source code browser. However, it should be noted that EXTRAVIS would still require a substantial amount of screen real estate to be used effectively.

Another potential factor that could have hindered the time performance of the Ecl+Ext group is that these subjects may not have been sufficiently familiar with EXTRAVIS' features, and were therefore faced with a time-consuming learning curve. This is partly supported by the debriefing questionnaire, which indicates that five out of the seventeen subjects found the tutorial too short. A more elaborate tutorial on the use of the tool could help alleviate this issue.

### 5.2 Reasons for correctness differences

We attribute the added value of EXTRAVIS to correctness to several factors. A first one is the inherent precision of dynamic analysis: the fact that EXTRAVIS shows the actual objects involved in each call makes the interactions

TABLE 5
Debriefing questionnaire results

| | *Eclipse* | | *Ecl+Ext* | |
|---|---|---|---|---|
| | **mean** | **stdev.** | **mean** | **stdev.** |
| **Miscellaneous** | | | | |
| Perceived time pressure (0-4) | 2.18 | 1.19 | 2.06 | 0.66 |
| Knowledge of dynamic analysis (0-4) | 2.26 | 1.22 | 2.53 | 1.12 |
| **Perceived task difficulty (0-4)** | | | | |
| T1 | 1.00 | 0.71 | 1.65 | 0.79 |
| T2.1 | 2.59 | 1.18 | 1.18 | 0.64 |
| T2.2 | 2.24 | 1.15 | 1.53 | 0.80 |
| T3.1 | 2.12 | 0.78 | 2.12 | 0.70 |
| T3.2 | 2.29 | 0.92 | 1.53 | 0.72 |
| T3.3 | 2.18 | 0.95 | 1.47 | 0.94 |
| T4.1 | 2.40 | 0.63 | 2.65 | 0.86 |
| T4.2 | 1.53 | 0.92 | 1.63 | 1.02 |
| Average | 2.04 | | 1.72 | |
| **Use of EXTRAVIS** | | | | |
| No. of features used | | | 7.12 | 2.67 |
| No. of tasks conducted w/ tool | | | 7.00 | 1.06 |
| No. of tasks successfully conducted w/ tool | | | 6.00 | 1.55 |
| Est. % of time spent in tool | | | 70.00 | 24.99 |
| Perceived tool speed (0-2) | | | 1.35 | 0.49 |

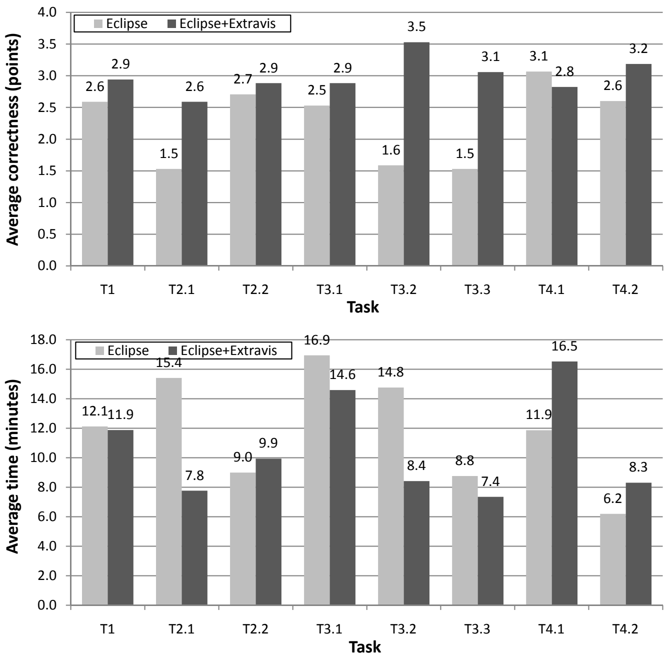


Fig. 4. Averages per task.

easier to understand. Section 5.3 discusses this in more detail through an example task.

Second, the results of the debriefing questionnaire (Table 5) show that the Ecl+Ext group used EXTRAVIS quite often: the subjects estimate the percentage of time they spent in EXTRAVIS at 70% on average. In itself, this percentage is meaningless: for example, in a related study it was observed that *"heavy use of a feature does not necessarily mean it (or the tool) helps to solve a task"*, and that *"repeated use may actually be a sign of frustration on the part of the user"* [40]. However, the questionnaire also shows that EXTRAVIS was used on seven of the eight tasks on average and that the tool was actually found useful in six of those tasks (86%). This is a strong indication that the Ecl+Ext subjects generally did not experience a resistance to using EXTRAVIS (resulting from, e.g., a poor understanding of the tool) *and* were quite successful in their attempts.

The latter assumption is further reinforced by the Ecl+Ext subjects' opinions on the speed and responsiveness of the tool, averaging a score of 1.35 on a scale of 0-2, which is between *"pretty OK: occasionally had to wait for information"* and *"very quickly: the information was shown instantly"*. Furthermore, all 34 subjects turned out to be quite familiar with dynamic analysis: in the questionnaire they indicated an average knowledge level of 2.3 on a scale of 0-4 on this topic, which is between *"I'm familiar with it and can name one or two benefits"* and *"I know it quite well and performed it once or twice"*.

As a side note, in a related study [48], no correlation could be identified between the subjects' experience levels and their performance. While in our experiment the same holds for the Ecl+Ext group and for correctness in the Eclipse group, there *does* exist a negative correlation between expertise and the time effort in the latter group: a high average expertise yielded lower time requirements, and vice versa. This observation partly underlines the importance of an adequate selection procedure when recruiting subjects for software engineering experiments.

## 5.3 Individual task performance

To address our third research question, whether there are certain types of comprehension tasks that benefit most from the use of EXTRAVIS (see Section 3.1) we examine the performance per task in more detail. Figure 4 shows the average scores and time spent by each group from a task perspective.

While the experiment concerned only eight tasks, our data does suggest a negative correlation between time spent and correctness, in the sense that relatively little effort and a relatively high score (and vice versa) often go hand in hand.

### Task T1

The goal of the first task was to identify and globally understand the most prominent stages in a typical CHECKSTYLE scenario (Table 3). The groups scored equally well on this task and required similar amounts of time. According to the motivations of their solutions, the Eclipse group typically studied the `main()` method: however, such important phases as the building and parsing of an AST were often missing because they are not directly visible at the `main()` level. On the other hand, the EXTRAVIS users mostly studied an actual execution scenario through the massive sequence view, which proved quite effective and led to slightly more accurate solutions.

### Task T2.1

Task T2.1 concerned a fan-in/fan-out analysis that turned out to be significantly easier for the Ecl+Ext group, who scored 1.1 more points and needed only half the time. This is presumably explained by EXTRAVIS' circular view, from which *all* classes and their interrelationships can be directly interpreted. The Eclipse group mostly carried out a manual search for utility-like classes, opening numerous source files in the process, which is time-consuming and does not necessarily yield optimal results.

**Task T2.2**

This task was similar to the previous one, except that the focus was more on coupling. While there still exists a performance difference, it is much smaller this time round. According to the given solutions, the Ecl+Ext group again resorted to the circular view to look for high edge concentrations, while the Eclipse group mostly went searching for specific imports. The fact that a more specific (and automated) search was possible in this case may account for the improved performance of the latter group.

**Task T3.1**

Task T3.1 asked the participants to study a certain check to understand its life cycle, from creation to destruction. The performance difference here was quite subtle, with the Ecl+Ext group apparently having had a small advantage. Eclipse users typically studied the check's source code and started a more broad investigation from there. EXTRAVIS users mostly used our tool to highlight the check in the given execution trace and examine the interactions that were found. Interestingly, only a handful of subjects discovered that the checks are in fact dynamically loaded, and both groups often missed the explicit destruction of each check at the end of execution, which is not easily observed in Eclipse nor in EXTRAVIS.

**Task T3.2**

The goal of this follow-up task was to understand the protocol between a check and a certain key class, and asked the subjects to provide a list of interactions between these classes. The fact that the check at hand is an extension of a superclass that is an extension in itself, forced the Eclipse group to distribute its focus across each and every class in the check's type hierarchy. EXTRAVIS users often highlighted the mutual interactions of the two classes at hand in the tool. As suggested by Figure 4, the latter approach is both faster and much more accurate (as there is a smaller chance of calls being missed).

**Task T3.3**

This task was similar to the previous one, except that it revolved around another type of check. The difference is that this check is dependent on the AST of the input source file, whereas the check in task T3.2 operates directly on the file. Finding the additional interactions was not too difficult for the EXTRAVIS users, who could follow a similar routine to last time. On the other hand, in Eclipse the subtle differences were often overlooked, especially if it was not understood that (and why) this check is fundamentally different from the previous one.

**Task T4.1**

Task T4.1 posed the challenging question of how CHECKSTYLE's error handling mechanism is implemented. It is the only task on which the Ecl+Ext group was clearly outperformed in terms of both time and correctness. The Eclipse group rated the difficulty of this task at 2.4, which is between *"intermediate"* and *"difficult"*, whereas EXTRAVIS users rated the difficulty of this task at 2.65, leaning toward *"difficult"*. An important reason might be that EXTRAVIS users did not know exactly *what to look for* in the execution trace, because the question was rather abstract in the sense that no clear starting point was given. On the other hand, the Eclipse group mostly used one of the checks as a baseline and followed the error propagation process from there. The latter approach is typically faster: the availability of EXTRAVIS may have been a distraction rather than an added value in this case.

**Task T4.2**

The focus in the final task was on testing the behavior of a check: given that a new check has been written and an input source file is available, how can we test if it works correctly? The Ecl+Ext group often searched the execution traces for communication between the check and the violation container class, which is quite effective once that class has been found. The Eclipse group had several choices. A few subjects tried to understand the check and apply this knowledge on the given input source file, i.e., understand which items the check is looking for, and then verify if these items occur in the input source file. Others tried to relate the check's typical warning message (once it was determined) to example outputs given in the handouts; yet others used the Eclipse debugger, e.g., by inserting breakpoints or print statements in the error handling mechanism. With the exception of debugging, most of the latter approaches are quite time-consuming, if successful at all. Still, we observe no large difference in time spent: the fact that eight members of the Eclipse group had already exceeded the time limit at this point may have caused them to hurry, thereby reducing not only the time effort but also the score.

**Summary**

Following our interpretation of the individual task performance, we now formulate an analytical generalization [55] based on the quantitative results discussed earlier, the debriefing questionnaire results, and the four case studies from our earlier work [7].

*Global structural insight.* From the results of tasks T2.1 and T2.2 it has become clear that EXTRAVIS' circular view is of great help in grasping the structural relationships of the subject system. In particular, the bundling feature ensures that the many relations can all be shown simultaneously on a single screen. This poses a great advantage to using a standard IDE, in which it often involves browsing through multiple files when a high-level structural insight is required. While any trace visualization technique could be helpful for such tasks, it should provide some means of visualizing the system's structural decomposition (e.g., UML sequence diagrams with hierarchically ordered lifelines [56]).

*Global behavioral insight.* In addition to structural insight, EXTRAVIS provides a navigable overview of an entire execution trace through the massive sequence view. As illustrated in earlier case studies and in task T1, this view visualizes the trace such that patterns can be visually distinguished. These patterns correspond to execution phases, the identification of which can be quite helpful in decomposing the subject system's behavior into smaller, more tractable pieces of functionality. In the case of CHECKSTYLE, this approach turned out to reveal more accurate information than could be derived from examining the `main()` method. A trace visualization technique must include some sort of navigable overview for it to be useful for such tasks.

*Detailed behavioral insight.* One of the main benefits of dynamic analysis is that occurrences of late binding are resolved, i.e., the maintainer can observe the actual objects involved in an execution scenario. This contributes to a more detailed understanding of a program's behavior. This is illustrated by tasks T3.2 and T3.3, which proved quite difficult for the Eclipse group as these tasks concerned the identification of inherited methods, which are difficult to track down unless some form of run-time analysis is possible. We expect this particular advantage of dynamic analysis to be exploitable by any trace visualization technique.

*Goal-oriented strategy.* Trace visualization is not always the best solution: the results for task T4.1 showed a clear advantage for the Eclipse group. We believe that the reason can be generalized as follows: dynamic analysis typically involves a goal-oriented strategy, in the sense that one must know what to look for. (This follows from the fact that an appropriate execution scenario must be chosen.) If such a strategy is not feasible, e.g., because there is no clear starting point (such as the name of a certain class), then a strong reliance on dynamic analysis will result in mere confusion, which means that one must resort to traditional solutions such as the IDE instead.

### 5.4 Related experiments

There exist no earlier studies in the literature that offer quantitative evidence of the added value of trace visualization techniques for program comprehension. We therefore describe the experiments that are most closely related to our topic.

The aforementioned article from Bennett et al. concerned a user study involving a sequence diagram reconstruction tool [40]. Rather than measure its added value for program comprehension, they sought to characterize the *manner* in which the tool is used in practice. To this end, they had six subjects perform a series of comprehension tasks, and measured when and how the tool features were used. Among their findings was that tool features are not often formally evaluated in literature, and that heavily used tool features may indicate confusion among the users. Another important observation was that much time was spent on *scrolling*, which supports our hypothesis that EXTRAVIS saves time as it shows all information on a single screen.

Quante performed a controlled experiment to assess the benefits of Dynamic Object Process Graphs (DOPGs) for program comprehension [48]. While these graphs are built from run-time data, they do not actually visualize execution traces. The experiment involved 25 students and a series of feature location tasks for two subject systems. The use of DOPGs by his experimental group led to a significant decrease in time and a significant increase in correctness in case of the first system; however, the differences in case of the second system were *not* statistically significant. This suggests that evaluations on additional systems are also desirable for EXTRAVIS and should be considered as future work. Also of interest is that the latter subject system was four times smaller than the former, but had three DOPGs associated with it instead of one. This may have resulted in an information overload on the part of the user, once more suggesting that users are best served by as little information as possible.

Among the contributions by Hamou-Lhadj and Lethbridge are encouraging quantitative results with respect to their trace summarization algorithm, effectively reducing large traces to as little as 0.5% of the original size [4]. However, the measurements performed relate to the effectiveness of the algorithm in terms of *reduction power*, rather than its added value in actual comprehension tasks.

## 6 THREATS TO VALIDITY

This section discusses the validity threats in our experiment and the manners in which we have addressed them. We have identified three types of validity threats: (1) internal validity, referring to the cause-effect inferences made during the analysis; (2) external validity, concerning the generalizability of the results to different contexts; and (3) construct validity, seeking agreement between a theoretical concept and a specific measuring procedure.

### 6.1 Internal validity

**Subjects.** There exist several internal validity threats that relate to the subjects used in this experiment. First of all, the subjects may not have been sufficiently competent. We have reduced this threat through the a priori assessment of the subjects' competence in five relevant fields, which pointed out that all subjects had at least an elementary knowledge of Eclipse (2.47 in Figure 2) and no expert knowledge of CHECKSTYLE. Furthermore, participants could ask questions on both tools during the experiments, and a quick reference chart was available.

Second, their knowledge may not have been fairly distributed across the control group and experimental group. This threat was alleviated by grouping the subjects such that their expertise was evenly distributed across the groups (Figure 2).

Third, the subjects may not have been properly motivated, or may have had too much knowledge of the experimental goal. The former threat is mitigated by the fact that they all participated on a voluntary basis; as for the latter, the subjects were not familiar with the actual research questions or hypotheses (although they may have guessed).

**Tasks.** The comprehension tasks were designed by the authors of this paper, and therefore may have been biased toward EXTRAVIS (as this tool was also designed by the authors). To avoid this threat, we have applied an established task framework [51] to ensure that many aspects of typical comprehension contexts are covered. As a result, the tasks concerned both global and detailed knowledge, and both static and dynamic aspects.

Another task-related threat is that the tasks may have been too difficult. We refute this possibility on the basis of the correctness results, which show that maximum scores were occasionally awarded in both groups for all but one task (T3.1), which in the Eclipse group often yielded 3 points but never 4. However, the average scores for this task were a decent 2.53 (stdev. 0.51) and 2.88 (stdev. 0.86) in the Eclipse group and Ecl+Ext group, respectively. This point of view is further reinforced by the subjects' opinions on the task difficulties: the task they found hardest (T4.1) yielded good scores, being 3.07 (stdev. 1.10) for the Eclipse group and 2.82 (stdev. 0.81) for the Eclipse+Extravis group.

Also related to the tasks is the possibility that the subjects' solutions were graded incorrectly. This threat was reduced in our experiment by creating concept solutions in advance and by having CHECKSTYLE's lead developer review and refine them. This resulted in a solution model that clearly states the required elements (and corresponding points) for each task. Furthermore, to verify the soundness of the reviewing process, the first two authors of this paper independently reviewed the solutions of five random subjects: on each of the five occasions the difference was no higher than one point (out of the maximum of 32 points), suggesting a high inter-rater reliability.

**Miscellaneous.** The results may have been influenced by time constraints that were too loose or too strict. We have attempted to circumvent this threat by performing two pilot studies, which led to the removal of two tasks. Still, not all subjects finished the tasks in time, but the average time pressure (as indicated by the subjects in the debriefing questionnaire) was found to be 2.18 (stdev. 1.19) in the Eclipse group and 2.06 (stdev. 0.66) in the Ecl+Ext group on a scale of 0-4, which roughly corresponds to only a *"fair amount of time pressure"*. Also, in our results analysis we have disregarded the last two tasks, upon which only three out of the 34 subjects still exceeded the time limit.

As several test subjects did not finish tasks T4.1 and T4.2 (within time), we decided to eliminate these tasks from the analysis of our results. This removal may have benefited the EXTRAVIS results because task T4.1 is one of the few tasks at which the Eclipse group outperformed the EXTRAVIS users. Fortunately, with EXTRAVIS shown to be 43% more accurate and 21% less time-consuming, the conclusion that EXTRAVIS constitutes a significant added value for program comprehension would likely still be valid if tasks T4.1 and T4.2 were taken into account. Future refinements of the experimental design should examine optimizations of the time limit policy.

The two execution traces that we provided to the experimental group for use in EXTRAVIS are relatively small, containing 31,260 and 17,126 calls respectively. The fact that these traces are relatively small might influence the usability of EXTRAVIS: in particular, large traces could render EXTRAVIS a little less responsive and therefore a bit more time-consuming to use. However, earlier case studies [7] that we performed with EXTRAVIS (involving much larger traces) lead us to believe that the usability impact of using larger traces is probably minor.

Furthermore, our statistical analysis may not be completely accurate due to the missing data points that we mentioned in Section 4. This concerned two subjects who did not finish the last two tasks and one subject who did not finish the last task. Fortunately, the effect of the missing timing and correctness data points on our calculations is negligible: had the subjects finished the tasks, their total time spent and average score could have been higher, but this would only have affected the analysis of all eight tasks whereas our focus has been on the first six.

Another validity threat could be the fact that the control group only had access to the Eclipse IDE, whereas the experimental group also received two execution traces (next to Eclipse and the EXTRAVIS tool). However, we believe that the Eclipse group would not have benefited from the availability of execution traces because they are too large to be navigated without any tool support.

Lastly, it could be suggested that Eclipse is more powerful if additional plugins are used. However, as evidenced by the results of the debriefing questionnaire, only two subjects named specific plugins that would have made the tasks easier, and these related to only two of the eight tasks. We therefore expect that additional plugins would not have had a significant impact.

### 6.2 External validity

The generalizability of our results could be hampered by the limited representativeness of the subjects, the tasks, and CHECKSTYLE as a subject system.

Concerning the subjects, the use of professional developers instead of (mainly) Ph.D. candidates and M.Sc. students could have yielded different results. Unfortunately, motivating people from industry to sacrifice two hours of their precious time is quite difficult. Nevertheless, against the background of related studies that often employ undergraduate students, we assume the expertise levels of our 34 subjects to be relatively high. This assumption is partly reinforced by the (subjective) a priori assessment, in which the subjects rated themselves as being *"advanced"* with Java (avg. 3.06, stdev. 0.65), and *"regular"* at using Eclipse (avg. 2.47, stdev. 0.90). We acknowledge that our subjects' knowledge of dynamic analysis may have been greater than in industry, averaging 2.26 (Table 5).

Another external validity threat concerns the comprehension tasks, which may not reflect real maintenance situations. We tried to neutralize this threat by relying on Pacione's framework [51], which is based on activities often found in software visualization and the comprehension evaluation literature. The resulting tasks were reasonably complicated: Both groups encountered a task of which they rated the difficulty between 2.5 and 3.0, roughly corresponding to "difficult" (See the debriefing questionnaire results in Table 5). Furthermore, they also included an element of "surprise": Task 3.1, for example, required the subjects to describe the life cycle of a given object, which made the majority of subjects enter in a fruitless search for its constructor, whereas the object was in fact dynamically loaded. Last but not least, the tasks concerned open questions, which approximate real life contexts better than multiple choice questions do. Nevertheless, arriving at a representative set of tasks that is suitable for use in experiments by different researchers is a significant challenge, which warrants further research.

Finally, the use of a different subject system (or additional runs) may have yielded different or more reliable results. CHECKSTYLE was chosen on the basis of several important criteria: in particular, finding another system of which the experimenters have sufficient knowledge is not trivial. Moreover, an additional case (or additional run) imposes twice the burden on the subjects or requires more of them. While this may be feasible in case the groups consist exclusively of students, it is not realistic in case of Ph.D. candidates or professional developers because they often have little time to spare.

### 6.3 Construct validity

In our experiment, we assessed the added value of our EXTRAVIS tool for program comprehension, and sought to generalize this added value to trace visualization techniques in general (Section 5.3). However, it should be noted that the experiment does not enable a *distinction* between EXTRAVIS and trace visualization: we cannot tell whether the performance improvement should be attributed to trace visualization in general or to specific aspects of EXTRAVIS (e.g., the circular bundle view). To characterize the difference, there is a need for similar experiments involving other trace visualization techniques.

As another potential threat to construct validity, the control group did not have access to the execution traces. This may have biased the experimental group because they had more data to work with. The rationale behind this decision was our intent to mimic real-life working conditions, in which software engineers often limit themselves to the use of the IDE. The subjects could still study the behavior of the application using, e.g., the built-in debugger in Eclipse (which in the experiment was available to both groups and was indeed used by some).

## 7 CONCLUSIONS

In this paper, we have reported on a controlled experiment that was aimed at the quantitative evaluation of EXTRAVIS, our tool for execution trace visualization. We designed eight typical tasks aimed at gaining an understanding of an open source program, and measured the performance of a control group (using the Eclipse IDE) and an experimental group (using both Eclipse and EXTRAVIS) in terms of time spent and correctness.

The results clearly illustrate EXTRAVIS' usefulness for program comprehension. With respect to time, the added value of EXTRAVIS was found to be statistically significant: on average, the EXTRAVIS group spent 22% less time on the given tasks. In terms of correctness, the results turned out even more convincing: EXTRAVIS' added value was again statistically significant, with the EXTRAVIS users scoring 43% more points on average. For the tasks that we considered, these results testify to EXTRAVIS' benefits compared to conventional tools: in this case, the Eclipse IDE.

To determine which types of tasks are best suited for EXTRAVIS or for trace visualization in general, we studied the group performance per task in more detail. While inferences drawn from one experiment and eight tasks cannot be conclusive, the experimental results do provide a strong indication as to EXTRAVIS' strengths. First, questions that require insight into a system's structural relations are solved relatively easily due to EXTRAVIS' circular view, as it shows *all* of the system's structural entities and their call relationships on a single screen. Second, tasks that require a user to globally understand a system's behavior are easier to tackle when a visual representation of a trace is provided, as it decomposes the system's execution into tractable parts. Third, questions involving a detailed behavioral understanding seem to benefit greatly from the fact that dynamic analysis reveals the actual objects involved in each interaction, saving the user the effort of browsing multiple source files.

This paper demonstrates the potential of trace visualization for program comprehension, and paves the way for other researchers to conduct similar experiments. The work described in this paper makes the following contributions:

- A systematic literature survey of existing trace visualization techniques in the literature, and a description of the 21 contributions that were found.
- The design of a controlled experiment for the quantitative evaluation of trace visualization techniques for program comprehension, involving eight reusable tasks and a validated solution model.
- The execution of this experiment on a group of 34 representative subjects, demonstrating a 22% decrease in time effort and a 43% increase in correctness.
- A discussion on the types of tasks for which EXTRAVIS, and trace visualization in general, are best suited.

### 7.1 Future work

As mentioned in Section 5.4, a related study has pointed out that results may differ quite significantly across different subject systems. It is therefore part of our future directions to replicate our experiment on another subject system.

Furthermore, we seek collaborations with fellow researchers to evaluate other trace visualization techniques. By subjecting such techniques to the same experimental procedure, we might be able to quantify their added values for program comprehension as well, and compare their performance to that of EXTRAVIS.

Finally, we believe that strong quantitative results such as the ones presented in this study could play a crucial role in making industry realize the potential of dynamic analysis in their daily work. In particular, they might be interested to incorporate trace visualization tools in their development cycle, and be willing to collaborate in a longitudinal study for us to investigate the long-term benefits of dynamic analysis in practice. Another aim of such a longitudinal study could be to shed light on how software engineers using a dynamic analysis tool define an execution scenario, how often they do this, and how much time they spend on it.

## ACKNOWLEDGMENTS

This research is sponsored by NWO via the Jacquard Reconstructor project. We would like to thank the 34 subjects for their participation, Danny Holten for his implementation of EXTRAVIS, Cathal Boogerd for performing one of the pilot studies, and Bart Van Rompaey for assisting in the experimental design. Also, many thanks to CHECKSTYLE's lead developer, Oliver Burn, who assisted in the design of our task review protocol.

## 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,” *J. Syst. Software*, vol. 38, no. 1, pp. 3–12, 1997.

[3] S. P. Reiss and M. Renieris, “Encoding program executions,” in *Proc. 23rd Int. Conf. Software Engineering*, pp. 221–230, IEEE Computer Society, 2001.

[4] A. Hamou-Lhadj and T. C. Lethbridge, “Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system,” in *Proc. 14th Int. Conf. Program Comprehension*, pp. 181–190, IEEE Computer Society, 2006.

[5] W. De Pauw, R. Helm, D. Kimelman, and J. M. Vlissides, “Visualizing the behavior of object-oriented systems,” in *Proc. Eighth Conf. Object-Oriented Programming Systems, Languages, and Applications*, pp. 326–337, ACM Press, 1993.

[6] D. F. Jerding, J. T. Stasko, and T. Ball, “Visualizing interactions in program executions,” in *Proc. 19th Int. Conf. Software Engineering*, pp. 360–370, ACM Press, 1997.

[7] B. Cornelissen, A. Zaidman, D. Holten, L. Moonen, A. van Deursen, and J. J. van Wijk, “Execution trace analysis through massive sequence and circular bundle views,” *J. Syst. Software*, vol. 81, no. 11, pp. 2252–2268, 2008.

[8] B. Cornelissen, A. Zaidman, B. Van Rompaey, and A. van Deursen, “Trace visualization for program comprehension: A controlled experiment,” in *Proc. 17th Int. Conf. Program Comprehension*, pp. 100–109, IEEE Computer Society, 2009.

[9] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, “A systematic survey of program comprehension through dynamic analysis,” *IEEE Trans. Software Eng.*, vol. 35, no. 5, pp. 684–702, 2009.

[10] A. Zaidman and S. Demeyer, “Managing trace data volume through a heuristical clustering process based on event execution frequency,” in *Proc. Eighth European Conf. Software Maintenance and Reengineering*, pp. 329–338, IEEE Computer Society, 2004.

[11] M. F. Kleyn and P. C. Gingrich, “Graphtrace - understanding object-oriented systems using concurrently animated views,” in *Proc. Third Conf. Object-Oriented Programming Systems, Languages, and Applications*, pp. 191–205, ACM Press, 1988.

[12] W. De Pauw, D. Lorenz, J. Vlissides, and M. Wegman, “Execution patterns in object-oriented visualization,” in *Proc. Fourth USENIX Conf. Object-Oriented Technologies and Systems*, pp. 219–234, USENIX, 1998.

[13] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. M. Vlissides, and J. Yang, “Visualizing the execution of Java programs,” in *Proc. ACM 2001 Symp. Software Visualization*, pp. 151–162, ACM Press, 2001.

[14] W. De Pauw, S. Krasikov, and J. F. Morar, “Execution patterns for visualizing web services,” in *Proc. ACM 2006 Symp. Software Visualization*, pp. 37–45, ACM Press, 2006.

[15] K. Koskimies and H. Mössenböck, “Scene: Using scenario diagrams and active text for illustrating object-oriented programs,” in *Proc. 18th Int. Conf. Software Engineering*, pp. 366–375, IEEE Computer Society, 1996.

[16] D. F. Jerding and S. Rugaber, “Using visualization for architectural localization and extraction,” in *Proc. Fourth Working Conf. Reverse Engineering*, pp. 56–65, IEEE Computer Society, 1997.

[17] T. Systä, “On the relationships between static and dynamic models in reverse engineering Java software,” in *Proc. 6th Working Conf. Reverse Engineering*, pp. 304–313, IEEE Computer Society, 1999.

[18] T. Systä, K. Koskimies, and H. A. Müller, “Shimba: an environment for reverse engineering Java software systems,” *Software, Pract. Exper.*, vol. 31, no. 4, pp. 371–394, 2001.

[19] T. S. Souder, S. Mancoridis, and M. Salah, “Form: A framework for creating views of program executions,” in *Proc. 17th Int. Conf. Software Maintenance*, pp. 612–620, IEEE Computer Society, 2001.

[20] R. Oechsle and T. Schmitt, “JAVAVIS: Automatic program visualization with object and sequence diagrams using the Java Debug Interface (JDI),” in *Proc. ACM 2001 Symp. Software Visualization*, pp. 176–190, ACM Press, 2001.

[21] A. Hamou-Lhadj and T. C. Lethbridge, “Compression techniques to simplify the analysis of large execution traces,” in *Proc. 10th Int. Workshop Program Comprehension*, pp. 159–168, IEEE Computer Society, 2002.

[22] A. Hamou-Lhadj, T. C. Lethbridge, and L. Fu, "Challenges and requirements for an effective trace exploration tool," in *Proc. 12th Int. Workshop Program Comprehension*, pp. 70–78, IEEE Computer Society, 2004.

[23] A. Hamou-Lhadj, E. Braun, D. Amyot, and T. C. Lethbridge, "Recovering behavioral design models from execution traces," in *Proc. Ninth European Conf. Software Maintenance and Reengineering*, pp. 112–121, IEEE Computer Society, 2005.

[24] M. Salah and S. Mancoridis, "Toward an environment for comprehending distributed systems," in *Proc. 10th Working Conf. Reverse Engineering*, pp. 238–247, IEEE Computer Society, 2003.

[25] M. Salah and S. Mancoridis, "A hierarchy of dynamic software views: From object-interactions to feature-interactions," in *Proc. 20th Int. Conf. Software Maintenance*, pp. 72–81, IEEE Computer Society, 2004.

[26] M. Salah, T. Denton, S. Mancoridis, A. Shokoufandeh, and F. I. Vokolos, "Scenariographer: A tool for reverse engineering class usage scenarios from method invocation sequences," in *Proc. 21st Int. Conf. Software Maintenance*, pp. 155–164, IEEE Computer Society, 2005.

[27] M. Salah, S. Mancoridis, G. Antoniol, and M. Di Penta, "Scenario-driven dynamic analysis for comprehending large software systems," in *Proc. 10th European Conf. Software Maintenance and Reengineering*, pp. 71–80, IEEE Computer Society, 2006.

[28] L. C. Briand, Y. Labiche, and Y. Miao, "Towards the reverse engineering of UML sequence diagrams," in *Proc. 10th Working Conf. Reverse Engineering*, pp. 57–66, IEEE Computer Society, 2003.

[29] L. C. Briand, Y. Labiche, and J. Leduc, "Tracing distributed systems executions using AspectJ," in *Proc. 21st Int. Conf. Software Maintenance*, pp. 81–90, IEEE Computer Society, 2005.

[30] L. C. Briand, Y. Labiche, and J. Leduc, "Toward the reverse engineering of UML sequence diagrams for distributed Java software," *IEEE Trans. Software Eng.*, vol. 32, no. 9, pp. 642–663, 2006.

[31] A. Kuhn and O. Greevy, "Exploiting the analogy between traces and signal processing," in *Proc. 22nd Int. Conf. Software Maintenance*, pp. 320–329, IEEE Computer Society, 2006.

[32] O. Greevy, M. Lanza, and C. Wysseier, "Visualizing live software systems in 3D," in *Proc. ACM 2006 Symp. Software Visualization*, pp. 47–56, ACM Press, 2006.

[33] J. Koskinen, M. Kettunen, and T. Systä, "Profile-based approach to support comprehension of software behavior," in *Proc. 14th Int. Conf. Program Comprehension*, pp. 212–224, IEEE Computer Society, 2006.

[34] S. Simmons, D. Edwards, N. Wilde, J. Homan, and M. Groble, "Industrial tools for the feature location problem: an exploratory study," *J. Software Maint. Evol.: Res. Pract.*, vol. 18, no. 6, pp. 457–474, 2006.

[35] B. Cornelissen, A. van Deursen, L. Moonen, and A. Zaidman, "Visualizing testsuites to aid in software understanding," in *Proc. 11th European Conf. Software Maintenance and Reengineering*, pp. 213–222, IEEE Computer Society, 2007.

[36] R. Voets, "JRET: A tool for the reconstruction of sequence diagrams from program executions," Master's thesis, Delft University of Technology, 2008.

[37] S. P. Reiss, "Visual representations of executing programs," *J. Vis. Lang. Comput.*, vol. 18, no. 2, pp. 126–148, 2007.

[38] J. Jiang, J. Koskinen, A. Ruokonen, and T. Systä, "Constructing usage scenarios for API redocumentation," in *Proc. 15th Int. Conf. Program Comprehension*, pp. 259–264, IEEE Computer Society, 2007.

[39] 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 *Proc. 15th Int. Conf. Program Comprehension*, pp. 49–58, IEEE Computer Society, 2007.

[40] C. Bennett, D. Myers, D. Ouellet, M.-A. Storey, M. Salois, D. German, and P. Charland, "A survey and evaluation of tool features for understanding reverse engineered sequence diagrams," *J. Software Maint. Evol.: Res. Pract.*, vol. 20, no. 4, pp. 291–315, 2008.

[41] A. R. Dalton and J. O. Hallstrom, "A toolkit for visualizing the runtime behavior of TinyOS applications," in *Proc. 15th Int. Conf. Program Comprehension*, pp. 43–52, IEEE Computer Society, 2008.

[42] M. J. Pacione, M. Roper, and M. Wood, "Comparative evaluation of dynamic visualisation tools," in *Proc. 10th Working Conf. Reverse Engineering*, pp. 80–89, IEEE Computer Society, 2003.

[43] A. Hamou-Lhadj and T. C. Lethbridge, "A survey of trace exploration tools and techniques," in *Proc. Conf. of the Centre for Advanced Studies on Collaborative Research*, pp. 42–55, IBM Press, 2004.

[44] N. Wilde and M. C. Scully, "Software Reconnaissance: Mapping program features to code," *J. Software Maint.: Res. Pract.*, vol. 7, no. 1, pp. 49–62, 1995.

[45] D. F. Jerding and J. T. Stasko, "The information mural: A technique for displaying and navigating large information spaces," *IEEE Trans. Vis. Comput. Graph.*, vol. 4, no. 3, pp. 257–271, 1998.

[46] M.-A. Storey, "Theories, methods and tools in program comprehension: past, present and future," in *Proc. 13th Int. Workshop Program Comprehension*, pp. 181–191, IEEE Computer Society, 2005.

[47] C. F. J. Lange and M. R. V. Chaudron, "Interactive views to improve the comprehension of UML models - an experimental validation," in *Proc. 15th Int. Conf. Program Comprehension*, pp. 221–230, IEEE Computer Society, 2007.

[48] J. Quante, "Do dynamic object process graphs support program understanding? – a controlled experiment," in *Proc. 16th Int. Conf. Program Comprehension*, pp. 73–82, IEEE Computer Society, 2008.

[49] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. van Deursen, "Mining software repositories to study co-evolution of production & test code," in *Proc. First Int. Conf. Software Testing*, pp. 220–229, IEEE Computer Society, 2008.

[50] B. Van Rompaey and S. Demeyer, "Estimation of test code changes using historical release data," in *Proc. 15th Working Conf. Reverse Engineering*, pp. 269–278, IEEE Computer Society, 2008.

[51] M. J. Pacione, M. Roper, and M. Wood, "A novel software visualisation model to support software comprehension," in *Proc. 11th Working Conf. Reverse Engineering*, pp. 70–79, IEEE Computer Society, 2004.

[52] B. Cornelissen, A. Zaidman, B. Van Rompaey, and A. van Deursen, "Trace visualization for program comprehension: A controlled experiment," Tech. Rep. TUD-SERG-2009-001, Delft University of Technology, 2009.

[53] M. Di Penta, R. E. K. Stirewalt, and E. Kraemer, "Designing your next empirical study on program comprehension," in *Proc. 15th Int. Conf. Program Comprehension*, pp. 281–285, IEEE Computer Society, 2007.

[54] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlesson, B. Regnell, and A. Wesslen, *Experimentation in software engineering - an introduction*. Kluwer Acad. Publ., 2000.

[55] R. K. Yin, *Case Study Research: Design and Methods*. Sage Publications Inc., 2003.

[56] C. Riva and J. V. Rodriguez, "Combining static and dynamic views for architecture reconstruction," in *Proc. Sixth European Conf. Software Maintenance and Reengineering*, pp. 47–55, IEEE Computer Society, 2002.