

Guided Exploration: A Method for Guiding Novice Users in Interactive Memory Monitoring Tools

MARKUS WENINGER, Institute for System Software, Johannes Kepler University Linz, Austria

ELIAS GANDER, Christian Doppler Laboratory MEVSS, Johannes Kepler University Linz, Austria

HANSPETER MÖSSENBOECK, Institute for System Software, Johannes Kepler University Linz, Austria

Many monitoring tools that help developers in analyzing the run-time behavior of their applications share a common shortcoming: they require their users to have a fair amount of experience in monitoring applications to understand the used terminology and the available analysis features. Consequently, novice users who lack this knowledge often struggle to use these tools efficiently.

In this paper, we introduce the *guided exploration* (GE) method that aims to make interactive monitoring tools easier to use and learn. In general, tools that implement GE should provide four support operations on each analysis step: they should automatically (1) *detect* and (2) *highlight* the most important information on the screen, (3) *explain* why it is important, and (4) *suggest* which next steps are appropriate. This way, tools *guide* users through their analysis processes, helping them to *explore* the root cause of a problem. At the same time, users learn the capabilities of the tool and how to use them efficiently.

We show how GE can be implemented in new monitoring tools as well as how it can be integrated into existing ones. To demonstrate GE's feasibility and usefulness, we present how we extended the memory monitoring tool AntTracks to provided guided exploration support during memory leak analysis and memory churn analysis. We use these guidances in two user scenarios to inspect and improve the memory behavior of the monitored applications.

We hope that our contribution will help usability researchers and developers in making monitoring tools more novice-friendly by improving their usability and learnability.

CCS Concepts: • **General and reference** → *Design*; • **Software and its engineering** → Software system structures; *Dynamic analysis*; *Software performance*; **Software maintenance tools**; *Software design techniques*; *Software defect analysis*; Maintaining software; • **Human-centered computing** → Graphical user interfaces; **User centered design**; *User interface design*; *User interface programming*.

Additional Key Words and Phrases: Monitoring Tools, Guided Exploration, Advisor, Onboarding, Intelligent Assistant, Context-Sensitive Help, Program Comprehension, Memory Comprehension, Memory Monitoring, Memory Leak Analysis, Memory Churn Analysis

ACM Reference Format:

Markus Weninger, Elias Gander, and Hanspeter Mössenböck. 2021. Guided Exploration: A Method for Guiding Novice Users in Interactive Memory Monitoring Tools. *Proc. ACM Hum.-Comput. Interact.* 5, EICS, Article 209 (June 2021), 34 pages. <https://doi.org/10.1145/3461731>

Authors' addresses: Markus Weninger, markus.weninger@jku.at, Institute for System Software, Johannes Kepler University Linz, Altenberger Straße 69, Linz, 4040, Austria; Elias Gander, elias.gander@jku.at, Christian Doppler Laboratory MEVSS, Johannes Kepler University Linz, Altenberger Straße 69, Linz, 4040, Austria; Hanspeter Mössenböck, hanspeter.moessenboeck@jku.at, Institute for System Software, Johannes Kepler University Linz, Altenberger Straße 69, Linz, 4040, Austria.

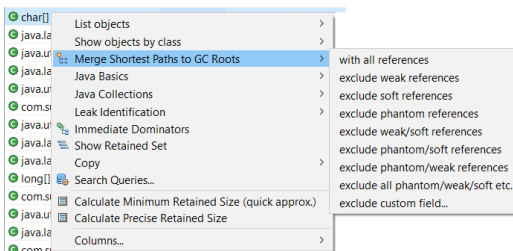
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of the ACM on Human-Computer Interaction*, <https://doi.org/10.1145/3461731>.

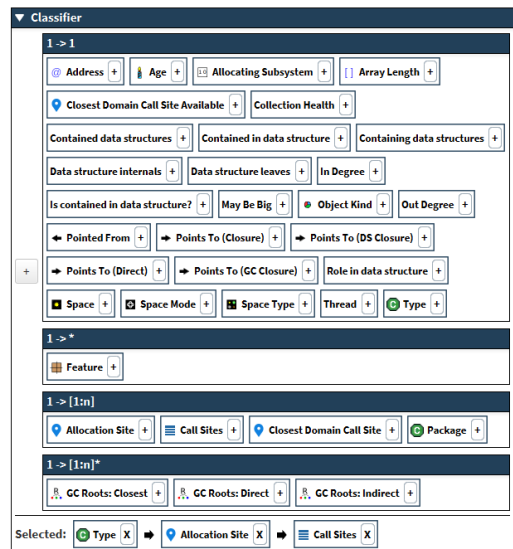
1 INTRODUCTION

The complexity of modern software makes monitoring tools essential, as their analysis features support users in inspecting, understanding, and fixing run-time problems. Unfortunately, most monitoring tools are designed for experts with extensive knowledge in their respective domain. Consequently, novices who lack this expertise are often unable to use these tools to their full potential [57, 79]. For example, Weninger et al. [91] observed during a user study on memory monitoring tools that especially novice users who were unfamiliar with typical memory monitoring activities and tool features struggled to extract the insights needed to fix a given problem. They were often overwhelmed by the complexity and number of available features and said that they wished to have more guidance throughout the analysis process. Based on these observations, the authors recommend that memory monitoring tool developers should *provide guidance and explanations to support exploratory learning of analysis capabilities* [91].

Having too many and too complex features is a common problem across interactive expert tools, which makes them hard to use for novice users, as illustrated in Figure 1. For example, in the memory analysis tool MAT [80], to inspect a suspicious heap object group in more detail, users are confronted with a long list of analysis features (Figure 1a). In AntTracks [85, 99], users can define how to group heap objects for inspection based on a number of different properties and criteria (Figure 1b). Without guidance or hints, novice users may feel overwhelmed and may not be able to decide which actions to take in a certain situation.



(a) MAT's type histogram provides a context menu that offers nine analysis features, six of which have (many) further sub-decisions.



(b) AntTracks's classification view offers around 35 grouping criteria for heap objects that can be freely combined by the user.

Fig. 1. Examples of complex decisions in memory monitoring tools.

In this paper, we present *guided exploration* (GE), a method that aims to increase the learnability and usability of monitoring tools, and GE's application in memory monitoring. In general, to implement GE, tool developers should first identify their tool's typical analysis processes. For example, in a memory monitoring tool, this may be the typical steps performed during memory leak analysis. GE aims to support users in performing and understanding these steps, especially

supporting those users without extensive knowledge in the tool's domain. To do so, tools that follow the method of GE should provide the following four support operations on every analysis step: they should automatically (1) *detect* and (2) *highlight* the most important information on the screen, (3) *explain* background knowledge and why the highlighted information is important, and (4) *suggest* which next steps are appropriate based on the findings.

To demonstrate how GE can be introduced in existing tools, we present how we extended the interactive memory monitoring tool AntTracks to support GE on its two main analyses: memory leak analysis and memory churn analysis.

Our main contributions in this work are:

- (1) an overview of our general *guided exploration* method that can be implemented in new monitoring tools as well as integrated into existing ones, see [Section 3](#).
- (2) guided exploration for memory leak analysis, integrated into AntTracks, see [Section 4](#).
- (3) guided exploration for memory churn analysis, integrated into AntTracks, see [Section 5](#).
- (4) a discussion of preliminary user feedback regarding AntTracks's GE ([Section 6](#)), an outlook on the possible application of GE in a domain other than memory monitoring ([Section 7](#)), and a discussion of GE's current limitations and possible future improvements ([Section 8](#)).

2 BACKGROUND AND RELATED WORK

In this section, we first discuss background and related work in the field of Human-Computer Interaction, more specifically related work on the usability and ease of use of analysis and monitoring tools. We then show different kinds of user guidance and how our approach fits into these classifications. Since this work focuses on novice user guidance in memory monitoring tools, we also introduce general memory monitoring concepts and typical memory problems that developers have to face. As we have implemented guided exploration in the memory monitoring tool AntTracks, AntTracks and its core features are also explained.

2.1 Usability, Ease of Use and Learnability

Ample studies have been performed on how to improve the user experience in software tools. For example, Johnson et al. [33] performed a study on the (under-)use of static analysis tools. Nineteen of their 20 study participants reported that they *felt that many static analysis tools do not present [...] enough information for them to assess what the problem is, why it is a problem and what they should be doing differently*, i.e., they missed explanations on how to interpret the presented data. Christakis and Bird [10] report that many of their findings match those of Johnson et al. In a study conducted by Riemenschneider and Hardgrave [70], *ease of use* (including *learnability*) was shown to be the major determinant for *tool usage*, i.e., ease of use is paramount for tools to attract and hold users. Holding users is important, since a continuous use of monitoring tools, especially application performance management (APM) tools, can have a positive impact on the quality of software [77]. Despite this, Tarek et al. [2] conclude their work on the effectiveness of APM tools as *[...] the reporting capability of APM tools must be improved to reduce the effort that is required to analyze detected performance regressions*. The logical consequence that follows from these results is that developers have to improve their tools' usability to reach a broad range of users. While some approaches try to achieve this using user-specific data aggregation [64, 82], our GE approach focuses on increased learnability by *guiding* the user through analysis processes.

2.2 User Guidance

The idea of user guidance is not new. Folmer and Bosch [18] classify two general *guidance patterns* that are typically used to increase tool usability [1, 38, 54]: (1) *wizards* and (2) *context-sensitive help*.

Most *wizards* are implemented as a rigid, linear series of dialog views [12]. These views ask a number of questions and then use this information to automate certain tasks [81]. Modern approaches involve the generation of user-specific wizards [102].

Various approaches exist for *context-sensitive help* [78], such as *coaches* [50], *guides* [12] or *advisors* [50]. *Coaches* are often implemented as context-sensitive hints or tips and typically provide the user “how to” information to overcome minor hurdles. *Guides* can be thought of as “intelligent coaches”, as they only display hints or tips whenever and wherever it is most likely useful, reacting to the user’s behavior. Coaches, guides, and more recent approaches such as micro-learning and gamification [22, 23, 28] are often used during *onboarding* [67], i.e., while introducing a person to a new tool to improve the person’s success using it [76]. Our approach differs from coaches and guides as GE does not only explain possible next steps or display certain hints, but it provides full guidance throughout a given task, including automatic decision making based on the underlying data. Thus, it better fits the description of an *advisor* system. Advisors are context-sensitive help systems that provide hints, tips, reasoning support, and explanations of complicated concepts. They help novice users to make decisions, to understand why certain steps should be performed, and to determine why certain decisions were suggested.

Rabiser et al. [62] provide a framework to compare monitoring approaches based on 21 different characteristics, including typical characteristics relating to guidance such as *target group*, *needed skills*, *input guidance*, and *output guidance*. They then compared 32 existing monitoring approaches and tools based on this framework. Even though they report that some monitoring tools partly provide certain unstructured guidance, they conclude that *many approaches do not provide much end-user tool support [...] and generally only very few provide fully-fledged tools with visualizations and guidance for users. The target user group seems to be mainly (experienced) engineers* [62].

To the best of our knowledge we are the first to describes a general guidance method in the context of interactive monitoring tools, especially in the domain of memory monitoring.

2.3 Memory Analysis

To reduce the risk for memory-related problems, modern programming languages such as Java use *garbage collection* (GC) to automatically reclaim unused memory. During a garbage collection, objects that are no longer (indirectly) reachable from GC roots (i.e., static fields and local variables) are automatically reclaimed, freeing up their reserved memory. This relieves programmers from the error-prone task of manual memory management. Nevertheless, garbage collection comes with its own set of possible memory problems that can slow down applications if developers handle object allocations and object storage carelessly. In the worst case, problems such as memory leaks can even crash the application.

Memory leaks occur when objects that are no longer needed remain reachable from GC roots due to programming errors [48]. For example, a developer may forget to remove objects from long-living data structures once they are not needed anymore. These objects cannot be reclaimed by the GC and will therefore accumulate over time [86, 88].

Another common memory anomaly that is often overlooked by novice users is *high memory churn*. High memory churn, also called *excessive dynamic allocations* [60, 75] or *high allocation density* [13], occurs when objects are (unnecessarily) allocated in high frequencies, just to be reclaimed shortly after their creation. For example, high memory churn is often the result of heavily-executed loops that contain allocations of short-living objects. This leads to increased work for allocating these objects on the heap and to an increased number of garbage collections to collect them, both of which negatively impact an application’s performance.

2.4 Introduction to AntTracks

This section presents the basics of AntTracks, a trace-based memory monitoring tool consisting of the AntTracks VM [39–41] (a modified Java Hotspot VM) and the AntTracks Analyzer [5, 86–90, 93, 94, 99]. We use AntTracks as an example throughout the paper to showcase how existing monitoring tools can be extended and refactored to support GE. We chose this tool since its source code is publicly available [85] and the authors already had prior experience with its code base.

2.4.1 Trace Recording by the AntTracks VM. The AntTracks VM records events such as object allocations and object movements during garbage collection by writing them into trace files [39, 40], introducing a run-time overhead of about 5%. To reduce the trace size, the VM does not record any redundant data and applies compression [41].

2.4.2 Reconstruction in the AntTracks Analyzer. The AntTracks Analyzer processes the events stored in a trace file, reconstructing the heap state at every garbage collection point [5]. A heap state is a set of heap objects that were live in the monitored application at a certain point in time. Properties such as the the address, the type, the allocation site, and the allocating thread can be reconstructed for each heap object, as well as GC root information and information about the references between the heap objects.

The tool’s core mechanism is object classification in combination with multi-level grouping [93, 99]. A classifier groups heap objects according to a certain criterion such as type, allocation site, or allocating thread. Grouping the heap objects according to the classification results of multiple classifiers results in a hierarchical *memory tree*. A common classifier combination is to group all heap objects by their types and then by their allocation sites, as exemplarily shown in Figure 2. Yellow rectangles represent tree nodes and blue circles represent the objects that were classified into the respective tree branch. For example, the objects 0 to 3 are of type `Object[]`, of which the objects 0, 1 and 3 have been allocated in the method `Stack: init()` and object 2 has been allocated in the method `MyService: foo()`.

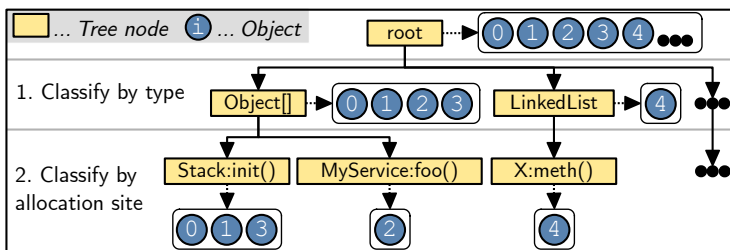


Fig. 2. A *memory tree* that first groups all objects by their types and then by their allocation sites.

Memory traces are used as a data basis for a variety of analyses within AntTracks. Two of these analysis, as well as their new guided exploration features, will be explained in more detail in Section 4 (memory leak analysis) and Section 5 (memory churn analysis).

3 GUIDED EXPLORATION

Without training, especially novice users may struggle to understand analysis features, terminology, metrics or visualizations in state-of-the-art interactive monitoring tools due to their steep initial learning curve. Being a novice monitoring tool user does not imply general inexperience. For example, even experienced software developers may have never used a memory monitoring tool before they encounter their first application crash due to a memory leak, which makes them a novice

in the domain of memory analysis. By incorporating learning-by-doing [71], our *guided exploration* method intends to simplify the onboarding process for this kind of users. As the method’s name suggests, tools implementing guided exploration should *guide* novice users through their analysis processes, helping them to *explore* the collected data until the root cause of a problem is found.

Even though the focus of this work is to present how GE can be applied in memory monitoring tools, this section presents the general idea of GE. As we discuss in Section 7, we think that GE may be a suitable guidance pattern for monitoring tools of other domains too, and thus we plan to further explore possible applications of GE in other domains in future work.

Section 3.1 discusses which steps are necessary before GE can be integrated into a tool. Section 3.2 explains GE’s four user support operations (depicted in Figure 4) in detail: *Detection*, *Highlighting*, *Explanation*, and *Suggestion*. These four user support operations can be gradually introduced in existing monitoring tools view by view, step by step.

3.1 Mapping of Analysis Process Steps to Views

Before introducing guided exploration in a monitoring tool, the tool developers have to define its typical analysis processes (such as memory leak analysis in a memory monitoring tool) and the steps performed within these processes. To do so, we suggest to create a (simple) process or task model. Various task model notations exist, for example ConcurTaskTrees [58], Task Flow [37], useML [51], or visualizations similar to UML statechart diagrams [46]. They all strive to capture the most important elements describing how a task (i.e., an activity that should be performed in order to reach a certain goal) is carried out by a particular user in a given context or in a given scenario [21, 45]. In the case of guided exploration, these task models should be designed from the perspective of novice users. This means that, even though analysis tasks (e.g., memory leak analysis) can often be performed in different ways across a number of multiple steps, the model should contain the *typical* flow of steps that should be performed to achieve the task’s goal. Tool developers and domain experts should be able to derive such a model, describing the “default” steps novice users should learn to perform, i.e., those steps that should be supported with GE in the future. Each step can then be mapped to one of the tool’s views to determine those views that have to be modified in order to support GE.

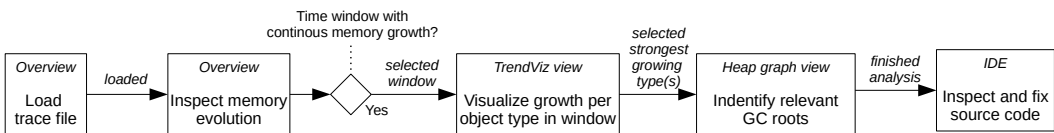


Fig. 3. Simplified task model of the typical steps performed during memory leak analysis, mapped to their corresponding AntTracks views.

For example, in Section 4 we show how GE has been implemented in AntTracks to guide users on the search for memory leaks. A typical memory leak analysis process, as shown in Figure 3, is (1) to search for a time window with continuous memory growth, (2) to find those objects that accumulate over time within this time window, and (3) to identify the GC roots that keep these accumulating objects alive. In AntTracks, each of these analysis steps is performed on a separate view, for each of which the four guided exploration support operations depicted in Figure 4 and explained in Section 3.2 have been implemented.

3.2 Guided Exploration Support Operations

This section discusses the four GE support operations a tool should perform on each analysis step: (1) First, the tool should automatically *detect* potential problems, i.e., suspicious patterns. (2) To help users in understanding from where the automatically gained insight was derived, the respective user interface (UI) region should be visually *highlighted*. (3) Since the user may require background knowledge to comprehend certain terminology and the highlighted information, *explanations* should discuss why the highlighted information is interesting. (4) Finally, based on the problem and the detected information, subsequent analysis steps should be *suggested*.

Since tools can greatly differ in their look-and-feel, we did not come up with a general rule on how to visualize notifications that a suspicious pattern has been found. It is thus up to the tool developer to appropriately inform the user about new guidance information. For example, in AntTracks, if guidance information is available a *guidance button* in the form of a light bulb is shown next to the respective UI element. Clicking such a light bulb then *highlights* the respective UI element and provides *explanations* and *suggestions*. This way of visualization was developed based on preliminary user feedback, as we will discuss in Section 6. Also, tool developers should keep guidance support optional, as experienced users may prefer to perform inspections without guidance elements visible.

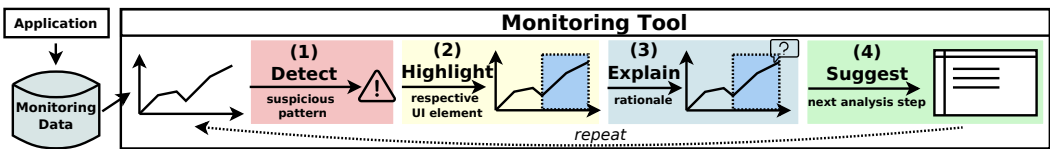


Fig. 4. The four guidance operations of GE: (1) Detection, (2) Highlighting, (3) Explanation, and (4) Suggestion.

Detection describes the task of automatically detecting potential problems, i.e., suspicious metrics or patterns.

Every view in an interactive monitoring tool is developed with the intention of supporting the user in achieving a certain goal. To this end, different kinds of visualizations are used to present data to the user. In non-guiding tools, it is up to the user to interpret these visualizations and to derive insights and findings from them. For example, a tool may present time-series charts for users to detect abnormal program behavior patterns. Others present tables, expecting the user to search for suspicious entries, e.g., metrics exceeding certain thresholds.

Most of these tasks require domain expertise that novice users generally do not have. For example, novice memory monitoring tool users may not search for data structures with a large *retained size*, i.e., the number of objects that are kept alive by a data structure [87], if this definition is unknown to the user. Thus, the first support operation of GE is to automatically *detect* suspicious patterns that may hint at problems in the monitored application. We think that intelligent tools should be able to perform this task, at least to a certain degree. After all, non-guiding tools expect their users to be able to detect suspicious patterns on their own, based on the displayed information. Since both the tool and the user have access to the same information, the tool should be capable of performing the same detection task, even if only detecting more obvious patterns using heuristics.

Even though the problem patterns that need to be detected may differ from domain to domain, we found certain similarities across different monitoring tools regarding the data they operate on and how this data is presented to users. Many monitoring tools inspect the *evolution* of a system, i.e., the evolution of certain metrics over time. These metric changes are often visualized

using charts, primarily time-series charts. Users are then expected to detect suspicious patterns within this evolution. Automated time series analysis [19] (using features such as regression analysis [53], seasonal and trend analysis [84], prediction [73], forecasting [4], or clustering and anomaly detection [43]) is a major research field in the domain of knowledge discovery and data mining. For example, time series analysis is used in memory monitoring to automatically detect suspicious time windows during which a monitored application behaves abnormally with regard to memory utilization [89].

Another typical way of depicting information is through the use of (hierarchical) tables, where features such as filtering or sorting should help the user to detect entries that (do not) meet certain criteria or those that exceed given thresholds. Such tasks may also be supported by automatic detection algorithms. For example, in the domain of memory monitoring, some memory inefficiencies and anti-patterns can automatically be detected based on memory metrics that exceed certain limits [8, 9]. *Intelligent user interfaces* [24, 30, 34, 49] often also apply artificial intelligence and machine learning for pattern detection, for example, by performing clustering or outlier detection. Also, research in the field of recommender systems [16, 65] may provide interesting ideas and algorithms that could be incorporated into the automated detection of patterns in monitoring tools.

Highlighting the relevant region on the user interface helps users to understand *where* the automatically gained insight can be found if the view was inspected manually.

The goal of GE is *not* to remove visualizations from monitoring tools, but to help users to understand and interpret these visualizations better. Thus, once a potential problem is detected, the UI region / element relevant for its detection should be *highlighted*. Different types and arrangements of UI elements may use different kinds of highlighting. The kind of highlighting should be chosen based on known UI design principles such as the principles of highlighting [44] or color coding [100]. Further, the developers should make sure that the style of highlighting is *consistent* throughout all views that support guided exploration [6]. For example, Figure 5 shows how AntTracks highlights rows in its tree tables by displaying them with a different background color.

Name	Objects	Retained size ▾
Overall	8,022,993	404 MB
Filtered	429,915	172.7 MB
HashMap	68,214	133.4 MB
SetMultimap::<init>()	6	91.8 MB
▶ AllocatedTypes::<init>()	3	35.2 MB
▶ SymbolsParser::parseAllo...	64,720	11.8 MB

Fig. 5. AntTracks now automatically *detects* and *highlights* suspicious parts, for example data structures that keep many other objects alive (i.e., those data structures that have a high retained size).

Explanations should help users in understanding why the highlighted information is important. They should clarify used terminology and concepts that are needed to understand the problem.


Let's continue with the example from Figure 5. First, AntTracks automatically *detects* objects with a high retained size [87], i.e., objects that keep a large number of other objects alive, and then

highlights these objects on the view (in this case six HashMap objects allocated at the same allocation site). Without knowing what a high retained size means or how to interpret it, the user will not be able to make sense of the highlighted information. Thus, the user can choose to display an *explanation* that should clarify needed *background knowledge*, *terminology*, as well as the *rationale* why the given pattern is considered suspicious.

Figure 6 exemplarily shows how AntTracks handles this in its guidance pop-ups. First, the explanation describes what retained size means (background knowledge + terminology), followed by an explanation of the currently highlighted area, i.e., “Over 22% of this heap is kept alive by 6 data structures of type HashMap that have been allocated in the constructor of class SetMultimap” (rationale).

Suggestions on which steps could or should be taken next to make it easier for the user to understand *what* operations are possible and *why* they might be useful.

Interactive monitoring tools often provide a vast amount of analysis features that can be applied in different situations. Intended for expert users, this flexibility may intimidate and overwhelm novice users. Despite the multitude of available features, as discussed in Section 3.1, most analysis processes have a default flow of tasks. *Suggestions* should guide the user through this process. We also recommend to not only display these suggestions as plain text, but to provide shortcuts to automatically perform the suggested actions. For example, Figure 6 shows how AntTracks presents suggested operations as buttons that automatically perform the next step.

Large data structures! 

The retained size describes **object ownership**, that means that objects / object groups with a high retained size keep a lot of other objects alive.

Over **22%** of this heap is kept alive by **6** data structures of type **HashMap** that have been allocated in the constructor of class **SetMultimap**.

Inspect the **GC roots** to see why these data structures are kept alive. The **data structures leaves** on the other hand will tell you why these data structures consume so much memory.

INSPECT GC ROOTS

INSPECT DATA STRUCTURE LEAVES

Fig. 6. This *explanation* and *suggestion* pop-up is shown upon user request in AntTracks when data structures with large ownership are detected. It explains terminology, e.g., retained size, and explains which insights might be gained following the provided suggestions.

4 GUIDED EXPLORATION OF MEMORY LEAKS

In this section, we present how we integrated *guided exploration* into AntTracks to facilitate the analysis of memory leaks. In Section 4.1, we describe a typical memory leak analysis process and how this process is mapped to AntTracks’s views. In Section 4.2, we describe how AntTracks’s views have been extended to support the four guided exploration operations *Detection*, *Highlighting*, *Explanation*, and *Suggestion*. To showcase how these new guidances support users in comprehending and investigating memory leaks, we use AntTracks’s new guided exploration and its suggestions to investigate and fix an application that contains a memory leak.

4.1 Mapping of Memory Leak Analysis Process Steps to Views

There are various ways to detect and analyze memory leaks. For example, AntTracks can detect memory leaks by searching for growing data structures and inspecting those with the strongest growth in more detail [86, 88]. Thus, we extended AntTracks’s data structure growth analysis by implementing guided exploration for it.

Yet, not every memory monitoring tool can access data structure information in the monitored application. Thus, we will focus on a more general memory leak analysis process that is not specific to AntTracks, which is visualized as a simplified task model in Figure 7. It consists of the following three steps:

- (1) Detect a time window with continuous memory growth, i.e., a continues time frame in which object accumulate over time.
- (2) Find out which kinds of objects accumulate over time in this window.
- (3) Find those GC roots that keep these strongly accumulating objects alive.

In AntTracks, each step is performed on a different view:

- (1) The *Overview view* plots the application’s memory evolution and GC activity in time-series charts. A growing number of heap objects over time may hint at a possible memory leak.
- (2) The *AntTracks TrendViz view* [94] shows how the heap evolves over time, i.e., which objects (grouped by, e.g., their types) accumulate the most.
- (3) The *Heap graph view* interactively visualizes a heap state in a graph-based visualization. It can be used to inspect *keep-alive* relations to drill-down to the root cause of a possible memory leak.

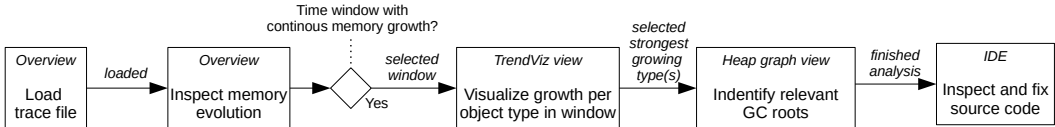


Fig. 7. Simplified task model of the typical steps performed during memory leak analysis, mapped to their corresponding AntTracks views.

The first two views have already existed in AntTracks and have been extended to provided guided exploration as part of this work. The heap graph view has been newly developed from scratch, including its GE support.

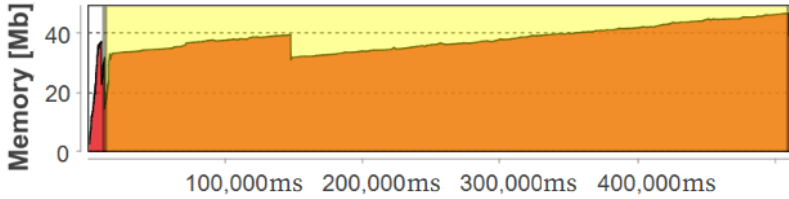
The analysis steps shown in Figure 7 are not restricted to AntTracks but can also be performed in a similar way in other memory monitoring tools such as VisualVM [56] or MAT [80]. Thus, the GE support operations that have been integrated into AntTracks could be integrated into these tools in a similar fashion as well.

4.2 Guided Exploration Support Operations for Memory Leak Analysis

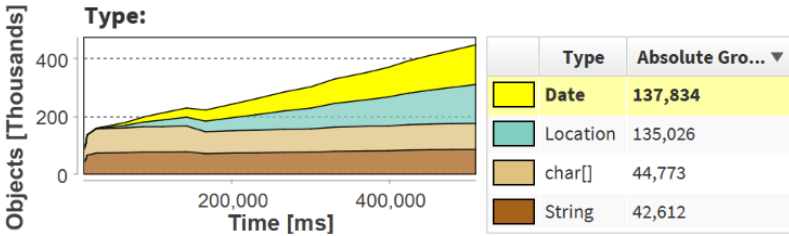
AntTracks now provides GE on the three views identified in Section 4.1. In this section, for each view we explain its general functionality and its new GE support operations *Detection*, *Highlighting*, *Explanation* and *Suggestion*.

To showcase how GE in AntTracks now supports users in comprehending and investigating memory leaks, we present how the newly introduced guidance features have been used to inspect *Dynatrace easyTravel* [14]. Dynatrace focuses on application performance monitoring (APM) and distributes easyTravel as their state-of-the-art memory leak demo application. It is a multi-tier application for a travel agency, using a Java backend and an automatic load generator simulating

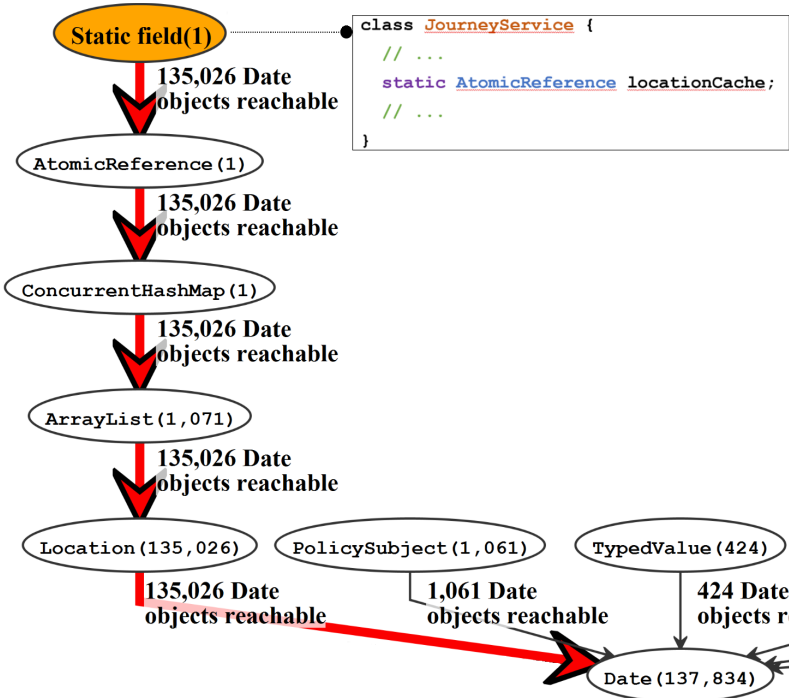
accesses to the service. All automatically detected and highlighted problem patterns are shown in Figure 8 and will be explained in detail in the following.



(a) The overview view highlights an automatically detected memory leak time window.



(b) AntTracks TrendViz shows how objects of various types accumulate over time.



(c) The graph view highlights the path from a selected group of objects (i.e., Date) to its most important garbage collection root (i.e., the static field locationCache), i.e., the path on which most Date objects are potentially kept alive.

Fig. 8. Memory leak analysis in AntTracks.

4.2.1 Overview View.

The overview (Figure 9) view gives the user a general impression on the application’s memory behavior. For example, a time-series chart plots the monitored application’s memory footprint over time. Users can select a single point in time to inspect the heap state at that point, or they can select a time window, i.e., two points in time, to inspect the heap evolution over this window.

We observed at different occasions, e.g., during studies or when AntTracks was used during hands-on tool presentations, that especially novice users are in need of guidance and support. Some users lacked the background knowledge to recognize abnormal behavior as such, or they struggled to select a suitable point in time or a suitable time window for certain analysis features. GE on AntTracks’s overview view should help the users by automating these steps.

Detection. Weninger et al. [89] showed how to automatically detect suspicious time windows in memory monitoring. We apply their heuristic-based algorithm that mimics human behavior to search for a memory leak window, i.e., a window with a continuous growth (except for minor drops) of reachable memory.

Highlighting. If a suspicious time window that may be the result of a possible memory leak is found, it is highlighted with a yellow rectangular overlay, as shown in Figure 8a.

Explanation. AntTracks explains to the user why the detected time window may be connected to a memory leak: *AntTracks has detected a time window over which the reachable memory is continuously growing. This is an indicator for a possible memory leak. If a memory leak exists, typically objects of a few common types accumulate over time.*

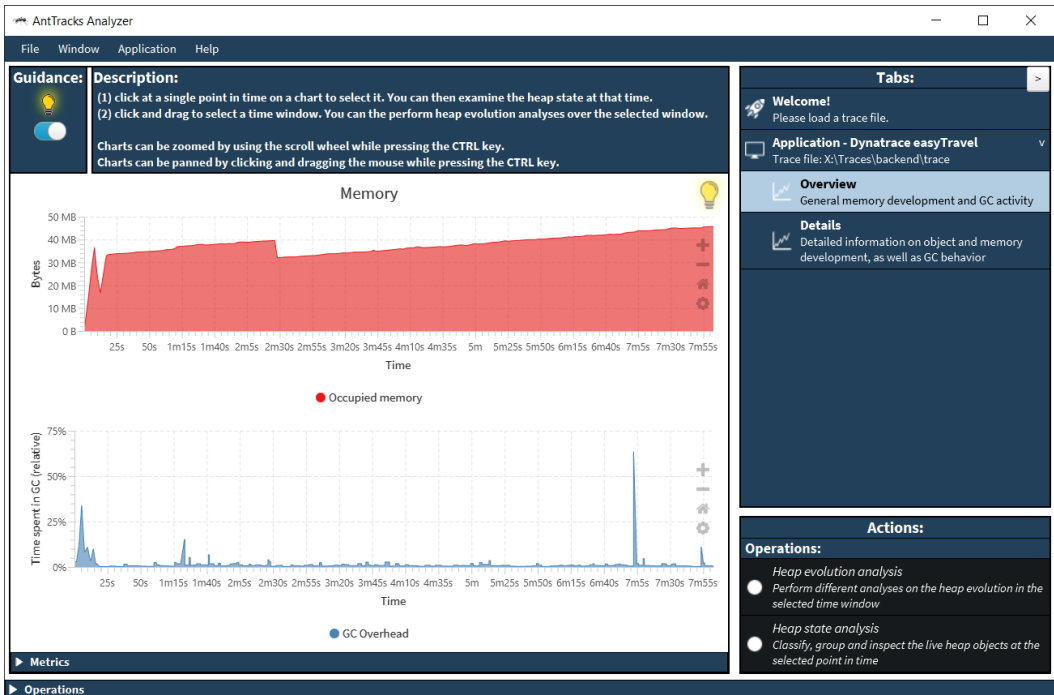


Fig. 9. The Overview provides initial information to assess the overall memory and garbage collection behavior of the monitored application.

Suggestion. We suggest the user to apply the *AntTracks TrendViz* [94] feature to explore how the heap’s contents changed over time in order to detect if certain types of objects accumulate over time.

Figure 8a shows the memory evolution of easyTravel over time, including an automatically detected and highlighted memory leak time window. *AntTracks explains* that the window exhibits strong memory growth (about 400%) up until the end of the application, an indication for a memory leak. It is worth mentioning that the initial memory spike during application startup is not part of this window. Objects allocated during this spike are freed shortly after and thus have no relevance for the memory leak, a fact that is obvious for experts (and the time window algorithm) but novice users may not be aware of. Following the *suggestion*, we applied the *AntTracks TrendViz* feature on the time window.

4.2.2 AntTracks TrendViz View.

The *AntTracks TrendViz* view [94] (Figure 10) classifies the live heap objects at every garbage collection based on a list of selected classifiers, as explained in Section 2.4. The evolution of the

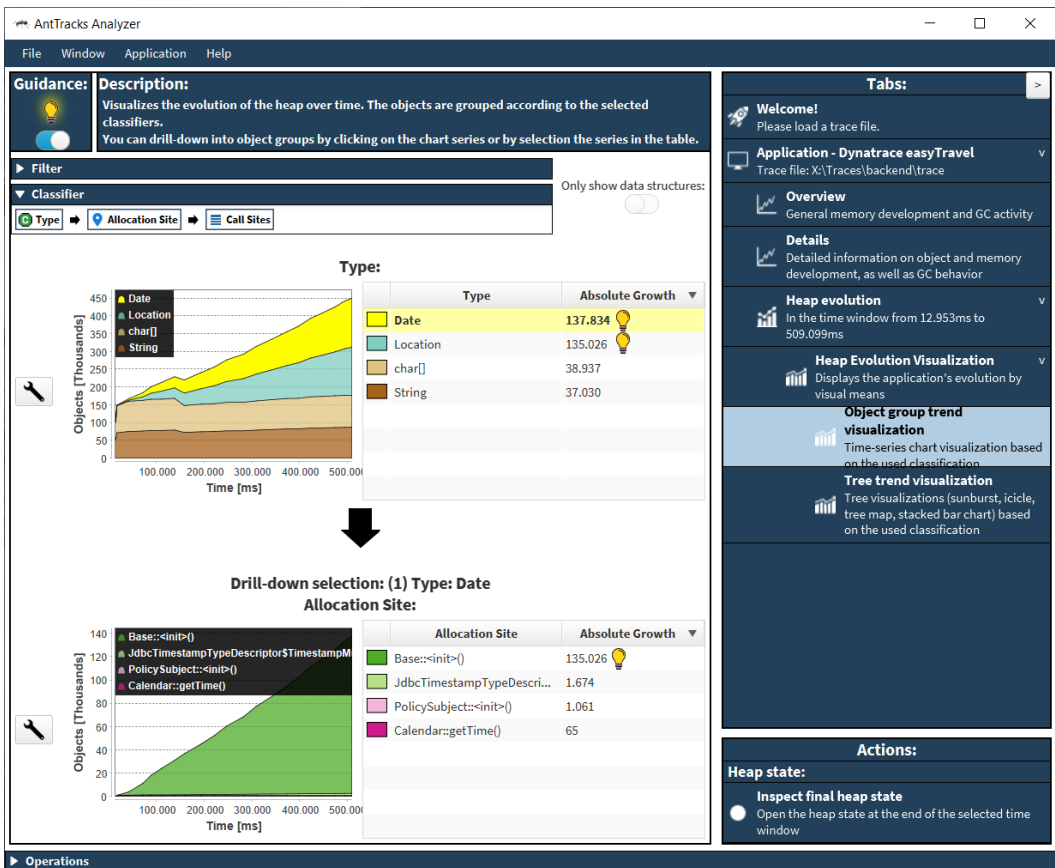


Fig. 10. The *TrendViz* view provides information on the heap evolution over time, i.e., which kinds of objects accumulated the most, including a drill-down feature to inspect suspicious object groups in more detail.

resulting memory trees is then visualized using time-series charts. Without guidance, users would have to select this list of classifiers on their own. Since GE aims to especially help novice users, we free them from this task by grouping the heap objects automatically by their types and their allocation sites. When opening the view, the evolution of the first level of the memory trees, i.e., the evolution of the objects grouped by type, is visualized, as shown on the top of [Figure 10](#) and in [Figure 8b](#). This is where GE comes into play.

Detection. We automatically detect the type of which the most objects accumulated over time. In easyTravel, the objects that accumulated the most are objects of type Date, as shown in [Figure 8b](#). Objects of this type are most probably involved in a possible memory leak. If multiple types exhibit similar strong growth (such as Location), all of them are suggested to the user for further inspection.

Highlighting. The chart series and the table entry of the suspicious type will be highlighted (yellow overlay) as shown in [Figure 8b](#).

Explanation. AntTracks’s explanatory text for the strongest growing type(s) reads “Over the selected time window, the number of <type> objects increased by <absolute growth>. This corresponds to <relative growth>% of the total heap growth and could be an indication of a memory leak.”

Suggestion. The view provides two suggestions: Either to stay on the view to drill-down to inspect where the suspicious objects have been allocated, or to go to the heap graph view to visually inspect the GC roots that keep the suspicious objects alive. Following the first suggestion opens a second time-series chart below the current one, which shows the evolution of the second level of the memory trees. For example, selecting the Date objects for drill-down opens a second chart that shows where the Date objects have been allocated over time, as shown on the bottom of [Figure 10](#). The same detection, highlighting and explanation steps as described above are then performed for the allocation sites, and users are suggested to visualize the objects of the strongest growing type that have been allocated at the allocation-heaviest allocation site in the heap graph view.

[Figure 8b](#) shows the memory evolution of easyTravel, grouped by type, on the AntTracks TrendViz view. Using the *Type classifier* as first grouping criterion was automatically performed by GE as we followed the suggestion on the overview. On the TrendViz view, AntTracks’s GE automatically *detected* that the objects that accumulated the most in easyTravel in the selected time window are those of type Date (*highlighted* in yellow) and Location. AntTracks’s GE *explains* that the Date objects are the major suspects for a possible memory leak since about 30% of the overall heap growth can be accounted to them. We then followed the *suggestion* to use the heap graph view to inspect the paths to the GC roots and thus find out which objects and GC roots (indirectly) keep the Date objects alive.

4.2.3 Heap Graph View.

Objects are kept alive because they are directly or indirectly reachable from GC roots. The heap graph view ([Figure 11](#)) is a newly introduced analysis view in AntTracks that is used to visually explore the references between heap objects and GC roots. This view was developed with GE support from the start to help users in detecting and understanding suspicious paths from objects to their GC roots, called *bottom-up analysis*. In the following, we will briefly explain the view’s interaction features before we present its guided exploration operations.

A heap may contain millions of objects, each of them referencing other objects. Thus, visualizing every object as a separate node and every reference as a separate edge is not feasible. Instead, our

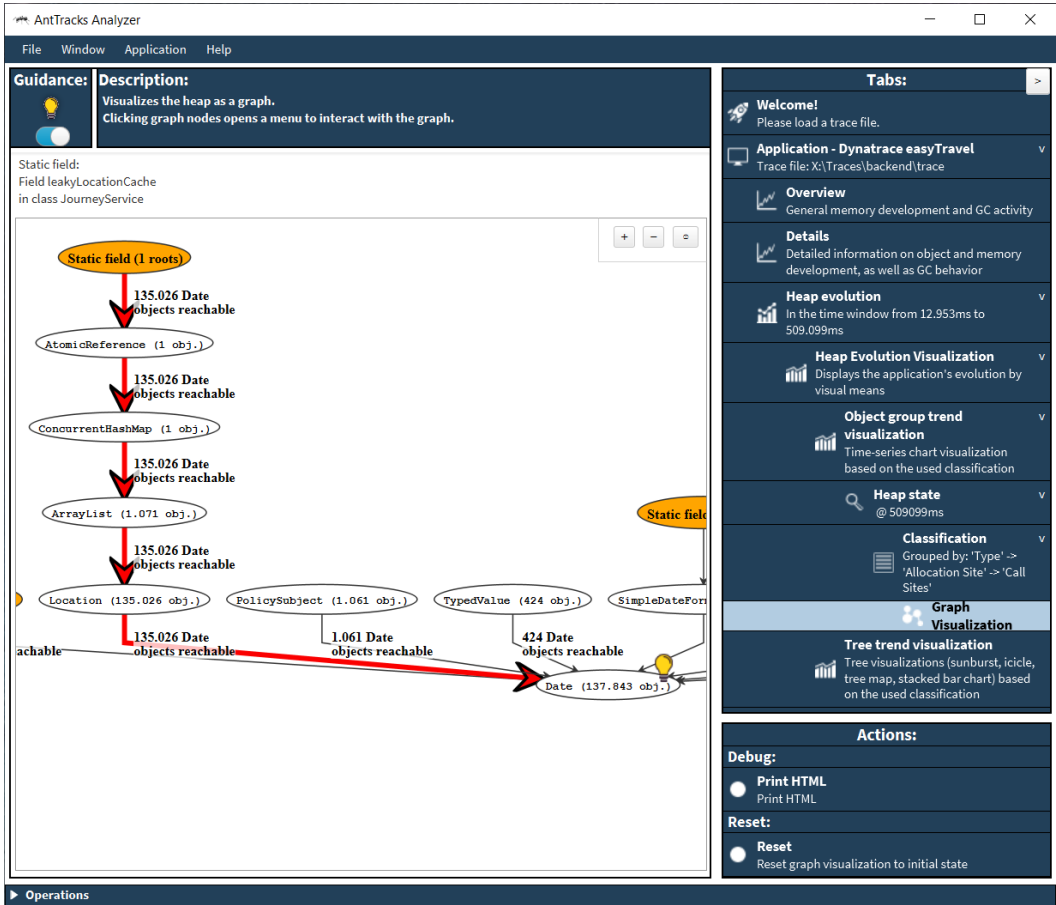


Fig. 11. The *Heap graph view* provides inspection features to analyze keep-alive relations between objects, paramount information to find the culprits of a possible memory leak.

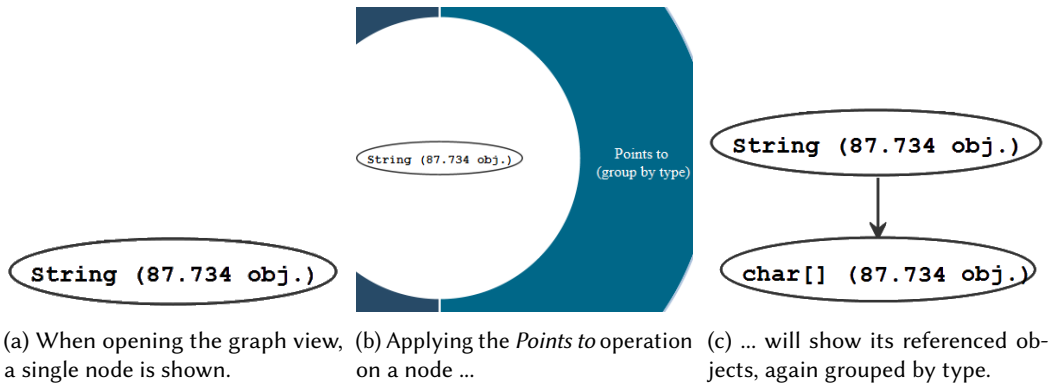


Fig. 12. Neighborhood analysis in the graph view.

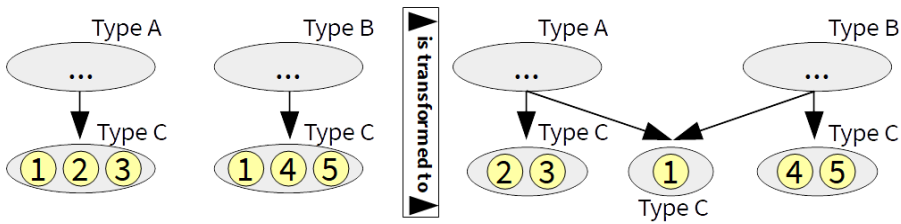


Fig. 13. The heap graph view groups objects by their types and extracts objects into separate nodes if they would be part of multiple nodes.

approach groups objects of the same type. When opening the heap graph view for objects of a specific type (e.g., inspecting all `String` objects), only a single node will be shown (for example 87.734 strings aggregated into a single node in Figure 12a). Users can explore the neighbors of a node by applying the *Points To* and *Pointed From* operations. These operations show the objects referenced by the node's objects or the objects referencing the node's objects respectively (again grouped into nodes based on their types). For example, applying the *Points To* operation (Figure 12b) on the `String` node will show all `char[]` objects referenced by the strings, as shown in Figure 12c.

Applying these neighborhood operations multiple times could lead to a situation in which one object is contained in two different nodes, which is illustrated on the left of Figure 13. The two top-most nodes represent objects of type A and type B, respectively. Assume that these objects point to objects of type C, namely the objects of type A point to the objects 1, 2 and 3, and the objects of type B point to the objects 1, 4 and 5. Since object 1 would be present in two different nodes, we extract it into a separate node, as shown on the right of Figure 13. Thus, after each operation that added nodes to the graph, we apply this technique on the whole graph to ensure that no heap object is contained in more than one graph node.

The view also supports operations that do not only involve the direct neighbors, but also operations to inspect *paths*. One of these operations is the *Paths to GC Roots* operation, which shows every path to GC roots starting from a selected node. Such a path represents a chain of objects that keep each other alive, starting at the object pointed by the GC root. While certain kinds of GC path inspection are also possible in other memory tools, nearly all these tools do not visualize these paths by graphical means but only in tree views. This has certain drawbacks. First, unwinding long paths in a tree view can be tedious. Second, it easily becomes confusing if multiple paths are shown. Third, tree views cannot display circular reference patterns. And lastly, most tools only support the inspection of GC root paths for a single object, not for object groups.

Since it can be rather complex to apply this view's features correctly and to interpret the resulting graph, AntTracks's guided exploration supports the user in multiple phases:

Detection #1. When the graph view is initially opened, a single node is shown, representing the heap objects for which we want to explore the paths to the GC roots.

Highlighting #1. This node is animated to draw attention to the fact that nodes are clickable.

Explanation #1. Since the user may have never investigated a memory problem before, AntTracks explains that the highlighted objects are kept alive because they are (indirectly) reachable from some GC roots that have yet to be explored.

Suggestion #1. To find out which GC roots keep the objects alive, we suggest to perform the *Paths to Most Interesting GC Roots* operation, an operation similar to the *Path to GC Roots* operations explained earlier. To create the paths to *all* roots, the *Pointed From* operation is automatically

applied multiple times, each time on the graph nodes that have been created in the previous step, until every path reaches a GC root. The *Paths to Most Interesting GC Roots*, instead of applying the *Pointed From* operation to every newly created node, applies this operation only to those nodes that reach at least 5% of the objects of the clicked node. For example, in [Figure 8c](#), the first *Pointed From* operation is applied to the `Date` node, which creates nodes for `Location`, `PolicySubject`, `TypedValue` and so on. While the *Path to GC Roots* algorithm would continue with all these nodes, the *Paths to Most Interesting GC Roots* algorithm only continues with the `Location` node. This is repeated until the graph cannot be expanded anymore, which results in the state shown in [Figure 8c](#).

Detection #2. Once the most important paths to the GC roots are shown, we automatically detect the path that reaches the most objects of the selected node, i.e., the path on which most objects may be kept alive.

Highlighting #2. The detected path is highlighted in red, and the thickness of the edges is adjusted according to the number of reachable objects, as shown in [Figure 8c](#).

Explanation #2. In the example from [Figure 8c](#), we explain that 135,026 `Date` objects are reachable from the leftmost path, while only 1,061 are reachable from the second-leftmost path. Consequently, we point out to the user that it is much more important to inspect the leftmost path than any other path. Once the most suspicious path is highlighted and its importance is explained, it is up to the user to try to “cut” the path somewhere. This cut must happen on the source code level by setting references that keep objects alive to `null`, or by removing objects from their containing data structures.

Suggestion #2. We currently suggest the user to start the source code inspection at the GC root and to traverse the references according to the types shown in the heap graph view. In future work, we will improve this suggestion step, for example by taking into account data structure boundaries or by including static source code analysis.

To inspect the root cause of `easyTravel`'s memory leak, we followed the *suggestion* to apply the *Paths to Most Interesting GC Roots* operation to find those GC roots that keep most of the `Date` objects alive. The result of this operation, including GE's *highlighting*, can be seen in [Figure 8c](#). `AntTracks`'s guided exploration *explains* that many `Date` objects are alive because they are reachable from objects along the path highlighted in red. The current GE implementation in `AntTracks` verbalizes the problem in the following way: 135,026 `Dates` are kept alive by 135,026 `Locations`. These `Locations` are kept alive by 1,071 `ArrayLists`. These `ArrayLists` are kept alive by a single `ConcurrentHashMap`. This `ConcurrentHashMap` is kept alive by a single `AtomicReference`. Finally, this `AtomicReference` is kept alive because it is stored in a static field called `locationCache` in the class `JourneyService`. To reduce the number of `Date` objects, you have to cut this path somewhere. You can achieve this by setting references to `null`, or by removing objects from their containing data structures. Also check why the `Date` objects are added in the first place. Are they contained in the mentioned data structures on purpose?

With this information, the user should be able to locate the reported objects in the source code. In this example, we looked up the variable name `locationCache` and checked the variable's usage. As the name suggests, the map serves as a cache, but its implementation was broken. There is a single line in the source code where `new ArrayList<Location>` objects are added if no matching key is already found in the cache. However, the class used as key in the `ConcurrentHashMap` neither implemented `hashCode` nor `equals`. Thus, every request (even for an already existing key) resulted in a cache miss and ultimately led to the problem that too many `Location` objects and `Date` objects

were created and kept alive. Implementing the two missing methods immediately resolved the problem.

5 GUIDED EXPLORATION OF MEMORY CHURN

To provide support for memory churn analysis, AntTracks encompasses a *short-living objects view* [90] that enables users to inspect those objects that are allocated in large quantities and die shortly afterwards. GE should help users to detect time windows that exhibit suspicious memory churn behavior as well as to guide them in finding those source code locations that should be inspected to reduce the churn.

5.1 Mapping of Memory Churn Analysis Process Steps to Views

The overall goal of memory churn analysis is to reduce the number of allocations happening in *memory churn hotspots*. Figure 14 shows a simplified model of such a memory churn analysis process. The first step is to detect a time window that covers a memory churn hotspot, i.e., a time window with strongly fluctuating memory utilization. The user then has to find out which types of objects are responsible for the churn and where these objects have been allocated. These locations can then be inspected in the source code to fix the problem. In AntTracks, these tasks are performed on two different views:

- (1) The *Details view* plots a detailed evolution of the memory footprint, where certain patterns indicate churn.
- (2) The *Short-living objects view* drills down into suspicious object groups to extract their types and allocation sites.

Both mentioned views already existed in AntTracks and have been extended with GE support as part of this work.

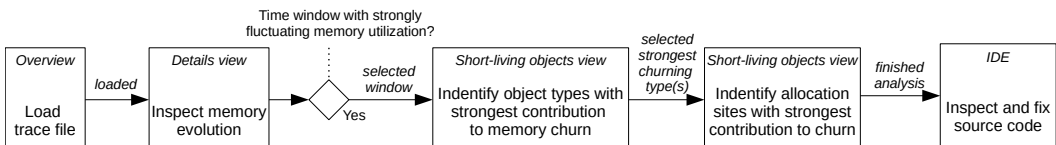


Fig. 14. Simplified task model of the typical steps performed during memory churn analysis, mapped to their corresponding AntTracks views.

5.2 Guided Exploration Support Operations for Memory Churn Analysis

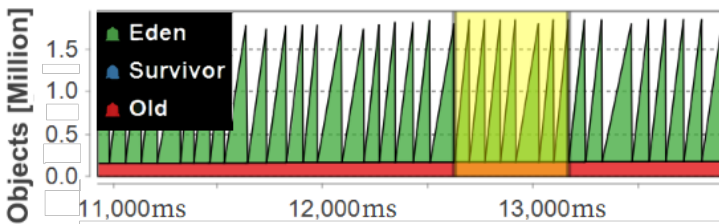
In this section, we show how GE is now supported in the two views identified in Section 5.1. We explain the views' general features and their newly supported GE support operations *Detection*, *Highlighting*, *Explanation*, and *Suggestion*.

To showcase how GE in AntTracks now supports users in analyzing and fixing memory churn, we present how the newly introduced guidance features have been used to inspect a benchmark of the *Renaissance benchmark suite* [61]. This suite is composed of modern, real-world, concurrent, and object-oriented workloads. Since this benchmark suite is rather new, it has not yet been the subject of a detailed memory study [42]. Thus, it is perfectly suited to test whether AntTracks's GE is able to guide users to the root cause of memory problems in applications even unknown to the inspector. We downloaded the benchmark suite in version 0.9, created a trace file of every benchmark and loaded these trace files into AntTracks and inspected the memory churn time windows that were automatically detected by GE. One benchmark that attracted our attention was

finagle-http. According to the benchmark’s documentation, it *sends many small Finagle HTTP requests to a Finagle HTTP server and awaits response*. All automatically detected and highlighted problem patterns in this application are shown in Figure 15 and will be explained in detail in the following.

5.2.1 Details View.

The details view (Figure 16) plots the memory consumption GC-wise, i.e., the view’s plots contain one data point at the beginning of a garbage collection (high memory consumption) and one at the end (low memory consumption). Thus, every garbage collection appears as a spike. When users investigate this view without guidance, they have to know that their task is to detect a time window



(a) Automatically detected memory churn hotspot in the *finagle-http* benchmark.

Name	Collected objects
Overall	10,019,784
▶ 0 GCs survived	10,012,077
▶ 4 GCs survived	7,686
▶ 1 GCs survived	21

(b) The guidance on the short-living objects reports that nearly all died objects did so without surviving a single garbage collection.

Name	Collected objects
Overall	10,019,784
▶ 0 GCs survived	10,012,077
▶ Promise\$WaitQueue\$anon\$4	2,494,576
▶ Promise\$Monitored	2,494,393
▶ Future\$anonfun\$onSuccess\$1	2,494,362
▶ FinagleHttp\$anonfun\$runIteration\$1\$anon\$2	2,494,361
▶ char[]	3,196

(c) Inspecting the types of the frequently dying objects reveals four types (that are automatically detected) that seem suspicious.

Name	Collected objects
Overall	10,019,784
▶ 0 GCs survived	10,012,077
...	
▶ FinagleHttp\$anonfun\$runIteration\$1\$anon\$2	2,494,361
▶ FinagleHttp\$anonfun\$runIteration\$1\$anon\$1	2,494,361

(d) Inspecting the allocation sites of these frequently dying objects leads to the source code locations that have to be checked.

Fig. 15. Memory churn analysis in AntTracks.

that contains high and frequent spikes. Yet, observing and interviewing memory monitoring tool users revealed that especially novices are often not aware of other memory problems beside memory leaks. They often lacked background knowledge to recognize high memory churn patterns as suspicious and worthy of inspection, a reason why we try to ease memory churn detection and analysis using guided exploration through the following support operations.

Detection. We apply the automatic memory churn time window detection algorithm by Weninger et al. [89] to detect memory churn hotspots.

Highlighting. Detected memory churn hotspots are highlighted with a yellow overlay (see Figure 15a), similar to memory leak time windows, following the HCI principle *consistency* [6].

Explanation. We explain the term *memory churn*, since most novice users may not be familiar with it: *AntTracks detected a time window where your application throws away over <garbage> MB per second, which is called high memory churn. This occurs when many short-living objects are being allocated in a short time span, leading to frequent garbage collections. Please note that too many GCs can slow down your application even if the GCs themselves are very quick.*

Suggestion. Our suggestion to the user is to use the *short-living objects* view to find out which objects cause the memory churn and where these objects have been allocated.

The mentioned memory churn hotspot detection algorithm is automatically run by AntTracks's GE every time a trace file is loaded. If such a hotspot is detected, as it was the case for the *finagle-http* benchmark, AntTracks suggests the user to switch to the details view to visualize it. Figure 15a shows the automatically detected memory churn hotspot in the *finagle-http* benchmark, for which AntTracks explains that about 500 MB are allocated and freed every second within the highlighted

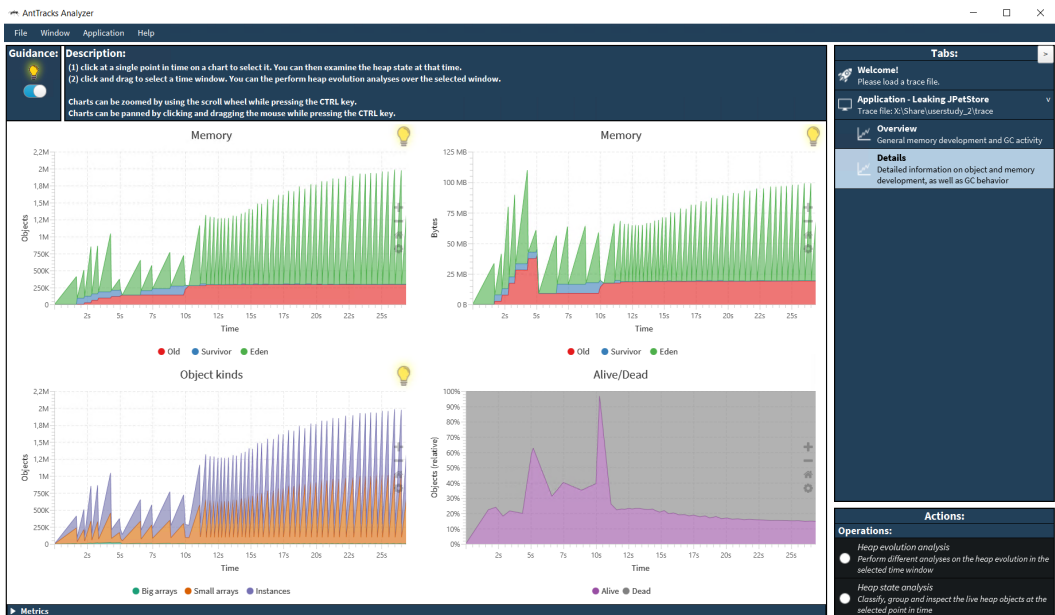


Fig. 16. The *Details* view provides more detailed information about the memory behavior, enabling the detection of spike patterns that hint at memory churn.

time window. Based on this, it suggests to explore the objects that make up this garbage in more detail on the short-living objects view.

5.2.2 Short-living Objects View.

The short-living objects view (Figure 17) calculates the *age* of each object that died within a selected time window. It uses this information to guide users to those objects that die shortly after their allocation, which are the major reason for memory churn. We define the age of a heap object as *the number of garbage collections it survived*. Even though more detailed death time algorithms exist, AntTracks uses this simple age definition as it can be reconstructed during trace parsing without additional overhead [90]. For example, the *Merlin* algorithm [25] used by *Elephant Tracks* [68, 69] can calculate more exact object death times, yet it causes a several 100-fold increase in the analyzed application’s run time [101].

The short-living objects view in AntTracks introduces a new classifier: the *Age classifier*. As shown in Figure 15b, applied on a died object, the age classifier returns the string “<x> GCs survived” as its classification. Like most views in AntTracks, the short-living objects view uses a tree table view to display the objects that were freed by the garbage collector. By default, these died objects are grouped first by age, then by type, and then by allocation site. They are sorted based on the number of objects that have been collected by the GC in the selected time window.

To make it easier for novices to learn and interpret this view, GE automatically detects suspicious objects and presents them to the user in multiple steps.

Detection #1. As the first classifier applied on the died objects is the *Age classifier*, we automatically detect how many objects died without surviving a single garbage collection.

Highlighting #1. Figure 15b shows how we highlight objects in the tree table view by assigning a special background color to the respective rows.

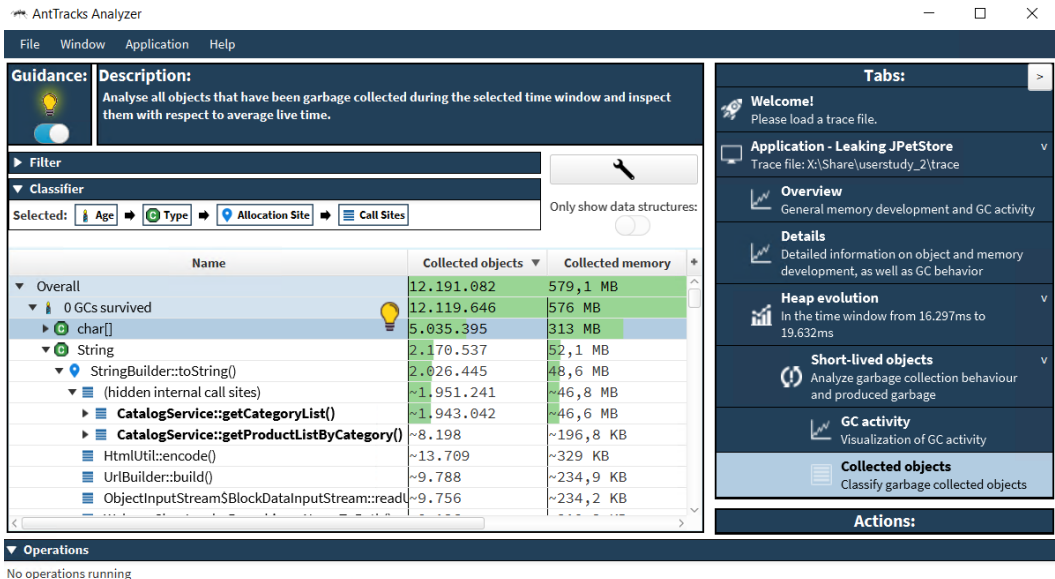


Fig. 17. The *Short-living objects view* helps the user to drill-down into object groups that contributed the most to the memory churn.

Explanation #1. A text explains that allocating large numbers of objects only to discard them shortly afterwards greatly increases the GC frequency which impacts the performance, followed by the info that *x% of the objects that died in the selected time window did not even survive a single garbage collection*. The text also explains that especially objects that do not even survive a single garbage collection are the main reason for memory churn and that they should be investigated.

Suggestion #1. As a next step, we suggest the user to expand the *0 GCs survived* row to check the types of the objects that died without surviving a single garbage collection.

Detection #2. GE detects those types that caused the most garbage within the selected time window. We empirically tested this view on various applications and determined that those types that account for at least 10% of the garbage should be highlighted. If no type accounts for at least 10% of the garbage, the type of which the most objects died is selected.

Highlighting #2. [Figure 15c](#) shows an example where four types are highlighted since each of them makes up about 25% of the overall garbage.

Explanation #2. Following text informs the users about suspicious types: *<x> types have been detected as the major suspects for memory churn: <list of types>. They account for <y>% of all objects that died without surviving a single garbage collection.*

Suggestion #2. For each suspicious type, GE suggests to inspect its allocation sites by expanding the type's row.

Detection #3. Among all allocation sites, the one at which the most objects were allocated is detected. According to our experience, most memory churn hotspots are caused by a single allocation site.

Highlighting #3. Again, the respective tree row is highlighted, as shown in [Figure 15d](#).

Explanation #3. AntTracks explains that an allocation site is the location in the code at which an object has been created, and that allocation sites where many short-living objects are created should be inspected in the source code. We further provide hints on what typical root causes of memory churn might look like. For example, allocations inside heavily-executed loops are dangerous. Another typical mistake is the careless adding and removing of boxed primitives to data structures, e.g., `ArrayList<Integer>`. Every time a primitive is added to such a data structure it is wrapped into a heap object, which can cause unnecessary memory overhead. One last example is the careless use of streams. Typical mistakes are (1) to perform multiple map operations that unnecessarily create many short-living intermediate objects, or (2) to use map when working with primitives instead of using the respective memory-efficient mapping operation such as `mapToInt`, or (3) to use `filter` operations too late in the chain of operations, leading to unnecessary operations and allocations to be performed.

Suggestion #3. We suggest to review the source code with regard to whether the executed allocations are really necessary. To reduce the number of allocations, existing objects could be reused [29, 47], for example by implementing a caching strategy and/or by using design patterns such as the *prototype pattern* or the *flyweight pattern* [20]. Future work encompasses to statically inspect the suspicious allocation site to derive context information about the allocations and to give more precise suggestions to the user.

When investigating the *finagle-http* benchmark, the first thing that is highlighted on this view is that over 99.9% of the objects that died in the selected time window (10,012,077 out of 10,019,784)

```

1 val response: Future[http.Response] = client(request)
2 for (i <- 0 until NUM_REQUESTS) {
3   Await.result(response.onSuccess { rep: http.Response =>
4     totalLength += rep.content.length
5   })
6 }

```

Listing 1. Problematic part of `FinagleHttp.runIteration()`.

```

1 val response: Future[http.Response] = client(request)
2 val h = { rep: http.Response => totalLength += rep.content.length }
3 for (i <- 0 until NUM_REQUESTS) Await.result(response.onSuccess(h))

```

Listing 2. Fixed version of `FinagleHttp.runIteration()`.

did not even survive a single garbage collection, as shown on [Figure 15b](#). Following the suggestion to inspect the types of the died objects (shown in [Figure 15c](#)) reveals that most of the died objects are divided almost equally among four types. It may be worth to mention that `finagle-http` is a Scala application, which typically produces longer type names than Java. For each of the four types, GE suggests to expand the respective row and to inspect the allocation sites of the different types. In this case, all objects of a given type that died in the selected time window were allocated at a single allocation site. The allocation sites of the first three types are within library methods which we cannot modify. Yet, the fourth type's allocation site is located in the `FinagleHttp` class, the benchmark's main class (see [Figure 15d](#)). Since Scala type names and allocation sites can be quite hard to read, we integrated rudimentary support into AntTracks's GE to translate them. For example, in the explanation text it translates `FinagleHttp$$anonfun$runIteration$1$$...` to *anonymous Scala function objects that have been allocated in method `runIteration` of the class `FinagleHttp`*.

Since such a rapid allocation and collection of anonymous function objects is unlikely to be intentional, we looked up the method's source code. [Listing 1](#) shows the problematic part. In the loop, a large number of anonymous function objects are created that wait for an HTTP request to succeed before incrementing the counter `totalLength`. [Listing 2](#) shows our fix for this problem. Only a single response handler is created which is reused for every HTTP request. This fix reduced the overall amount of allocated temporary objects by about 25% and sped up the application by about 5%.

6 PRELIMINARY USER FEEDBACK

Even though a detailed user study is still missing (but planned as future work, see [Section 8](#)), we wanted to gather preliminary user feedback to get a general idea of how AntTracks's new GE features may help novice users. To this end, we asked three PhD students and two master students that work as assistants at our institute¹ to use AntTracks and its new guidance features. All of them have a background in computer science and software engineering, and the participants reported experience in software development ranging from four to eight years. None of them had used AntTracks before and all of them stated that they had no background in memory analysis (i.e., they classified themselves as *novices* with regard to memory analysis). In separate sessions, each participant was given a memory leak and a memory churn analysis task, both of which were taken from a user study on the usability of memory monitoring tools [91]. We asked the participant to 'think aloud' [27, 31, 55], i.e., to describe what they are doing, to comment on any of their

¹None of them is involved in the development of AntTracks or this research.

concerns, and to say whatever comes to their mind while solving the given tasks. This way, we were able to collect a number of interesting *observations* and *think-aloud statements*, which we used in combination with feedback collected during a short final *interview* to initially assess AntTracks's GE system.

6.1 Study System

We selected the web application JPetStore 6 [52] as our study system. JPetStore has been widely used in research projects [17, 32, 35, 36, 83]. It models a minimalistic web shop for pets and uses a clearly structured class hierarchy. We chose JPetStore since its straightforward structure can be expressed well in a simple UML class diagram [7]. This UML diagram was handed out at the beginning of each session, which made it easy for the participants to comprehend the system's structure without being familiar with its source code. This helped to mitigate the risk of participants not finishing the study tasks [92]. To prepare the system for the study, we modified the JPetStore source code to contain two memory anomalies. We seeded the system with a memory leak by keeping shop item objects alive after their web page has been requested and a memory churn hotspot by using a Java stream inefficiently to process database responses.

We created AntTracks trace files before the user study for both the memory leak and the memory churn problem. In particular, we simulated heavy load by sending numerous requests to the different web pages of the application.

6.2 Tasks

The participants were given the trace files and had to complete the following five tasks. On each view, they were allowed to use AntTracks's GE features to receive guidance by the tool:

- *Memory leak detection*, i.e., they had to recognize and correctly classify a suspicious memory growth time window as such.
- *Trend analysis*, i.e., the participants had to find out which kinds of objects accumulated the most over this window.
- *Graph-based GC root analysis*, i.e., on the graph view, the participants had to find out which GC roots cause the memory leak. They were then shown the source code to try to fix the problem.
- *Memory churn detection*, i.e., after (hopefully) fixing the memory leak, the participants had to recognize and correctly classify a memory churn hotspot (i.e., a frequent spike pattern in the memory chart).
- *Short-living objects analysis*, i.e., they had to collect information (such as object types and allocation sites) about the churning objects. They were then again asked to locate and fix the memory problem in the source code based on their findings.

6.3 Feedback

All users were able to find the problematic source code locations relevant to the memory problems and actively expressed that they liked the guidance system (“*The light bulbs were great if you got lost or did not know what to do*”). They stated that the analysis flow and the hints during the memory leak analysis (cf. Section 4) as well as the memory churn analysis (cf. Section 5) are well chosen (“*The number and order of steps seemed natural to me*”). All users agreed that the amount of text shown in the individual hints is reasonable and not overwhelming. Despite their length, the explanations of common root causes for an observed problem and how to fix them (e.g., a list of possible root causes for high memory churn) have been praised as being very helpful to novices

(“*Experienced or power users may not need them, but they were great help for me as a beginner*”). One user also positively highlighted the use of formatting in the information text.

Nevertheless, some participants also reported that, even though the guidance helped them to *find* the problematic source code locations, they struggled to *fix* the problem (“*I know that the problem is due to a lot of Strings and Products being allocated here, but I cannot find out how to fix it; I think it has something to do with this stream*”). This suggests that future work should explore how we can further support users after their final analysis step in the monitoring tool, for example by adding guidance features to the IDE.

One improvement we already implemented based on this preliminary feedback concerns the way how we present available hints. In our initial version, each view had a single light bulb in its top-left corner that stored all the hints for this view. If new hints became available, e.g., after performing a certain action, they would be added to that light bulb’s list of hints. The users expressed that they would rather have a separate light bulb for every hint that is placed next to the UI element it refers to. This makes it more clear when a new hint becomes available, as a new light bulb appears.

As a final question, we asked the users whether they think that they would have been able to use AntTracks to find the root causes of the problems without guidance. All of them said that they think they would have eventually succeeded using a trial-and-error approach. However, all of them also stated that they are certain that it would have taken them far more time to complete the tasks.

7 GUIDED EXPLORATION IN ANOTHER DOMAIN: THREAD LOCK CONTENTION MONITORING

While the main focus of this work was to show how GE can be integrated into an interactive memory monitoring tool, we are confident that the general GE method presented in Section 3 can also be useful for monitoring tools of other domains. While more work and research still has to be performed in this direction, we initially asked authors² of an interactive thread lock contention monitoring tool [26, 72] on their opinion whether they think if GE could also be integrated into their tool.

In their positive response, they outlined how they would proceed to integrate guided exploration. They explained the typical analysis process for lock contention, all its involved steps and outlined how these steps map to views in their tool. Even though GE has not yet been integrated into their tool, we will outline their detailed response on how they could use GE to guide users on their search for thread locking problems in their applications.

7.1 Mapping of Thread Lock Contention Analysis Process Steps to Views

According to the tool authors, users who use their tool for the first time are mostly interested in (1) which shared resource (monitor) is blocking threads the most, (2) which method spends the most time waiting for the resource and (3) which method holds the resource the most and thus causes the most waiting time. This well-defined default analysis flow is also visualized in Figure 18. The analysis steps for all these tasks happen on the same view of their tool, its *drill-down view*.

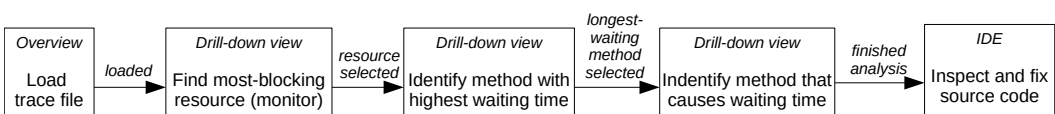


Fig. 18. Simplified task model of the typical steps performed during thread lock contention analysis, mapped to views in the monitoring tool.

²None of them is involved in the development of AntTracks or this research.

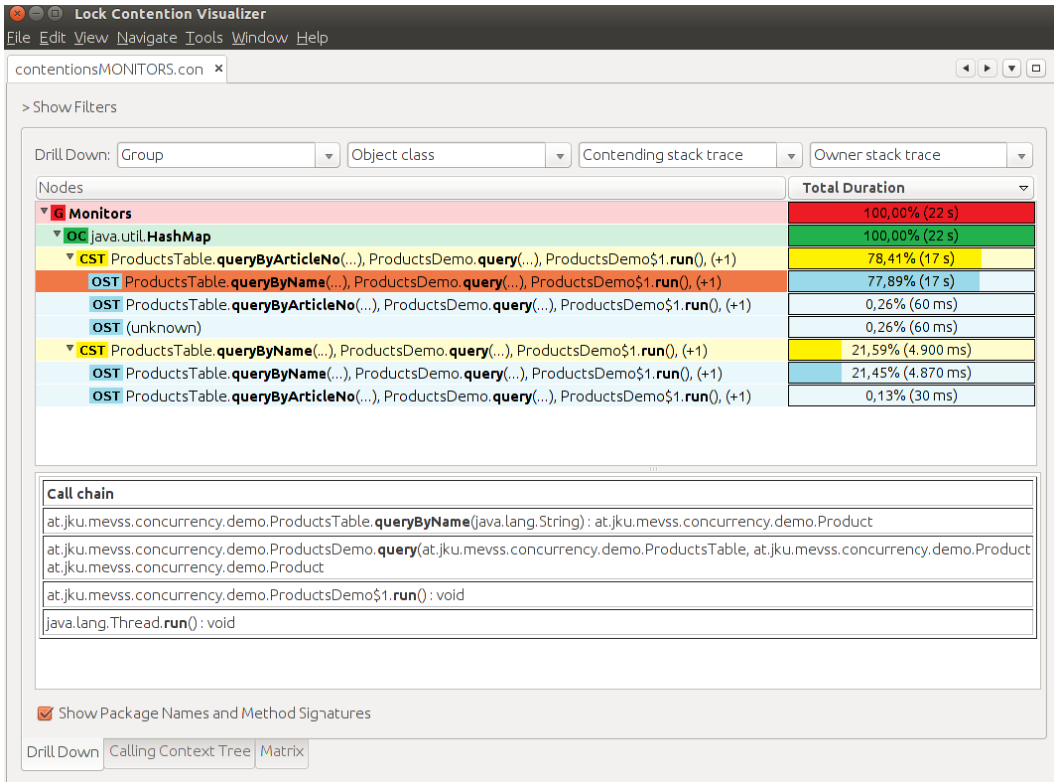


Fig. 19. The *drill-down* view in this thread lock contention monitoring tool could also be extended with guided exploration features (slightly modified figure taken with permission from [74] - Figure 4.4).

7.2 Guided Exploration Support Operations for Thread Lock Contention Analysis

The tool authors replied that they think that users could be guided well through the mentioned analysis steps and their tool's drill-down view (Figure 19) using on the four GE support operations detection, highlighting, explanation and suggestion. They also mentioned that they think that a "lightbulb-based" info mechanism to inform users about possible guidances, similar to the one used in AntTracks, could be easily integrated. In the following, we report their ideas on how to integrate GE's guidance operations into their tool.

Detection #1. Once a trace file (containing information about lock contentions that happened in the monitored application) has been loaded, the user can select criteria based on which the drill-down view groups the data. By default, the data is automatically grouped to best support the default task given in Figure 18. In this grouping, the first level of the tree splits all lock contentions that happened based on the resources that caused them. Following our GE method, the authors would improve their tool to automatically detect the resource that caused the most waiting time.

Highlighting #1. Since the majority of their UI is also structured in a table view (see Figure 19), highlighting could work in a similar way to how it is performed in AntTracks, i.e., by using colored overlays.

Explanation #1. They would explain to the user that lock contention happens when a thread T2 wants to lock a resource R, but that resource is already locked by another thread T1. T2 thus has to wait until R is released by T1, an undesired behavior in multi-threaded applications. Their tool would continue to explain that it just detected the resource that caused the most waiting time, and that the locks involving this resource should be inspected in more detail.

Suggestion #1. They would inform the user that they should expand the respective tree row to inspect *where* the most threads are waiting for the shared resource.

Detection #2. The second tree level groups all lock contentions involving a given shared resource by the methods in which threads had to wait for the resource. The tool could automatically detect the method where threads had to wait the most.

Highlighting #2. Again, the respective tree view row could be highlighted with an overlay.

Explanation #2. The tool authors would first introduce certain terminology such as *contending method* or *contending stack trace*. Following, they would explain that the highlighted row shows the method that had to wait the most for the most-blocking shared resource.

Suggestion #2. Since we just found the method that had to wait the most, the last vital information is to find out where the lock was held the most during this waiting time. Thus, the tool would suggest to step one level deeper into the tree to gather this information.

Detection #3. On the final tree level, the tool would automatically search for the method that caused the most waiting time by holding the respective shared resource while another thread wanted to obtain it.

Highlighting #3. Again, the respective tree view row could be highlighted with an overlay.

Explanation #3. After the final piece of information was collected, the explanation would summarize *which* object caused the most waiting time, *where* threads were waiting the most for this object, and *where* the object was mostly held while others were waiting.

Suggestion #3. To investigate the problem in the source code, the tool would suggest to look up both mentioned methods. Most locks in Java are caused by requests to shared resources in synchronized blocks in the form of `synchronized(sharedResource) { ... }` or by operations performed by classes of the `java.util.concurrent` package. It is the developer's task to ensure that these locked regions span as few operations as possible, and that locking is only performed where needed.

Even though further evaluation is needed, this detailed description by the lock contention monitoring tool authors fosters our belief that the GE method can also be useful to monitoring tool developers of various monitoring domains.

8 CURRENT LIMITATIONS AND FUTURE WORK

In this section, we discuss current limitations of the general GE approach, GE in the memory monitoring tool AntTracks, and how we plan to tackle these limitations in future work.

8.1 Guided Exploration in AntTracks

The main focus of this work was to explore how novice users could be better guided in interactive memory monitoring tools such as AntTracks. While the reactions during a preliminary user

feedback (see Section 6) were promisingly positive, our approach still has limitations that should be investigated in the future.

User Study. Based on the preliminary feedback we collected, we strongly believe that GE in AntTracks makes it easier to use and learn the tool, especially for novice users without expertise in memory monitoring. We presented user scenarios to demonstrate the usefulness of GE and to showcase how the guidance supports users when inspecting applications. Nevertheless, a more thorough evaluation is still missing. We thus plan to conduct a user study to compare the performance of participants who use AntTracks's GE support with the performance of those who try to resolve memory problems without guidance. It would also be interesting to check whether GE is helpful to both novice users as well as advanced users, or if advanced users prefer to use the tool without guidance.

Guided Exploration Integration in IDEs. Currently, there is a clear separation between the guided analyses in AntTracks and unguided source code inspection in the IDE. For example, after users were guided to a suspicious allocation site in AntTracks, they still have to fix the source code in their IDE without further guidance. Developing an IDE plugin [3, 11] and using hybrid static and dynamic analysis [15] would allow us to highlight suspicious code segments in the IDE, continuing GE on the source code level.

Heap Graph Visualization. In Section 4, we presented AntTracks's new heap graph view to inspect a heap state in a visual way. This feature is still under development and evolves constantly. In the future, we plan to report in more detail on this new visualization technique, how it compares to other techniques for heap visualization [59, 66, 95–98], and how its guidance can be further extended.

8.2 Guided Exploration in General

In general, our GE method presented in Section 3 is not restricted to the domain of memory monitoring. Yet, while its core idea could also be useful to monitoring tools of other domains, future work still has to be performed to evaluate the approach's general applicability across domains.

Generalization. To mitigate the generalizability problem of our approach, we asked other monitoring tool developers for their opinion regarding the feasibility of implementing GE in their tool and if they think that their users would profit from it, as shown in Section 7. Despite their positive feedback and a detailed explanation on how they would integrate GE into their tool, more monitoring tools from different domains should be inspected for possible GE support in the future.

Furthermore, the GE process outlined in Section 3 currently serves more as an overview of the four guidance operations (detection, highlighting, explanation and suggestion). Yet, we did not discuss in detail how to implement them, for example based on certain characteristics the monitoring tool exhibits. Rabiser et al. [62, 63] already explored various kinds of characteristics based on which different monitoring tools could be classified and compared, even across different domains. We think that it may be possible to link certain characteristics of a tool to suggested ways of how to implement the different guidance operations, for example different ways of problem detection or highlighting. Such relations between tool characteristics and possible guidance operation implementations could be explored and discussed in more detail in the future.

Guided Exploration Tool Integration. In this work, we presented how GE has been integrated into AntTracks, using clickable lightbulb icons near UI elements for which guidances exist. Yet, we are certain that there are other possibilities on how to visualize guidances. To explore these possibilities, for example, one could organize a workshop where the participants should inspect

existing monitoring tools and discuss commonalities across them. Separated into groups, they could work out GE prototypes, i.e., how they would integrate guidance into these tools, and comment on each others ideas. These discussions could lead to a more detailed description and understanding of the GE approach in the future, as well as to better ways on how to visually support it.

Rule-based Guidance Definition. AntTracks uses a consistent way to display *explanations* and *suggestions* throughout its various views. Unfortunately, the code to *detect* suspicious information as well as to *highlight* the respective UI region is currently hard-coded within every view.

In the future, instead of modifying the underlying source code, we would like to be able to define rules for guided exploration externally using a domain specific language that allows definitions such as “*If pattern X is detected: highlight UI element Y, show explanation text Z, suggest steps A and B*”. On the one hand, this poses various challenges such as how to access and abstract the data used by the view or how to specify (customized) UI element highlighting. On the other hand, it would make it much easier to integrate guided exploration into other tools besides AntTracks.

9 CONCLUSIONS

In this work, we presented *guided exploration*, a method that can be integrated into interactive monitoring tools in order to improve their learnability and usability. The goal of guided exploration is to support novice users, i.e., users who may lack the experience to recognize and analyze program behavior anomalies on their own. Guided exploration makes a tool easier to use by *guiding* users through the analysis process and helping them to *explore* the collected data until the root cause of a problem is found.

In general, guided exploration is an iteration of four support operations performed by a tool. According to GE, a tool should automatically (1) *detect* the most interesting piece of information in the current view, (2) *highlight* the UI elements where this information can be found, (3) *explain* the required background knowledge and the rationale why the highlighted information is important, and (4) *suggest* further analysis steps based on these findings.

In this work, we focused on guided exploration in interactive memory monitoring tools. We integrated our guidance approach into the memory monitoring tool AntTracks, namely for the processes of memory leak analysis and memory churn analysis. For both analyses, we explained in detail how the four support operations of guided exploration have been implemented. To demonstrate their applicability, we presented two user scenarios where two applications have been analyzed by following the explanations and suggestions of AntTracks’s new guided exploration system.

We hope that guided exploration can be of help to researchers and developers of interactive (memory) monitoring tools to better structure their analysis processes, making them more accessible to novice users.

ACKNOWLEDGMENT

The financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development, and Dynatrace is gratefully acknowledged.

REFERENCES

- [1] Alain Abran, Adel Khelifi, Witold Suryn, and Ahmed Seffah. 2003. Usability Meanings and Interpretations in ISO Standards. *Software Quality Journal* 11, 4 (2003), 325–338. <https://doi.org/10.1023/A:1025869312943>
- [2] Tarek M. Ahmed, Cor-Paul Bezemer, Tse-Hsun Chen, Ahmed E. Hassan, and Weiyi Shang. 2016. Studying the Effectiveness of Application Performance Management (APM) Tools for Detecting Performance Regressions for Web

- Applications: An Experience Report. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR)*. 1–12. <https://doi.org/10.1145/2901739.2901774>
- [3] Sebastian Baltes, Peter Schmitz, and Stephan Diehl. 2014. Linking Sketches and Diagrams to Source Code Artifacts. In *Proceedings of the 22nd ACM SIGSOFT International Symp. on Foundations of Software Engineering (FSE)*. 743–746. <https://doi.org/10.1145/2635868.2661672>
- [4] André Bauer, Marwin Züfle, Johannes Grohmann, Norbert Schmitt, Nikolas Herbst, and Samuel Kounev. 2020. An Automated Forecasting Framework based on Method Recommendation for Seasonal Time Series. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE)*. ACM, 48–55. <https://doi.org/10.1145/3358960.3379123>
- [5] Verena Bitto, Philipp Lengauer, and Hanspeter Mössenböck. 2015. Efficient Rebuilding of Large Java Heaps from Event Traces. In *Proceedings of the Principles and Practices of Programming on The Java Platform (PPPJ)*. 76–89. <https://doi.org/10.1145/2807426.2807433>
- [6] Alan Blackwell and Thomas Green. 2003. CHAPTER 5 - Notational Systems - The Cognitive Dimensions of Notations Framework. In *HCI Models, Theories, and Frameworks*. Morgan Kaufmann, 103 – 133. <https://doi.org/10.1016/B978-155860808-5/50005-8>
- [7] Grady Booch, James E. Rumbaugh, and Ivar Jacobson. 2005. *The Unified Modeling Language User Guide - Covers UML 2.0 (Second Edition)*. Addison-Wesley.
- [8] Adriana E. Chis. 2008. Automatic Detection of Memory Anti-Patterns. In *Comp. to the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 925–926. <https://doi.org/10.1145/1449814.1449911>
- [9] Adriana E. Chis, Nick Mitchell, Edith Schonberg, Gary Sevitsky, Patrick O’Sullivan, Trevor Parsons, and John Murphy. 2011. Patterns of Memory Inefficiency. In *Proceedings of the 25th European Conference on Object-Oriented Programming (ECOOP)*. 383–407. https://doi.org/10.1007/978-3-642-22655-7_18
- [10] Maria Christakis and Christian Bird. 2016. What Developers Want and Need from Program Analysis: An Empirical Study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 332–343. <https://doi.org/10.1145/2970276.2970347>
- [11] Jürgen Cito, Philipp Leitner, Christian Bosshard, Markus Knecht, Gene Mazlami, and Harald C. Gall. 2018. PerformanceHat: Augmenting Source Code with Runtime Performance Traces in the IDE. In *Comp. of the 40th International Conference on Software Engineering (ICSE)*. 41–44. <https://doi.org/10.1145/3183440.3183481>
- [12] D. Christopher Dryer. 1997. Wizards, Guides, and Beyond: Rational and Empirical Methods for Selecting Optimal Intelligent User Interface Agents. In *Proceedings of the 2nd International Conference on Intelligent User Interfaces (IUI)*. 265–268. <https://doi.org/10.1145/238218.238347>
- [13] Bruno Dufour, Karel Driesen, Laurie Hendren, and Clark Verbrugge. 2003. Dynamic Metrics for Java. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 149–168. <https://doi.org/10.1145/949305.949320>
- [14] Dynatrace. 2017. *Demo Applications: easyTravel*. <https://community.dynatrace.com/community/display/DL/Demo+Applications+-+easyTravel>
- [15] Michael D. Ernst. 2003. Static and Dynamic Analysis: Synergy and Duality. In *Workshop on Dynamic Analysis (WODA)*. 24–27. <https://homes.cs.washington.edu/~mernst/pubs/staticdynamic-woda2003.pdf>
- [16] Alexander Felfernig, Gerald Ninaus, Harald Grabner, Florian Reinfrank, Leopold Weninger, Dennis Pagano, and Walid Maalej. 2013. An Overview of Recommender Systems in Requirements Engineering. In *Managing Requirements Knowledge*. 315–332. https://doi.org/10.1007/978-3-642-34419-0_14
- [17] Florian Fittkau, Phil Stelzer, and Wilhelm Hasselbring. 2014. Live Visualization of Large Software Landscapes for Ensuring Architecture Conformance. In *Proceedings of the Workshops & Tool Demos Track of the European Conference on Software Architecture (ECSAW)*. 28:1–28:4. <https://doi.org/10.1145/2642803.2642831>
- [18] Eelke Folmer and Jan Bosch. 2003. Usability Patterns in Software Architecture. In *Proceedings of the 10th International Conference on Human-Computer Interaction (HCI)*. 93–97. <https://doi.org/10.1109/DSAA.2018.00057>
- [19] Tak-Chung Fu. 2011. A Review on Time Series Data Mining. *Eng. Appl. Artif. Intell.* 24, 1 (2011), 164–181. <https://doi.org/10.1016/j.engappai.2010.09.007>
- [20] Erich Gamma, Richard Helm, Ralph E. Johnson, and John M. Vlissides. 1993. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In *Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP)*. 406–431. https://doi.org/10.1007/3-540-47910-4_21
- [21] Josefina Guerrero Garcia, Jean Vanderdonck, and Christophe Lemaigre. 2008. Identification Criteria in Task Modeling. In *Proceedings of the 1st TC 13 IFIP Human-Computer Interaction Symposium (HCIS)*, Vol. 272. 7–20. https://doi.org/10.1007/978-0-387-09678-0_2
- [22] Bernhard Göschlberger and Peter A. Bruck. 2017. Gamification in Mobile and Workplace Integrated Microlearning. In *Proceedings of the 19th International Conference on Information Integration and Web-based Applications & Services*

- (iiWAS) (Salzburg, Austria). 545–552. <https://doi.org/10.1145/3151759.3151795>
- [23] Juho Hamari, Jonna Koivisto, and Harri Sarsa. 2014. Does Gamification Work? - A Literature Review of Empirical Studies on Gamification. In *Proceedings of the 47th Hawaii International Conference on System Sciences (HICSS)*. 3025–3034. <https://doi.org/10.1109/HICSS.2014.377>
- [24] William E. Hefley and Dianne Murray. 1993. Intelligent User Interfaces. In *Proceedings of the 1st International Conference on Intelligent User Interfaces (IUI)*. 3–10. <https://doi.org/10.1145/169891.169892>
- [25] Matthew Hertz, Stephen M. Blackburn, J. Eliot B. Moss, Kathryn S. McKinley, and Darko Stefanovic. 2006. Generating Object Lifetime Traces with Merlin. *ACM Trans. Program. Lang. Syst.* 28, 3 (2006), 476–516. <https://doi.org/10.1145/1133651.1133654>
- [26] Peter Hofer, David Gnedt, Andreas Schörghenhuber, and Hanspeter Mössenböck. 2016. Efficient Tracing and Versatile Analysis of Lock Contention in Java Applications on the Virtual Machine Level. In *Proceedings of the 7th ACM/SPEC International Conference on Performance Engineering (ICPE)*. 263–274. <https://doi.org/10.1145/2851553.2851559>
- [27] Andreas Holzinger. 2005. Usability Engineering Methods for Software Developers. *Commun. ACM* 48, 1 (2005), 71–74. <https://doi.org/10.1145/1039539.1039541>
- [28] Michal Hucko, Ladislav Gazo, Peter Simún, Matej Valky, Róbert Móro, Jakub Simko, and Mária Bielíková. 2019. YesElf: Personalized Onboarding for Web Applications. In *Adjunct Publication of the 27th Conference on User Modeling, Adaptation and Personalization (UMAP)*. 39–44. <https://doi.org/10.1145/3314183.3324978>
- [29] Alejandro Infante and Alexandre Bergel. 2017. Object Equivalence: Revisiting Object Equality Profiling (An Experience Report). In *Proceedings of the 13th ACM SIGPLAN International Symp. on Dynamic Languages (DLS)*. 27–38. <https://doi.org/10.1145/3133841.3133844>
- [30] V. López Jaquero, F. Montero, J.P. Molina, and P. González. 2009. *Intelligent User Interfaces: Past, Present and Future*. Springer London, 1–12. https://doi.org/10.1007/978-1-84800-136-7_18
- [31] Monique W. M. Jaspers, Thiemo Steen, Cor van den Bos, and Maud M. Geenen. 2004. The Think Aloud Method: A Guide to User Interface Design. *I. J. Medical Informatics* 73, 11–12 (2004), 781–795. <https://doi.org/10.1016/j.ijmedinf.2004.08.003>
- [32] Zhen Ming Jiang, Ahmed E. Hassan, Gilbert Hamann, and Parminder Flora. 2009. Automated Performance Analysis of Load Tests. In *Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM)*. 125–134. <https://doi.org/10.1109/ICSM.2009.5306331>
- [33] Brittany Johnson, Yoonki Song, Emerson R. Murphy-Hill, and Robert W. Bowdidge. 2013. Why Don't Software Developers Use Atomic Analysis Tools to Find Bugs?. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*. 672–681. <https://doi.org/10.1109/ICSE.2013.6606613>
- [34] Vivien Johnston. 2019. A Framework for the Development of a Dynamic Adaptive Intelligent User Interface to Enhance the User Experience. In *Proceedings of the 31st European Conference on Cognitive Ergonomics (ECCE)*. 32–35. <https://doi.org/10.1145/3335082.3335125>
- [35] Reiner Jung and Marc Adolf. 2018. The JPetStore Suite: A Concise Experiment Setup for Research. In *Proceedings of the 9th Symposium on Software Performance (SSP)*. <http://eprints.uni-kiel.de/48775/>
- [36] Reiner Jung, Marc Adolf, and Christoph Dornieden. 2017. Towards Extracting Realistic User Behavior Models. *Softwaretechnik-Trends* 37, 3 (2017). <http://eprints.uni-kiel.de/40365/>
- [37] Marius Koller and Gerrit Meixner. 2016. Task Models in Practice: Are There Special Requirements for the Use in Daily Work?. In *Proceedings of 18th International Conference on Human-Computer Interaction (HCI) - Theory, Design, Development and Practice*. 488–497. https://doi.org/10.1007/978-3-319-39510-4_45
- [38] Steinar Kristoffersen. 2008. Learnability and Robustness of User Interfaces. Towards a Formal Analysis of Usability Design Principles. In *Proceedings of the 3rd International Conference on Software and Data Technologies (ICSOFI), Volume SE/MUSE/GSDCA*. 261–268.
- [39] Philipp Lengauer, Verena Bitto, Stefan Fitzek, Markus Weninger, and Hanspeter Mössenböck. 2016. Efficient Memory Traces with Full Pointer Information. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ)*. 4:1–4:11. <https://doi.org/10.1145/2972206.2972220>
- [40] Philipp Lengauer, Verena Bitto, and Hanspeter Mössenböck. 2015. Accurate and Efficient Object Tracing for Java Applications. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE)*. 51–62. <https://doi.org/10.1145/2668930.2688037>
- [41] Philipp Lengauer, Verena Bitto, and Hanspeter Mössenböck. 2016. Efficient and Viable Handling of Large Object Traces. In *Proceedings of the 7th ACM/SPEC International Conference on Performance Engineering (ICPE)*. 249–260. <https://doi.org/10.1145/2851553.2851555>
- [42] Philipp Lengauer, Verena Bitto, Hanspeter Mössenböck, and Markus Weninger. 2017. A Comprehensive Java Benchmark Study on Memory and Garbage Collection Behavior of DaCapo, DaCapo Scala, and SPECjvm2008. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering (ICPE)*. 3–14. <https://doi.org/10.1145/3151759.3151795>

[//doi.org/10.1145/3030207.3030211](https://doi.org/10.1145/3030207.3030211)

- [43] Jinbo Li, Hesam Izakian, Witold Pedrycz, and Iqbal Jamal. 2021. Clustering-based Anomaly Detection in Multivariate Time Series Data. *Appl. Soft Comput.* 100 (2021), 106919. <https://doi.org/10.1016/j.asoc.2020.106919>
- [44] Jie Liang and Mao Lin Huang. 2010. Highlighting in Information Visualization: A Survey. In *Proceedings of the 14th International Conference on Information Visualisation (IV)*. 79–85. <https://doi.org/10.1109/IV.2010.21>
- [45] Quentin Limbourg and Jean Vanderdonckt. 2003. *The Handbook of Task Analysis for Human-Computer Interaction*. CRC Press, Chapter Comparing Task Models for User Interface Design, 135–154.
- [46] Victor López-Jaquero and Francisco Montero Simarro. 2007. Comprehensive Task and Dialog Modelling. In *Proceedings of the 12th International Conference on Human-Computer Interaction (HCI) - Interaction Design and Usability*, Vol. 4550. Springer, 1149–1158. https://doi.org/10.1007/978-3-540-73105-4_125
- [47] Darko Marinov and Robert O’Callahan. 2003. Object Equality Profiling. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. 313–325. <https://doi.org/10.1145/949305.949333>
- [48] Evan K. Maxwell, Godmar Back, and Naren Ramakrishnan. 2010. Diagnosing Memory Leaks using Graph Mining on Heap Dumps. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 115–124. <https://doi.org/10.1145/1835804.1835822>
- [49] Mark T. Maybury. 1999. Intelligent User Interfaces: An Introduction. In *Proceedings of the 4th International Conference on Intelligent User Interfaces (IUI)*. 3–4. <https://doi.org/10.1145/291080.291081>
- [50] Karen L. McGraw and Bruce A. McGraw. 1997. Wizards, Coaches, Advisors, and More: A Performance Support Primer. In *Ext. Abstr. on Human Factors in Computing Systems (Atlanta, Georgia)*. 152–153. <https://doi.org/10.1145/1120212.1120318>
- [51] Gerrit Meixner, Marc Seissler, and Kai Breiner. 2011. Model-Driven Useware Engineering. In *Model-Driven Development of Advanced User Interfaces*. 1–26. https://doi.org/10.1007/978-3-642-14562-9_1
- [52] MyBatis. 2016. *JPetStore*. <http://mybatis.org/jpetstore-6/>
- [53] Raymond H Myers and Raymond H Myers. 1990. *Classical and modern regression with applications*. Vol. 2. Duxbury press Belmont, CA.
- [54] Jakob Nielsen. 1993. *Usability Engineering*. Academic Press.
- [55] Mie Nørgaard and Kasper Hornbæk. 2006. What do Usability Evaluators do in Practice?: An Explorative Study of Think-aloud Testing. In *Proceedings of the Conference on Designing Interactive Systems (DIS)*. 209–218. <https://doi.org/10.1145/1142405.1142439>
- [56] Oracle. 2020. *VisualVM: All-in-One Java Troubleshooting Tool*. <https://visualvm.github.io/>
- [57] J. D. Ornelas, J. C. Silva, and J. L. Silva. 2016. USS: User support system. In *Proceedings of the 11th Iberian Conference on Information Systems and Technologies (CISTI)*. 1–6. <https://doi.org/10.1109/CISTI.2016.7521412>
- [58] Fabio Paternò, Cristiano Mancini, and Silvia Meniconi. 1997. ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models. In *Proceedings of the IFIP TC13 International Conference on Human-Computer Interaction (INTERACT)*. 362–369.
- [59] Wim De Pauw and Gary Sevitsky. 1999. Visualizing Reference Patterns for Solving Memory Leaks in Java. In *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP)*. 116–134. https://doi.org/10.1007/3-540-48743-3_6
- [60] Manjula Peiris and James H. Hill. 2016. Automatically Detecting "Excessive Dynamic Memory Allocations" Software Performance Anti-Pattern. In *Proceedings of the 7th ACM/SPEC International Conference on Performance Engineering (ICPE)*. 237–248. <https://doi.org/10.1145/2851553.2851563>
- [61] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tuma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: Benchmarking Suite for Parallel Applications on the JVM. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 31–47. <https://doi.org/10.1145/3314221.3314637>
- [62] Rick Rabiser, Sam Guinea, Michael Vierhauser, Luciano Baresi, and Paul Grünbacher. 2017. A Comparison Framework for Runtime Monitoring Approaches. *J. Syst. Softw.* 125 (2017), 309–321. <https://doi.org/10.1016/j.jss.2016.12.034>
- [63] Rick Rabiser, Klaus Schmid, Holger Eichelberger, Michael Vierhauser, Sam Guinea, and Paul Grünbacher. 2019. A Domain Analysis of Resource and Requirements Monitoring: Towards a Comprehensive Model of the Software Monitoring Domain. *Inf. Softw. Technol.* 111 (2019), 86–109. <https://doi.org/10.1016/j.infsof.2019.03.013>
- [64] Mukund Raghothaman, Sulekha Kulkarni, Kihong Heo, and Mayur Naik. 2018. User-Guided Program Reasoning Using Bayesian Inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 722–735. <https://doi.org/10.1145/3192366.3192417>
- [65] Shaina Raza and Chen Ding. 2019. Progress in Context-aware Recommender Systems - An Overview. *Comput. Sci. Rev.* 31 (2019), 84–97. <https://doi.org/10.1016/j.cosrev.2019.01.001>

- [66] Steven P. Reiss. 2009. Visualizing the Java Heap to Detect Memory Problems. In *Proceedings of the 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*. 73–80. <https://doi.org/10.1109/VISSOFT.2009.5336418>
- [67] J. Renz, T. Staubitz, J. Pollak, and C. Meinel. 2014. Improving the Onboarding User Experience in MOOCs. In *Proceedings of the 6th International Conference on Education and New Learning Technologies (EDULEARN)* (Barcelona, Spain). 3931–3941.
- [68] Nathan P. Ricci, Samuel Z. Guyer, and J. Eliot B. Moss. 2011. Elephant Tracks: Generating Program Traces with Object Death Records. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java (PPPJ)*. 139–142. <https://doi.org/10.1145/2093157.2093178>
- [69] Nathan P. Ricci, Samuel Z. Guyer, and J. Eliot B. Moss. 2013. Elephant Tracks: Portable Production of Complete and Precise GC Traces. In *Proceedings of the International Symposium on Memory Management (ISMM)*. 109–118. <https://doi.org/10.1145/2491894.2466484>
- [70] Cynthia K. Riemenschneider and Bill C. Hardgrave. 2001. Explaining Software Development Tool Use with the Technology Acceptance Model. *Journal of Computer Information Systems (JCIS)* 41, 4 (2001), 1–8. <https://www.tandfonline.com/doi/abs/10.1080/08874417.2001.11647015>
- [71] Roger C Schank, Tamara R Berman, and Kimberli A Macpherson. 1999. Learning by Doing. *Instructional-design theories and models: A new paradigm of instructional theory 2*, 2 (1999), 161–181.
- [72] Andreas Schörgenhumer, Peter Hofer, David Gnedt, and Hanspeter Mössenböck. 2017. Efficient Sampling-based Lock Contention Profiling for Java. In *Proceedings of the 8th ACM/SPEC International Conference on Performance Engineering (ICPE)*. 331–334. <https://doi.org/10.1145/3030207.3030234>
- [73] Andreas Schörgenhumer, Mario Kahlhofer, Paul Grünbacher, and Hanspeter Mössenböck. 2019. Can we Predict Performance Events with Time Series Data from Monitoring Multiple Systems?. In *Companion of the ACM/SPEC International Conference on Performance Engineering ICPE*. 9–12. <https://doi.org/10.1145/3302541.3313101>
- [74] Andreas Schörgenhumer. 2017. *Efficient Sampling-based Lock Contention Profiling in Java*. Master's thesis. Johannes Kepler University, Institute for System Software. <https://epub.jku.at/obvulihs/content/titleinfo/1825350>
- [75] Connie U. Smith and Lloyd G. Williams. 2000. Software Performance Antipatterns. In *Proceedings of the Second International Workshop on Software and Performance (WOSP)*. 127–136. <https://doi.org/10.1145/350391.350420>
- [76] Ken Soong, Xin Fu, and Yang Zhou. 2018. Optimizing New User Experience in Online Services. In *Proceedings of the 5th IEEE International Conference on Data Science and Advanced Analytics (DSAA)*. 442–449. <https://doi.org/10.1109/DSAA.2018.00057>
- [77] Miroslaw Staron, Wilhelm Meding, Jörgen Hansson, Christoffer Höglund, Kent Niesel, and Vilhelm Bergmann. 2014. Dashboards for Continuous Monitoring of Quality for Software Product under Development. In *Relating System Quality and Software Architecture*. 209–229. <https://doi.org/10.1016/b978-0-12-417009-4.00008-9>
- [78] Piyawadee Noi Sukaviriya and James D. Foley. 1990. Coupling a UI Framework with Automatic Generation of Context-Sensitive Animated Help. In *Proceedings of the 3rd Annual ACM Symp. on User Interface Software and Technology (UIST)*. 152–166. <https://doi.org/10.1145/97924.97942>
- [79] Claudia Szabo. 2015. Novice Code Understanding Strategies during a Software Maintenance Assignment. In *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering (ICSE)*. 276–284. <https://doi.org/10.1109/ICSE.2015.341>
- [80] Eclipse Foundation. 2020. *Eclipse Memory Analyzer (MAT)*. <https://www.eclipse.org/mat/>
- [81] Doug Tidwell and Jeanette Fucella. 1997. TaskGuides: Instant Wizards on the Web. In *Proceedings of the 15th Annual International Conference on Computer Documentation (SIGDOC)*. 263–272. <https://doi.org/10.1145/263367.263401>
- [82] Omer Tripp, Salvatore Guarnieri, Marco Pistoia, and Aleksandr Aravkin. 2014. ALETHEIA: Improving the Usability of Static Security Analysis. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 762–774. <https://doi.org/10.1145/2660267.2660339>
- [83] Jinshui Wang, Xin Peng, Zhenchang Xing, and Wenyun Zhao. 2011. An Exploratory Study of Feature Location Process: Distinct Phases, Recurring Patterns, and Elementary Actions. In *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM)*. 213–222. <https://doi.org/10.1109/ICSM.2011.6080788>
- [84] Qingsong Wen, Jingkun Gao, Xiaomin Song, Liang Sun, Huan Xu, and Shenghuo Zhu. 2019. RobustSTL: A Robust Seasonal-Trend Decomposition Algorithm for Long Time Series. In *Proceedings of the Thirty-Third Conference on Artificial Intelligence (AAAI)*. 5409–5416. <https://doi.org/10.1609/aaai.v33i01.33015409>
- [85] Markus Weninger et al. 2020. *AntTracks*. <http://mevss.jku.at/AntTracks>
- [86] Markus Weninger, Elias Gander, and Hanspeter Mössenböck. 2018. Analyzing the Evolution of Data Structures Over Time in Trace-Based Offline Memory Monitoring. In *Proceedings of the 9th Symp. on Software Performance (SSP)*. 64–66. http://pi.informatik.uni-siegen.de/stt/39_3/01_Fachgruppenberichte/SSP18/WeningerGanderMoessenboeck18.pdf
- [87] Markus Weninger, Elias Gander, and Hanspeter Mössenböck. 2018. Utilizing Object Reference Graphs and Garbage Collection Roots to Detect Memory Leaks in Offline Memory Monitoring. In *Proceedings of the 15th International*

- Conference on Managed Languages & Runtimes (ManLang)*. 14:1–14:13. <https://doi.org/10.1145/3237009.3237023>
- [88] Markus Weninger, Elias Gander, and Hanspeter Mössenböck. 2019. Analyzing Data Structure Growth Over Time to Facilitate Memory Leak Detection. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering (ICPE)*. 273–284. <https://doi.org/10.1145/3297663.3310297>
- [89] Markus Weninger, Elias Gander, and Hanspeter Mössenböck. 2019. Detection of Suspicious Time Windows In Memory Monitoring. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR)*. 95–104. <https://doi.org/10.1145/3357390.3361025>
- [90] Markus Weninger, Elias Ganer, and Hanspeter Mössenböck. 2020. Investigating High Memory Churn via Object Lifetime Analysis to Improve Software Performance. In *Proceedings of the 11th Symp. on Software Performance (SSP)*. http://pi.informatik.uni-siegen.de/stt/39_4/01_Fachgruppenberichte/SSP2019/SSP2019_Weninger.pdf
- [91] Markus Weninger, Paul Grünbacher, Elias Gander, and Andreas Schörghener. 2020. Evaluating an Interactive Memory Analysis Tool: Findings from a Cognitive Walkthrough and a User Study. *Proc. ACM Hum.-Comput. Interact.* 4, EICS, Article 75 (June 2020), 37 pages. <https://doi.org/10.1145/3394977>
- [92] Markus Weninger, Paul Grünbacher, Huihui Zhang, Tao Yue, and Shaikat Ali. 2018. Tool Support for Restricted Use Case Specification: Findings from a Controlled Experiment. In *Proceedings of the 25th Asia-Pacific Software Engineering Conference (APSEC)*. 21–30. <https://doi.org/10.1109/APSEC.2018.00016>
- [93] Markus Weninger, Philipp Lengauer, and Hanspeter Mössenböck. 2017. User-centered Offline Analysis of Memory Monitoring Data. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering (ICPE)*. 357–360. <https://doi.org/10.1145/3030207.3030236>
- [94] Markus Weninger, Lukas Makor, Elias Gander, and Hanspeter Mössenböck. 2019. AntTracks TrendViz: Configurable Heap Memory Visualization Over Time. In *Comp. of the 2019 ACM/SPEC International Conference on Performance Engineering (ICPE)*. 29–32. <https://doi.org/10.1145/3302541.3313100>
- [95] Markus Weninger, Lukas Makor, and Hanspeter Mössenböck. 2019. Memory Leak Visualization using Evolving Software Cities. In *Proceedings of the 10th Symp. on Software Performance (SSP)*. 44–46. http://pi.informatik.uni-siegen.de/stt/39_4/01_Fachgruppenberichte/SSP2019/SSP2019_Weninger.pdf
- [96] Markus Weninger, Lukas Makor, and Hanspeter Mössenböck. 2020. Memory Cities: Visualizing Heap Memory Evolution Using the Software City Metaphor. In *Proceedings of the Working Conference on Software Visualization, (VISSOFT)*. 110–121. <https://doi.org/10.1109/VISSOFT51673.2020.00017>
- [97] Markus Weninger, Lukas Makor, and Hanspeter Mössenböck. 2020. Heap Evolution Analysis Using Tree Visualizations. In *Proceedings of the 11th Symp. on Software Performance (SSP)*. http://pi.informatik.uni-siegen.de/stt/39_4/01_Fachgruppenberichte/SSP2019/SSP2019_Weninger.pdf
- [98] Markus Weninger, Lukas Makor, and Hanspeter Mössenböck. 2020. Memory Leak Analysis using Time-Travel-based and Timeline-based Tree Evolution Visualizations. In *Proceedings of the Conference on Smart Tools and Apps for Graphics - Eurographics Italian Chapter Conference*. <https://doi.org/10.2312/stag.20201241>
- [99] Markus Weninger and Hanspeter Mössenböck. 2018. User-defined Classification and Multi-level Grouping of Objects in Memory Monitoring. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering (ICPE)*. 115–126. <https://doi.org/10.1145/3184407.3184412>
- [100] Jen-Her Wu and Yufei Yuan. 2003. Improving Searching and Reading Performance: The Effect of Highlighting and Text Color Coding. *Inf. Manag.* 40, 7 (2003), 617–637. [https://doi.org/10.1016/S0378-7206\(02\)00091-5](https://doi.org/10.1016/S0378-7206(02)00091-5)
- [101] Guoqing (Harry) Xu. 2013. Resurrector: a Tunable Object Lifetime Profiling Technique for Optimizing Real-World Programs. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*. 111–130. <https://doi.org/10.1145/2509136.2509512>
- [102] N. Zhang, N. Jiang, Y. Zhang, and G. Huang. 2010. Towards Automated Generation of User-Specific Eclipse Wizard. In *Proceedings of the International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*. 490–497. <https://doi.org/10.1109/CyberC.2010.95>

Received February 2021; accepted April 2021