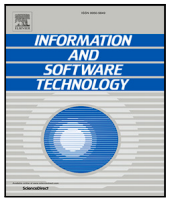
Information and Software Technology 150 (2022) 106989

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



Trace visualization within the Software City metaphor: Controlled experiments on program comprehension

Veronika Dashuber [a,*], Michael Philippsen [b]

[a] QAware GmbH, Aschauer Str. 32, 81549 Munich, Germany
[b] Programming Systems Group, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Martensstr. 3, 91058 Erlangen, Germany

ARTICLE INFO

Keywords:
Trace visualization
Software city
Program comprehension
Aggregation
Heatmap
Root cause analysis

ABSTRACT

Context: Especially with the rise of microservice architectures, software is hard to understand when just the static dependencies are known. The actual call paths and the dynamic behavior of the application are hidden behind network communication. To comprehend what is going on in the software the vast amount of runtime data (traces) needs to be reduced and visualized.

Objective: This work explores more effective visualizations to support program comprehension based on runtime data. The pure DynaCity visualization supports understanding normal behavior, while DynaCity$_{rc}$ supports the comprehension of faulty behavior.

Method: DynaCity uses the city metaphor for visualization. Its novel trace visualization displays dynamic dependencies as arcs atop the city. To reduce the number of traces, DynaCity aggregates all requests between the same two components into one arc whose brightness reflects both the number and the total duration of the requests. DynaCity also encodes dynamic trace data in a heatmap that it uses to light up the building: the brighter a building is, the more active it is, i.e., the more and the longer the requests are that it receives and/or spawns. An additional color scheme reflects any error/status codes among the aggregated traces. In a controlled experiment, we compare our approach with a traditional trace visualization built into the same Software City but showing all dependencies (without aggregation) as individual arcs and also disabling the heatmap. We also report on a second study that evaluates if an error-based coloring of only the arcs is sufficient or if the buildings should also be colored. We call this extension DynaCity$_{rc}$ as it is meant to support *r*oot *c*ause analyses. The source code and the raw data of the quantitative evaluations are available from https://github.com/qaware/dynacity.

Results: We show quantitatively that a group of professional software developers who participated in a controlled experiment solve typical software comprehension tasks more correctly (11.7%) and also saved 5.83% of the total allotted time with the help of DynaCity and that they prefer it over the more traditional dynamic trace visualization. The color scheme based on HTTP error codes in DynaCity$_{rc}$ supports developers when performing root cause analyses, as the median of them stated that the visualization helped them *much* in solving the tasks. The evaluation also shows that subjects using DynaCity$_{rc}$ with colored arcs and buildings find the responsible component 26.2% and the underlying root cause 33.3% more correctly than the group with just colored arcs. They also ranked it 40% more helpful to color both.

Conclusion: The DynaCity visualization helps professional software engineers to understand the dynamic behavior of a software system better and faster. The color encoding of error codes in DynaCity$_{rc}$ also helps them with root cause analyses.

1. Introduction

Modern software is often built according to a microservice architecture so that it can run as a distributed system in the cloud and thus benefits from the elasticity and scalability of this technology. Since a microservice is independent and focuses on a specific use case, its source code becomes smaller, less complex, and easier to maintain and comprehend than a monolith. However, the complexity is still present; by distributing the system, the complexity only shifts to the mostly asynchronous communication between components. This makes the

* Corresponding author.
E-mail addresses: veronika.dashuber@qaware.de (V. Dashuber), michael.philippsen@fau.de (M. Philippsen).

https://doi.org/10.1016/j.infsof.2022.106989
Received 10 December 2021; Received in revised form 9 June 2022; Accepted 21 June 2022
Available online 25 June 2022

traceability of invocations and the understanding of program sequences more difficult.

For understanding how and why a distributed system behaves the way it does, the so-called three pillars of observability matter: logs, traces, and metrics [1]. Logs and metrics are mainly useful for debugging. Traces provide information about the call structure of a distributed application. As static dependencies only reveal possible call paths, dynamic dependencies are conceptually more useful because they show the actual program flow. In practice however, the vast amount of dynamic trace data that a running application generates restricts the possibility to gain insights from dynamic trace data and to exploit them for program comprehension. Hence, much effort has been made to address these issues by reducing [2] or visualizing [3–6] of execution traces.

The city metaphor is widely used to visualize software [7]. Components, such as Java classes, are mapped to buildings. Containers of components, such as Java packages, are mapped to districts. Dependencies between components are usually represented with arcs drawn between buildings. We also use this metaphor in our visualization. But the representation of dependencies as arcs atop the software city does not scale as in general representing every single of the many traces as an arc overloads the visualization. Therefore, our visualization reduces the number of arcs. We aggregate all traces between the same components into one representative. Its brightness reflects the number and the duration of the aggregated traces. The visualization is a lot less cluttered. To also show the activity of each component we use a heatmap to light up the buildings. The brighter a building is, the more active it is, i.e., the more and the longer the requests are that the building receives and/or spawns.

This article extends our work presented at the VISSOFT 2021 conference [8] and also visualizes error/status codes in DynaCity$_{rc}$. With this extension the software engineers cannot only understand the normal behavior of an application, when for example HTTP status codes like 403 are frequent and expected in its operation and do not pose a problem. But they can also visualize faulty behavior and perform root cause analyses.

The remainder of the paper is structured as follows: Section 2 covers related work, specifically addressing trace reduction techniques, trace visualizations, and heatmaps. Section 3 presents (pure) DynaCity, highlighting our aggregation algorithm for displaying only a reduced set of trace representatives as well as projecting the heatmap onto the Software City. Section 4 describes our controlled experiment to gauge the impact of DynaCity's trace visualization on program comprehension. Section 5 discusses the results as well as the threats to validity. Section 6 covers and evaluates the coloring according to error/status codes that we added in DynaCity$_{rc}$. We keep the quantitative evaluation separate from Sections 4 and 5 as the numbers came from a separate controlled experiment with a different set of participants. Finally, Section 7 concludes.

2. Related work

Trace visualizations that only show the active traces at a certain point of time (either live monitoring systems or playback systems) do not need reduction techniques. For example, the playback Software City from Waller et al. [9] for the analysis of concurrency issues visualizes single execution traces that come and go over time. The monitoring system of Fittkau et al. [3] also does not need reductions.

2.1. Trace reduction techniques

Trace visualization systems for time ranges in general need reduction techniques to avoid that many arcs clutter the view. Cornelissen et al. [10] distinguish four categories of trace reduction techniques: (1) summarization techniques apply a grouping criterion and replace traces with fewer class representations. Unselected traces without a representation are not shown. (2) Metrics-based filtering approaches have in common that there are threshold values. If the metrics value of an event is below that threshold, it is hidden and does not appear in the visualization. Example metrics are a simple stack depth limitation [11,12] or the fan-in/fan-out degree of method invocations [13–15]. (3) Language-based filtering reduces traces by removing for instance getters/setters. (4) Ad hoc approaches include for example sampling.

Our approach is a combination of (1) and (2). We aggregate arcs with common endpoints. Representatives are weighted with our metrics that reflect all the call durations and frequencies of all aggregated traces. In contrast to the purely metrics-based approaches there is no threshold, i.e., all traces get visualized, but some rare invocations may end up as arcs that are almost as dark as the background.

2.2. Trace visualizations

Even after reduction techniques have been applied there may still be too many arcs so that additional techniques are needed to clean up the visualization. Frequently used are, among others, graph representations [6,16,17], hierarchical edge bundles [5], circular bundles [18], and massive sequences [19].

Caserta et al. [20] present a Software City with 3D hierarchical edge bundles. They display a line for every trace. To avoid the clutter, they organize the lines so that along common segments the lines run in parallel, i.e., they appear as thick bundles (like cable bundles). Instead of a spaghetti style chaos the visualization is tidy and easier to read. SArF Map [21] also uses edge bundles to visualize dependencies between components. Their clustering algorithm further cleans up the visualization of bundled lines. In contrast, DynaCity does not show all traces but aggregates them into fewer representatives. That also cleans out the chaos. The comparative study in Section 4 compares edge-bundling to aggregation and gauges their effects in software comprehension tasks.

In addition to cleaning up the visualization by organizing the arcs, there are also approaches that encode information from arcs into other features of the Software City. The general idea is to use heatmaps.

In the Software City by Benomar et al. [22] there are no arcs. They only project a heatmap to areas of the city's floor, while we use a more specific mapping to the brightness of the buildings/components. The authors only report on a small case study.

Krause et al. [23] add a heatmap overlay based on trace data to their Software City. This approach is similar to DynaCity, but instead of projecting the heatmap directly onto buildings, it is overlaid as additional information. Moreover arcs are not colored at all and there is no quantitative evaluation in their paper at all.

EvoSpaces [24] is a playback system that therefore does not need reductions. In addition to the usual arcs, it encodes the incoming/outgoing calls in the colors of buildings. This is similar to our heatmap. However, our metrics also reflect the duration of calls. Also, in EvoSpaces a user must explicitly map call frequencies to colors, which is hard to do for an unknown application while trying to comprehend it. Instead of predefined colors, we automatically scale light intensities to the metrics. Furthermore, in contrast to the EvoSpaces literature, we quantitatively examine the impact of our visualization on program comprehension.

There are also many approaches of software cities based on purely static analysis [7,25–27]. We do not discuss them in detail here, since they do not analyze dynamics and are therefore not close enough to our approach and goals.

Distributed tracing systems

Distributed tracing systems like Zipkin,[1] Jaeger,[2] or HTrace[3] are widely used in the industrial context. They also provide visualizations

[1] https://zipkin.io/.
[2] https://www.jaegertracing.io/.
[3] https://htrace.org/.