SynchroVis: 3D Visualization of Monitoring Traces in the City Metaphor for Analyzing Concurrency

Jan Waller, Christian Wulf, Florian Fittkau, Philipp Döhring, and Wilhelm Hasselbring
Software Engineering Group, Department of Computer Science, Kiel University, Kiel, Germany
Email: (jwa, chw, ffi, pdo, wha) @informatik.uni-kiel.de

Abstract—**The increasing code complexity in modern software systems exceeds the capabilities of most software engineers to understand the system's behavior by just looking at its program code. The addition of concurrency issues through the advent of multi-core processors in the consumer market further escalates this complexity.**

A solution to these problems is visualizing a model of the system to ease program comprehension. Especially for the comprehension of concurrency issues, static information is often not sufficient. For this purpose, profiling and monitoring can provide additional information on the actual behavior of a system. An established visualization approach is the 3D city metaphor. It utilizes the familiarity with navigating a city to improve program comprehension.

In this paper, we present our trace-based SynchroVis 3D visualization approach for concurrency. It employs the city metaphor to visualize both static and dynamic properties of software systems with a focus on illustrating the concurrent behavior. To evaluate our approach, we provide an open source implementation of our concepts and present an exemplary dining philosophers scenario showing its feasibility.

I. Introduction

Through the advent of multi-core processors in the consumer market, parallel systems became a commodity. The resulting addition of concurrent behavior in software systems leads to further challenges in program comprehension.

For this reason, specifications and tools have been invented which facilitate the analysis of software systems to be more clearly represented. For instance, the Unified Modeling Language (UML) provides several diagrams to visualize both the static structure of a system (in terms of software entities and their relationships) and the dynamic aspects, namely class instantiations and message interactions in the context of time.

Recent approaches [e. g., 1, 2, 3, 4, 5] expand the 2D visualization of, for example, UML diagrams by another dimension to describe a software system's static and dynamic properties. The city metaphor is such a 3D visualization approach displaying a software system as a large city. The viewers day-to-day familiarity in navigating a city (e. g., reading a street map, orienting with the help of large buildings, etc.) supports the understanding of the visualized application [2].

In addition, there has been work on tools analyzing the runtime behavior of software systems. So-called monitoring tools, such as Kieker [6], are able to collect information on operations, e. g., associated classes, response times, and concurrent behavior, by means of software probes inserted into the target system. However, these execution traces produce enormous quantities of information [4] making them challenging to analyze and to visualize.

In our SynchroVis approach [7] we chose the city metaphor to improve program comprehension for software systems. Our visualization of the system's static structure is based on a source code analysis, while the dynamic behavior is gathered from information collected in monitoring traces. The focus of our approach is on providing a detailed visualization of the system's concurrent behavior. The SynchroVis tool including some examples is available as open source software.[1]

In summary, our main contributions are:

- A 3D city metaphor to simultaneously visualize the static and dynamic properties of software systems
- A visualization of the system's concurrent behavior within the city metaphor
- Additional support for displaying and navigating collected program traces within the city metaphor

The rest of the paper is organized as follows. In Section II, we describe our SynchroVis approach and our contributions. The evaluation scenario is presented in Section III, while related work is discussed in Section IV. Finally, we draw the conclusions and present future work (Section V).

II. The SynchroVis Tool

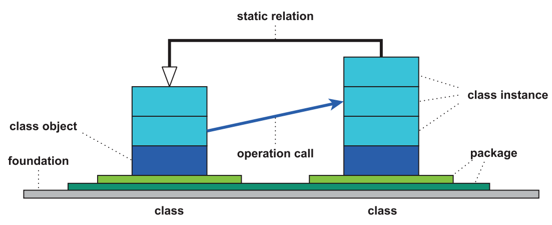
We present our SynchroVis approach according to the nested model for visualization design by Munzner [8].

The *domain* of our approach is program comprehension with a focus on concurrency. Typical tasks include understanding the existence and interaction (e. g., the mutual calling behavior) of classes or components in a software system under study, as well as understanding concurrency (e. g., locking and starvation).

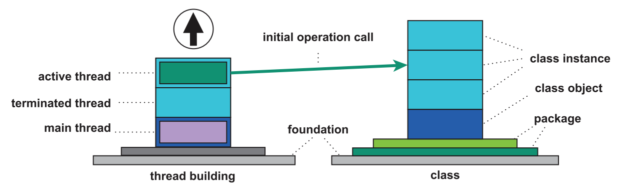
The information used for program comprehension are gathered from a static analysis for the static structure of the software system and from monitoring program traces [6] for the dynamic and concurrent behavior. This information is transformed into internal data formats (e. g., class instances, relationships, and traces) used for our visualization. This *operation and data type abstraction* is described in greater detail by Döhring [7].

The main contribution of our approach is in the *design encoding and interaction technique level*. We employ the city metaphor to visualize the gathered and prepared information. This city metaphor is detailed in the following.

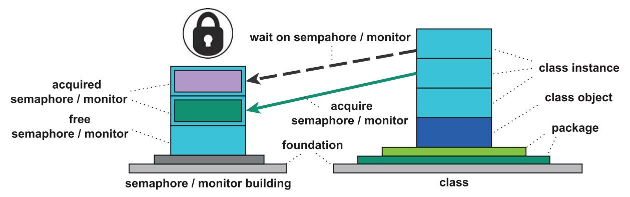
[1]http://kieker-monitoring.net/download/synchrovis/



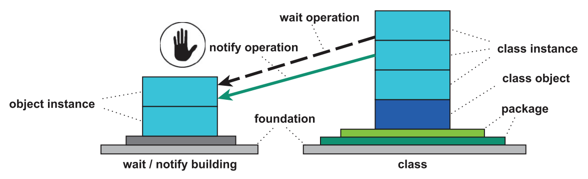
(a) Our city metaphor



(b) Thread building visualizing started threads


wait on sempahore / monitor
class instance
class object
package
acquired semaphore / monitor
free semaphore / monitor
acquire semaphore / monitor
foundation
semaphore / monitor building
class

(c) Synchronization with semaphores / monitors



(d) Synchronization with wait / notify

Figure 1. Schematic view of our SynchroVis approach (based upon [7])

A. Our City Metaphor

A schematic view of our city metaphor used as the basis for our visualization approach is depicted in Figure 1a. We employ the three general concepts of (1) *districts* to break our city into parts, of (2) *buildings* to represent static parts of the software system, and of (3) *streets* to connect our static parts according to dynamic interactions.

Districts: Packages (e. g., in Java) or components (as a more general concept) form the districts of our city metaphor. Each package is visualized as a rectangular layer with a fixed height. Several packages are stacked upon one another to display their subpackage hierarchies with increasing lightness.

Buildings: Each class is represented by a building which is placed in its corresponding district (determined by its package). The ground floor of the building represents the actual class object, while the upper floors represent dynamically created class instances.

Streets: Operation calls contained in the program trace are represented by streets, i. e., colored arrows between floors of the same or different buildings. Each color corresponds to a single thread to simplify the comprehension of different traces. Arrows entering the ground floor represent either calls

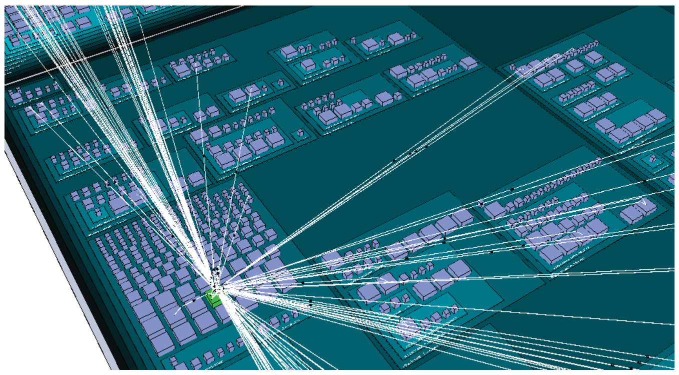


Figure 2. Visualizing the static structure of the large software system Vuze

of static operations or of constructors. A constructor call also adds a new floor to the building. Arrows entering upper floors represent operation calls on the corresponding object instances.

In addition, SynchroVis allows to show or to hide static relations of either a selected class or of all classes in our visualization. All of these relations are displayed as arrows connecting building roofs. Black arrows symbolize inheritance relationships, gray arrows symbolize interface implementations, while white arrows symbolize general associations.

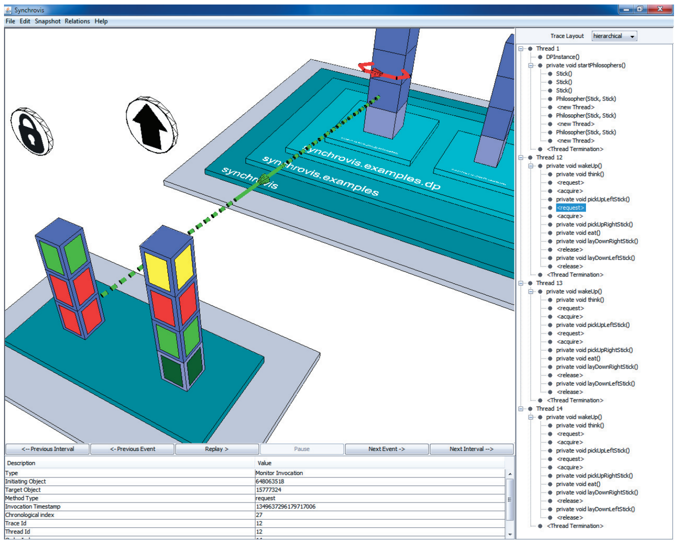
An example of visualizing the static information of a large software system is depicted in Figure 2. In this case, the packages, classes, and relations of the Java-based Vuze Bittorrent Client are displayed.

B. Mechanisms for Visualizing Concurrency

Besides visualizing the interleaving of concurrent threads as described in the previous section, we also provide mechanisms to visualize four specific synchronization concepts.

Threads: All program traces start at a special thread building in an external district. The ground floor of the thread building represents the starting thread of the program execution. Each time a new thread starts, a new floor is added, colored in the respective thread's color. The initial arrow of the new trace starts within the respective floor of this thread building. In case of thread termination, the associated floor looses its color. Refer to Figure 1b for a schematic representation.

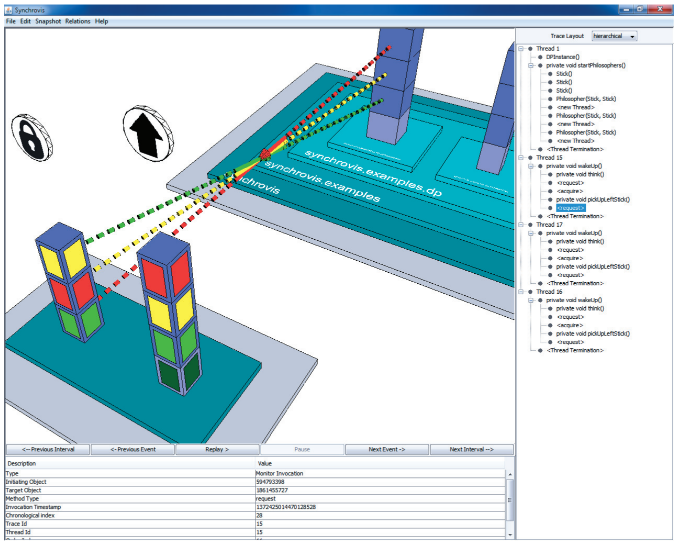
Monitor / Semaphore: A classic synchronization concept is the monitor, e. g., realized by the `synchronized` keyword in Java. It is similar to the concept of binary semaphores. Each semaphore or monitor is visualized by a separate floor on a specific semaphore building next to the thread building. If the monitor gets acquired, the floor gets colored by the respective thread's color, otherwise it is uncolored. Furthermore, any successful acquire or entry operation is depicted with a solid arrow directed between source class instance floor and the semaphore floor. Blocking operations are depicted with similar dashed arrows. Thus, a waiting thread is visible by its directed arrow entering a differently colored semaphore. Similarly, it is possible to spot deadlocks by comparing the different semaphore floors and their respective waiting threads. This semaphore / monitor mechanism is depicted in Figure 1c.



```
Thread 1
    DPInstance()
    private void startPhilosophers()
        Stick()
        Stick()
        Stick()
        Philosopher(Stick, Stick)
        <new Thread>
        Philosopher(Stick, Stick)
        <new Thread>
        Philosopher(Stick, Stick)
        <new Thread>
    <Thread Termination>
Thread 12
    private void wakeUp()
        private void think()
        <request>
        <acquire>
        private void pickUpLeftStick()
        <request>
        <acquire>
        private void pickUpRightStick()
        private void eat()
        private void layDownRightStick()
        <release>
        private void layDownLeftStick()
        <release>
    <Thread Termination>
Thread 13
    private void wakeUp()
        private void think()
        <request>
        <acquire>
        private void pickUpLeftStick()
        <request>
        <acquire>
        private void pickUpRightStick()
        private void eat()
        private void layDownRightStick()
        <release>
        private void layDownLeftStick()
        <release>
    <Thread Termination>
Thread 14
    private void wakeUp()
        private void think()
        <request>
        <acquire>
        private void pickUpLeftStick()
        <request>
        <acquire>
        private void pickUpRightStick()
        private void eat()
        private void layDownRightStick()
        <release>
        private void layDownLeftStick()
        <release>
    <Thread Termination>
```

Description	Value
Type	Monitor Invocation
Initiating Object	648063518
Target Object	15777324
Method Type	request
Invocation Timestamp	1349637296179717006
Chronological index	27
Trace Id	12
Thread Id	12

Figure 3. Visualizing a normal run of the dining philosophers problem



```
Thread 1
    DPInstance()
    private void startPhilosophers()
        Stick()
        Stick()
        Stick()
        Philosopher(Stick, Stick)
        <new Thread>
        Philosopher(Stick, Stick)
        <new Thread>
        Philosopher(Stick, Stick)
        <new Thread>
    <Thread Termination>
Thread 15
    private void wakeUp()
        private void think()
        <request>
        <acquire>
        private void pickUpLeftStick()
        <request>
    <Thread Termination>
Thread 17
    private void wakeUp()
        private void think()
        <request>
        <acquire>
        private void pickUpLeftStick()
        <request>
    <Thread Termination>
Thread 16
    private void wakeUp()
        private void think()
        <request>
        <acquire>
        private void pickUpLeftStick()
        <request>
    <Thread Termination>
```

Description	Value
Type	Monitor Invocation
Initiating Object	594793398
Target Object	1861455727
Method Type	request
Invocation Timestamp	1372425014470128528
Chronological index	28
Trace Id	15
Thread Id	15

Figure 4. Visualizing a deadlock in the dining philosophers problem

Wait / Notify: The next synchronization concept supported by our SynchroVis approach is the wait and notify mechanism. Again, we add a special building to the extra district and assign a floor to each object a thread is waiting on. Each `wait` operation is depicted with a dashed arrow between the source floor and the respective floor of the special building, while each `notify` or `notifyAll` operation is depicted with a solid arrow. This allows for a visualization very similar to the visualization of locking and deadlock behavior of semaphores. This mechanism is illustrated in Figure 1d.

Thread Join: The final synchronization concept realized within our approach is the joining concept of threads, where threads wait upon the completion of another thread. This is visualized by dashed arrows into the respective floor of the thread building that the other threads are waiting upon.

C. Displaying and Navigating Collected Program Traces

The SynchroVis tool provides the usual means for interactively navigating the city (e. g., moving, rotating, and zooming), as well as for searching and locating specific entities within. It is also possible to select an arbitrary scene element to get further information on it.

Furthermore, SynchroVis provides specific support for navigating program traces. The user is able to use both a chronological and a hierarchical display of program traces. The *chronological display* allows to iterate over all events of all threads sorted by time in ascending order, while the *hierarchical display* allows to iterate over the events of a single thread. Additionally, it is possible to use a *time-based stepping* with a configurable interval.

For enhanced usability, SynchroVis provides the option to directly jump to a specific point in time or to a specific event. Moreover, it offers the possibility to automatically step through the trace by means of a *movie mode*. In this mode, the user is able to watch and to study the recorded behavior of the application.

III. Evaluation

Due to space concerns, we limit ourselves to a single evaluation scenario in this paper. Refer to Döhring [7] for an discussion of the benefits of our approach and for an anecdotal expert interview conducted with several internal developers concerning questions of comprehensibility and usability.

As an example scenario, we describe a typical concurrency analysis of the well known dining philosophers problem with SynchroVis. This scenario is often used to study locking behavior, especially deadlocks. For the sake of simplicity, we assume three present philosophers, realized by separate threads, and three shared forks, realized by monitors.

A normal run of the dining philosophers problem is presented in Figure 3. Here, the philosopher represented by the red thread acquired his two forks (monitors) and is executing his eating time. The green philosopher acquired only a single fork and is waiting for the release of his second fork, currently held by the red thread. This is visualized by the green arrow pointing at the red floor of the semaphore / monitor building.

A run resulting in a deadlock is presented in Figure 4. In this case, all three philosophers managed to acquire a single fork. Thus, all three threads are waiting upon each other to release the respective monitors. This is visualized by the colored arrows pointing at the differently colored floors of the semaphore / monitor building. The navigation within the program traces allows for an analysis of the cause of this deadlock.

Both runs of this evaluation scenario are included in our distribution of SynchroVis.

IV. Related Work

For reasons of limited space, we only discuss closely related approaches to our visualization, i. e., visualizations based on the city metaphor and visualizations of concurrency. A general overview on several 3D visualizations of software systems is provided, for example, by Teyseyre and Campo [9].

A. 3D Visualization with the City Metaphor

Software World by Knight and Munro [1] was one of the first approaches employing the 3D city metaphor to visualize software systems. A similar more recent approach is provided by Wettel and Lanza [2]. Contrary to SynchroViz, both approaches only visualize the static structure of a system.

Panas et al. [3] present a realistic 3D city visualization focusing on the static structure and the communication based on program traces. In contrast to our approach, the authors use a more realistic visualization (e. g., cars to represent communication) but provide no direct support for concurrency.

EvoSpace [4] utilizes the 3D city metaphor with a day view to display a system's static structure and a night view to display its dynamic behavior. Contrary, our approach combines both views into one and also offers support for concurrency.

Caserta et al. [5] visualize large software systems with their static and dynamic properties using hierarchical edge bundles to aggregate dynamic relations. In contrast, our approach is focused on visualizing single execution traces and concurrency.

B. Visualizing Concurrency

Much research has been conducted for visualizing concurrency utilizing UML diagrams. Mehner and Weymann [10] enhance UML communication diagrams to visualize Java monitors. Artho et al. [11] and Malnati et al. [12] provide extensions of UML sequence diagrams to explicitly support concurrency. Leroux et al. [13] utilize state charts in combination with an extension to UML sequence diagrams to visualize the concurrent behavior of a software system. SynchroVis builds upon several of these extensions [12, 13] and adapts them to our city metaphor. Furthermore, we combine the visualization of static and dynamic properties and provide better scalability by utilizing the third dimension.

Further approaches [e. g., 14, 15] focus on supporting program comprehension of concurrent applications without using the UML. In contrast to these two approaches, our approach is able to visualize both static and dynamic properties (including concurrency) of the software system in a single view.

To the best of our knowledge, SynchroVis is the first tool to visualize concurrency with the help of the 3D city metaphor.

V. Conclusions & Outlook

3D visualization of monitoring traces with the help of the city metaphor provides an effective means to ease program comprehension of concurrency. Our SynchroVis approach and its corresponding tool enables the visualization of static and dynamic properties of a software system. Furthermore, it supports software engineers in debugging and understanding the concurrent behavior of their applications.

As future work for our SynchroVis tool, we will implement a space-optimizing layout algorithm, also minimizing edge lengths and crossings. Furthermore, our city metaphor can be improved by diversifying the look of our buildings, e. g., by using cylinders instead of cubes for the special buildings. Finally, we will focus on providing greater scalability of our visualizations for larger software systems.

References

[1] C. Knight and M. Munro, "Virtual but visible software," in *Proc. of the Int. Conf. on Information Visualisation*. IEEE Computer Society, 2000, pp. 198–205.

[2] R. Wettel and M. Lanza, "Visualizing software systems as cities," in *Proc. of the 4th Int. Workshop on Visualizing Software For Understanding and Analysis*. IEEE Computer Society, 2007, pp. 92–99.

[3] T. Panas, R. Berrigan, and J. Grundy, "A 3D metaphor for software production visualization," in *Proc. of the 7th Int. Conf. on Inf. Visualization*, 2003, pp. 314–320.

[4] P. Dugerdil and S. Alam, "Execution trace visualization in a 3D space," in *Proc. of the 5th Int. Conf. on Information Technology: New Gen.*, 2008, pp. 38–43.

[5] P. Caserta, O. Zendra, and D. Bodenes, "3D hierarchical edge bundles to visualize relations in a software city metaphor," in *Proc. of the 6th Int. Workshop on Visualizing Software for Understanding and Analysis*. IEEE Computer Society, 2011, pp. 1–8.

[6] A. van Hoorn, J. Waller, and W. Hasselbring, "Kieker: A framework for application performance monitoring and dynamic software analysis," in *Proc. of the 3rd Int. Conf. on Perf. Eng.* ACM, 2012, pp. 247–248.

[7] P. Döhring, "Visualisierung von Synchronisationspunkten in Kombination mit der Statik und Dynamik eines Softwaresystems," MSc thesis, Kiel University, 2012.

[8] T. Munzner, "A nested model for visualization design and validation," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 6, pp. 921–928, 2009.

[9] A. R. Teyseyre and M. R. Campo, "An overview of 3D software visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, pp. 87–105, 2009.

[10] K. Mehner and B. Weymann, "Visualization and debugging of concurrent Java programs with UML," in *Proc. of the Workshop on Soft. Vis.*, 2001, pp. 59–64.

[11] C. Artho, K. Havelund, and S. Honiden, "Visualization of concurrent program executions," in *Proc. of the 31st Annual Int. Comp. Software and Applications Conf.* IEEE Computer Society, 2007, pp. 541–546.

[12] G. Malnati, C. M. Cuva, and C. Barberis, "JThreadSpy: Teaching multithreading programming by analyzing execution traces," in *Proc. of the ACM Workshop on Par. and Distr. Systems: Testing and Debugging*, 2007, pp. 3–13.

[13] H. Leroux, C. Mingins, and A. Réquilé-Romanczuk, "JACOT: A UML-based tool for the run-time inspection of concurrent Java programs," in *Proc. of 1st Workshop on Adv. the State-of-the-Art in Run-Time Insp.*, 2003.

[14] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. M. Vlissides, and J. Yang, "Visualizing the execution of Java programs," in *Int. Sem. Revised Lectures on Software Visualization*. Springer, 2002, pp. 151–162.

[15] J. Trümper, J. Bohnet, and J. Döllner, "Understanding complex multithreaded software systems by using trace visualization," in *Proc. of the 5th Int. Symp. on Software Visualization*. ACM, 2010, pp. 133–142.