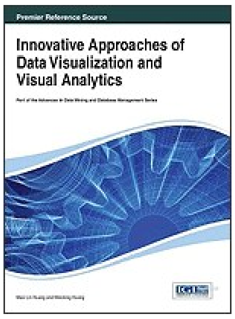


Innovative Approaches of Data Visualization and Visual Analytics

by Mao Lin Huang and Weidong Huang (eds)
IGI Global. (c) 2014. Copying Prohibited.

Chapter 18: A Programmer-Centric and Task-Optimized Object Graph Visualizer for Debuggers

Anthony Savidis,
Institute of Computer Science-FORTH & Department of Computer Science, University of Crete
Greece

Nikos Koutsopoulos,
Institute of Computer Science-FORTH
Greece

ABSTRACT

Today, existing graph visualizers are not popular for debugging purposes because they are mostly visualization-oriented, rather than task-oriented, implementing general-purpose graph drawing algorithms. The latter explains why prominent integrated development environments still adopt traditional tree views. The authors introduce a debugging assistant with a visualization technique designed to better fit the actual task of defect detection in runtime object networks, while supporting advanced inspection and configuration features. Its design has been centered on the study of the actual programmer needs in the context of the debugging task, emphasizing: 1.) visualization style inspired by a social networking metaphor enabling easily identify who deploys objects (clients) and whom objects deploy (servers); 2.) inspection features to easily review object contents and associations and to search content patterns (currently regular expressions only); and 3.) interactively configurable levels of information detail, supporting off-line inspection and multiple concurrent views.

INTRODUCTION

Debugging is the systematic process of detecting and fixing bugs within computer programs and can be summarized (Zeller, 2005) by two main steps, bug detection and bug fixing, as outlined under Figurer 1. The examination of the runtime state requires inspection of object contents, associations and dependencies.

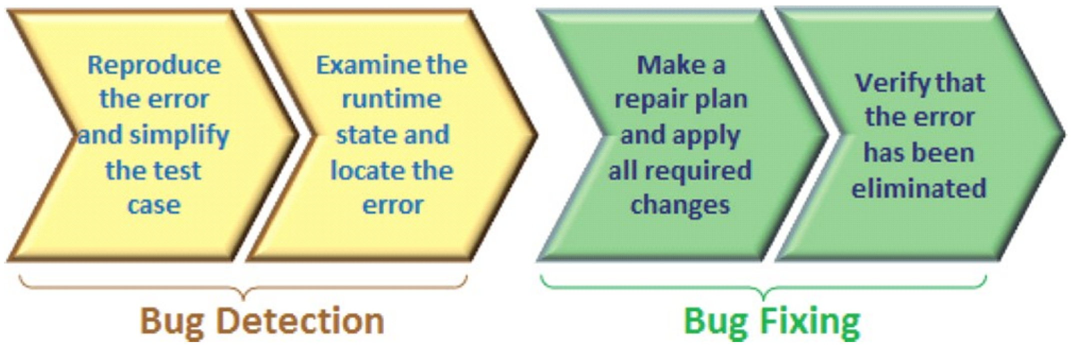

Reproduce the error and simplify the test case
Examine the runtime state and locate the error
Make a repair plan and apply all required changes
Verify that the error has been eliminated
Bug Detection
Bug Fixing

Figure 1: The overall workflow of the debugging process

The latter is a difficult task, even for small-scale applications, where traditional graph visualizations proved to be rather ineffective. The remark is justified by the fact that most popular commercial IDEs like Visual Studio (Microsoft) and IntelliJ IDEA (Jet Brains) do not provide them, while open source IDEs like Eclipse and NetBeans have a couple of relevant third-party plug-ins which are rarely used. But still, during debugging, objects remain the primary information unit for gaining insights (Yi et al., 2008) on how errors infect the runtime state. Interestingly, while the notion of visualization generally receives positive attention, relevant implementations failed to essentially improve the state examination process.

We argue that this is due to the primary focus of present tools on visualization, adopting general-purpose rendering algorithms being the outcome of graph drawing research, however, lacking other sophisticated interactive features.

More specifically, general graph drawing algorithms aim to better support supervision and pattern matching tasks through the display of alternative clustering layouts. This can be valuable when visualizing static features like class hierarchies, function dependencies and recursive data structures. However, during the debugging process the emphasis is shifted towards runtime state analysis involving primarily state exploration and comparison tasks, requiring detailed and advanced inspection facilities. We argue that the inability to systematically address this key issue explains the failure of object graph visualizers in the context of general-purpose debugging. As we discuss later, most existing tools are no more than mere implementations of graph drawing algorithms. Our primary goal is to provide interactive facilities which improve the runtime state examination process for defect detection. This goal is further elaborated into three primary design requirements:

- Visualization style inspired by a social networking metaphor enabling easily identify who deploys objects (clients) and whom objects deploy (servers)
- Inspection features to easily review object contents and associations and to search content patterns (currently regular expressions only)

- Interactively configurable levels of information detail, supporting off-line inspection and multiple concurrent views

The reported work (i-views) has been implemented as a debugger plug-in on top of the Sparrow IDE for the Delta programming language being publicly available (Savidis, 2010). This language has been chosen for ease of implementation, since it offers an XML-based protocol for extracting object contents during runtime (Savidis & Lilis, 2009). Overall, our results may be applied to any other debugging tool.

BACKGROUND

Graphs have been widely deployed in the context of object-oriented visualizations (Lange & Nakamura, 1997) for various purposes, besides debugging, such as: tracing of dynamic object-allocation characteristics, revealing static ownership-based object relationships (Hill et al., 2002), tracking event generation in event-based systems (Kranzlmuller et al., 1996), and investigating object constraints (Muller, 2000). General purpose visualization toolkits exists too, such as Prefuse (Heer et al., 2005) for visualization of large-scale data offering predefined layout algorithms, and the work of (Hendrix et al., 2004) for visualizing data structures in a lightweight IDE. Another state visualization tool is HDPV (Sundararaman & Back, 2008) whose purpose is to provide a global image of the stack, heap and the static data segment in a way reflecting the precise set of object instances during runtime. It offers an option to animate the changes in the memory state in response to instruction execution, assuming users will perceive potential state anomalies by observing state snapshots and eventually lead to defect detection. The emphasis of such previous tools is primarily put in enabling visualization while letting users navigate and explore in a free manner, rather than providing interactive layouts optimized for the particular task at hand being debugging.

Tools offering interactive object graphs exclusively for the debugging process exist for various languages. Memory visualization tools such as those by Aftandilian et al. (2010), De Pauw and Sevitsky (1999), and Reiss (2009) are essentially memory graph visualizers targeted in displaying memory usage by programs, enabling programmers to detect memory leaks or identify memory corruption patterns. They are low-level since their focus is on memory maps and memory analysis, while fundamental program notions like objects and associations (clients and servers) are not handled. A tool working on objects is DDD (GNU, 2009), however, it allows incremental expansion only for referent (outgoing / server) structures.

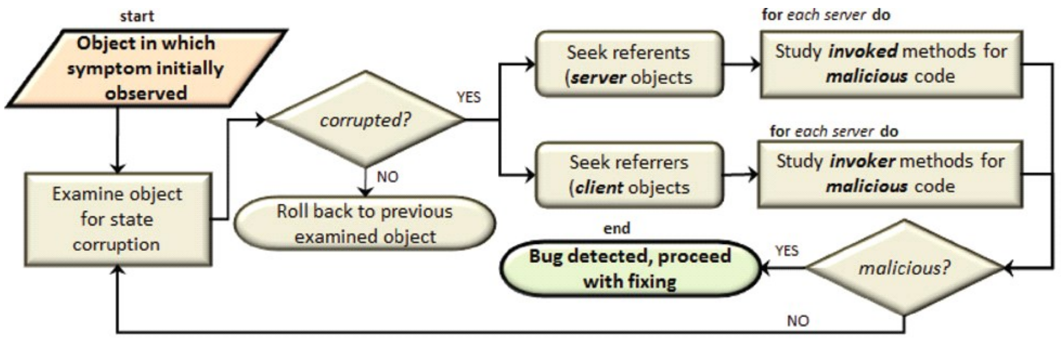
Comparing with previous work, it is clear that all existing tools focus merely on the graph drawing quality rather than on optimizing the primary task being object state inspection for defect detection. As a result, while they tend overall offer attractive visualizations, those are not perfectly aligned with the typical activity patterns programmers actually perform during debugging. As an example, none of the existing tools allows to identify directly the clients and servers of an object, information being very critical when examining error propagation patterns.

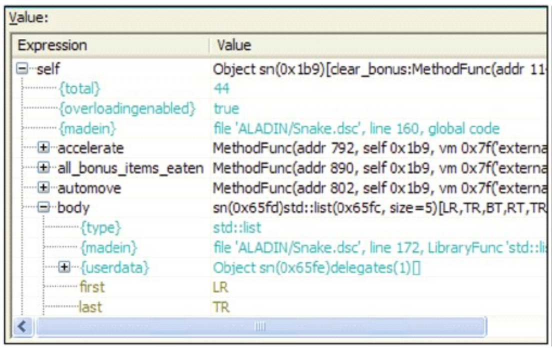
In our approach, the previous has been treated as a primary requirement and played a key role in designing a layer-based graph visualization style. Additionally, we treated inspection as a genuine data mining process by providing users with facilities not met in existing tools, including: bookmarking, text search, lens tools, concurrent views, off-line inspection, dropping interactively examined objects, and configurable visualization parameters in real-time.

DEBUGGING TASK DETAILS

Our graph visualization approach does not adopt a generic graph drawing algorithm (Battista et al., 1999), like those provided by GraphViz (Gansner & North, 1999), but is explicitly designed for defect detection in object networks. We elaborate on the design details of our method showing that it is primarily task-oriented, rather than supervision-oriented or comprehension-oriented as with most graph visualizers.

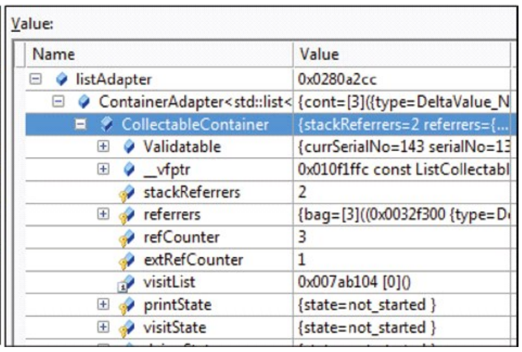
More specifically, the bug detection process is initiated from a starting object on which the undesirable symptoms are initially observed. Then, an iterative process is applied with essentially two categories of analysis activities to detect state corruption and identify malicious code (see Figure 2, to part). In this context, programmers will usually have to further inspect objects by seeking referent objects (outgoings links) from a starting object. Such referent inspection is well supported by existing visualizers like DDD (GNU, 2010) or HeapViz (Aftandilian et al., 2010). However, it is also supported by common tree views (like those of Figure 2, bottom part), which, due to their ease of use, remain at present the most preferable inspection tool.


start
Object in which symptom initially observed
Examine object for state corruption
corrupted?
YES
NO
Roll back to previous examined object
Seek referents (server objects
for each server do
Study invoked methods for malicious code
Seek referrers (client objects
for each server do
Study invoker methods for malicious code
end
Bug detected, proceed with fixing
YES
malicious?
NO



Value:

Expression	Value
⊟ self	Object sn(0x1b9)[clear_bonus:MethodFunc(addr 11
{total}	44
{overloadingenabled}	true
{madein}	file 'ALADIN/Snake.dsc', line 160, global code
⊞ accelerate	MethodFunc(addr 792, self 0x1b9, vm 0x7f('externa
⊞ all_bonus_items_eaten	MethodFunc(addr 890, self 0x1b9, vm 0x7f('externa
⊞ automove	MethodFunc(addr 802, self 0x1b9, vm 0x7f('externa
⊟ body	sn(0x65fd)std::list(0x65fc, size=5)[LR,TR,BT,RT,TR
{type}	std::list
{madein}	file 'ALADIN/Snake.dsc', line 172, LibraryFunc 'std::li
⊞ {userdata}	Object sn(0x65fe)delegates(1)[]
first	LR
last	TR



Value:

Name	Value
⊟ listAdapter	0x0280a2cc
⊟ ContainerAdapter<std::list<	{cont=[3]({type=DeltaValue_N
⊟ CollectableContainer	{stackReferrers=2 referrers={...
⊞ Validatable	{currSerialNo=143 serialNo=13
⊞ __vfptr	0x010f1ffc const ListCollectabl
stackReferrers	2
⊞ referrers	{bag=[3]((0x0032f300 {type=D
refCounter	3
extRefCounter	1
visitList	0x007ab104 [0]()
⊞ printState	{state=not_started }
⊞ visitState	{state=not_started }

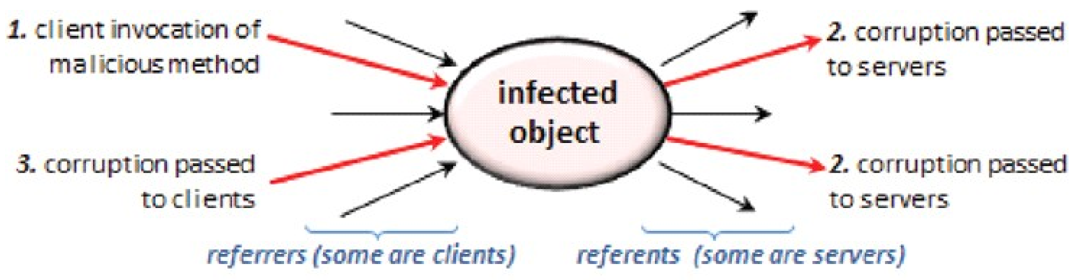
Figure 2: (Top) flowchart of the bug detection task and (bottom) typical debugger tree views

Once the type and level of state corruption is verified on an object, the analysis proceeds so as to identify the offensive code. For the latter, all use sites of infected objects must be investigated since they constitute potentially malicious code. For this step referrer objects (incoming links) should be manually traced. Since many infected objects may coexist, it is crucial to *allow programmers quickly switch back and forth to referent/referrer inspection for different objects*.

This step is the most critical and most demanding part of debugging known as the bug finding process. Only when the offensive statement is eventually found the defect detection step completes and the bug fixing process is initiated. Based on the previous remarks, we follow with the design details of our object graph visualizer.

PROGRAMMER CENTRIC DESIGN

To better support defect detection in object networks, visualizers should support referrer and referent inspection since they directly affect the generation and propagation of defects. In particular (see Figure 3, top part), objects are infected if a method with an offensive statement is invoked by clients (step 1).


1. client invocation of malicious method
3. corruption passed to clients
infected object
2. corruption passed to servers
2. corruption passed to servers
referrers (some are clients)
referents (some are servers)

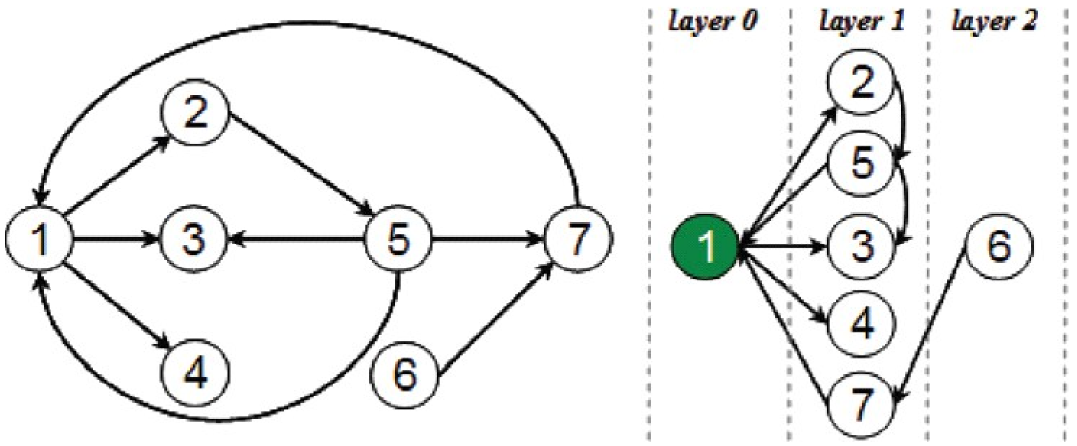


Figure 3: (Top) defect generation and propagation in object networks and (bottom) a typical digraph and a layered graph for node (1)

Such an infection is propagated when servers (referents) are used by (step 2), or clients (referrers) are using an infected object (step 3). Based on the characteristics of the debugging process we focused on a visualization method allowing improved inspection practices for detecting object anomalies. We observed that given an object, all its direct referrers and referents are essentially its close runtime peers, semantically playing as either clients or servers. If such an inner social circle for objects is directly traceable it becomes easier to examine the actual level of infection.

Following this concept, given a starting object A and a maximum social distance N we introduce layers of social peers by the rules: (i) all direct referrers and referents of A are put in *layer 1*; (ii) *layer i+1* contains the direct referrers and referents of objects from *layer i* not included in *layer i* or *layer i-1*; and (iii) when *layer i+1* becomes empty the process terminates.

An example is provided under Figure 3 (bottom part) showing how from a typical directed graph we get a layered graph, for a given starting node. In a layered graph the runtime peers of an object fall either in its own layer or a neighbor one. Every layer encompasses the client and server objects of its previous and next layers. By tracing layers, clients of clients or servers of servers are easily tracked down. Typically, from a starting node, the number of subsequent layers to visit during inspection is usually small, while the number of objects within layers may get very large. It should be noted that the layered view is not designed to offer a generally more attractive, or easier to assimilate, image of the graph. It is a task-specific visualization technique which for a mission other than debugging may be proved to be suboptimal.

INTERACTIVE OBJECT GRAPH VISUALIZER

The detailed software architecture of the interactive object graph visualizer is provided under Figure 4 and a general snapshot of the system used in real practice is shown within Figure 5 (objects are shown with full contents as tables with slot–value pairs). Although the default view is crowded with crossing / overlapping edges, we will briefly discuss how this is handled via the large repertoire of interactive configuration and inspection features allowing programmers to inspect object contents and identify potential state faults. We discuss the key features below.

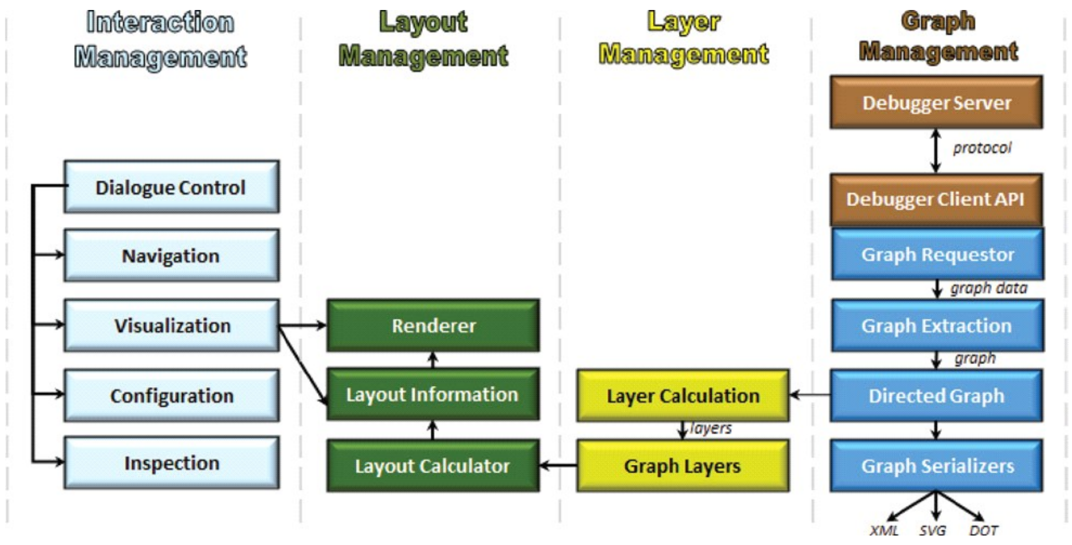


Figure 4: Software architecture of the visualizer

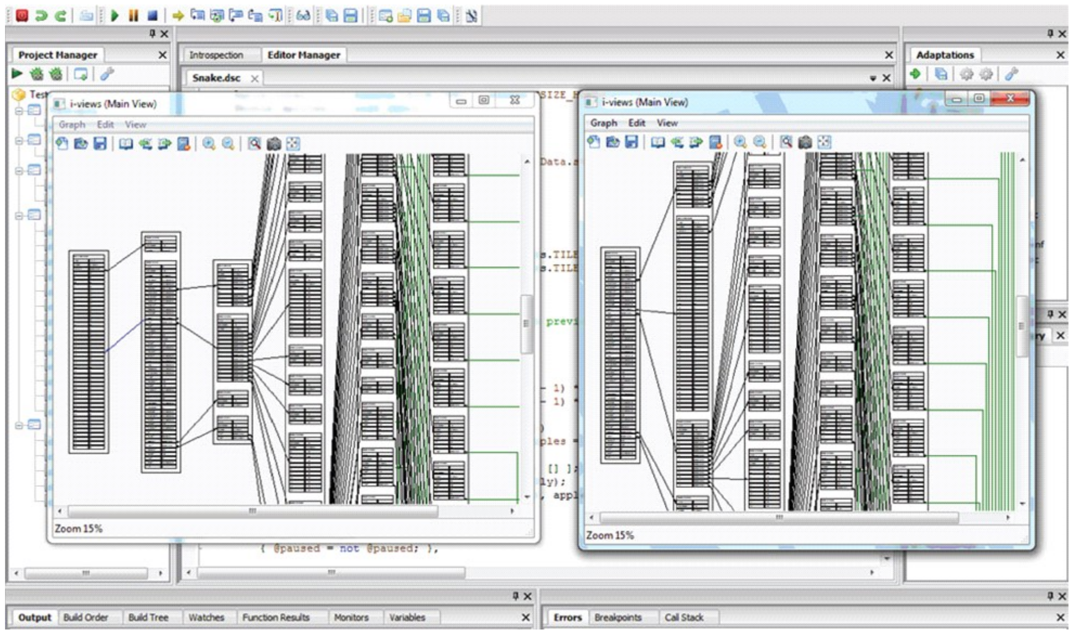
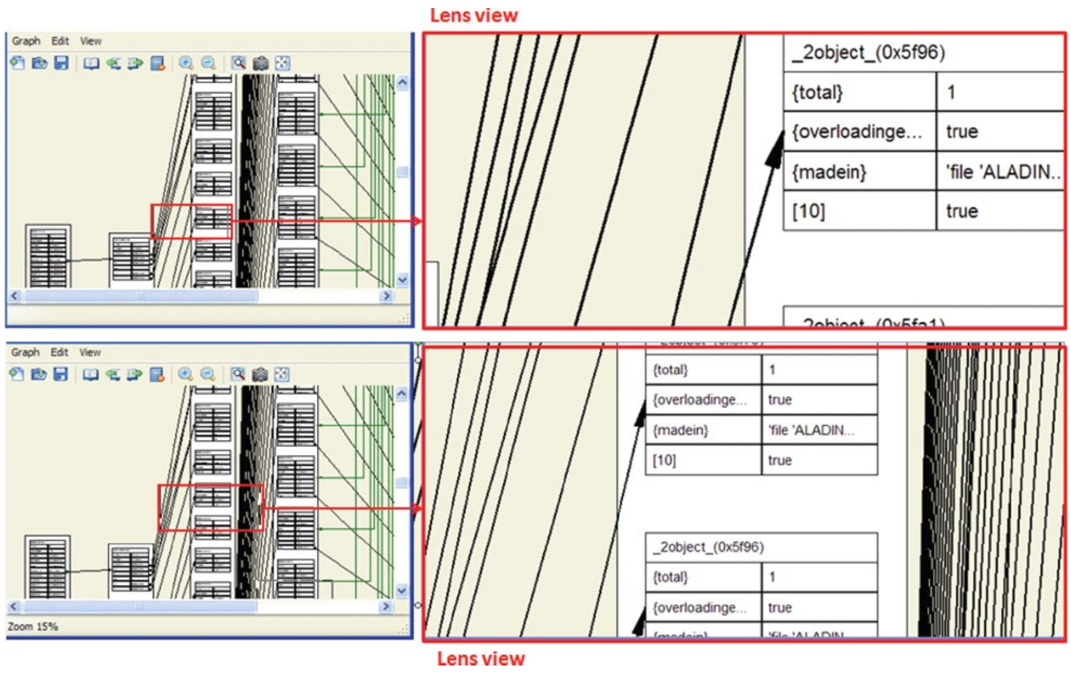


Figure 5: Two parallel inspection sessions with different configurations

Adjustable Lens View

The main view offers zooming and resizing to enable inspect the object graph from different view scales and with varying view sizes. Similarly, the lens view scale and size (see Figure 6) can be separately adjusted. The latter, combined with the main view, offers two independent levels of detail for the inspection process. One of the most typical uses of the lens view during the inspection phase relates to the snapshot of Figure 6: 1.) the main view is chosen with a high zoom-out factor enabling to supervise the largest part of the graph although with a low-level of information detail; 2.) the lens view is configured to combine a large view size (window) with a high zoom-in factor to offer an increased level of detail; and 3.) the lens is dragged across the main view to inspect object contents and their respective associations.



2object(0x5f96)	
{total}	1
{overloadinge...	true
{madein}	'file 'ALADIN..
[10]	true

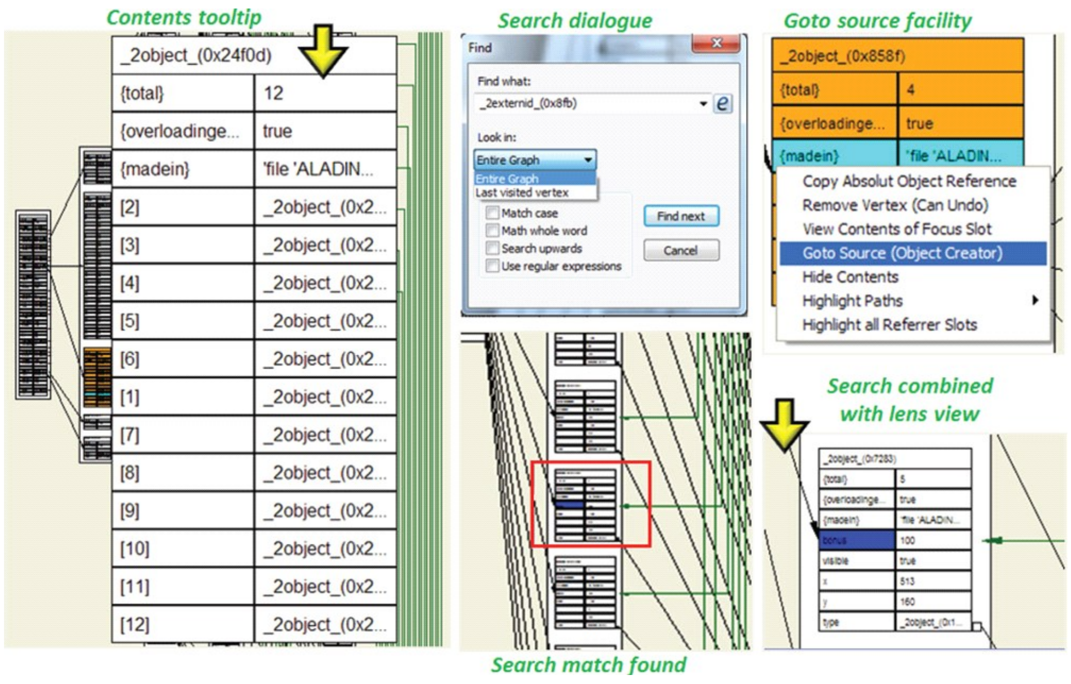
{total}	1
{overloadinge...	true
{madein}	'file 'ALADIN...
[10]	true

2object(0x5f96)	
{total}	1
{overloadinge...	true

Figure 6: The lens view in various (adjustable) view scales

Content Tooltips, Text Search, and Goto Source

As we discuss later, the visualization can be configured in various ways, such as displaying full object contents (all slots shown) or only the object reference identifier. The latter, while it makes the resulting graphs visually less crowded, it also reduces the conveyed information content. For this purpose, content tooltips allow to quickly view object contents (see Figure 7). Also, under a zoom-out factor (value is configurable, default is 30%), the tooltips will remain active even if full contents are shown. Another feature is text search which, amongst others, enables to immediately spot objects with content-corruption patterns (supporting regular expressions too). As shown in Figure 7 (right part), with every match, the tool focuses on the target object and highlights the matching slots; the search can be combined with lens views. Finally, the debugger cores of some languages, like the Delta language, are capable to record in every object instance the source point in which it has been originally constructed during execution. If such information is available, the graph viewer offers a *goto source line* option opening in the editor the respective source file and also positioning at precise source line (see Figure 7, up right).


Contents tooltip
Search dialogue
Goto source facility
Search combined with lens view
Search match found
Find
Find next
Cancel
Copy Absolut Object Reference
Remove Vertex (Can Undo)
View Contents of Focus Slot
Goto Source (Object Creator)
Hide Contents
Highlight Paths
Highlight all Referrer Slots

2object(0x24f0d)	
{total}	12
{overloadinge...	true
{madein}	'file 'ALADIN...
[2]	_2object_(0x2...
[3]	_2object_(0x2...
[4]	_2object_(0x2...
[5]	_2object_(0x2...
[6]	_2object_(0x2...
[1]	_2object_(0x2...
[7]	_2object_(0x2...
[8]	_2object_(0x2...
[9]	_2object_(0x2...
[10]	_2object_(0x2...
[11]	_2object_(0x2...
[12]	_2object_(0x2...

Find what: _2externid_(0x8fb)
Look in: Entire Graph
Entire Graph
Last visited vertex
Match case
Math whole word
Search upwards
Use regular expressions

2object(0x858f)	
{total}	4
{overloadinge...	true
{madein}	'file 'ALADIN...

2object(0x7283)	
{total}	5
{overloadinge...	true
{madein}	'file 'ALADIN...
bonus	100
visible	true
x	513
y	160
type	_2object_(0x1...

Figure 7: Content tooltips, text search combined with lens view and go to source

Bookmarking and Path Highlighting

During the inspection process, content analysis and comparison between different objects is frequently performed. The goal is to identify cases where object contents disobey the patterns expected in a correct program execution. For this task, in existing debugging tools, programmers have to apply tedious heuristics.

For example, they manually copy contents in an editor, trace another target object in the debugger with which they need to compare, and then switch back and forth between the editor and the debugger inspection tool to identify differences or commonalities. To facilitate such activities, we introduced bookmarks in the graph viewer which record the focus object, view origin, and zooming factor. As a result, when switching context to a bookmark, the view state is restored exactly as it was at the time the bookmark was set. Bookmarked objects are indicated with an extra marker. Another useful facility for debugging is highlighting all client and server reference paths recursively (i.e. clients of clients and servers of servers) for a given object (see Figure 8, upper part). This provides a clear picture of the runtime interactions of an object at a given time. Additionally, the object slots involved in creating a reference path are highlighted as well.

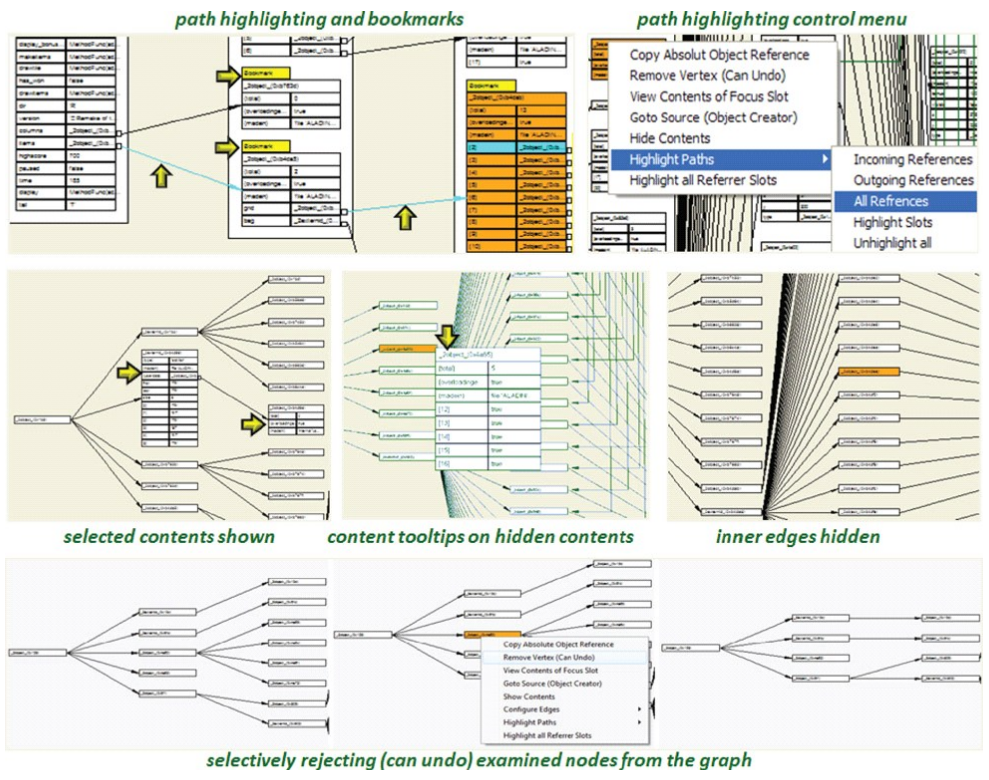

path highlighting and bookmarks
path highlighting control menu
Copy Absolut Object Reference
Remove Vertex (Can Undo)
View Contents of Focus Slot
Goto Source (Object Creator)
Hide Contents
Highlight Paths
Highlight all Referrer Slots
Incoming References
Outgoing References
All Refrences
Highlight Slots
Unhighlight all
selected contents shown
content tooltips on hidden contents
inner edges hidden
Copy Absolut Object Reference
Remove Vertex (Can Undo)
View Contents of Focus Slot
Goto Source (Object Creator)
Show Contents
Configure Edges
Highlight Paths
Highlight all Referrer Slots
selectively rejecting (can undo) examined nodes from the graph

Figure 8: Various interaction configuration features in the object graph

The interactive configuration features of the graph viewer have been designed to enable switching between different levels of information detail. A few examples are provided under Figure 8 (middle and bottom parts) showing the outcome by applying configuration features. They mostly concern the way either objects (vertices) or their associations (edges) are drawn. When combined with other inspection features typical process patterns during debugging are supported. For example, consider path highlighting combined with the ability to hide contents of all objects and the option for selective content expansion on individual objects. This combination allows programmers focus on a specific client-server dependency path, commonly studied when likely corruption patterns are detected. This way, programmers avoid the information overload caused when objects irrelevant to the current investigation context are fully expanded.

Additionally, it is allowed to drop (with undo support) from the graph any object that is considered of no interest during inspection. This allows programmers incrementally simplify the examined graph, thus eventually keeping only the objects assumed as candidates for state infection.

Concurrent Inspection Sessions

In existing graph / tree view tools, inspection is initiated from a starting object whose content is taken using a language library for communicating with the debugger core. Further requests to the debugger core are made when the user requests graph / tree expansion. Most languages support a single communication session with the core, thus the user-interface is made modal. The latter disables concurrent inspection sessions from different starting objects, severely restricting the overall inspection process.

Because of the specific structure of layered graphs, semantically related objects of a starting object fall within a usually small number of proximate layers. This property allowed us to overcome the incremental communication bottleneck and support concurrent non-modal inspection sessions. More specifically, the user is initially requested to provide a maximum depth (semantic distance) from a starting node. Then, an intensive communication round with the debugger core takes place for extracting all required objects and their contents. Once completed, the session is terminated and the tool enables the inspection facilities. To gain such information via existing tree or graph views, a large number of incremental expansion steps are needed, involving synchronous communication with the debugger backend. Additionally, because no debugger core is actually deployed during inspection, we allow the graphs to be saved (in XML format) enabling to reopen at any point in time for off-line inspection.

EVALUATION RESULTS

Due to the nature of the domain, the conduct of a systematic evaluation process with users was inevitable, while we dropped the deployment of subjective evaluation methods. More specifically, because subjective evaluation prescribes expert-based analysis we decided that concrete results with programmers involved in actual debugging tasks would provide more valuable and reliable data. In this context, we initially performed early evaluation trials with prototypes; however, the results were rather inconclusive and partial. In particular, we observed that even small discounts on functionality, something mandatory in order to be able carry out such early sessions, tended to seriously affect the way programmers organize debugging activities. As a result, we have decided to firstly fully implement features, and then *test them in combination* with all the features becoming available to programmers. Thus, following user suggestions as well, we treated all *debugging features as a unit*, shifting emphasis to the conduct of an iterative application of *integration, testing and evaluation* processes with real users. The latter was carried out following a scenario-based conduct:

- We introduced various defects within two applications, all causing malfunctions due to propagated object infections, and then we required users to locate them. To enable the extraction of comparative results we had to produce similar errors, however involving different packages, classes and methods, and then challenged programmers to located offensive statements by either using or not our visualization tool. Also, the bugs were classified in three categories, namely *simple*, *average* and *tough*, implying an increasing level of difficulty for overall bug detection.
- The process was organized with two parallel sessions and two respective groups, with only one group at a running session actually deploying the visualization instrument. With every session we also switched the group that was assigned the task to use the tool to avoid conclusions biased to a group. After every three sessions the groups were reorganized, with about fifteen sessions overall executed.
- The users involved in the sessions were programmers (graduate students) with considerable programming experience and knowledgeable in using typical debugging visualization facilities in popular IDEs like Eclipse, Net Beans and Visual Studio. In total, six students were involved, with two groups of three students per session.

Our conclusions were very interesting regarding the utility of the tool. In particular, they were heavily related to the difficulty level of the bugs, and in some cases indicated a bias due to the required learning curve of the tool. More specifically, for simple bugs most programmers expressed that they found the adoption of such a comprehensive visual instrument to be rather tedious and unnecessary, as they manage to detect infected objects quickly with traditional watches and tree views. One of the ten programmers involved in the study mentioned that the inherent learning effort to use the tool, for detecting simple anomalies, is overall "annoying" in terms of the required investment. Now, when it came to analyzing the results for the bugs of an average complexity scale the conclusions were surprising. Although we did expect some changes, we thought that we could likely get positive feedback only with the tough bugs.

What we observed, and what apparently most of the programmers did in the study, was a very quick shift from the tree views to the layered graphs, even after five to eight repeated inspection trials once failing to capture the originally infected objects. While they expressed that the exploded views with global referent and referrer associations seemed a little crowded in the opening visualization screen, they begun using lenses and configuration features directly and quickly managed to reach infected objects. We asked if they were to abandon the tree views so quickly anyway, irrespective of the presence of the object graphs. All of them responded negatively in an emphatic way. They mentioned that the reason they switched this time so quickly is because they were very eager to find the bug, and they considered that the graph tool with its visual associations would help them to do so. The comparative results for the tough bugs were not very different from the average ones, although it took more time for programmers to locate them (two of them actually failed to identify the bug at all). Finally, in a post analysis session involving the programmers, they mentioned that from their experience, the vast majority of the bugs they are faced in a software project are simple and pretty straightforward to detect. In this case, they would never switch to graph views. However, once bugs get more difficult, it is the psychology of the task that makes them more open to the adoption of tools enabling inspect objects in a more elaborate manner, with the hope that anomalies will be more quickly spotted.

SUMMARY AND CONCLUSIONS

We have discussed a debugging assistant offering interactive object graphs, putting emphasis on improved visualization, inspection (navigation) and configuration facilities. The design of the tool was focused on the optimal support of the primary task, rather than on the general-purpose graph drawing applicability as such. This has led to a considerable number of interactive features, each with a distinctive role in object state inspection, not met in existing tools.

As explained earlier, our visualization approach has been also a spin-off of the task analysis process, which emphasized effective and efficient inspection, rather than overall comprehension. The latter is a novel view regarding the utility of graphical debugging aids. In particular, we observed that programmers study object paths based almost exclusively on infection criteria. Practically, views like dynamic object topology and linkage patterns were of less interest during debugging. They seemed to be appropriate for the design stages of a subsystem, to assimilate its runtime behavior, but not during the bug finding process.

Our evaluation trials have shown that programmers tend to spend more time in detailed inspection of object contents, in identifying corruption patterns and in tracking down referrers and referrers. They reported that they prefer alternative drawing tools mostly for static aspects such as class hierarchies and module dependencies. In conclusion, we believe our work offers a novel insight on the design of interactive graph visualizations for debuggers. Further systematic analysis and support of debugging activities in the future may result in more advanced facilities, effectively leading to more usable and useful interactive debugging environments.

REFERENCES

Aftandilian, E., Kelley, S., Gramazio, C., Ricci, N., Su, S., & Guyer, S. (2010). Heapviz: Interactive heap visualization for program understanding and debugging. In *Proceedings of ACM SoftVis 2010*. Salt Lake City, UT: ACM Press.

Battista, D. G., Eades, P., Tamassia, R., & Tollis, I. G. (1999). *Graph Drawing: Algorithms for the Visualization of Graphs*. Upper Saddle River, NJ: Prentice-Hall.

De Pauw, W., & Sevitsky, G. (1999). Visualizing reference patterns for solving memory leaks in java. In *Proceedings of the ECOOP 1999 Conference*. Berlin: Springer.

Gansner, E., & North, S. C. (1999). An open graph visualization system and its applications to software engineering. *Software, Practice & Experience*, *30*(11), 1203–1233. doi:10.1002/1097-024X(200009)30:11<1203::AID-SPE338>3.0.CO;2-N

GNU DDD. (2009). *Data display debugger.* Retrieved from http://www.gnu.org/software/ddd/.

Heer, J., Card, S., & Landay, J. (2005). Prefuse: A toolkit for interactive information visualization. In *Proceedings of the CHI 2005 Conference on Human Factors in Computing Systems*. Portland, OR: ACM Press.

Hendrix, D., Cross, J., & Barowski, L. (2004). An extensible framework for providing dynamic data structure visualizations in a lightweight IDE. In *Proceedings of SIGCSE'04 35th Technical Symposium on Computer Science Education*. New York: ACM Press.

Hill, T., Noble, J., & Potter, J. (2002). Scalable visualizations of object-oriented systems with ownership trees. *Journal of Visual Languages and Computing*, *13*, 319–339. doi:10.1006/jvlc.2002.0238

Kranzlmuller, D., Grabner, S., & Volkert, J. (1996). Event graph visualization for debugging large applications. In *Proceedings of ACM SPDT'96*. Philadelphia: ACM Press.

Lange, D. B., & Nakamura, Y. (1997). Object-oriented program tracing and visualization. *IEEE Computer*, *30*(5), 63–70. doi:10.1109/2.589912

Muller, T. (2000). Practical investigation of constraints with graph views. In *Proceedings of the International Workshop on the Implementation of Declarative Languages*. Paris: Springer.

Reiss, S. (2009). Visualizing the java heap to detect memory problems. In *Proceedings of VISSOFT '09 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, 73–80. Washington, DC: IEEE Press.

Savidis, A. (2010). *Delta programming language*. Retrieved from http://www.ics.forth.gr/hci/files/plang/Delta/Delta. html.

Savidis, A., & Lilis, Y. (2009). Support for language independent browsing of aggregate values by debugger backends. *Journal of Object Technology*, *8*(6), 159–180. Retrieved from http://www.jot.fm/issues/issue_2009_09/article4.pdf doi:10.5381/jot.2009.8.6.a4

Sundararaman, J., & Back, G. (2008). HDPV: Interactive, faithful, in-vivo runtime state visualization for C/C++ and Java. In *Proceedings of SoftVis'08 4th Symposium on Software Visualization*, 47–56. New York: ACM Press.

Yi, J. S., Kang, Y., Stasko, J., & Jacko, J. (2008). Understanding and characterizing insights: How do people gain insights using information visualization? In *Proceedings of ACM BELIV '08*. Florence, Italy: ACM Press.

Zeller, A. (2005). *Why programs fail: A guide to systematic debugging*. Boston: Morgan Kaufmann.

Zimmermann, T., & Zeller, A. (2002) Visualizing memory graphs . InDiehl, S (Ed), *Software Visualization* (191-204). New York: Springer. doi:10.1007/3-540-45875-1_15