VizzAnalyzer– A Software Comprehension Framework

Welf Löwe Morgan Ericsson Jonas Lundberg Thomas Panas Niklas Petersson

Software Technology Group
MSI/University of Växjö
Vejdes Plats 7, SE 351 95 Växjö, Sweden
{Welf.Lowe|Morgan.Ericsson|Jonas.Lundberg|Thomas.Panas|Niklas.Petersson}@msi.vxu.se
http://www.msi.vxu.se/~rics

Abstract— **We advocate that successful software comprehension methods (and tools) need the synergy of low-level code analyses known from the field of compiler construction, high-level analyses from the field of re-engineering and software visualization techniques. We argue that each individual technique would be either not goal directed or too shallow (or both). This paper describes a software architecture – the VizzAnalyzer– allowing for an easy integration of low- and high-level analysis as well as visualization components.**

I. Introduction

Understanding a legacy system is essential for the further development, the maintenance and the re-engineering of the system. Unfortunately, the architecture of such systems is hardly documented. The only trustworthy source of information is the system implementation. Hence, the architecture has to be retrieved from this source. As real world legacy systems tend to be large, (semi-)automatic *program analyses* must support the extraction of the information.

These analyses are not autonomous, i.e. system engineers have to be involved to accept or reject certain results proposed by the automatic analyses. Hence, the result of such analyses should be presented in a way that is intuitive to the system engineer. Therefore, the program analysis must go hand in hand with interactive *software visualisations*.

The combination of program analyses and software visualisation techniques is crucial to the comprehension of legacy systems, since plain program analysis results are hard to utilise by software engineers. Moreover, the analyses have to be controlled by the software engineers interactively, making it necessary to assess intermediate results. Such an assessment is based on software visualisations. However, plain visualisation of the system's structure (e.g. the abstract syntax tree) or behaviour (e.g. program traces) cannot provide the right information to gain a understanding of the system. Software engineers would simply drown in the flood of information. In order to achieve an understanding, one must therefore reduce the complexity of the original system by program analysis. Implicitly this means to reduce the amount of information in order to achieve comprehension.

This paper is a status report of a project developing a framework, the VizzAnalyzer, aiming at easily combining program analysis and software visualisation techniques. With this framework at hand, we will be in a position to simplify experiments answering our actual research questions like:

- Does a synergy of program analysis and visualisation really leverage software understanding?
- What is a reasonable tradeoff between precision of the analyses and scalability in manageable legacy systems size?
- Does the combination of different heuristics improve software understanding?

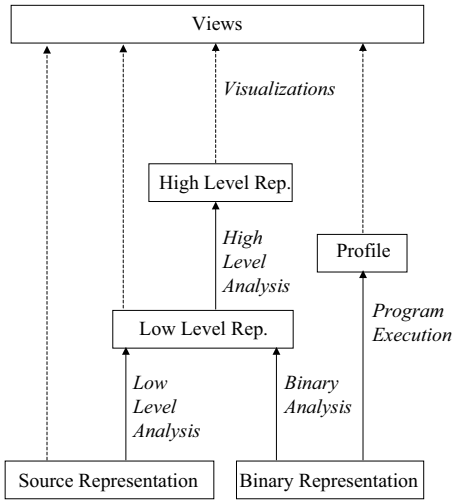

Fig. 1. Taxonomy of analyses techniques and representations.

A. Taxonomy

Software comprehension is a non-trivial task involving various analyses and representations, as shown in Figure 1.

We denote analyses as *high-level* if designed to retrieve a view on the system that is directly suitable to support program comprehension. Examples of such techniques are dominance, coherence, concept, and metric analysis, as well as pattern recognition. Each one of these techniques provides *high level representations* of the system. Examples of such representations are class interaction graphs, lists of possible component or pattern candidates, and architectural description languages – ADLs. The visualisation on

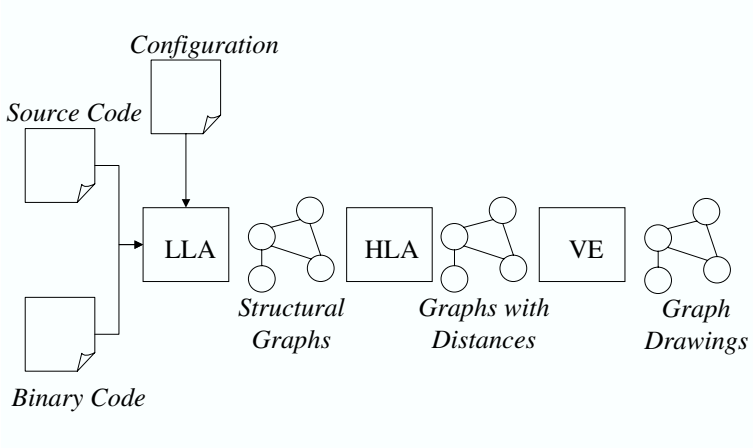

Fig. 2. General Architecture of the VizzAnalyzer framework.

this level includes dominance tree drawings, dendograms, UML diagrams, graph browsers with "collapsing" capabilities, component/connector graph drawings.

High level analysis is usually not applied directly on source level representations like Java or Java byte code. Most papers dealing with high level analysis assume that some *low level analysis*, like lexical and syntactical analyses as well as typing and data-flow analyses, have already been applied, and that the system is given in some kind of *low level representation*. Examples of low level representations are abstract syntax trees (AST), basic block graphs, single static assignment (SSA) representations, call graphs, and control flow graphs. The low level representation is still much to complex to provide any understanding of any non-trivial program. Low level visualisation tools like AST explorers or graph browsers are therefore more useful for those developing high-level analysis techniques rather than those trying to understand a system.

A somewhat separate technique to achieve program comprehension is based on execution of the actual system at hand. We refer to this approach as *run-time analysis*. Examples of run-time analysis are profiles or event traces from a program execution visualised, for example in debugging or profiling tools, or call graphs based on a single program execution.

Furthermore, we distinguish between *structural*, *behavioural*, and run-time information. With structural information we mean the basic static information that can be retrieved by compiler front ends, that is, syntactic and rough type information. Behavioural information is the result of data flow analysis that allows us to derive more accurate approximations of the actual types, call graphs (with resolved polymorphic calls), and control flow graphs associated with the system. Finally, we have run-time information that is gathered during single, or multiple, executions of the program at hand. The run-time information can be considered as exact, but not complete behavioural information.

Finally, the complete source code of a legacy system may not be available. For example libraries, run-time systems, and foreign components may only be available in binary form and parts of the source code could be lost. This requires *binary analyses* of those parts before structural or behavioural information can be retrieved.

B. Article Focus and Overview

This paper presents a framework architecture supporting easy integration of different analysis and visualisation techniques. It is a first attempt to match the requirements on software comprehension tools discussed previously [1]. Moreover, we introduce a prototype implementation of this architecture – the VizzAnalyzer.

The paper is organised as follows: Section II gives an overview of the VizzAnalyzer architecture. The subsequent sections describe its main components in detail. Section VI discusses an extended example. Finally, Section VII concludes the paper and describes plans for future extensions of the framework.

II. Architectural Overview

The architecture consists of three major parts, each prepared as an extension point of the framework where user defined algorithms can be added, cf. Figure 2.

- A low/binary-level analysis engine (LLA).
- A high-level analysis and metrics engine (HLA).
- A visualisation engine (VE).

The LLA takes source and binary code, respectively, and produces model of the program under investigation. The program model is captured in form of one or more structural graphs. These graphs are configurable according to the needs of the HLA. A detailed description on the LLA gives III.

The HLA works on structural graphs. It adds additional information captured in either further graphs or in annotations of existing ones. Special algorithms of the HLA compute annotation that are interpretable as distance metrics between nodes of the graphs. In order to draw a graph it needs to have at least one distance metric computed. A detailed description on the HLA gives IV.

The VE computes a layout of a graph containing distance information and depicts the graph. It provides a user in-

terface allowing for zooming, rotating and aggregating the depicted graphs. A detailed description on the VE gives V.

III. Low-level Analyses

The LLA consists of three major parts, as well. It is depicted in Figure 3.

- A compiler front-end engine (CFE).
- A filter engine (FE).
- An annotation engine (AE).

The CFE reads in source code, constructs an abstract syntax tree (AST), and performs static semantic analysis. To this end it differs in no way from a normal compiler front-end. In contrast to such a component, it filters some sources that are of no interest for the forthcoming analyses. For example classes that are part of the system environment are excluded. In an interactive comprehension process, one could also exclude parts that are already understood. Filters are specified in a configuration file.

In our VizzAnalyzer implementation, we rely on the RecoderR meta-programming library [2], providing a compiler front-end for Java programs and an API for accessing AST and semantic analysis results.

The FE filters the AST produced by the CFE. Filtering of AST nodes is also guided by the configuration file. Basically, all AST node types can be removed individually. Depending on the high-level analyses, this filter can remove already superfluous information.

Moreover, one can create several different graphs by defining several different filters. It is, for instance, possible to construct a class graph containing only declarations or a call graph containing reference and method declaration nodes by defining two different filters.

Based on the assumption that one usually wants to remove more node types than one wants to keep, our implementation requires one to specify specifies the node types to keep rather than the types to filter out.

The AE adds additional edges to the remaining trees. These edges correspond to constructor/method calls, field accesses, and inheritance relations derived by the static semantic analyses of the CFE. For example, the method references are connected to the (static) target method declaration to create a call edge.

If the direct source or target node of an edge is filtered by the FE, the edges are propagated to their first (transitive) parent that is not filtered. If by any chance all transitive parents of source or target are filtered, the edge is not inserted at all.

Again, the edges to add to the different filtered ASTs can be chosen in the configuration.

Additionally, the AE perform some adjustments refining the conservative analysis result from the static semantic analysis of the CFE. These refinements regard the targets of constructor/method calls and field accesses, and support later high-level analyses. In the remainder of this paper we will refer to them as references.

A constructor reference might point to a default constructor that is implicitly defined in any class. This default constructor does not correspond to an AST node since it is not defined explicitly in the source program. The AE adds a default constructor node explicitly to the AST of the corresponding class (if necessary). Hence, any edge corresponding to a constructor reference has a target AST node.

A method or field reference might address an entity defined in a superclass of the static target class. Consider, e.g., the situation given in Figure 4.

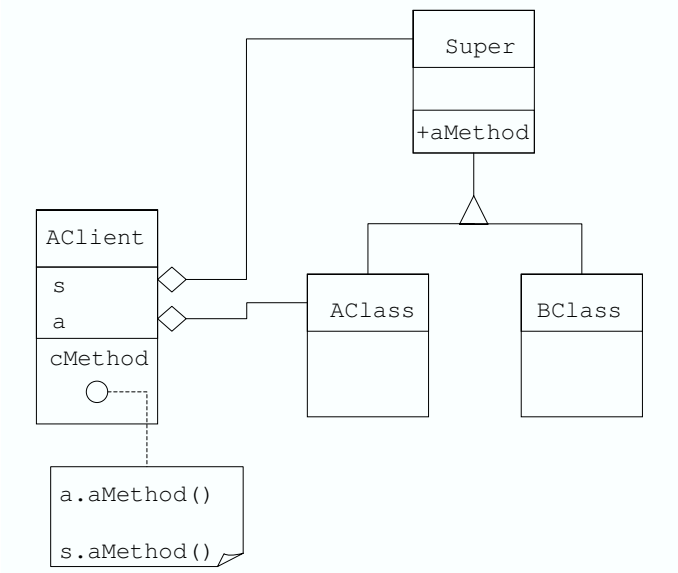

Fig. 4. Example Program Structure.

The method a call `a.aMethod()` reaches at runtime could be defined in `Super` and in any subtype of `AClass` but never in a subtype of `BClass`. In contrast, the method of a call `s.aMethod()` could be defined in `Super` and in any subclass of `AClass` and `BClass`. Provided we keep method and class declarations after filtering the AST, both calls would lead to an edge from `AClients`'s `cMethod` declaration to `Super`'s `aMethod` declaration. High-level analyses cannot distinguish both situations and should assume conservatively that the edge represents a polymorph call to `Super`'s `aMethod`.

One could resolve this with edge annotations or additional edges. To keep our graph representation simple, we have decided not to introduce new kinds of information. Instead, we attach an explicit dispatch method to the static target of such references. Like the solution to the default constructor, this is semantically equivalent to the original program and only affects the AST, and not the source program.

In our example, the call `a.aMethod()` is processed by a dispatch method defined in `AClass` calling monomorphically either `Super`'s `aMethod` or a method defined in a subclass of `AClass`. In contrast, the call `s.aMethod()` reaches the dispatcher in `Super` distributing the call either to the local `aMethod` declaration or to a `aMethod` declaration in the proper subclass of `AClass` and `BClass`.

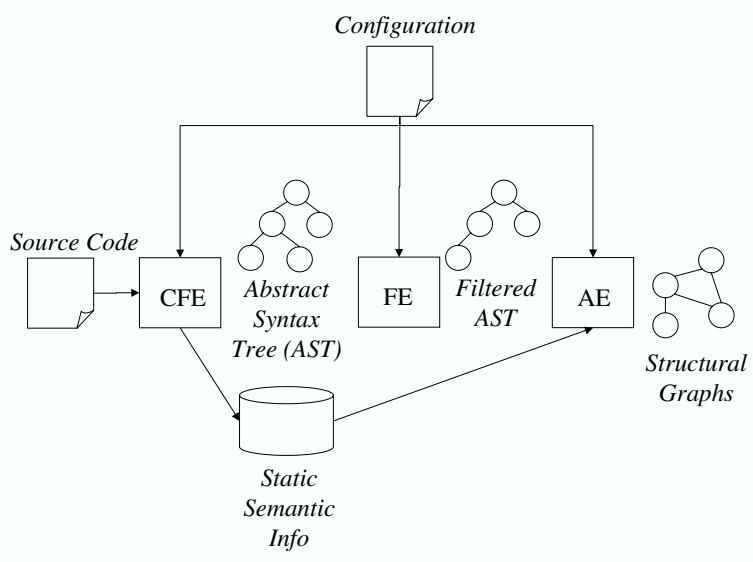

Fig. 3. Architecture of the VizzAnalyzer's low-level analysis.

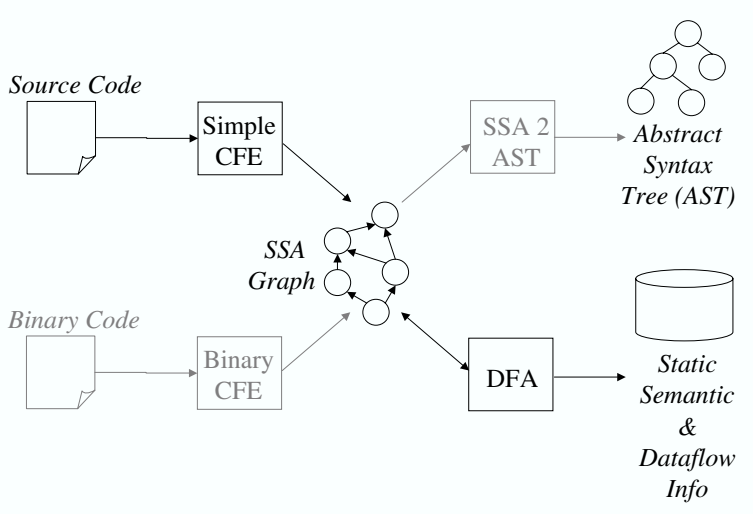

Fig. 5. Extended Compiler Front-End. Grey components are under construction.

However, this fix cannot replace more advanced low-level analyses, i.e. data-flow, binary and dynamic analyses discussed below.

A. Data-flow and Binary Analyses

In any object oriented language, the major reason for conservative assumptions on references is polymorphism: the static type of a reference (analysed by the CFE) is a super-type of the actual reference (occurring at runtime). Reference analyses (or points-to analyses) is generally undecidable and approximations require exponential time on increasing precision (context sensitivity). Much research has been done in the past, mainly in the field of compile time optimisations. A good overview and classification gives [3].

The VizzAnalyzer implements a simple but fast reference analysis on the level of Rapid Type Analysis (RTA) [4], [5]. In order to support more advanced analyses as plug-ins, we have extended the CFE: in addition to the simple CFE generating the AST directly, our CFE can also generate an intermediate representation of program in *Static Single Assignment (SSA)* form [6]. It computes essential data dependencies once and store them as edges in the a graph simplifying subsequent data-flow analyses. We use memory-SSA [7], an extension of SSA that allows for the representation and analysis of object-programs. Our implementation uses the FIRM-library [8].

As stated earlier, parts of the legacy system may only be available in binary form. Information about these parts must be approximated very conservatively, since their real behaviour is unknown. For example, a reference passed to a binary component may be captured by the component and the object referred may be changed at any point during program execution. The binary analysis needed is designed to exclude as many of these situations as possible and allow a less conservative and more correct approximation.

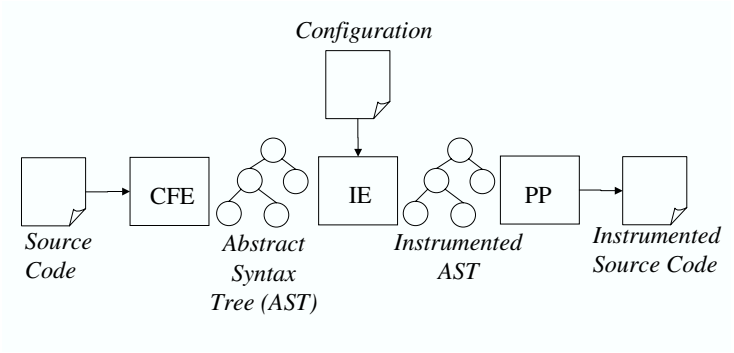

Fig. 6. Preprocessing for dynamic analyses.

To achieve a less conservative approximation, the binary analysis must try to recreate as much of the information contained in the original program as possible. This process is very much like compilation: it requires syntax and semantic (mainly type) analyses. The area is well researched since every tool that operates on binaries, for instance decompilers, needs to do these analyses [9].

In our architecture, a binary front-end reads binary (ELF) formats, and constructs an SSA graph. As *firm* provides operations on machine code level, this step is rather a coding problem. The second step of the binary analysis is to recreate composite data types, like arrays and records. The analysis uses so called "synchronisation points", that is points in the program where the types are known, for instance via interface definitions. By tracing accesses backward and forward from a synchronisation point it is possible to infer the types. This approach has been used to analyse Java byte-code [10], but not "real" binaries[1].

An architectural overview of the extended front-end is given in Figure 5.

B. Dynamic Analyses

Static analyses provide information stratified by all program runs. This necessarily leads to imprecisions in any specific run. Hence, many high-level analyses use program observations instead of or in addition to static analyses, e.g. for architectural recovery [11], [12], [13] or our own contributions to design pattern detection [14], [15]. These dynamic analyses deliver exact results on specific program runs and can be used, e.g., to identify classes needed in certain application use cases or to false positives of a static analysis.

Dynamic analyses work pretty much like a debugger: at certain points (breakpoints) of the analysed program (debuggee) the control is handed over to an analyser (debugger) that observes the state of the analysed program. Our first attempt implemented this idea directly [16]. Programs were executed in a standard Java Virtual Machine controlled by the VizzAnalyzer via the Java Debug Interface. Whenever a relevant program point was executed, the analyser received an event, accessed and stored the relevant program state, and resumed the program execution.

The approach described above was to slow for larger programs, so code instrumentation was used instead. Relevant program points, like method call, exit, entry, and attribute access, can be extended by calls to the VizzAnalyzer. All program information needed for the analyses is handed over at once along with these calls. Code instrumentation allows for the analyses of large programs without a considerable slow down, as our experiences with design pattern detection showed.

However, it requires a preprocessing phase preceding the actual analysis. After AST construction, a configurable instrumentation engine (IE) adds calls to the VizzAnalyzer. Before the actual dynamic analysis starts, a pretty-printer (PP) serialises the instrumented code, cf. Figure 6, which then can be compiled and executed with standard Java compilers and VMs (not depicted). On execution, the instrumented program provides the specified runtime information.

For pre-processing, we again exploit the meta-programming facilities of RecodeR.

IV. High-level Analyses

The high-level analyses precedes the low-level analyses in a series of abstractions. Implicitly, this means that we reduce the amount of information in order to achieve comprehension at the cost of accuracy.

In architectural recovery, e.g., we first extract the information needed to construct a call graph, which is our low-level representation of the program. The low-level representation contains information that we consider to be essential in order to recover the system architecture. However, for any non-trivial real-world program this structure is still much to complex to provide the system engineer with architectural insight. For example, a medium sized program often results in a call graph with thousands of nodes and about twice as many edges. In order to achieve an architectural understanding, we must therefore reduce the complexity of the low-level representation by further abstractions.

The low-level representation computed by the LLA consists of structural graphs. Nodes correspond to AST nodes, edges to relations between them like calls, field accesses,

[1]It should be mentioned that the SSA construction from binaries and the generation of the AST from SSA are not currently functional. These components represent work in progress.

and inheritance. Further information is attached, e.g. type information from the semantic analysis. A user configuration determines node and edge types contained in the single structural graphs.

The HLA engine consumes these graphs and pushes the abstraction even higher. It supports two kinds of analyses, both represented by abstract classes in the framework. These analyses are:

- Graph analyses (GA),
- Metric analyses (MA).

Special subclasses of the GA are filters and aggregations. Both filters and aggregations perform a reduction of information. While filters disregard (removes) parts of the graph, aggregations map several nodes to a representative node. These special GAs need not to be implemented individually but are predefined. Instantiations just define the node and edge types to filter and and aggregation function, respectively.

A MA takes a graph and attaches properties to the graph, its nodes, and/or its edges. A graph global property is, e.g., planarity or sparseness; common node properties are the different software metrics of class complexity.

A special subclass of the MA are those computing distances between nodes. These distances are required if the graph is supposed to be visualised later on. They calculate the "ideal" distance of every two nodes according to some distance measure. The distance measure is determined by the comprehension problem. It could model similarity of nodes or coupling to the rest of the graph. Note, that a distance metric must not necessarily regard the edges in the graph.

The HLA engine consists of a series of GA and MA until the information supporting software understanding directly is computed. Each needs certain types of nodes and edges to be available or certain properties to be computed before. To guarantee (dynamic) type safety of the high-level analyses, each analysis must implement a boolean `accept` function accepting a graph if it provides sufficient information to perform the analysis.

The hierarchy of abstract classes that the HLA provides is depicted in Figure 7. Users of the framework can extend the classes `DistanceAnalysis`, `MetricAnalysis`, and `GraphAnalysis`. Additionally, they can instantiate the predefined `Filter` and `Aggregate` classes or, of course use one of the predefined analyses like the package hierarchy constructor (not depicted).

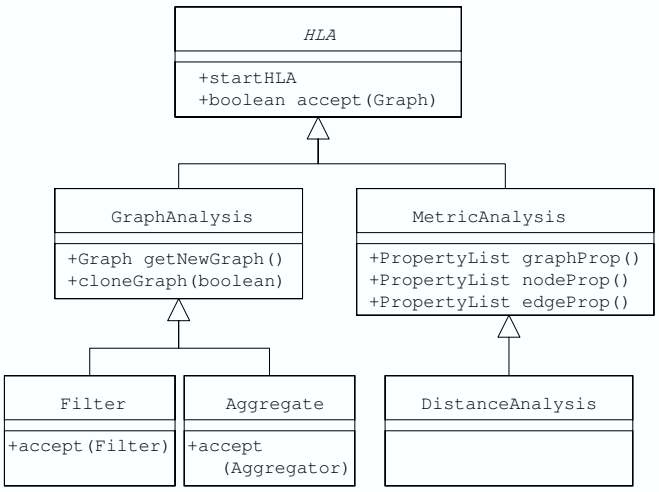

Fig. 7. Hierarchy of classes in the HLA engine.

V. Visualisations

The scale and complexity of software and its visualisation are pressing issues, as is the associated information overload problem that this brings. Abstractions, as mentioned before is the key to reduce the enormous quantities of information contained in real world programs. Further attempts to address the problem of understanding complex systems are considered to be [17]:

- **Metaphors** The mapping from a program model (lower level of abstraction) to an image (higher level of abstraction) is defined through a metaphor, specifying the type of visualisation. Most visualisation techniques and tools are based on the graph metaphor (including the extensive research on graph layout algorithms). Other initiatives are the representation of programs as, for instance 3D cities [18], solar systems [19], video games [20], [18], nested boxes [21], [22], or 3D Spaces [23].
- **Visualisations** It is not feasible to depict all kinds of program model phenomena in just one picture when the model carries too much information. Therefore each program model is depicted through various views, guaranteeing that the right subset of objects and their relations are depicted and understood. Most efforts to solve the problem of software complexity is put into different visualisation forms, mainly the graph metaphor, including UML diagrams, to depict various program class and architecture views [24], [25], [26], [27].

As a result, we intend to provide the user with interactive, parallel and natural views to simplify perception of a program and its comprehension. Metaphors, when depicting real world entities and established social interactions, especially in virtual reality, is considered to be important. The choice of which metaphors to use when displaying the program analysis results may similarly affect the usability of a software comprehension tool. A problem with many graphic designs is that they have no intuitive interpretation, which requires much training of the user in order to understand them. Metaphors found in nature or in the world avoid this by providing a graphic design that the user understands.

The VizzAnalyzer provides the user with the flexibility to choose among a number of different metaphors and layout algorithms. It maps the graph representations of the LLA and HLA to a Java3D SceneGraph. A Java3D SceneGraph is a directed acyclic graph arranged in a tree structure, where the viewer (camera) usually represents one subgraph and the scene another. The nodes of the scene subgraph correspond to nodes in an abstract syntax tree. This maps e.g. a class, method, etc. in the AST to a

building or sphere in the SceneGraph. The objects of the SceneGraph and their layout are described within the VizzAnalyzer as dynamically loadable metaphors and layout algorithms. Hence, an AST can be visualised as either globes and lines or buildings and roads, etc. Nevertheless, a metaphor can not be illustrated with any layout algorithm, e.g. a city metaphor would somehow restrict the layout on the y axis (so the houses do not fly in the air). Therefore, for any metaphor the layout algorithm must be chosen wisely.

Java3D provides us with all the necessities to illustrate our analysis results interactively in parallel views with user individual metaphors and layout algorithms. While a metaphor is described through a metaphor-XML file, the mapping between the elements in the actual reverse engineered graph (e.g. AST) and the SceneGraph happens in a binding-XML file. With the latter, a transfer function is used to enable metric information to be passed on to the SceneGraph. For example, if an AST has the metric information "class size", there will be a mapping from an actual value of the size of a class in the AST to a new value in the SceneGraph based on the transfer function. The new value might have a range from for instance zero to ten, indicating the size that the rendered object should have. What kind of object is drawn, is specified in the metaphor-XML file. Figure 9 (bottom) shows an example of a city metaphor laid out with a leveller layout algorithm. The shape and texture of the houses are part of the metaphor-XML file. The metaphor-engine supports the import of meshes from 3DStudio Max.

VI. Example

The project (program) to be analysed is specified in an XML structure like the one below. In this example we use the VizzAnalyzer is analysing a simple version of itself. After some general information in beginning, it contains the usual compiler information required by RecodeR to parse a given program: entry point of the program to analyse, its source and classpath. Finally it defines the first set of filter indicating that we are interested in packages with prefix `viz`. When the analysis reaches classes packages beginning with `java`, further analysis is aborted. leads to an analysis of our example program but excludes the of the Java library classes.

```
<?xml version="1.0" encoding="UTF-8"?>
<project>
  <projectname>Analyze Example</projectname>
  <projectdescription>
    Analyzes a small example application
  </projectdescription>
  <projectspecification>
    <entrypoint>vis.vis3d</entrypoint>
    <sourcepath>
        C:\My Working Set\Example\
    </sourcepath>
    <classpath>
        C:\Program Files\...rt.jar;...
    </classpath>
    <filter>
      <include>vis</include>
      <exclude>java</exclude>
    </filter>
  </projectspecification>
</project>
```

Next we discuss the configuration of the graph that the low-level analysis should deliver. Our example specification below specifies that there is just one graph to construct, namely a `Class Reference Graph`. Its edges are those outgoing from `FieldReference`, `MethodReference`, and `New` AST nodes. These edges are (by definition) attribute accesses, method calls and object creations, respectively. However, we are not interested in these nodes themselves. Instead, we propagate source and targets of detected edges to enclosing `ClassDeclaration` and `InterfaceDeclaration` AST nodes. Both edge and node types contained in a graph can easily be changed by adding new program elements to the `<edges>` and `<nodes>` lists, respectively. Any kind and number of such graph specifications could be added to the `<graphs>` list.

```
<?xml version="1.0" encoding="UTF-8"?>
<graphs>
    <graph>
      <name>Class Reference Graph</name>
      <description>
        Register calls between classes
      </description>
      <edges>
        <edge>FieldReference</edge>
        <edge>MethodReference</edge>
        <edge>New</edge>
      </edges>
      <nodes>
        <node>ClassDeclaration</node>
        <node>InterfaceDeclaration</node>
      </nodes>
    </graph>
</graphs>
```

The low level analyses delivers a directed graph: nodes represent class and interface declarations, edges different kinds of usage (attribute access, method call, object creation) between them. It is a multi-graph; any detected usage between two classes is represented by a single edge.

The first high-level analysis to be discussed accumulates multiple edges and replaces them by a single edge, labelled with the number of edges in the previous graph. Note, that this is a reduction of information contained in the previous graph where information about source and target AST nodes were kept in each individual edge. This level of detail is lost in the abstractions performed by the high-level analysis.

As a side effect, the analyses prints the nearest-neighbour-coupling-strength to a table, importable by MS Excel. Figure 8 depict the coupling strength of two classes, `vis.java3d.Visualizer` (top) and the main class `vis.vis3d` (bottom) to their nearest neighbours.

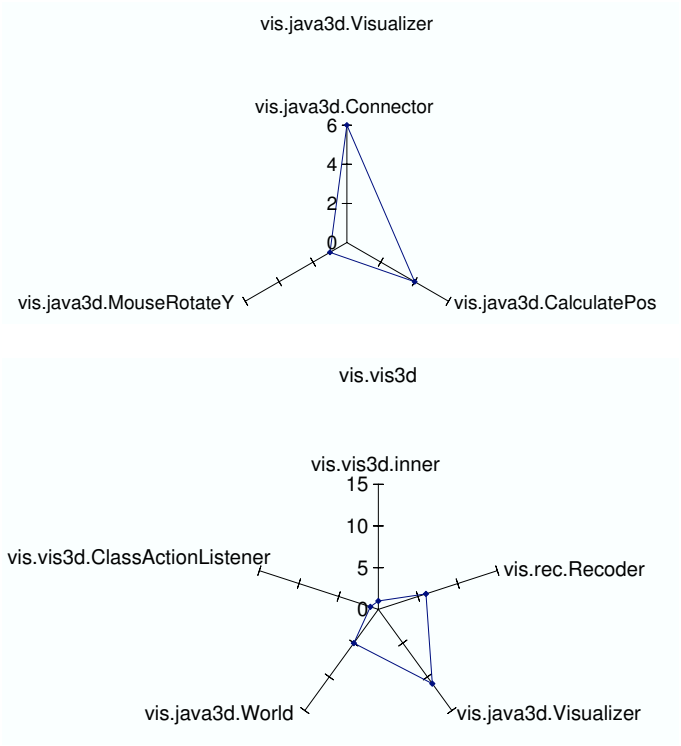

Fig. 8. Nearest-neighbour coupling of classes in the example program. Zero usages relations are not depicted. Values on the axis represent the number of usages of the corresponding class.

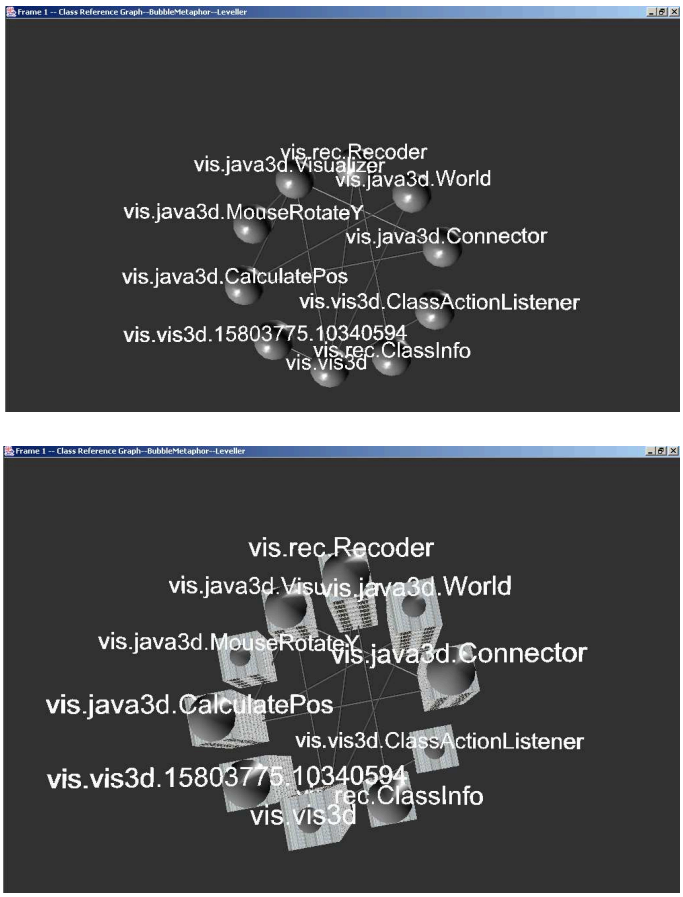

Fig. 9. Changing the metaphors of a graph visualization.

Figure 9 shows the graph constructed by the first analysis using our own 3D visualisation prototype. It demonstrates the facility to change the binding between program elements and metaphor configurations.

The second example high-level analysis adds further structure to the graph. It builds a package hierarchy using the fully qualified class names. Although not defined like this in the Java language, programmers structure packages in a hierarchical way. This "virtual" package contains relation is made explicit in this high-level analysis. For our examples, package `vis` is assumed to contain the class `vis.vis3d` and the package `vis.java3d`, which in turn contains the class `vis.java3d.Visualizer`. This hierarchy is represented with additional edges and nodes, added to the accumulated class usage graph. This situation is depicted in Figure 10 where transparent nodes represent packages, containing other packages or classes. The latter are not transparent.

The main loop controlling the described high-level analyses is actually as simple as the code below shows:

```
//Get the graphs from the low-level analysis:
Iterator graphs=
    dataManager.getGraphList().iterator();

//For all these graphs:
while (graphs.hasNext()){
    GraphInterface graph =
        ((GraphInterface)graphs.next());

    //First high-level analysis:
    HLA hLAEngine =
        new UsageGraphConstructor();
    if (!hLAEngine.accept(graph)) continue;
    hLAEngine.startHLA();
    GraphInterface usageGraph =
        hLAEngine.getNewGraph();

    //Second high-level analysis:
    hLAEngine = new PackageGraphConstructor();
    if (!hLAEngine.accept(usageGraph)) continue;
    hLAEngine.cloneGraph(false);
    hLAEngine.startHLA();
    GraphInterface packageGraph =
        hLAEngine.getNewGraph();

    //Export to Wilma as a side effect:
    hLAEngine = new PrintToWilma();
    if (!hLAEngine.accept(packageGraph)) continue;
    hLAEngine.startHLA();
}
```

Other predefined high-level analyses can easily be added by creating objects of other `HLA` subclasses. Alternatively, the sequence of high-level analyses and subsequent visualisations can be selected via a GUI. Adding new HLA

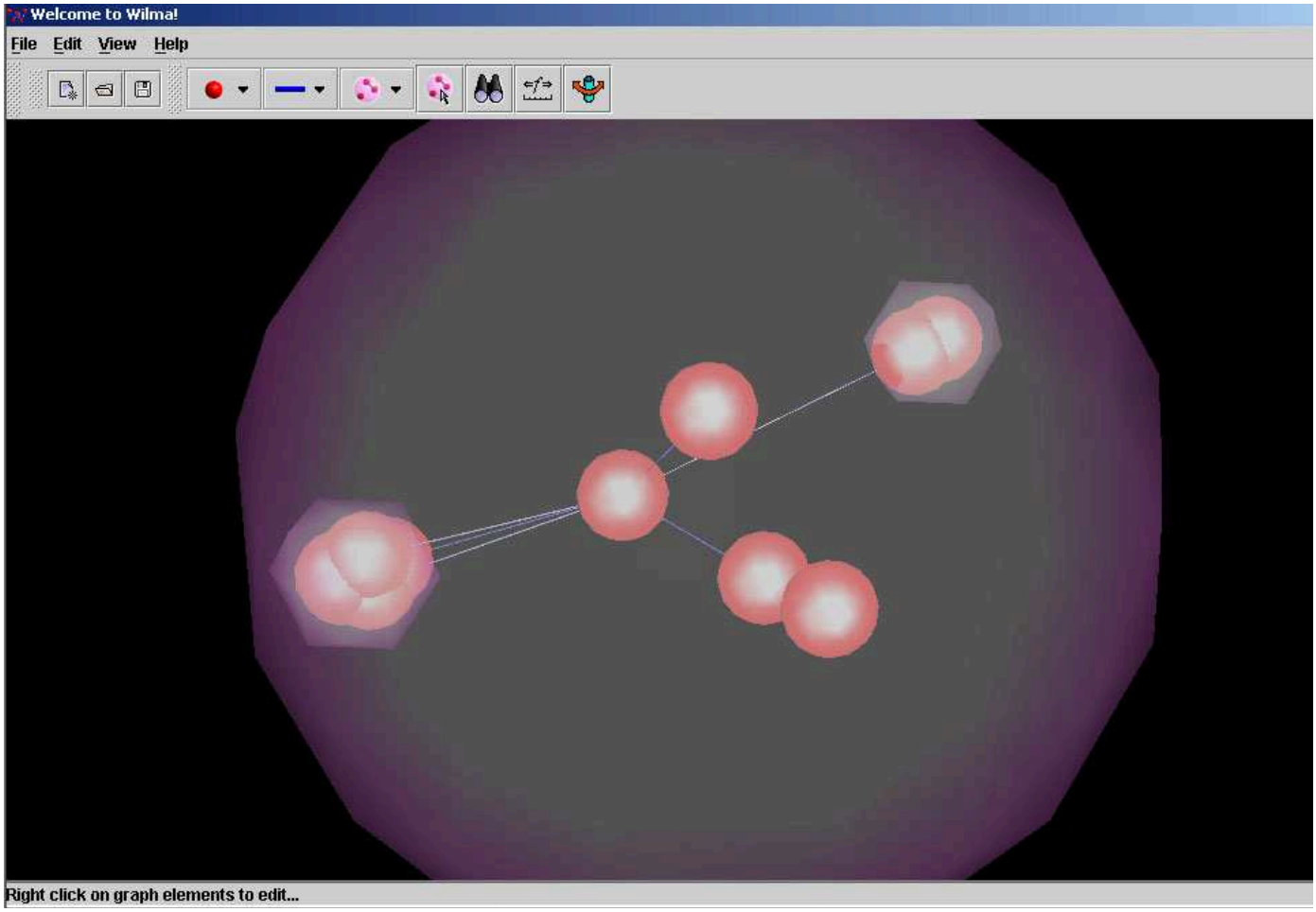

Fig. 10. Package hierarchy and class usage in an example program, visualised using Wilma.

requires the implementation of a new HLA subclass.

VII. Conclusion, Current and Future Work

This paper presents is a project status report about work in progress.

It presents a framework for program analysis and visualisation, the VizzAnalyzer. Its main design goals are flexibility (to add further low- and high-level analyses as well as further software visualisations) and robustness. The flexibility is achieved by:

Configurations: The low-level analysis allows to filter out source files not to be analysed and AST nodes not to be regarded. Later in the visualisation, the mapping of abstract program model elements to metaphors is controlled by a configuration, as well.

Framework extension: New high-level analyses can be added as extensions of predefined abstract analyses classes.

Configurations are robust by definition of the configuration language; we use dynamic type checking for guaranteeing robust framework extensions.

The framework architecture and the implementation of the core functionality are currently quite stable. A lot of implementation work, however, needs to be done in the data-flow, dynamic and binary analysis parts of the low-level analysis engine and in layouting the graphs. When this is done, we will apply the tool on itself looking for both design flaws uncovered by the analyses and visualisation that work, and implementation errors uncovered by those that do not.

References

[1] W. Löwe, M. Ericsson, J. Lundberg, and T. Panas, "Software comprehension - integrating program analysis and software visualization," in *2nd Conf. on Software Engineering Research and Practise in Sweden*, 2002.

[2] Andreas Ludwig, *RECODER Homepage*, `http://recoder.sf.net`, 2001.

[3] B. Ryder, "Dimensions in precision in reference analysis of object-oriented languages," in *Compiler Construction*, 2003.

[4] David F. Bacon and Peter F. Sweeney, "Fast static analysis of C++ virtual function calls," *ACM SIGPLAN Notices*, vol. 31, no. 10, pp. 324–341, 1996.

[5] Frank Tip and Jens Palsberg, "Scalable propagation-based call graph construction algorithms," *ACM SIGPLAN Notices*, vol. 35, no. 10, pp. 281–293, 2000.

[6] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck, "An efficient method of computing static single assignment form," in *POPL*, 1989, pp. 25–35.

[7] M. Trapp, *Optimierung Objektorientierter Programme*, Xpert.press. Springer, 2001.

[8] Martin Trapp and Götz Lindenmayer, "Explizite

abhängigkeitsgraphen," Tech. Rep., Universität Karlsruhe, 1999.

[9] C. Cifuentes, *Reverse Compilation Techniques*, Ph.D. thesis, Queensland Univ. of Technology, School of Computing Science, July 1994.

[10] Shin ya Katsumata and Atsushi Ohori, "Proof-directed decompilation of low-level code," in *European Symposium on Programming*. 2001, LNCS 2028, pp. 352–366, Springer.

[11] Rainer Koschke, *Atomic Architectural Component Recovery for Program Understanding and Evolution*, Ph.D. thesis, Institute for Computer Science, University of Stuttgart, 2000.

[12] Rick Kazman and S. Jeromy Carrière, "Playing detective: Reconstructing software architecture from available evidence," *Automated Software Engineering: An International Journal*, vol. 6, no. 2, pp. 107–138, April 1999.

[13] R. Kazman, S. G. Woods, and S. J. Carriere, "Requirements for integrating software architecture and reengineering models: Corum II," in *Working Conference on Reverse Engineering (WCRE'98)*, October 1998.

[14] D. Heuzeroth, T. Holl, and W. Löwe, "Combining Static and Dynamic Analyses to Detect Interaction Patterns," in *6th Int. Conf. Integrated Design and Process Technology (IDPT)*, June 2002.

[15] Dirk Heuzeroth, Gustav Högström, Thomas Holl, and Welf Löwe, "Automatic Design Pattern Detection," in *11th IWPC*, May 2003.

[16] Dirk Heuzeroth and Welf Löwe, *Software-Visualization - From Theory to Practice*, chapter Understanding Architecture Through Structure and Behavior Visualization, Kluwer, 2003.

[17] C. Knight, *Visual Software in Reality*, Ph.D. thesis, University of Durham, 2000.

[18] C. Knight and M. C. Munro, "Virtual but visible software," in *IV00*, 2000, pp. 198–205.

[19] P. Damien, "Building program metaphors," in *PPIG'96 Post-Graduate Students Workshop at Matlock, UK*, September 1996.

[20] K. Kahn, "Drawing on napkins, video-game animation, and other ways to program computers," *Communications of the ACM*, vol. 39, no. 8, pp. 49–59, 1996.

[21] J. Rekimoto and M. Green, "The information cube: Using transparency in 3d information visualization," 1993.

[22] S. P. Reiss, "An engine for the 3d visualization of program information," *Visual Languages and Computing (special issue on Graph Visualization)*, vol. 6, no. 3, 1995.

[23] J. Rilling and S.P. Mudur, "On the use of metaballs to visually map source code structures and analysis results onto 3d space," in *Ninth Working Conference on Reverse Engineering (WCRE'02)*. IEEE, October 2002.

[24] T. Ball and S. G. Eick, "Software visualization in the large," *IEEE Computer*, vol. 29, no. 4, 1996.

[25] C. Lewerentz and F. Simon, "Metrics-based 3d visualization of large object-oriented programs," in *1st Int. Workshop on Visualizing Software for Understanding and Analysis*, June 2002.

[26] J. Stasko, J. Domingue, M. H. Brown, and B. A. Price, Eds., *Software Visualization*, MIT Press, 1998.

[27] P. Eades, *Software Visualization*, World Scientific Pub Co, 1996.