JexVis: An interactive visualization tool for exception call graphs in Java.

Anirban Sinha[1]
Department of Computer Science
University of British Columbia, Canada

ABSTRACT

Just as the structure of any system degrades over time as it evolves & becomes more complicated, software systems are no exception to this. A particular aspect of software systems is its exception handling mechanism which also tends to get dirty with time. Jex, a tool developed by Martin Robillard at University of British Columbia is a static analyzer that extracts exception flow information from Java programs. It generates a list of exceptions that can be raised by any method of a class & presents this information textually in the context of exception handling structure in the program. In this paper, we present JexVis, a tool that uses the textual information to generate an interactive visualization of exception structure in any program or its modules. In the process of design of JexVis, the author actually modifies the original tool, Jex so that it produces a textual output in the form of exception call graph which is then visualized using java's swing capabilities.

C.R Categories: H.5.2 [User Interfaces]: Graphical User Interfaces (GUI); D.2.5 [Testing and Debugging]: Error handling and recovery; D.2.2 [Tools and Techniques]: User interfaces.

Keywords: Software Visualization, Exception Handling Design, Error Handling, Jex, Exception Call Graph.

1 INTRODUCTION

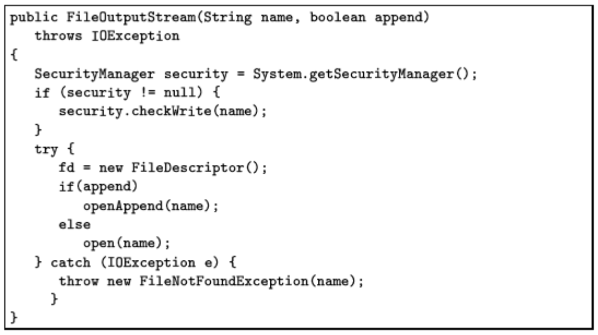
To make design of robust software systems easy, modern languages like Java incorporate well defined mechanism to handle exception conditions. To be specific, these languages provide mechanisms to explicitly raise exceptions at a point in the program & a separate block of code or several blocks of code to handle the exception or several types of exceptions that can be raised from that point. Thus, the exception handling block is separated from the normal program execution logic. This helps programmers to deal with the exceptional situations & runtime errors separately from the original program logic. However, despite the availability of these structures built into the language & developer's best intensions to structure the system as neatly & as efficiently as possible, addition of new modules invites new additional calls between methods & incorporation of new classes & methods derived from existing classes & methods. This increases complexity & invites degradation of code. Exception generation & their handling are also not protected from this degradation. Martin & Gail in their technical paper [4] explain some of the root causes of exception degradation. These include unanticipated or unchecked exceptions; exception handler overload leading to a single exception handler unknowingly subsuming unanticipated exceptions; exception propagation upwards through the method call stack leading to inappropriate or uninformed handling of the exception etc.

Manually understanding how exceptions are handled within each method requires elaborate & concrete knowledge about the exceptions that might arise as a method is executing, the exceptions that are handled and the exceptions that are passed on through the method call stack. This, needless to say, would be highly tedious, even for simple programs & virtually impossible for larger projects.

Jex [10], designed at University of British Columbia by Martin Robillard & Gail Murphy is a static analysis tool that provides information about the exceptions that can be raised in a Java program. For each method of each class, Jex outputs a description of all exceptions that can be raised in the method. It includes the types of the exceptions raised by subsumption and by polymorphic calls. For example, if a method declares to raise IOExceptions, but in reality raises both IOException and FileNotFoundExceptions (a subtype of IOException), Jex reports each of the different subtypes of IOExceptions that can be raised. Jex also reports the origin of exceptions stemming from polymorphic calls by using a conservative class hierarchy analysis algorithm. The information reported by Jex is also more complete because it includes information about the types of unchecked exceptions that can be raised. This can be easily explained through the codes shown in Figures 1 to 4 which are taken from the Jex paper [1]. Figure 1 shows the constructor for the Java.io.FileOutputStream class. Figure 2 illustrates the information generated by Jex that shows precise information of various exceptions generated by the methods. A naïve user, without the precise knowledge of various exceptions generations would use FileOutputStream class in a way similar to Figure 3 where a generic catch block accepts all different kinds of exceptions. However, with the power of Jex, a more learned user would now separate out various exception traps & will now have different catch blocks for each of the different types of exceptions generated as shown in Figure 4.

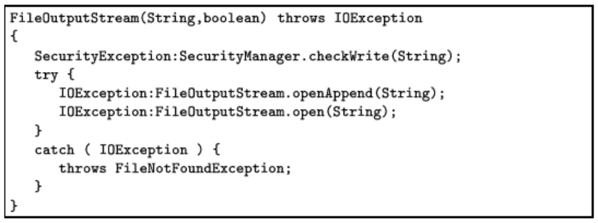
However, as is seen from the figures, the output generated by Jex is in textual format where the user needs to parse the Jex output file manually in order to extract meaningful information. This is not very useful & again involves careful analysis & introspection on the part of the programmer. JexVis tries to address this very issue by creating a graphical interactive visualization of the exception call graph structure so that a naïve user can easily understand how exceptions are generated & propagated within classes & modules. In order to achieve our target the author uses information visualization concepts & ideas. For example, the same output generated by Jex is now visualized into a tree structure in JexVis as shown in Figure 5. Clearly, the interrelationships between the method calls & the exceptions they generate are easily understandable.

[1] anirbans@cs.ubc.ca



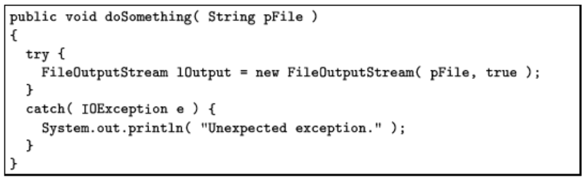
```
public FileOutputStream(String name, boolean append)
   throws IOException
{
   SecurityManager security = System.getSecurityManager();
   if (security != null) {
      security.checkWrite(name);
   }
   try {
      fd = new FileDescriptor();
      if(append)
         openAppend(name);
      else
         open(name);
   } catch (IOException e) {
      throw new FileNotFoundException(name);
     }
}
```

Fig. 1. The source code for the constructor of class FileOutputStream



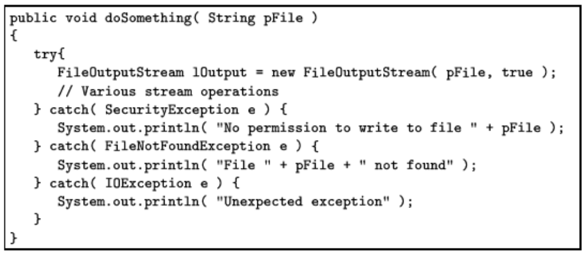
```
FileOutputStream(String,boolean) throws IOException
{
   SecurityException:SecurityManager.checkWrite(String);
   try {
      IOException:FileOutputStream.openAppend(String);
      IOException:FileOutputStream.open(String);
   }
   catch ( IOException ) {
      throws FileNotFoundException;
   }
}
```

Fig. 2. The structure of exceptions for the constructor of class FileOutputStream



```
public void doSomething( String pFile )
{
  try {
    FileOutputStream lOutput = new FileOutputStream( pFile, true );
  }
  catch( IOException e ) {
    System.out.println( "Unexpected exception." );
  }
}
```

Fig. 3. An example of code not using Jex information



```
public void doSomething( String pFile )
{
   try{
      FileOutputStream lOutput = new FileOutputStream( pFile, true );
      // Various stream operations
   } catch( SecurityException e ) {
      System.out.println( "No permission to write to file " + pFile );
   } catch( FileNotFoundException e ) {
      System.out.println( "File " + pFile + " not found" );
   } catch( IOException e ) {
      System.out.println( "Unexpected exception" );
   }
}
```

Fig. 4. An example of code making use of Jex information

On placing the mouse pointer on the node corresponding to the "open" method, the path from the parent to the exception generated by the method gets highlighted in blue & the tool tip shows the full method name. This helps the user to clearly understand which method generates which exceptions. The user can use pan & zoom to focus into a specific node, trace call execution along a particular branch etc.

The current version of JexVis was designed using eclipse SDK [14], along with Prefuse toolkit for graphical rendering [11]. The author used Fedora Core 3 as his operating system platform for development. JexVis, in the backend, calls Jex for actual static analysis.

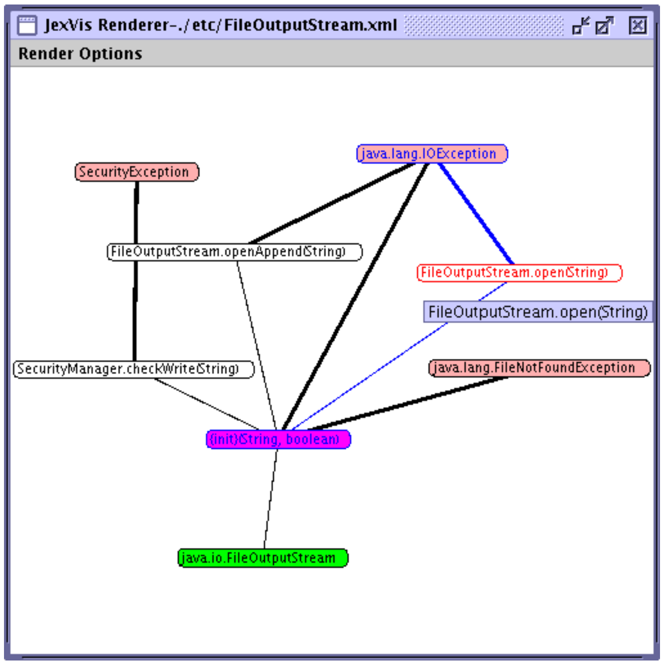

JexVis Renderer-./etc/FileOutputStream.xml
Render Options
java.lang.IOException
SecurityException
FileOutputStream.openAppend(String)
FileOutputStream.open(String)
FileOutputStream.open(String)
SecurityManager.checkWrite(String)
java.lang.FileNotFoundException
(init)(String, boolean)
java.io.FileOutputStream

Figure 5: The JexVis rendering of the Jex file shown in Fig 2.

The rest of this paper is organized as follows. Section 2 discusses some related work in this direction. Section 3 discusses JexVis in detail including its several features & user interfaces. Section 4 discusses evaluation of the tool and scalability issues. Section 5 discusses some of the important information visualization lessons learnt during the course of the project. Section 6 goes into some of the main challenges faced during design of JexVis. Section 7 gives some future directions for this work & finally section 8 concludes the paper.

2 RELATED WORK

The Aristotle Research Group [13] has designed a set of tools for static analysis of source codes from various programs. The main infrastructure consists of parsers that gather language-specific information, such as control-flow, local data-flow, and symbol table. Other sets of tools use this information to perform additional analysis, such as data-flow, control-dependence, pointer, slicing, and visualizations. The various components of the program-analysis infrastructure include the Aristotle Analysis System, the Java Architecture for Byte code Analysis and the Rational Apex System. Still other tools provide information for software-engineering tasks, such as testing, regression testing, debugging, test-suite management, and program understanding. Figure 6 elaborates this point.

However, Aristotle Analysis tool is only based on SunOS 5.6 (Solaris 2.6) or higher and Perl & designers do not plan to port it to other platforms. Further, it does not explicitly have Java exception call graph analysis & visualization. On the contrary, JexVis & Jex are built using Java and is therefore portable to different platforms, as long as the platform supports java run time environment.

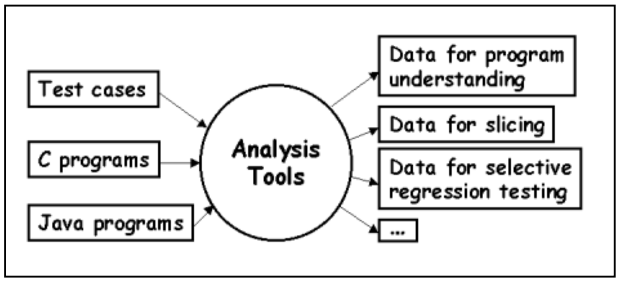

Test cases
C programs
Java programs
Analysis Tools
Data for program understanding
Data for slicing
Data for selective regression testing
...

Figure 6: The Aristotle Analysis System

Prawn [6] & Pathfinder [7] are some other interesting projects aimed at analysis of Java source codes & package structures but neither addresses the exception structure effectively. Grove et al [5] describe some theoretical analysis & representations of exception flow in java with the help of Factored Control Flow Graph but does not address the issue of effective visualization of exception flow control.

3 JEXVIS DESIGN, FEATURES AND LIMITATIONS

This section discusses JexVis & its functionalities in detail. Section 3.1 deals with JexVis architecture & design from a software designer's point of view. Section 3.2 discusses the various steps required to actually analyze & debug a software system through JexVis. Section 3.3 discusses some visualization tools & features designed into JexVis for effective analysis. Section 3.4 describes some limitations & bugs still present in JexVis. Finally, section 3.5 describes some sample scenarios where JexVis can prove to be useful to software designers & debuggers.

3.1 JexVis architecture

JexVis consists of several modules, each performing its own defined set of functions. The modules & their interdependencies are shown in figure 9. The main module is the GUI generator & user control module. It is through this module that the user interacts with JexVis, providing input source files to analyze, setting directories, setting visualization options & getting the output from JexVis. Figure 7 shows the diagram of this main interface module.
The main static analysis of the code is done by Jex but before it does so, a configuration file must be generated & provided to it which it would parse for input source directories, packages, output files & other options. This is done by the configuration file generator module in JexVis which generates the configuration file for Jex based on the inputs the user provides before the actual analysis begins. Jex itself uses an open source java compiler KJC [17] along with two other tools for analyzing the source code, gnu.getopt [16] which is a command line parser & ANTLR [15] (ANother Tool for Language Recognition) which is used for constructing recognizers, compilers, and translators from grammatical descriptions containing Java, C#, C++, or Python actions.

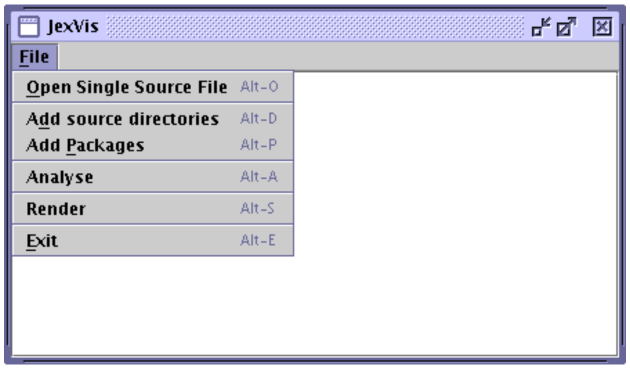


Figure 7: JexVis main user interface

The main module responsible for rendering the visualization is the JexVis renderer which itself has several menu options so that users can customize the rendering of the current tree. Figure 8 shows a sample screenshot. The menu at the top can be hidden by the user at his convenience and again restored by right clicking the renderer window. This module takes the help from the XML parser that actually parses the XML file generated by Jex to render the visualization. Several helper modules are used by this main renderer module so as to bring about effective visualization of the tree.
The entire rendering engine uses some inbuilt features, libraries & methods from the prefuse toolkit [11] & some custom action commands.

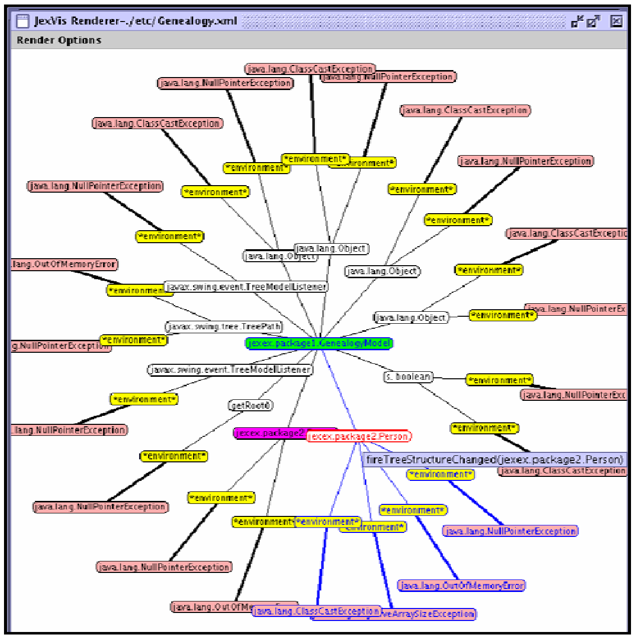

JexVis Renderer-./etc/Genealogy.xml
Render Options
java.lang.ClassCastException
java.lang.NullPointerException
environment
java.lang.Object
java.lang.OutOfMemoryError
javax.swing.event.TreeModelListener
javax.swing.tree.TreePath
jexex.package1.GenealogyModel
boolean
getRoot()
jexex.package2
jexex.package2.Person
fireTreeStructureChanged(jexex.package2.Person)

Figure 8: Rendering Module

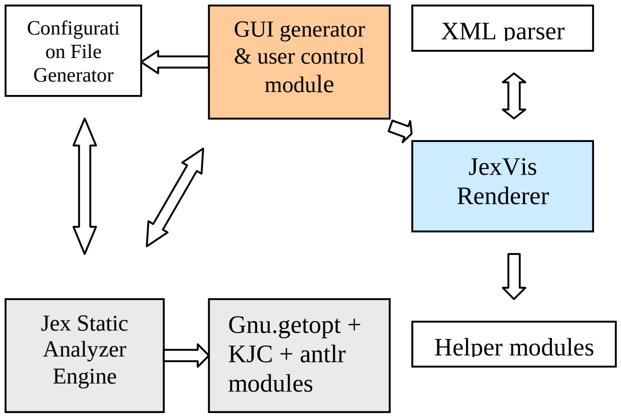
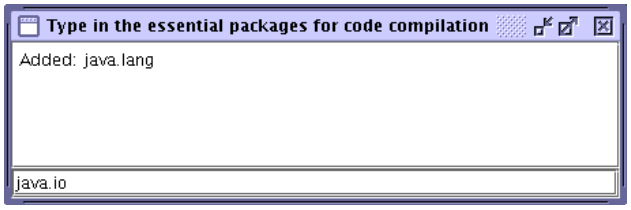

Configuration File Generator
GUI generator & user control module
XML parser
JexVis Renderer
Jex Static Analyzer Engine
Gnu.getopt + KJC + antlr modules
Helper modules

Figure 9: JexVis Design

3.2 Analysis using JexVis

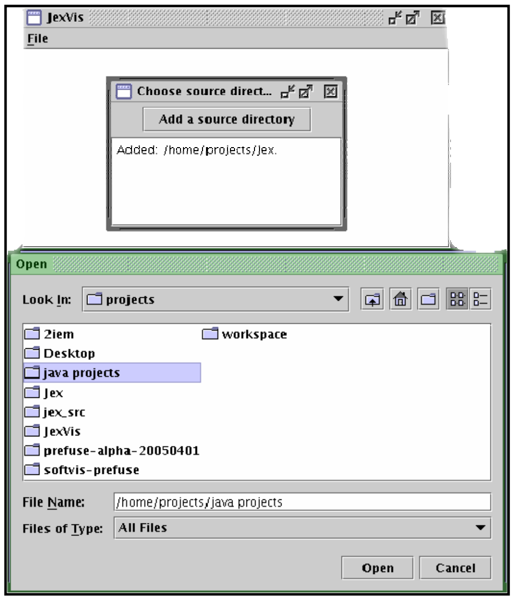
The analysis begins with collecting all the source .java & corresponding .class files of various modules into specific directories, putting the packages into proper sub directories & then firing up JexVis. As of now, JexVis exists as a single eclipse project. The author intends to distribute JexVis as a single java JAR file in future so that running JexVis would simply mean running the JAR file directly. The main user interface is shown in Figure 7. From the menu, one should first add all the packages necessary to compile the source codes. This can be done through a dialogue box shown in figure 10 where the user can type in specific Java packages & libraries. JexVis uses this information to generate the configuration file for Jex as previously described. Next, the user should input the source directories or single source files (if necessary, not mandatory) into JexVis by clicking the menu as shown in Figure 7. The corresponding diagram of this interface is shown in figure 11. When one selects "open single source file" from the menu, a usual common dialog box opens up allowing user to select an individual source file.



Added: java.lang

java.io

Figure 10: Dialogue box to input essential packages for the code analysis.



Added: /home/projects/Jex.

File Name: /home/projects/java projects

Figure 11: Interface to specify input source directories.

With all these information gathered from the user, JexVis generates the configuration file when the user selects "analysis" from the main menu. JexVis then calls Jex engine to perform actual analysis of source codes.
Rendering is performed by clicking "render" on the main menu as shown in figure 7. JexVis shows a list of available XML files corresponding to the input source classes for rendering as shown in Figure 12(a) & 12(b).
Clicking OK renders the corresponding XML file. A sample rendering window is shown in Figures 5 & 8.

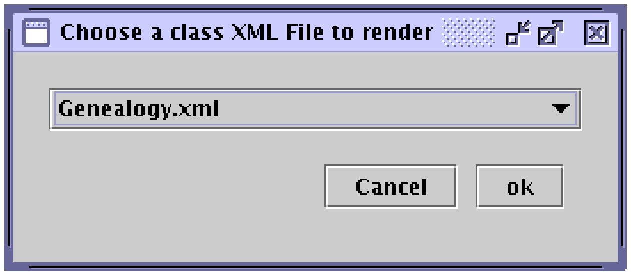


Figure 12(a): Dialogue box shown on clicking render in the main menu.

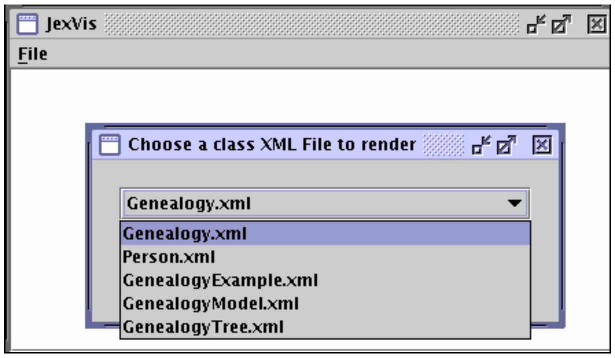


Figure 12(b): List of XML Files to render

3.3 JexVis features & strengths

JexVis incorporates some nifty tools to help programmers analyze the source code effectively. Some of these tools are inbuilt with rendering; some can be configured according to the user preferences from the rendering menu shown in Figure 16. These are discussed in detail in the subsequent sections.

3.3.1 Use of path highlighting

JexVis has a very cool feature by virtue of which whenever an user places mouse pointer on any node representing a method, immediately, the entire path (i.e., the edges) leading up to its children (which are the exceptions raised by this method directly or indirectly) gets highlighted. Along with it the edge connecting this node to its immediate parent also gets highlighted. This helps the user to easily understand the kind of exceptions raised by a method & the exception generation path. This feature is demonstrated in the Figure 13 where we can see the entire highlighted path from the node jexex.package2.person leading up to its children. Also, the edge connecting this node to its immediate parent jexex.package1.GenealogyModel also gets highlighted. We can immediately realize that the jexex.package1.GenealogyModel creates an instance of jexex.package2.Person class, thereby calling its constructor, which in turn generates four different types of exceptions each raised by the Java runtime environment.

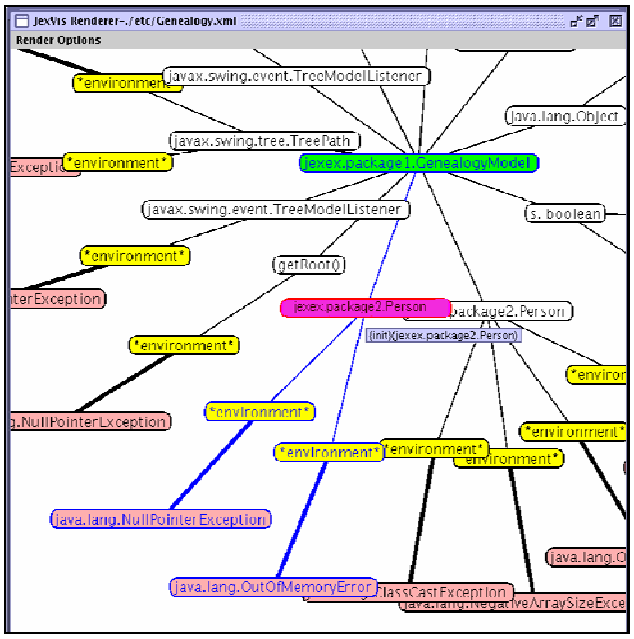

JexVis Renderer-./etc/Genealogy.xml
Render Options
environment
javax.swing.event.TreeModelListener
java.lang.Object
javax.swing.tree.TreePath
jexex.package1.GenealogyModel
getRoot()
s. boolean
jexex.package2.Person
(init)(jexex.package2.Person)
java.lang.NullPointerException
java.lang.OutOfMemoryError
ClassCastException
java.lang.NegativeArraySizeExce

Figure 13: Path Highlight leading to an exception node.

3.3.2 Zooming

Ability to zoom-in to a particular section of the graph & zoom out to get an overall context is a feature built-in with JexVis. Zoom in can be done by pressing the right mouse button and dragging it upward. Similarly, zoom out can be done by pressing the right mouse button & dragging it downward. Figure 13 shows a portion of the graph zoomed in to show the relevant edges & nodes. Similarly Figure 14 shows a zoomed out view of the same graph.

3.3.3 Panning

JexVis also provides ability to pan & move the graph to a comfortable position to view the relevant edges & nodes. Figure 14 shows the graph of Figure 13 zoomed out & panned in to a corner of the window.

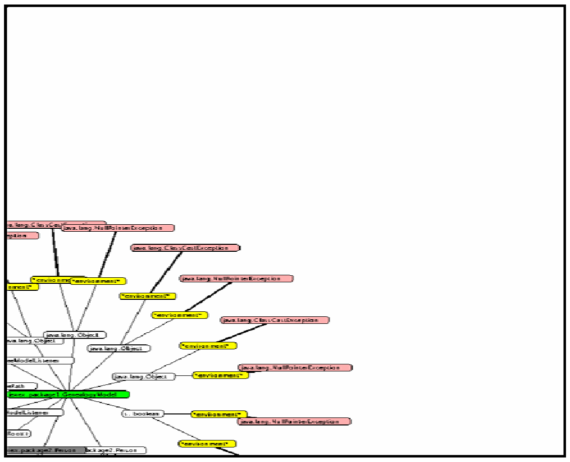


Figure 14: Pan & zoom out in action.

3.3.4 DOI & force directed layout

JexVis by default uses force directed layout design for graph drawing. This means that, the nodes automatically align themselves at an optimal distance from each other so that nodes & edges are clearly legible with no zoom-in or zoom-out in effect. JexVis uses degree of interest calculation [10] to put nodes that have DOI below a threshold outside the viewable region. To overcome occlusion, one can drag out a node out of a region by holding down the left mouse button and dragging it out. This is shown in Figure 15 where the node representing the method getRoot() has been dragged out.
When the node is released, the tree automatically adjusts itself to its previous position on virtue of its predefined forces between the nodes. It is clear from the figure that the highlighted blue edges still show the relationship between the node being dragged, its parent & the exceptions it generates.

3.3.5 Window border enforcement

Another cool feature in JexVis is the ability to enforce the graph to stay within the window boundary, thus ignoring the DOI conditions. All the nodes of the graph remain visible within the window. This is particularly useful for small graphs or when force directed layout is disabled so that the user has control over all the nodes & can manually manipulate their positions. This enforcement can be activated by selecting “Enforce Border” from the render options menu in the render window as shown in figure 16.

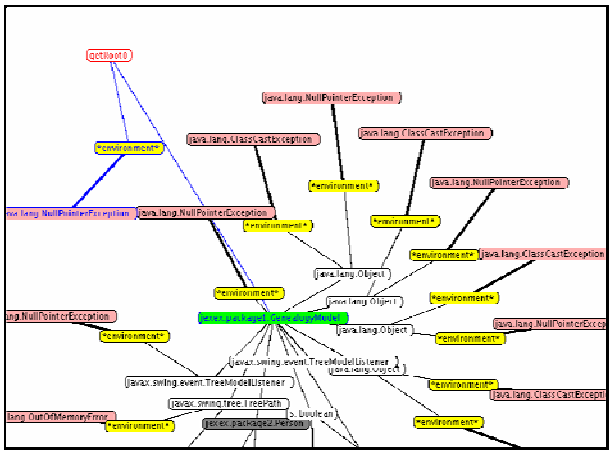

getRoot()
java.lang.NullPointerException
java.lang.ClassCastException
environment
java.lang.Object
jexex.package1.GenealogyModel
javax.swing.event.TreeModelListener
javax.swing.tree.TreePath
s. boolean
java.lang.OutOfMemoryError
jexex.package2.Person

Figure 15: Dragging a node out

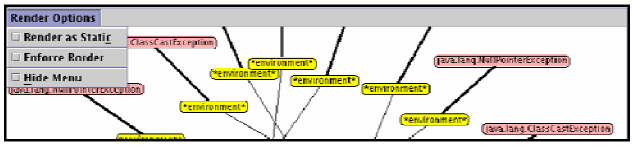


Figure 16: Render Menu

3.3.6 Static & dynamic rendering

Static rendering disables force directed layout. All the nodes & edges become fixed in the window & do not move on their own until user drags them to a different position. This manual control is very useful in many occasions when the user does not want the graph to reposition itself but instead needs manual control. This feature can be selected by selecting "Render as static" from the render options menu from the JexVis renderer window shown in Figure 16.

3.3.7 Use of color

All exception nodes are colored pink, environment nodes are colored yellow, parent node (class node) is colored green, constructor nodes are colored in magenta & the currently highlighted node is colored red. All other nodes are white. This helps the user to clearly classify different kinds of nodes & their relative importance. This of color also creates a vivid tree-diagram on the render window.

3.3.8 Use of tool tip

JexVis has a nifty feature whereby the full method name corresponding to a node is displayed as a tool tip when the user hovers the mouse over the node. For example, in Figure 17, a node is only partially visible in the display window since the display is heavily magnified. Thus, in normal course the node's method name is not visible. However, when the user hovers the mouse over the node, the full name of the method appears as tool tip text & at the same time the node gets highlighted in red as shown in the figure.

Tool tips are very useful in another context. If the fully qualified name corresponding to a node has a text width larger than a predefined value (set at 200 character-points), the names are automatically truncated so that the nodes do not cause occlusion. However, this truncation has some weird consequences and may cause confusion as to the name of the class & the method of the corresponding node. Tool tips come in rescue. When the user hovers the mouse on one of such nodes, fully qualified names of the corresponding methods are shown in tool tips. This is exemplified in Figure 18 where we can see that the node in focus just shows the method arguments in the node text & not the fully qualified method name.

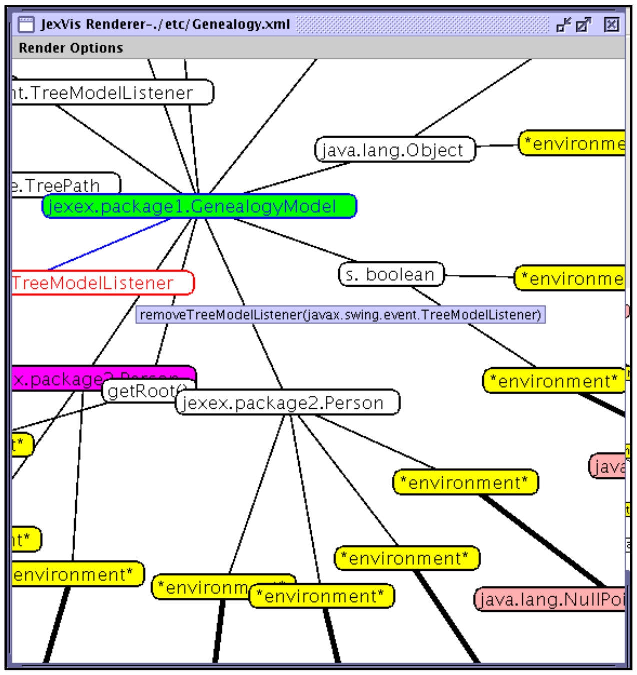


Figure 17: Tool tip text

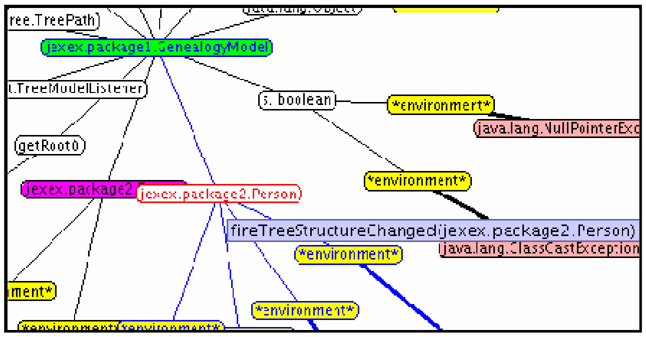


Figure 18: Tool Tip Text comes handy when displaying full method names.

Without the tool tip, it would virtually be impossible to make any guesses about the actual name of the method. Fortunately however the corresponding tool tip provides complete information about the method & its class.

3.3.9 Juxtaposed rendering

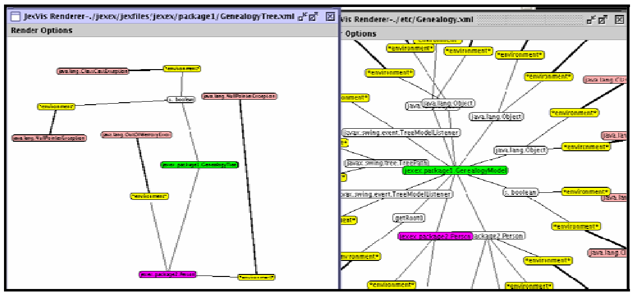


Figure 19: Juxtaposed Rendering

Another important feature of JexVis is its ability to show multiple rendering side by side. The user can fire up more than one instance of the renderer, each showing separate call graphs in different windows, each independent of each other. Thus, programmers can quickly view the call graph structure of some other class and compare it with another. The XML file names are shown on the title bar for the ease of the observer.

3.4 JexVis limitations & bugs

As is the case with all software tools, JexVis has its limitations too. Some of the limitations of JexVis are directly related to the limitations of Jex & the Prefuse toolkit. The most important drawbacks of JexVis are:

1. JexVis does not scale very well with large projects having their own packages & libraries. This is directly attributed to the fact that Jex itself has several issues with CLASSPATH settings & we need to modify the source codes to reflect the package hierarchy relative to their parent directory. This we found had to be done manually and this becomes a very big issue with a large project that has several multiple files & packages. With smaller projects though, Jex & JexVis performs relatively satisfactorily as is discussed in the evaluation section.
2. Another drawback of JexVis is that it can not handle very large graphs within the viewable window effectively. To address this issue, the author provides several options to the user so that he can tweak the displayable region to his convenience. However, much work is needed to be done to make it really user friendly & convenient.
3. If a method is encountered more than once during parsing & analysis stages, Jex writes the node corresponding to that method more than once in the XML file. This creates multiple nodes for the same method in the rendered graph. The author found however that this drawback has some side benefits as well. It helps to maintain symmetry & balance in the graph which otherwise would become too congested and/or unbalanced. Having a symmetry & balance helps positively in the cognitive process, penalty for this being the addition of extraneous nodes in the graph. This addition can cause real problems when a single class has too many method calls in it. Removal of redundant nodes would require us either to tweak the hash tables maintained by Jex or maintain a hash table of our own to keep track of the nodes already visited & check against this hash table every time we create a new node in the XML file. Unfortunately, due to time constraints, the author could not incorporate this in his code.
4. Another critical problem the author discovered is the problem related to node names when the fully qualified method name has characters greater than a threshold. The "*TextItemRenderer*" class in Prefuse tends to abbreviate the name in some weird way that it creates confusion to the user. The author could not eliminate this bug, though a work around to this was designed through use of tool tips as discussed earlier.
5. Due to limited time available for this project, exhaustive debugging & testing of all components of JexVis could not be performed. Thus, JexVis might not perform as is expected to for all different kinds of source codes & java projects. If we could get more time we could have performed an exhaustive analysis & checking of JexVis to ensure it gives consistent performance with all different kinds of source codes.

3.5 JexVis application scenario

JexVis can be effectively applied to analyze portions of large projects, taking modules & sub systems separately to investigate for possible effective & flawless use of exception handlers. It can also be applied for code maintenance, upgradation & debugging. Its graphical interface is easier to use than the original Jex utility which should make it a popular tool in the hands of software programmers, project managers, testers & software support engineers.

4 EVALUATION & SCALABILITY ISSUES

JexVis failed miserably in the context of very complex Java projects like the Vibes project [19]. The author tried hard to get some results visualized using JexVis but due to the underlying weakness of Jex related to CLASSPATH settings, JexVis always failed at the point where it called Jex engine to analyze the code. The failure often always was Jex's inability to recognize source classes & the classpath confusion. We take this failure as the choice of a wrong backend to analyze the code. Probably, either Jex needs to be modified to make it more scalable, or we may need to look into finding an alternative open source java static analyzer which at the moment of writing this paper could not be found.
The author evaluated JexVis in the context of Genealogy graph example having a very small set of java classes, provided in the Sun java swing tutorial website [18]. We found that in this smaller example, JexVis scaled well enough. All the figures shown in this paper are from the same example except figure 6 & figure 20. The author also tested JexVis against analyzing portions of large inbuilt java libraries from Sun Java SDK. Figure 6 shows the analysis results from JexVis for one of those classes, the java.io.FileOutputStream. Since Jex analyses every sub classes separately & generates corresponding .XML file, each of the rendered graphs scaled well, since each of them dealt with the

individual classes separately. The author also took another relatively moderate sized Java project on “Minesweeper” game developed by him to evaluate JexVis. It has about 8 source classes and 15 user-defined packages, including some JDK packages. A sample screenshot is shown in Figure 20. JexVis did not perform too badly though the weakness issues still remained areas that required more work.
Due to time constraints, more exhaustive testing could not be done.

5 LESSONS LEARNT

The author learnt valuable lessons during the course of the project. They are listed chronologically below.

1. It’s really difficult to design a good static analyzer. The initial target of the author was to design a static analyzer of his own for his specific needs, but later it was realized that this would really make the project too big to be completed within stipulated deadlines. Even when he found one, its weaknesses made the overall performance of JexVis very poor.
2. Coming up with good information visualization ideas to effectively portray data across to the end user is really challenging & requires lot of experience & thinking. The author feels that to some extent he really ran out of good ideas for creating an effective visualization.
3. Choosing an effective & useful layout scheme for a large graph is a complex problem.
4. Trying to provide more useful information to the user without causing clutter and occlusion is tough. It takes time & lot of thinking to build effective filters to address this issue.
5. Zooming may not be the ideal solution for displaying more information of a small region. It can cause more problems than simplifying them. Deciding alternative approaches to zoom is a difficult problem.
6. Scalability is an important issue for every information visualization system. Designing scalable system for large projects requires lot of planning, experience & ideas.
7. Use of specific toolkit can stifle creativity because one tends to think in lines of the features & interfaces provided by the toolkit which may not be suitable for all different types of visualization problems. However, toolkits can helps in rapid coding & design when time is scarce.

6 CHALLENGES FACED & LINGERING ISSUES

The main challenge faced by the author was modifying Jex & recompiling it so as to produce an output in conformation with the XML file syntax & rules. Precise information related to the functioning of Jex was not available & much time was spent reading the source codes, trying to understand how Jex worked before its code could be modified. When that was figured out, next challenge was to make Jex compile in eclipse after setting the environment variables. Another challenge was to create a proper user friendly interface for the users so that appropriate configuration files could be generated & Jex could be called directly from within the main JexVis application. A major issue with Jex is still its environment variable settings & appropriate configuration script. One strange problem faced by the author was that java.lang was not found by default by Jex in spite of setting the classpath variable in Linux as well as automatic classpath settings done by eclipse. Therefore, java.lang (rc.jar) had to be manually incorporated into eclipse project environment. This issue still remains a major problem that has to be solved before JexVis becomes fully portable.
Another challenge the author faced was to address the scalability issue & in making the rendering effective & meaningful. Several options were thought but unfortunately there was strict time limitations, therefore some of these ideas could not be actually implemented. These are discussed further in future directions section. However, the author still believes that JexVis with its currently built-in features will still prove to be useful to some degree for software analysis.

7 FUTURE DIRECTIONS

JexVis provides basic functionalities for visualizing exception call graphs. Much work is needed in this direction to make it commercially useful & popular. Some of these are discussed in the subsequent subsections.

7.1 Removal of repeated nodes

As discussed earlier, repeated nodes are a consequence of old Jex behavior. This needs to be corrected so that nodes corresponding to a unique method are represented only once & not multiple times. This can be done by creating a hash table of those nodes already written to XML file & checking against that table every time a new node is being written into it.

7.2 Unintuitive node name truncation

As previously discussed, the default text node renderer used by Prefuse truncates long texts in such a way that it becomes confusing to the user. This default behaviour can be modified so as to provide more intelligible truncation that does not confuse analyzers. Due to time constraints we could not address this issue.

7.3 Visualization of package structure

Currently, package structure is shown by naming the nodes preceded by package names. This is not very good means to show package structure of the existing modules. A separate visualization tool can be designed that shows the package hierarchy along with & juxtaposed with the exception call graph.

7.4 Use of filters

Another effective way to show only certain specific nodes is to use filters to filter out unimportant, uninteresting nodes & edges. Users can be asked to enter the nodes they are interested in & only those nodes & their path to the leaves (the exception nodes) can be made visible leaving out the rest. This might also result in loss of context. So better still, the other nodes & edges can be displayed in light shades of gray so that they are faintly visible, keeping the user apprised of the context. This is particularly useful if the graph rendered has many nodes & edges.

7.5 Enhanced tool tip

The tool tip text provided by JexVis shows only the basic information about a particular node. Tool tips can be enhanced showing additional information about the class & its members and/or other attributes of the node in question. There is in effect no limit to the amount of text that can be displayed in a tool tip. For example, eclipse uses tool tips to provide detail information about classes & methods, even shows sample example from java documentation on their use. The same features can be added to JexVis.

7.6 Use of a wizard style dialogue to easy users

Currently, JexVis uses menus for various user inputs. It would be so much better if there were a wizard style dialogue that would guide users through the various stages of the analysis process.

7.7 Use of appropriate color scheme

The color scheme the author used may not be the ideal color scheme, especially people with disabilities. Probably more work is needed to find out a more suitable color scheme.

7.8 Exhaustive testing with real life project codes

The author did not get enough time to test his implementation exhaustively on a real life big enough project source codes. This was partly because Jex did not prove to be scalable enough. Exhaustive testing & debugging on huge dataset is necessary to make this tool come to of any real use. This might require us to redesign some Jex modules so that it becomes more portable.

8 CONCLUSION

In this paper, the author discussed JexVis, a static, offline visualization system that visualizes exception call graphs in Java. By using an interactive graphical visualization system, the programmers will be better equipped to analyze, maintain & debug large projects developed in Java. The front end was developed using Java Swing with the use of Prefuse toolkit taking advantage of many information visualization techniques. The author believes that through use of this tool, much of the problems faced by programmers dealing with complex java projects could be easily solved though much work is still needed on JexVis to make it really useful.

9 ACKNOWLEDGEMENTS

The author is really grateful to Martin Robillard who provided useful information about Jex & its source code. The author also drew inspiration from the 2004 fall term-1 information visualization course project "NAV" by Meghan & Peter for writing this paper & from the Prawn paper [6] for giving final shape to the project. I am also grateful to Shawn Minto for giving me suggestion on the static analysis part. Finally, the author is grateful to the instructor, Tamara Munzner providing critical (& stern) comments after the update presentations & otherwise that helped him to address some of the important issues & give him directions in his work during the project.

REFERENCES

[1] Analyzing Exception Flow in Java Programs, Martin P. Robillard and Gail C. Murphy, Proceedings of the 7th European Software Engineering Conference and 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering; Toulouse, France; 6--10 September 1999
[2] Designing Robust Java Programs with Exceptions, Martin P. Robillard and Gail C. Murphy, Proceedings of the ACM SIGSOFT Eighth International Symposium on the Foundations of Software Engineering (FSE-8): Foundations of Software Engineering for Twenty-First Century Applications (San Diego, California, USA; 8--10 November 2000; SIGSOFT '00), ACM Press, pp. 2--10, 2000.
[3] Analyzing Exception Flow in Java Programs, M.Sc. thesis, Martin P. Robillard, UBC, September 1999, http://www.cs.ubc.ca/labs/se/theses/robillard99-msc.html.
[4] Regaining control of exception handling, Martin Robillard, Gail Murphy, Technical Report Number TR-99-14, UBC, December 1st 1999.
[5] Efficient and Precise Modeling of Exceptions for the analysis of Java Programs, David Grove et al, ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software tools & Engineering (PASTE 1999).
[6] Prawn: An interactive Tool for Software Visualization, Andrew Chan & Reid Holmes, Information visualization course project, UBC, spring 2003.
[7] Pathfinder: Exposing the mental map of program navigation, Mik Kersten (beatmik-at-acm.org), Information visualization course project, UBC, March 1 2004.
[8] J. Wu and M.-A.D. Storey. A multi-perspective software visualization environment. In Proceedings of CASCON'2000, pages 41-50, 2000.
[9] S. Tilley H. Mueller, M. Orgun and J. Uhl. A reverse engineering approach to subsystem structure identification. Journal of Software Maintenance: Research and Practice, 5(4):181-204, 1993.
[10] Degree-of-Interest Trees: A Component of an Attention-Reactive User Interface, Card, S.K. and D. Nation, Advanced Visual Interfaces. 2002.
[11] Prefuse: a toolkit for interactive information visualization, Proceedings of the SIGCHI conference on Human factors in computing systems table of contents Portland, Oregon, USA, Pages: 421 – 430.
[12] Jex, a Tool for Analyzing Exception Flow in Java Programs, http://www.cs.ubc.ca/~mrobilla/jex/
[13] The Aristotle Research Group, Program Analysis based software Engineering, http://www.cc.gatech.edu/aristotle/
[14] The Eclipse SDK, http://www.eclipse.org/
[15] ANTLR, http://www.antlr.org/
[16] GNU GETOPT, http://www.urbanophile.com/arenn/hacking/getopt/gnu.getopt.Getopt.html
[17] KJC, http://www.dms.at/kopi
[18] Java Tutorial on use of swing components, http://java.sun.com/docs/books/tutorial/uiswing/components/
[19] VIBES (Variational Inference for Bayesian Networks), http://vibes.sourceforge.net/

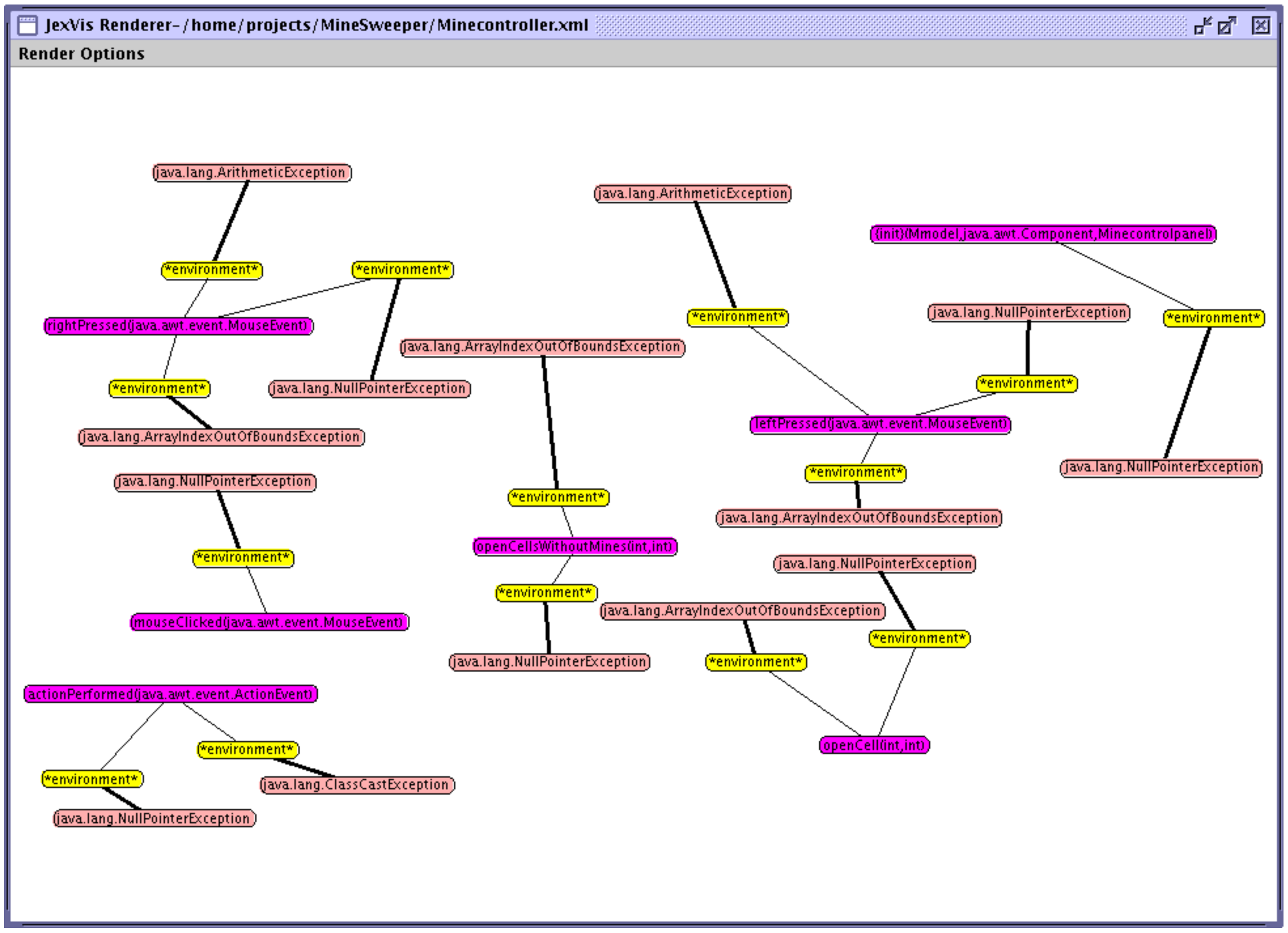

JexVis Renderer- /home/projects/MineSweeper/Minecontroller.xml
Render Options
java.lang.ArithmeticException
environment
rightPressed(java.awt.event.MouseEvent)
java.lang.ArrayIndexOutOfBoundsException
java.lang.NullPointerException
mouseClicked(java.awt.event.MouseEvent)
actionPerformed(java.awt.event.ActionEvent)
java.lang.ClassCastException
openCellsWithoutMines(int,int)
leftPressed(java.awt.event.MouseEvent)
(init)(Mmodel,java.awt.Component,Minecontrolpanel)
openCell(int,int)

Figure 20: Sample screen shot of JexVis renderer on a MineSweeper game module.