A Case Study in Software Adaptation

August 1st, 2009

http://www.cc.gatech.edu/morale/tools/isvis/2009

Andrew Bernard
College of Computing, Georgia Tech
abernard@gatech.edu

***Abstract*—The objective of this study is to examine the challenges that come with adapting legacy software. We will discuss the software adaptation of a 13 year old system called ISVis. ISVis supports the browsing and analysis of program execution scenarios. It is written in C++ and comprises 48 source and header files defining 57 classes totaling around 18 KLOC. Our adaptation process involved analysis of operating system calls, C++ language standards, third-party library dependence, and build environments. The project yielded results and experiences that are applicable to a variety of topics including software migration difficulties, upgrading and debugging legacy code, and regression testing GUI's.**

***Index Terms*—software migration, software adaptation, legacy systems, refactoring**

I. Introduction

SOFTWARE products are constantly evolving in order to adapt to emerging technologies. Unfortunately, many legacy systems were not designed with this in mind. Software engineering researchers, in an effort to create long lasting and maintainable software, have explored a variety of system configurations and architectures to ease the process of upgrading. The purpose of this study is to explore some of the issues relevant to modifying a legacy system in order to adapt it to a changed environment as well as to get a general understanding of the optimal characteristics of software designed to live through changes in technologies.

The legacy program that was the target of our adaptation study is the **I**nteraction **S**cenario **Vis**ualizer, or ISVis for short. ISVis was originally released in 1996 [1] and has not been updated since that time. It is an application created to support the browsing and analysis of program executions. Through a set of visualization techniques, ISVis provides users with a depiction of large amounts of information to understand a program's behavior through the discovery of interaction patterns and recurring scenarios. Section II describes the program's functionality in further detail.

ISVis was originally developed in an older (pre-standard) version of C/C++ using third-party libraries. The user interface portion was written using the Motif toolkit and its operating system was Solaris [2]. Since then, the program has become dated with the standardization of C++, upgrades to the third party libraries it depends on, and the need for portability to different platforms. The process of updating ISVis was started in Fall of 2006. This paper discusses the progress that has been made since that time. We will go into detail about the difficulties we faced in Section III and then describe our approach to solving each issue in Section IV. The following is a brief overview of the main issues that were encountered during the adaptation process:

1) OS Dependence: As was mentioned above, ISVis was originally developed to run solely on a Solaris machine (SUN Solaris 2.5). Our short term goal was to get it to run on a more recent version of Solaris. A long term goal is to enable it to run on a variety of platforms (Linux, Mac, Windows).

2) Language: We needed to update the pre-ANSI standard version of C++ to use the current standard version of the language.

3) Third Party Dependence: ISVis originally used a third party library called RogueWave [3] to handle persistence and collection classes. This dependence needed to be replaced since the version of the library being used was no longer supported.

4) Build Environment: The Makefile used to build the program had to be adapted to coincide with the other modifications being made.

5) GUI: ISVis was dependent on out of date versions of Motif (1.2) and X Windows (X11R4) for its GUI. Adaptations were made to bring these up to date.

II. ISVis

[Section II is copied from [4] with minor rewording and organization]

A. Purpose

ISVis aids in understanding the architecture of a software system. It does this by helping the analyst cope with the abundance of detail inherent in large systems. In particular, it provides two key features for managing detail: graphical visualization techniques and support for analyst-defined abstractions.

ISVis provides an analyst with a process and a tool within which a program's behavior can be visualized, filtered, and abstracted and with which the analyst can build and save views of the behavior appropriate for the particular program understanding task. Using these visualizations analysts can detect recurring patterns of component interactions, manifested as repeated sequences of program events such as function calls, object creation, and task initiation. Instances of these interaction patterns occur at various levels of abstraction. Using these abstractions, the analyst can help bridge the gap between low-level execution events and architectural models of program behavior.

B. Overview

1) Process Flow: The overall process of performing architectural localization using ISVis is depicted in Figure 1.

It comprises a static analysis of the subject system, instrumentation of that system to track interesting events, execution of the instrumented system in particular usage scenarios to generate event traces, and visualization and abstraction of the event traces using the ISVis tool. The visualization and abstraction steps are repeated until the analyst is satisfied that a high-level understanding of the relevant parts of the system have been obtained. In particular, the analyst defines high level components in terms of constituent actors and designates interaction patterns denoting recurring scenarios. The goal is to provide support for the process whereby the analyst formulates a high-level understanding of a system from its voluminous details.

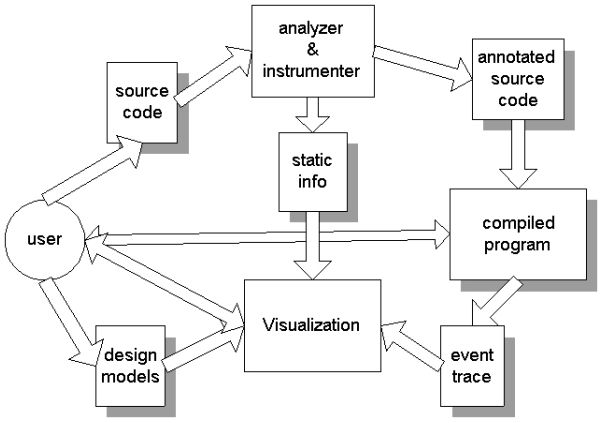

Fig. 1. ISVis Process Flow

2) Architecture: Figure 2 provides an architectural description of ISVis, including its components, connectors, and input-output files. The Static Analyzer reads the Source Browser database files produced by Solaris C and C++ compilers and generates a static information file. The Instrumentor takes the source code, the static-information file, and information supplied by the analyst about what actors to instrument (specified in the trace information file), and generates instrumented source code. This source must be compiled externally to the ISVis tool, and then, when the instrumented system is executed using relevant test data, event traces are generated. The Trace Analyzer in ISVis uses the trace information files and the Event Stream to read event traces and convert them into scenarios, stored in the Program Model. As scenarios are created, the actors involved are also added to the Program Model. The user then interacts with the Views of the Program Model to do the analysis. A Program Model can be stored for later use in a session file. For the version of ISVis being adapted, actors consist of C/C++ functions, methods, classes and files; events comprise calls and returns between functions and methods. However, the algorithms that ISVis employs are independent of the specific nature of the actors and events.

3) GUI: ISVis provides the analyst two views, the Main View shown in Figure 3 and the Scenario View shown in Figure 4. The top portion of the Main View lists the actors in the Program Model, including user-defined components, files, classes, and functions. The middle portion includes lists of the scenarios and interactions in the Program Model, as well as an area for displaying information about the currently selected item. The key area allows users to assign colors to actors or interactions. The bottom portion of the view is a text window for user interaction.

The Scenario View provides several features to help an analyst build abstract models of the subject system and to localize behavior. An option menu allows the actors in the scenario to be grouped by containing file, class, or component. Another option allows the user to select a class of interactions or just a single instance of an interaction. Once a sequence of interactions are selected, they can be defined as a scenario, and then all occurrences of that scenario in the original event trace are replaced with a reference to the newly defined scenario. While a simple interaction is shown as a line connecting the source and destination actors, an analyst-defined scenario that occurs within the Scenario View appears graphically as a thin, horizontal rectangle containing all of the actors involved in the scenario.

C. Usage

The focus of this paper is the adaptation of ISVis, but the reader should refer to [4] for a more detailed discussion on ISVis's capabilities, including a case study depicting its usage.

D. Code

ISVis is written in C++ and comprises 48 source and header files defining 57 classes. The code totaled 17,829 LOC at the start of this study and 18,177 in its present form.

[Insert a "class" diagram (use source files as classes)]

III. Adaptations

This section will go into the details about the main areas that were targeted during the adaptation process.

A. OS Dependence

ISVis was originally developed to run on SUN Solaris 2.5. Initially, a review of newer versions of Solaris was performed to identify differences. Although having the system running on Solaris computers was always a priority, it was important to take into consideration the fact that Linux and Windows based machines are now more popular, and therefore it was of value to transform ISVis to run on a variety of operating systems.

Additionally ISVis has some functionality that is platform dependent. For ISVis to perform a static analysis of a subject program, that program must have been compiled using either the Solaris CC or cc compiler, with the `-xsb` flag set to generate source browser information (static information used by Sun's Source Browser, a tool suite no longer available). The result of a static analysis of the subject program is a static information file describing the files, classes, and functions in the source code. ISVis relies on this file to extract the necessary static information about the subject program which is then visualized through the GUI. To completely break our dependence on Solaris we must find a replacement tool/script that will produce the same static information. Currently the static

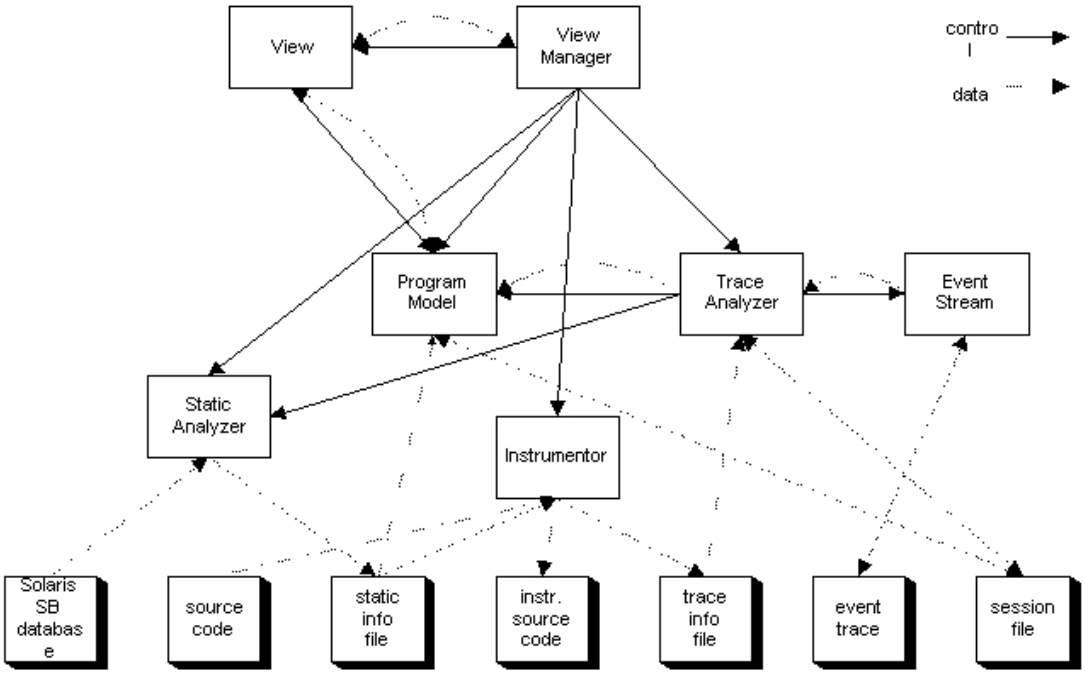

Fig. 2. ISVis Architecture

analysis, instrumentation, and trace generation are inoperable due to this deficiency and all ISVis execution is performed using legacy data.

B. Language

The original ISVis program was written in an older version of C/C++ and since then C++ has been standardized, creating differences between the older non-standard version and the new standardized version of C++ (see [5]). Because of these differences, ISVis would not compile using the current compilers that are ANSI C++ compliant, requiring the code to be upgraded. Table VIII (in Appendix F) shows a list of the errors that were encountered when the code was first compiled (using a modern compiler, gcc 4.3.2). This is not a complete list because the compilation stopped on some files due to the high number of errors found. The most prominent errors were:

- initialization errors that were minor corrections, like leaving off the "`int`" when using an integer (40 cases)
- warnings about string literals converted to char in initialization (55 cases)
- errors because of the use of "const" (9 cases)

Additionally, we also needed to examine the function calls ISVis was making to ensure their compatibility with the C++ standard. These calls are listed in Table I along with how many times they are used in ISVis.

TABLE I
FUNCTION CALLS

Function Call	Count
abs	2
assert	1
atoi	6
fabs	2
floor	1
gettimeofday	4
longjmp	4
printf	8
signal	6
sprintf	16
strcpy	5
strlen	27
strncpy	6

C. Third Party Dependence

1) Persistence: ISVis provides its user the ability to store an analysis session for later use. RogueWave, a third party library, was used to provide this ability. The version of RogueWave that was used when ISVis was first created was no longer available so that left us with the choice to either update it or replace it with something else. RogueWave is now an expensive enterprise-level tool so we decided it would be easier to try and replace it with calls to the C++ Standard Template Library (STL) [6] wherever possible. Also, it is generally better to use a standard library than an external one, making STL the most natural alternative to RogueWave. We conducted an analysis of RogueWave's use in the application and produced the results shown in Table II. A major issue was that there was no STL alternative to much of the persistence functionality that is available in RogueWave.

2) Containers: In addition to persistence, ISVis also relied on RogueWave for some other data types. The `RWCString` is used extensively throughout the whole ISVis program most

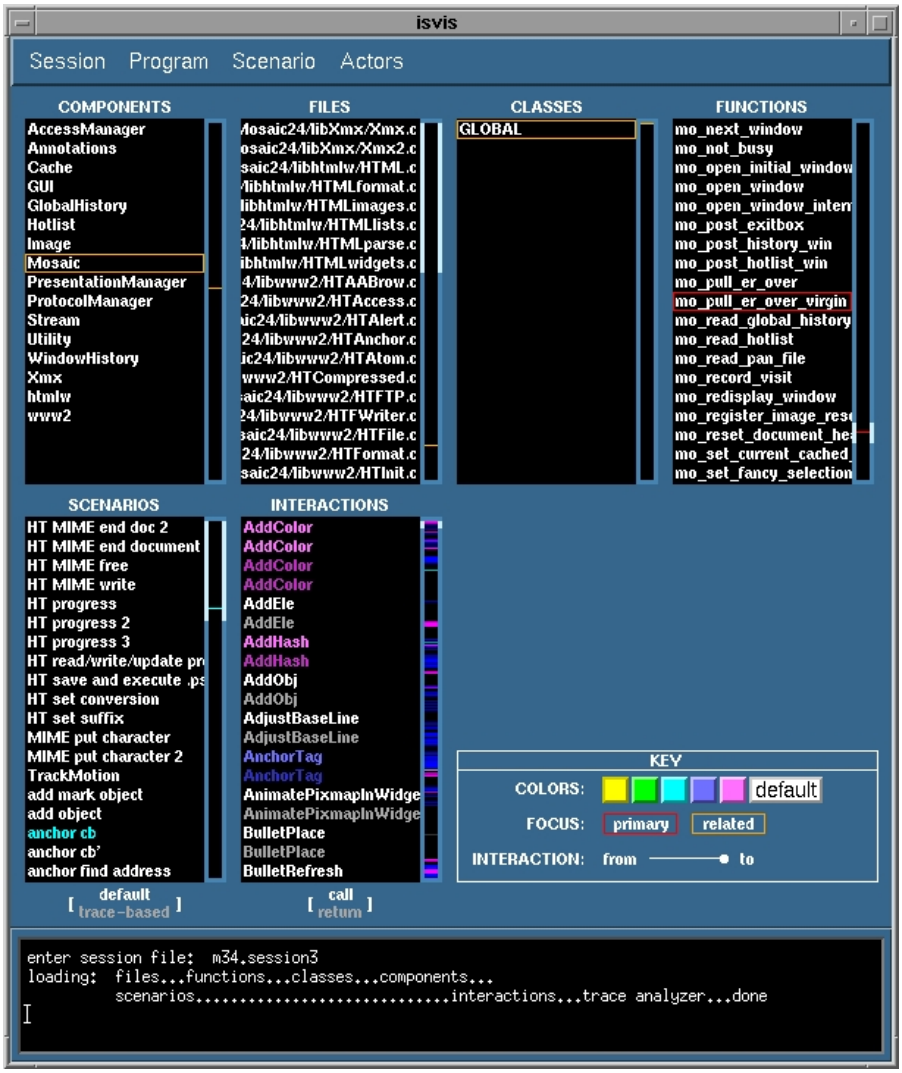

Fig. 3. ISVis Main View

likely because `std::string` was not available when the program was originally written and using `RWCString` was a better alternative to using `char` arrays. The functionality of `RWCString` is very similar to that of `std::string`, but there is some extra functionality that is not available in `std::string`. However, this additional functionality is not used within ISVis, so it was not a concern.

The other major dependency on the RogueWave library came in the use of container classes, notably the `RWTPtrHashTable` class. The notion of a hash table is not a built-in data type within C++ without using the STL. Whilst the `std::map` provides similar functionality to a hash table it is not clear what the performance implications would be of using `std::map` instead of a hash table. A possible alternative is to use `hash_map`, which is a common STL extension although, at the time of this writing, it is not officially included as part of the STL. While `hash_map` is included in most compiler libraries as an extension, the location seems to be different for each. For example in gcc you need to include `<ext/hash_map.h>`, and `hash_map` is in the `__gnu_cxx` namespace, whereas for Microsoft Visual Studio you have to include `<hash_map.h>` and `hash_map` is in the `stdext` namespace. This could be overcome with some convoluted use of `#ifdef` and `typedef` statements.

D. Build Environment

The biggest change that needed to be made to the build environment had to do with the Makefile since the one that came with the original version of ISVis was overly complex. [We should inspect the original Makefile to see if there was also any platform dependence - write about it here if there is] We also needed to update it as it referenced libraries and file paths that were changing as a result of the adaptation process.

E. GUI

In this section we will discuss the adaptations that needed to occur for the GUI to function properly.

1) Motif: Being a visualization tool, ISVis heavily relies on its user interface. The interface was originally developed using Motif 1.2 [7]. In order to have a fully up-to-date system, it was necessary to update the Motif libraries to the latest stable

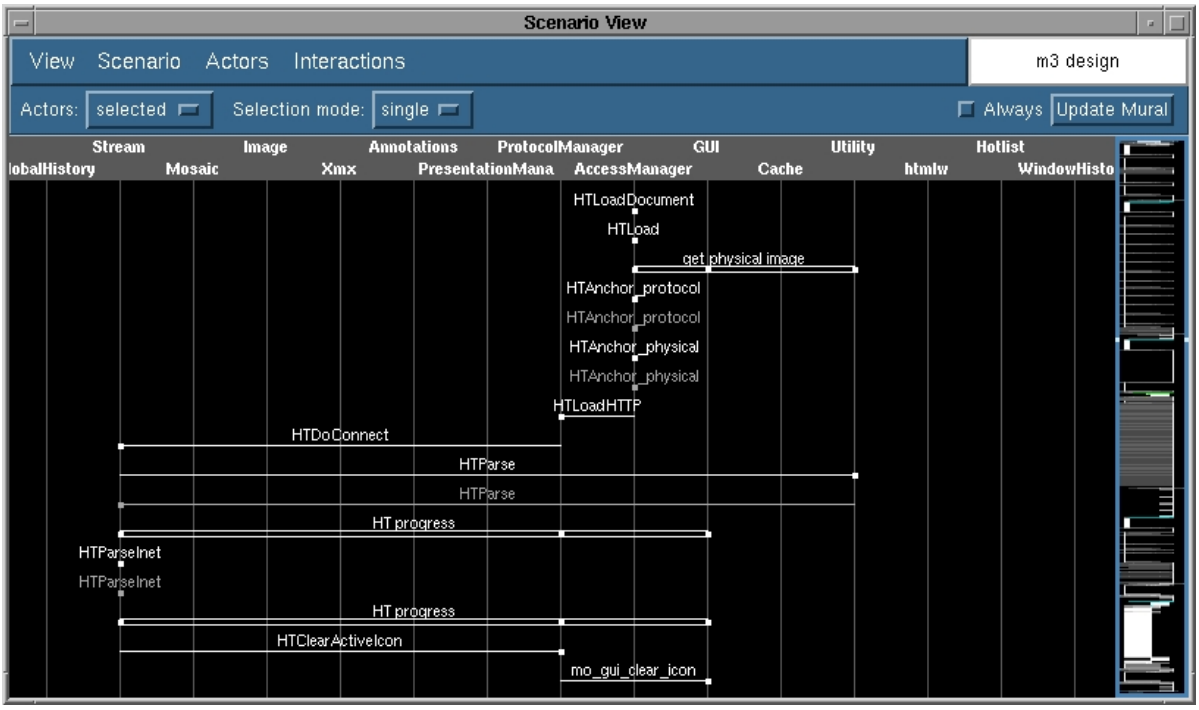

Fig. 4. ISVis Scenario View

TABLE II
ROGUEWAVE ANALYSIS

RogueWave Element	Count
`RWDEFINE_COLLECTABLE`: Macro for assigning a classID to a class	16
`RWCString`: String manipulation class	587
`RWBoolean`: Boolean wrapper	139
`RWCollectable`: Abstract class for collectable objects	85
`RWClassID`: ClassID wrapper	4
`RWspace`: return type for function that defines the number of bytes needed to store an object in a binary file	49
`RWvostream`: Virtual output stream	46
`RWvistream`: Virtual input stream	45
`TPtrHashMap`: Bespoke wrapper class that uses Rogue Wave hash tables	40
`RWTPtrSlist`: RogueWave singly linked list container	180
`RWTPtrHashSet`: HashTable where only one item of a given value will be accepted	8
`RWTPtrHashTable`: RogueWave hash table container	38
`RWTPtrHashDictionary`: container for storing pairs of hash keys and values	209
`RWTPtrDlist`: RogueWave doubly linked list container	43
`RWFile`: file encapsulation	13
`RWCTokenizer`: splits a string into tokens	18
`RWTPtrOrderedVector`: vector container class	20
`RWBitVec`: resizable bitfield	27

version 2.1. Starting in Motif 2.0, The Open Group cleaned up much of the Motif API. The public API (intended for application development) was left as is, but the private API (needed by widget writers, i.e. ISVis code) was heavily modified. Prior to the 2.0 release, there were 2 types of symbols: public (prefixed by `Xm`) and private (prefixed by `_Xm`). In the 2.0 release, a new symbol type `Xme` was introduced and most `_Xm` symbols were renamed to `Xme`. The ones left as `_Xm` are internal to the library. Several symbols (ex: `EditDone`, `EditError`, etc.) have been moved to the public header file `Xm/Xm.h` from their original location in the private header file `Xm/TestStrsoP.h`. These changes can cause problems in applications and libraries that use these symbols themselves that results in a namespace conflict at compile time. Table III is a list of the Motif function calls made in the ISVis application. These calls indicated a dependence on Motif forcing us to analyze each of the listed files when performing our adaptation.

[Elaborate on Table III giving a summary of impact on the source code]

2) X Windows: A deeper analysis of ISVis proved its interface to be incompatible with modern graphics hardware. Therefore it was necessary to update the interface code accordingly to begin ISVis's transformation into a fully cross-platform piece of software. This required some research into the X Windows library to better understand how and where it was being used in ISVis. Appendix E shows a list of all the X Windows dependencies that were found during the analysis.

Because of this device evolution, one of the biggest obstacles encountered when upgrading ISVis had to do with how colors were managed in X Windows and how they were used by ISVis. There was a part of the code that required an analyst to have knowledge about how to work with

TABLE III
MOTIF DEPENDENCIES

File	Motif Function/Macro/Class
actor_list.c	xmFormWidgetClass
io_shell.c	xmDrawingAreaWidgetClass
main_view.c	xmTextWidgetClass
mural.c	XmTextInsert
scenario_view.c	XmTextGetLastPosition
xapp.c	XmTextSetEditable, XmTextGetString, XmTextSetString, XmTextSetCursorPosition, xmPushButtonWidgetClass, xmSeparatorWidgetClass, xmCascadeButtonWidgetClass, xmSeparatorWidgetClass, XmDrawingAreaCallbackStruct, xmToggleButtonWidgetClass,XmStringFree, XmString, XmStringCreateSimple, XmStringGetLtoR, XmProcessTraversal, XmMenuPosition, XmCreateSimplePopupMenu, XmCreatePulldownMenu, XmCreatePopupMenu, XmCreateOptionMenu, XmCreateMenuBar

different display devices so that the manipulation of color was performed correctly. We had to gain that knowledge in order to successfully upgrade ISVis's code.

After ISVis was operational and we could see the main GUI screen we encountered a bug that was preventing the scenario view mural from drawing. The Mural is the portion of the GUI that displays a visualization of the actors being shown. It is a key feature to ISVis's use and so we deemed ISVis to still be "broken" until this bug was fixed.

IV. APPROACH

A. OS Dependence

ISVis only used one system call - the `exit()` method - which required no code change. Addressing the static analysis tool, we have looked at a number of alternatives ([8], [9], [10], [11]) to replace the outdated Solaris command, but we have not yet decided on or tested any of them. Whichever tool we decide on, the biggest factor to consider is whether or not it produces output that can easily be translated into the information file that ISVis relies on.

B. Language Versions

A significant amount of time and effort was invested in updating ISVis's source code to make it ANSI C++ compliant. During this upgrading process, several errors were introduced into the source code, which led to lengthy debugging sessions. For example, ANSI standards forbid the initialization of class variables in header files. This style rule was not common practice when ISVis was first developed; therefore there were numerous instances of variable initialization which needed to be moved from the header files into the source files. Other common discrepancies that needed to be fixed were related to the explicit use of namespaces required by the ANSI standard, as well as with the use of Strings and character arrays interchangeably. The process of fixing compiler errors of this type was tedious, which is likely the reason why the programmers involved overlooked a few occurrences and introduced faults into the system by removing initializations from a header file and not replacing them in the corresponding source file.

At the end of the language migration, all compilation errors regarding the differences between the original and the ANSI versions of C++ were fixed yielding a fully upgraded code base compliant with the ANSI C++ standards. There are still some compilation warnings regarding `char` strings. Table IV shows a summary of the number of changes that had to be made to reach a clean compile (i.e. compilation with no warnings or errors). The totals are grouped by the categories the changes applied to. For each category, needed changes where also classified as critical and non-critical, depending on their importance. This upgrade will prevent ISVis from facing issues regarding incompatibility with C++ compilers similar to those resolved during this study, as it is likely compilers for standard versions of the C++ programming languages will be available. ISVis is currently operational with ISO/IEC 14882:2003 C++ [12] compiled under gcc 4.3.2.

C. Third Party Dependence

1) Persistence: Interaction visualizations are built from large trace files which take some time loading. After trace files are loaded the user can manipulate the interaction information through the GUI to create different scenarios and identify patterns. It is important for the user not only to be able to rapidly reload the information contained in a previously read trace file, but also to have later access to the various scenarios created during a particular program run.

Originally, ISVis used RogueWave serialization libraries to load and save system state in binary format. When looking into replacing RogueWave, two main options were considered: the development of a custom serialization system and the use of an existing library. There was some skepticism about using a third party serialization system, because it could potentially become unavailable and cause the same problems RogueWave was causing. A custom serialization system, on the other hand, would be lengthy and complex to develop.

After some research on the available options, a decision was made to use the Boost serialization libraries [13]. Boost provides free portable C++ libraries; several of them standardized by the C++ Standards Committee. The set of libraries provided by Boost were appealing for two main reasons. First, Boost is amongst the most popular C++ libraries, largely supported by both its developers and the C++ community. It is also reviewed and freely distributed, characteristics that reduce the possibility of facing similar issues as those resulting from the unavailability of RogueWave. Second, Boost's Serialization

TABLE IV
SUMMARY OF CHANGES TO REACH CLEAN COMPILATION

Category	Critical	Non-Critical	Totals
Makefile	6	0	6
Motif	4	6	10
C++	78	92	170
RogueWave	188	102	290
Totals	276	200	476

libraries have a similar architecture to that of RogueWave's serialization. This proved to be an important characteristic, because it facilitated the migration from RogueWave to Boost, while leaving unchanged the original architecture of ISVis.

Not only was it important to select the optimal serialization mechanism, but it was also crucial to select the correct data file format for saved sessions. Originally, saved runs were stored using binary data files. Although this was effective and efficient, saving binary data posed an obstacle in making ISVis platform independent. Several alternatives where considered when selecting the optimal file format for the new ISVis serialization; binary, plain text, XML and binary XML where the main options considered. After careful consideration it was decided that XML and binary XML were the best data file options. Binary XML had to be ruled out because it is still in the process of standardization, which in the future could lead to issues similar to the ones that were caused by the difference between ISVis's version of C++ and the C++ standard. This left XML as the best option for saving the program's data.

XML is standard, therefore reading files even years after they have been created should not be a problem. It is sufficient to have enough information about the schema to be able to interpret the data in the files (although we have yet to define a formal schema for ISVis's output). Furthermore, platform differences become irrelevant as all machines can write XML documents as well as interpret them correctly. XML documents, however, can be quite large and thereby decrease performance. Although a potential downside for using XML, performance was shown not to be an issue for ISVis; files of approximately 2MB can be loaded and saved on average of 3 seconds.

2) Containers: Changing from the `RWCString` RogueWave class to using `std::string` was trivial (although tedious) — it just involved find and replace, with the occasional fix for replacing operations on strings. The more widely used RogueWave constructs were collection classes such as vectors, lists and maps. After in-depth investigation it was decided to replace the use of RogueWave with that of the STL. The set of libraries provided by STL are not likely to present the same problems as RogueWave did, given that they are not commercial, yet they are standardized and heavily supported and used by the C++ community.

Although there is no doubt about the appropriateness of the decision made about STL, this effort introduced a significant amount of bugs into the system. Most of the problems arose due to a misunderstanding on the programmer side with respect to differences on the basic functionality of collection classes such as adding, removing and traversing through elements. Because the program could not be executed during this process, it was not possible to identify such problems as they were introduced, and instead they had to be traced once the program was running during lengthy debugging sessions. It took over 50 hours of debugging time to get the program into a runnable state. Table V lists the replacements we used to replace the RogueWave elements ISVis relied on.

TABLE V
ROGUEWAVE SOLUTIONS

RogueWave Element	**Replacement**
`RWDEFINE_COLLECTABLE`	Custom
`RWCString`	`std::string`
`RWBoolean`	`bool`
`RWCollectable`	Custom
`RWClassID`	Custom
`RWspace`	Custom
`RWvostream`	`std::ostream`
`RWvistream`	`std::istream`
`TPtrHashMap`	??
`RWTPtrSlist`	`std::list<T>`
`RWTPtrHashSet`	??
`RWTPtrHashTable`	??
`RWTPtrHashDictionary`	??
`RWTPtrDlist`	`std::list<T>`
`RWFile`	`std::fstream`
`RWCTokenizer`	??
`RWTPtrOrderedVector`	`std::vector`
`RWBitVec`	`std::vector<bool>` or `std::bitset`

D. Build Environment

After spending many hours trying to modify the existing Makefile to meet our changing needs, we eventually decided to just scrap the old version and start a new one from scratch. As we were creating this new file we also took the opportunity to clean up the variables that were no longer being used and to make the variable names more descriptive of their function. Additionally, to ease the use of the Makefile on different machines, we created OS specific files which define the library and include paths needed for each machine we are working on. The name of the file is passed to the `make` command as a parameter (ex. `make linux`) allowing those specific paths to be used for the build.

E. GUI

To get the GUI up and running took significant effort. We had to upgrade Motif and gain an understanding about how X Windows was being used to control the display. We also had to understand what the code was trying to accomplish with the GUI in terms of displaying colors and drawing the mural visualization.

1) Motif: Upgrading to Motif 2.1 turned out to be the easiest of the GUI related migration activities. We first started out by analyzing the changes between versions 1.2 and 2.1 - these can be seen in Appendix A. Our initial review of the ISVis Motif code did not indicate any apparent issues with migrating to 2.1. ISVis did not make use of any of the changed symbols nor did the Motif classes, functions and macros used appear in the list of changes made by The Open Group for Motif 2.1. Ultimately, we predicted that if there were any errors resulting from this upgrade, they would be relatively minor. Our predictions held true - all of the issues we encountered related to this change were syntax related and easy to fix.

2) X Windows: ISVis is a visualization application so it relies a great deal on using color to accurately display information to the user. According to chapter 7 from [14], we need the following information to manipulate a color on a display device in the X Windows environment:

- Pixel position value - designates the color of the pixels on the display
- RGB value - the red/green/blue value of the color
- Visual class - defines the color capability of the display; there are a variety of ways of handling and managing the different visual aspects of a computer screen - these management systems are commonly referred to as *visuals*.
- Colormap - indexes each pixel in the screen; used for lookup when the pixel needs to be manipulated

The X Windowing system provides three major visual modes, each with two classes defining whether colormaps are read/write or read-only. They are (listed as read-write/read-only):

- GrayScale/StaticGray - A pixel value indexes a single colormap that contains monochrome intensities. Each pixel value is represented by up to 8 bits.
- PseudoColor/StaticColor - A pixel value indexes a single colormap that contains color intensities. Each pixel value is represented by up to 8 bits.
- DirectColor/TrueColor - A pixel value is decomposed into separate red, green, and blue subfields. Each subfield indexes a separate colormap and is represented by 8 bits per field, giving each pixel 24 bits total to describe a color.

Differences between these types of visuals consist primarily in the manner color values can be manipulated in hardware. The visual class being used determines the exact structure of the colormap for any given window in a particular application (each window having a single associated colormap).

There are three strategies described in the chapter on how to interact with colormaps:

- Shared color cells - like its name suggests, this is simply the strategy of sharing the same color cells across multiple applications.
- Standard color maps - an application can query X Windows for a standard colormap which contains preloaded red/green/blue values for all their color cells.
- Private color cells - allows for the dynamic update of any primary color value stored in the the colormap

ISVis uses the third strategy so it can guarantee access to the colors the user will be manipulating. For this reason it must always be used with a read/write visual class - either PseudoColor or DirectColor. Consequently it can only be run on displays with those classes available.[1] The original version of ISVis was built under the assumption that the program would run on machines with hardware supporting PseudoColor visuals. Modern hardware, however, rarely supports PseudoColor displays and instead DirectColor has become the most popular visual type, followed by TrueColor. The discrepancy between the supported visual types from former and newer hardware caused a myriad of unexpected runtime errors that needed to be fixed in order to have a usable system.

The process of identifying and resolving these issues was a particularly lengthy one, and several decisions had to be made along the way. Because our main goal was first to get the program to run (on any computer) we took the easy route and hard-coded the visual class that should be used (DirectColor) knowing that our monitors would support it. Obviously this is not optimal since the code would eventually get out-dated (again) as display technologies advance. One solution to this could be the X Windows function `XDefaultVisual` which can be used to query the display from the code and retrieve a structure, `Visual`, containing the display device's default visual class. We could make use of this class to check if ISVis can run on the display device it is using. If it can't, we would be able to exit gracefully explaining why ISVis could not execute. Alternatively, several modern UI toolkits support various visual types without any extra effort from the programmer. It would be ideal to refactor ISVis to incorporate a user interface toolkit that is independent from graphics hardware to prevent similar issues from arising in the future.

3) Mural: In order to figure out why the Mural was not being drawn properly we did an analysis of the code responsible for its construction and display. This allowed us to formulate a solution which resulted in us making some minor updates to the code. The entire process is described in Appendix B.

V. Results

[For each class of changes, can we figure out how much code was changed and how long it took? I could not find this level of detail in any of the written documents from the students that did this work, but maybe we could get an estimate by examining the CVS logs]

The status of ISVis after the above adaptations is described in this section.

A. Current Status

1) OS: All that remains to completely break the dependence on Solaris is to find an alternative for the scripts ISVis relies on.

2) Language: The code was updated to comply with the latest ANSI standards. All compile errors have been fixed. Compilation still produces warning messages generated from `Mural.C`.

3) Third Party Dependence: The dependency on the RogueWave library was fully removed. Uses of RogueWave were replaced by the equivalent STL classes or custom implementations. Some persistence functionality has been turned off. [Elaborate on what has been turned off if more information can be found - this was handled by Angela]

4) Build Environment: A new Makefile was created and updated accordingly throughout the course of this study. There are variables in the Makefile that need to be modified manually by the user so it has the right paths to the X Windows libraries.

[1] the `xdpyinfo` command can be used from the command line to query the display device to see the visual modes that are available

5) GUI:

- Compiler errors and warnings concerning Motif were fixed
- Motif was upgraded to version 2.1
- All colors are being displayed normally
- The mural works for both the main and scenario view

B. Time Spent

The total time spent in all the categories of work we have been discussing — including research, debugging, analysis, and adaptations — was 648.26 hours broken down as follows:

1) OS: 55 hours on the OS-specific portion of the code

2) Language: 101.6 hours to bring the code up to ANSI standard

3) Third Party Dependence: 200.26 hours on RogueWave related adaptations

4) Build Environment: 78 hours on the Makefile

5) GUI: 88.4 hours on Motif upgrades, 52 hours on the code related to X Windows and the color display, 73 hours spent on the mural

VI. Discussion

One of the major issues we ran into was the learning curve associated with nearly every adaptation we had to undertake. To adapt the OS we had to learn about Solaris. To adapt the language we had to learn about the differences that were created from standardizing C++. To adapt the overall code base we had to learn about a third party library we had never used before. To adapt the GUI we had to learn about some of the intricacies of X Windows. This is all aside from the learning curve associated with our design discovery. Because the ISVis code base is not documented very well, we spent a great deal of time trying to understand the reasons for the author's original design decisions. In essence, this is one of the challenges that come with legacy systems — when the original authors are no longer available design discovery becomes part of the adaptation process.

To aid us in our discovery process we relied on each other to become experts in specific areas of interest related to our adaptation tasks. We would not have been able to make the progress we did if we did not divide the work in this manner. The use of an IDE (Eclipse [15]) was also a great help to speed the process of adapting the code and detecting compile errors.

One thing we wish was part of ISVis's original code was better log messages. It was surprising to us how long it took to find the right data to report and to display information at the right time in the code execution. It was an unexpected time sink. A logging framework of some kind would have been a huge help in tracking down some of the bugs we encountered. This "epiphany" came to us much later in our process, after we had already littered the code with `printf` statements. Those should definitely be replaced with the use of a framework like Log4cxx [16] where log levels can be set. Logging became so important because we were dealing with a GUI application in which figuring out the series of events/calls/callbacks that triggered a certain condition was not trivial.

Another major concern about continuing to refactor the ISVis code has to do with testing. We currently have no automated process to test the GUI-related portions of the code. That was not too much of a problem during our initial adaptation effort because the set of tests we had to conduct to verify our changes were minimal (see Appendix D). However, as more refactoring is conducted the list of cases to test will grow and we predict it will quickly reach the point where executing each test case manually will become impossible. For this reason, we have done some research into this specific topic which we will discuss later in Section VII-D. Ideally we would be able to set up an environment in which we could make a change and then run all of our test cases without any human involvement other than to figure out the cause of failed tests.

In the rest of this section we will discuss our experience as it relates specifically to refactoring and software adaptation.

A. Refactoring

During our examination of the code, we produced the class diagram shown in Figure 5 showing the code related to the GUI display. `ViewManager` is the entry point to the system - when ISVis starts, it is the `main` function in `ViewManager` that gets called. `X_Application` acts like a utility class giving access to many of the X library calls that ISVis uses although it is unclear if there is a specific set of functionality it is meant to encapsulate. It is unclear because many of the other classes make calls to the X library without going through `X_Application`. `IOShell` is another utility class with methods provided for I/O functions. `MainView` is responsible for instantiating, positioning, and drawing all the subview classes. `View` is used as an abstract class, although it is not defined as one - meaning its type is used as a reference, while only it's children are ever instantiated. The rest of the classes shown relate directly to the GUI components they are named for.

This diagram, along with the statistics shown in Appendix E, was developed using a combination of the techniques described in [17] and [18]. In order to "flush out" a hidden concern, [17] describes a process that combines text-based and type-based mining. The paper illustrates why doing them separately doesn't prove to be as effective as doing them together and then introduces the Aspect Mining Tool, developed by the authors. The techniques described in this paper would have proven to be very useful in refactoring ISVis since the display-related code is scattered in various places throughout the application, and a quick way of finding it would have helped in determining what needed to be changed. Unfortunately that specific tool is only available for Java at the time of this writing.

Nevertheless, while it might not be as a ideal, text-based mining was done manually to search for references to the X library, pointing us to the classes that were related to the display. Type-based mining was done using the compiler strategy described in [18]. The compiler strategy involved commenting out include statements that referenced X libraries and building the code to find what data type references broke. The compiler strategy was needed to catch references to

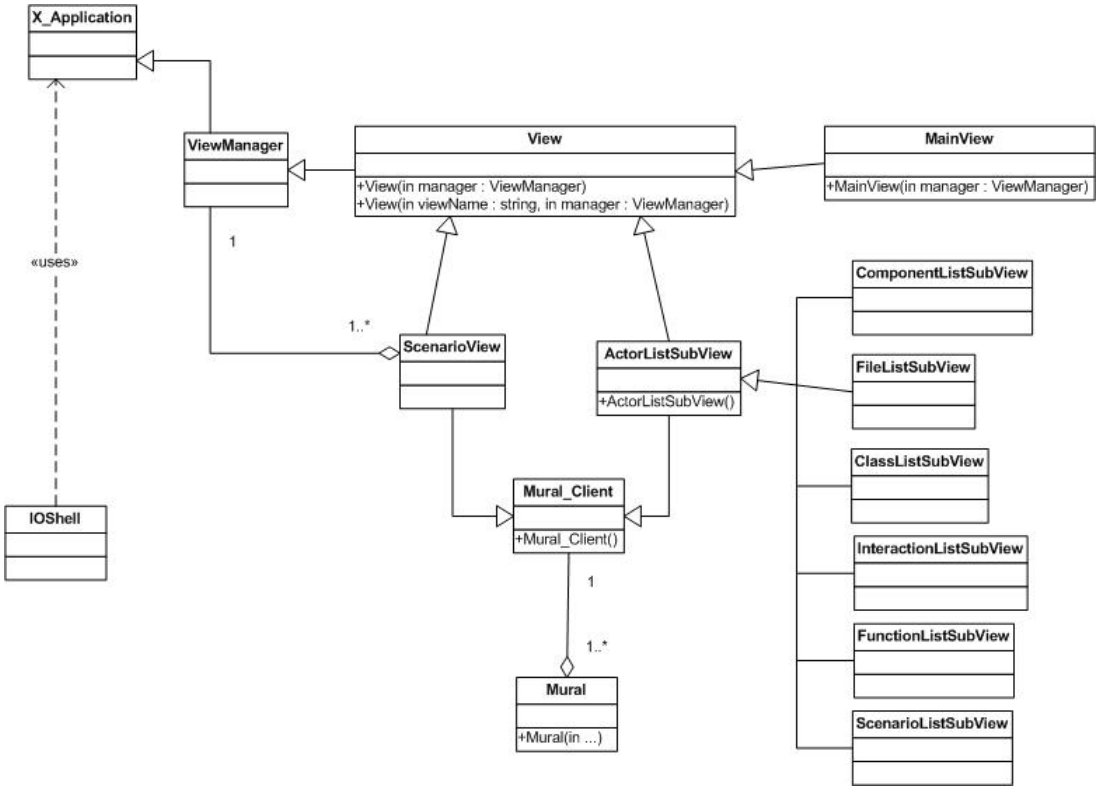

Fig. 5. Class Diagram of display-related code

classes in the X library that did not have the typical X* naming convention.

1) Transformation Approach: One of the main purposes behind identifying which classes are related to the display is to facilitate its encapsulation. Our eventual goal is to create an interface around the current code associated with the display. All the code interacting with the display would do so through interfaces thereby decoupling the display implementation from the program's functionality. Once this layer of abstraction is in place we will be able to "plug out" the current code used for the display and plug in something else - like code for displaying on Windows or Linux. This will make ISVis much more portable and much less brittle to future changes.

2) Refactoring Plan: To accomplish this transformation, the refactoring plan shown in Figure 6 was developed. At the start there are two parallel processes occurring. The left arc starts with an Analyze phase in which aging symptoms in ISVis are identified. From that identification we then choose a candidate to focus our refactoring efforts on. After making that choice we redesign the component in question and then implement the new design by refactoring.

While all of this is happening, the right arc shows two other steps that should also be taking place: building the code and capturing data of the current build to be used in regression tests after the code has changed. Once both arcs have been completed we can proceed with building the newly refactored code. The last box labeled Regression Test includes executing the code and gathering data that can be reliably compared to the data gathered from the pre-refactor execution. This begs the question: what is reliable? That cannot really be determined ahead of time as it is dependent on how fine or coarse the scope of the refactoring is - the candidate for refactoring can be anything from one method to an entire package. An example of the use of this approach is described in Appendix C.

This refactoring will serve as a model for our future design goals of creating an abstract layer for the display code to live under. We will progressively migrate all of the X Windows specific calls out of the main classes and into "X" classes. Once we reach the point where all the main classes are only calling their corresponding "X" classes for X Window related functions, then we will be able to add a layer of abstraction between those calls. Taking our color button method as an example, once we have our abstract layer we would replace our call to `XMainView::CreateColorButton` with `Abstract_View::CreateColorButton`. Once all of our main classes call our abstract layer we will be free to vary the implementation beneath that layer as needed.

B. Software Adaptation

At the beginning of this project, the system was clearly outdated as well as incompatible with newer technologies. Upgrading ISVis made evident the importance of constantly maintaining software in order to keep it up to date with platform changes. Upgrading ISVis frequently could prevent the program from requiring a massive refactoring session like the ongoing one. Because there are several aspects of the program dependent on third party software, maintenance of the system can be extremely complex and lengthy if all of

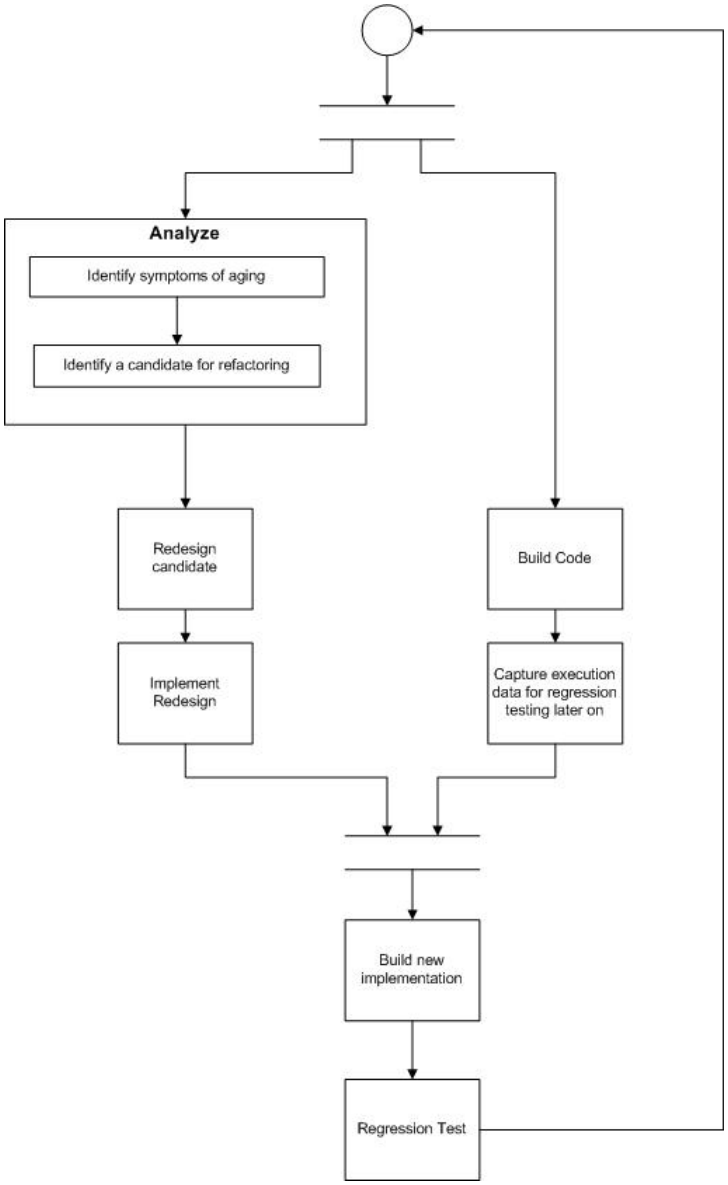

Fig. 6. Refactoring Plan for ISVis

these technologies are critically outdated, therefore it is ideal to constantly maintain ISVis and upgrade it whenever it is necessary.

Many of the issues found when attempting to run ISVis for the first time during this study were introduced into the system during the upgrade of the code base. Uninitialized variables, invalid access to data and incorrect use of collection classes were amongst the most common faults preventing the program from running correctly. Tracking down these faults was a very costly process in terms of time and effort. When systems require large numbers of changes, it is of primary importance to be able to test the system as it undergoes such changes. Unfortunately, because of the nature of the initial upgrades, it was not possible to test ISVis as its code base was modified. Had this not been the case, valuable time could have been spent resolving other important issues. On the other hand, a more careful look into the differences between the original code base and the ANSI standards could have prevented such faults from being introduced into the system. It is important to mention, however, that the number of faults introduced was relatively small considering the large number of changes and source files which were modified. Furthermore, it would have been ideal to have the developer responsible for such changes debugging the system, rather than to have a new developer come in and learn about these changes while attempting to correct any errors.

As shown during this study, it is of primary importance to carefully evaluate various alternatives when considering the usage of third party software and libraries. It is clear that in many cases it is not necessary to spend resources developing custom functionality that can be provided by other existing systems. It is crucial nonetheless to select the appropriate system to reduce any overhead and prevent future issues as in the case of ISVis. RogueWave becoming unavailable was very

expensive to mitigate. When selecting third party software it is important to look into the future and select systems that are widely used by the software community, seem stable, and are well documented and supported by either developers or user groups.

ISVis's user interface was the piece of the system with the most problems. Even if at the time ISVis was developed there were very few user interface toolkits, some valuable lessons can be learned from the use of Motif. Many of the issues with Motif arose from the usage of procedures tightly related to computer hardware. Hardware changes rapidly, and developing software too closely linked to hardware shortens the potential life of the system. Similarly, a system too strongly connected to hardware is hard to port. Fortunately, nowadays most user interface toolkits take care of the platform dependent issues and provide developers with more abstract functionality that can be easily ported.

In general, it is recommended (whenever possible) to avoid the usage of any library or system that is highly dependent on a specific hardware environment. Similarly, the usage of non-standard versions of protocols, programming languages or any other software artifact should be avoided in order to prevent issues costly to resolve like the ones found throughout this study. It is also of primary importance to constantly update software products to prevent them from becoming completely obsolete.

Although the work on ISVis can be considered successful, slight differences in the approach taken could have made the study more successful. As shown before, the fact that ISVis could not be run while the source code was going through major changes introduced many problems into the system. Even if it is expected to encounter difficulties when refactoring a system, the amount of time and effort spent on finding and fixing the introduced bugs was much larger than expected. Similarly, having one individual making source code changes and another one finding the faults and fixing them was not particularly efficient. It is highly likely that, had the same developer been in charge of both tasks, the debugging time would have been shorter and more improvements would have been introduced into ISVis in the same period of time.

C. Future Work

One quick task that we will perform to improve the code's understandability is to separate all classes into individual source/header files. The code base presently contains many classes that are grouped together in the same source and header file. This was somewhat confusing when trying to read the code and why an IDE was so important because it allowed us to quickly find where a class was defined since we could not rely on the file names. It would be helpful to remedy this and put each class in its own file.

After that is completed ISVis needs to be thoroughly tested. Before we proceed with any additional refactoring, we need to generate a comprehensive list of tests for every feature of the program and fix any bugs that turn up from that process (see Appendix D). Preferably this would turn into some automated process that would execute our tests for us whenever needed. As we have described above, not being able to execute the code and see the results of our changes until much later was an error-prone process (although unavoidable in our case). We need to have these test cases to give us a more solid foundation which we can use to reliably perform our refactoring effort.

VII. Related Work

The background work that was needed to perform the adaptation to ISVis covers four main areas: legacy systems, refactoring, software adaptation, and GUI testing.

A. Legacy Systems

Legacy systems are, by definition, old systems whose health has often deteriorated over time. Aging symptoms of legacy systems are described in [19] where Visaggio et al. advise developers on how to measure and handle design deterioration. The authors describe five symptoms to watch for:

- Embedded Knowledge - knowledge about the system that can no longer be derived from existing documentation
- Pollution - components that are no longer used by the users
- Poor Lexicon - variables and components that have names with no meaning
- Coupling - the flow of control/data between components is tightly linked
- Layered Architectures - several different solutions spread out across a system's architecture.

ISVis exhibits all of the above symptoms to varying degrees, suffering most severely from embedded knowledge.

Because of these symptoms, we know ISVis has to be refactored, and we know from our definition above that after a refactoring, a program must be syntactically correct. There are ways to preserve a program's behavior which rely on the compiler catching mistakes, but there are some errors which could change the behavior of a program that a compiler would not be able to catch. A particular set of program properties have been found to be easily violated if explicit checks are not made before a program is refactored. Those safety conditions are described in Opdyke's dissertation [18]. As ISVis is refactored more and more, the properties described by Opdyke should be kept in mind.

When that time comes, there are a variety of ideas to draw upon for guidance on how to proceed. In [20], Doblar and Newcombe argue for an increased focus on automating transformation processes based on the statistic that, on average, a well trained programmer can only transform about 160 lines of code per day. To transform an entire system at that rate would be too costly. The innovative part of the process described by Doblar et al. is that they propose using a suite of artificial intelligence technology tools to automate 99 percent of the transformation work. What is left for manual transformation should be what truly requires a human decision maker. They make the point that while high levels of automation are achievable for transformation tasks there will always be tasks that are manually intensive. Nevertheless, they argue that anything that can be automated should be.

Unfortunately, the paper did not go into how the AI performs its tasks, but that is most likely because the research comes from a private company dealing with defense software. Also, the techniques described were applied to languages to convert them to C++ - it is unclear whether they would work if trying to perform a transformation to the same language. Regardless, these techniques would probably be overkill for reengineering ISVis.

[21] summarizes the efforts of an XP team from ThoughtWorks Technologies to introduce new features to a legacy system. It provides an informal guide to working with a legacy system using an agile development process. Since agile methods are relatively new and most legacy systems were developed under a non-agile development process, it is interesting to see those techniques applied to an older system to try and make it more maintainable. We have tried to adopt the same mentality as we have begun to refactor ISVis - the refactoring plan in Section VI-A2 is an example of this. As the adaptation of ISVis continues, we expect agile methods to be a part of the process we use.

In [22], Thomas argues that the older the code base and the more mission critical the application, the more difficult it is to maintain or enhance it. One primary difficulty is just understanding the code base which may not be well-designed or well-written. For this reason, the author promotes the idea that when working with legacy software, Discovery and Transformation should be the focus before considering Design and Development. Discovery combines the stories obtained from experienced developers and customers with knowledge gained by analyzing the code, associated documentation, and test cases. Transformations of the code should be done systematically in small pieces, testing each change before moving to the next. These two concepts are important parts of the plan for refactoring ISVis.

The Discovery process is also a primary concern of the research described in [17], where the authors address the problems associated with identifying hidden concerns in legacy code. A hidden concern (HC) is any issue that has been grafted into existing code to extend it's functionality in some way that is different from the piece of code it is being added to. This decreases the code quality by reducing it's coherence. More often than not, an HC suffers from two problems that make it hard to track down: It is usually scattered throughout the project and entangled with other code. Consequently the primary focus as it relates to HC's is how to identify and extract the code related to a hidden concern - a task which the authors recognize as being non-trivial. In order to create the abstract display layer that we want for ISVis's code, that identification process will have to be conducted on a more extensive scale to find all the HC's related to the display of the interface.

A possible approach to identifying hidden concerns could be feature-oriented refactoring (FOR) [23]. This is the process of decomposing a program into features, thus recovering a feature based design and giving it an important form of extensibility. A characteristic of FOR that makes it challenging is that a feature cannot always be implemented by a single module. To try and deal with this problem Batory et al. describe a methodology for expressing programs and features as mathematical equations. Engaging in this practice gives the practitioner a reliable way of predicting the effects of adding or removing features from the base program because the derived equations communicate properties that decompositions and their underlying base modules and derivatives must have. These properties are automatically checked by the tools the authors developed to aid in the FOR process. The authors developed an Eclipse plugin to be used with legacy Java applications. The plugin guides the user through a five-step process which defines the base modules of the program, optional modules, and then reconstitutes the program for the user.

Pertaining to ISVis, the tool the authors developed would be useful to have because it would give us an automated way of discovering the classes directly involved with any of the features we might be interested in. Unfortunately, the plugin only works for Java programs and the work involved in implementing our own feature discovery tool by hand would probably not be worth the effort. Additionally, this method requires a substantial knowledge of the program being examined and that would make using it on ISVis difficult since ISVis suffers from the "embedded knowledge" symptom of an aged system.

Another idea could be to use a goal model - a graph structure representing stakeholder goals and their inter-dependencies. [24] describes a methodology for obtaining a goal model from legacy code, but it is a process that is currently unnecessary for the ISVis project since the main focus of the reengineering process has already been identified. However, the methods described by Yu et al. could potentially be useful in future refactoring efforts on ISVis as they would allow for the categorization of the code to help identify candidates for refactoring.

In [25], Heineman and Mehta describe a 3-step methodology for evolving a system into components based on the features the system implements. They applied their proposed process to a 14-year old product, and they describe the lessons they learned from their work. Two of those lessons are applicable to the ISVis adaptation project: The authors conclude that features are good candidates for evolution into components if they "change often, are concentrated in fewer functions, or depend on or share global variables as a means of communication." While the "display feature" of ISVis might not necessarily change often, it was chosen as the primary candidate for refactoring because it is currently nonfunctional on modern machines and this holds it back from being used or updated.

Secondly, the authors comment that "the true measure of a successful evolution methodology is in reduced future maintenance costs." From that perspective, the adaption of ISVis to have a plugin-style interface for its display code will certainly be a success since that accomplishment will drastically reduce the cost of maintaining that portion of the code.

[26] describes a process for incrementally re-engineering a legacy application whose architecture has degraded over time. Specifically, the authors' goal was to develop a target architecture for a system and then re-engineer the system to the desired architecture using a defined series of steps.

Reengineering ISVis will most likely produce the architecture erosion Coelho et al. talk about since there is no clearly defined documentation on the design of the system. The lack of architectural documentation will ensure that any changes made to the code will not be in line with the original author's thinking, thus producing the erosion.

[27] describes a process model for re-engineering a legacy system - particularly a system that is in use and cannot be shut down for an extended period of time. One of the main issues the paper addresses is the traditional approach of reengineering a system all at once and consequently prohibiting the use of the system while it is being changed. To combat this, the authors propose a method which share features with the Chicken-Little Strategy [28] and the Butterfly Methodology [29] while alleviating some of their weaknesses. [27] was used as inspiration for the refactoring plan that was developed for the ISVis display (found in Section VI-A2). While [27] was targeted for data-centric systems, the more general idea of iterating over a reengineering process will definitely be applied to adapting ISVis.

We can see from the limited set of work we have looked at regarding legacy systems that dealing with them is not trivial. It involves recognizing what ails the system, deciding on how to fix it, and then implementing the solution. Each of those tasks take significant effort as our experience with ISVis has shown us.

B. Refactoring

There are a variety of reasons people have encountered prompting them to re-examine the structure of a software application. This re-structuring is often referred to as either "refactoring" or "reengineering" - but mistakenly, they are sometimes used interchangeably. According to Fowler in [30] "refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure." Reengineering has a much broader scope. Chikofsky and Cross define reengineering as "the examination and alteration of a system to reconstitute it in a new form." [31] While a main goal of refactoring is to maintain the behavior of the code being changed, it is often the case that new functionality is added (or old functionality changed) during a reengineering project. Reengineering often necessitates some sort of refactoring, but refactoring rarely has the impact that a reengineering process would.

The main characteristic that is common to both processes is that the practitioner is trying to change the target application (or system) in some way. Generalizing the motives for each of these processes might lead us to the notion that we refactor to improve maintainability, and we reengineer to add functionality. Taking these motives into consideration, we quickly come to realize that when dealing with a legacy application, both processes become necessary and both bring with them challenges (as we have seen from our discussion above).

To help handle these challenges, Mens and Tourwe define five activities in [32] that should be performed during each individual refactoring:

1) Identify where to refactor the software
2) Determine which refactoring to apply
3) Guarantee the applied refactoring preserves behavior
4) Apply the refactoring
5) Evaluate the success of the refactoring

One reason to take on these challenges is because of the benefit we gain from modularity. Sullivan et. al. talk about this in [33] where they describe a study to find quantitative means of measuring the value of a module in a software design. To do this, the authors make use of design structure matrices (DSM), one of the components of Baldwin and Clark's options model [34]. The other component is the Net Option Value formula. The authors adapt both parts of this model to the subject of software modularity and show how they applied it to Parnas's famous KWIC example [35]. Their results did yield the same conclusions that Parnas came to, predicting that the modularized design of KWIC was better than the strawman.

The authors claim that their results serve as evidence for the potential to quantify the value of a design and thus support a discipline of "design for added value." This was written eight years ago so we are curious to find out if time has lead to more concrete results in this matter since it would have very practical effects on a developer's effort to estimate the cost of a refactoring. It would certainly be beneficial to our adaptation process to be able to measure the potential benefits of different modularization options before deciding on one to implement.

[36] describes the use of a formula to calculate the cost effectiveness of a planned refactoring. While it still remains the job of the developer to decide what to refactor, the proposed formula aims to help determine when that refactoring should happen by calculating the return on investment (ROI) for the planned activity. If the ROI is greater than or equal to one, then the planned refactoring is deemed to be cost effective. It would be hard to determine the ROI for the planned refactoring of ISVis since there are currently no regression tests available, a key component to calculating the formula. It would be a rather meaningless effort anyway since the planned refactoring is necessary to get the application running on modern machines — so we would not refactor it if our ROI was below one. Nevertheless, this could prove to be useful in the future, after ISVis is working properly, to determine how to maximize the effort spent on further modifications.

Without the ability to reliably and consistently quantify refactorings, an easier alternative is to refactor continuously and liberally. Roock and Lippert [37] claim that such behavior will lead to an emergent design. They describe this concept as a design that is constructed incrementally as more is learned about the software behavior and requirements. They prefer this to the traditional method of a big upfront design because constant refactoring keeps the code from ageing. Refactorings should be conducted to support changing requirements and to eliminate weaknesses in the code. "Weaknesses are present when the existing system structure hampers or even prevents modifications." These bad smells should not be ignored since they degrade the quality of the code over time.

In conducting these refactorings, small refactorings are always better since they reduce the risk associated with the changes by allowing the code to remain in a usable state.

Sometimes it is necessary to make the code more complex in order to guarantee it stays in a usable state until the refactor is completed. This is essentially what we have done by creating the `XMainView` class (described in Section VI-A2). The authors refer to this as a "detour" with the appropriate analogy: "build a detour first and then tear up the road."

A real world example of this can be found in [38] where the authors describe a case study in which they refactored a portion of a commercial platform comprised of 1.5 MLOC. To carry out the refactoring the authors first developed a conceptual architecture model of the system based on discussion with developers. They then used architecture analysis tools to help target dependencies in need of refactoring. After coming up with a new architecture for the modules in question, the authors discussed it with the developers to get their feedback and to help discover any patterns that would be worthy of abstraction. After their consultation with the developers, they ended up changing their original refactoring plan to better accommodate the developers' concerns. This resulted in a "less clean" version of the new architecture, but it was a necessary detour to make all parties comfortable with the proposed changes.

Looking at ISVis with all of this knowledge is a primary motivation for refactoring the code. We could say a new requirement for ISVis is that it be platform-independent so refactoring the display-related code is intended to achieve that end. Additionally, ISVis is plagued with bad smells as we have discussed earlier so there will undoubtedly be more refactorings than what we have initially planned.

C. Software Adaptation

In [39], Heineman and Ohlenbusch make a distinction between evolution and adaptation stating that "evolution occurs when a software component is modified by the original component design team or by programmers hired to maintain and extend the component ... In contrast, adaptation occurs when an application builder acquires a third-party component and creates a new component to use within the target application." ISVis has both of these scenarios - the instrumentation tools being the third-party components that have been used in the program. Our current focus is on the evolution of the application to bring it up to date with today's standards.

The paper lists several "requirements" that are needed for the adaptation of a component - they are really more like guidelines and are not all necessary (or possible) depending on the adaptation being performed. The paper then discusses five techniques that can be used in the adaptation of software:

- Active interfaces - a component that allows for the execution of user-defined callback methods; in contrast to the other techniques discussed this one relies on the component being adapted to have this functionality
- Binary Component Adaptation (BCA) - a technique that applies adaptations to component binaries (no source code access)
- Inheritance - built into OO languages
- Wrapping - this technique wraps the component being adapted providing an interface that hides the component's usage
- In-place - occurs when changes are made directly to the source code of a component

Obviously our adaptation of ISVis falls into the last category. Our plans for refactoring the display code might employ some use of wrapping, but it is too early to tell. One thing we anticipate being necessary is some sort of dynamic way of detecting the display capabilities of the platform being used to run ISVis and then adapting the execution accordingly.

In [40], Camara et. al summarize a strategy that can be used to design a dynamic adaptation management framework — i.e. a system that can automatically integrate components that enter the context of its execution at runtime. The authors provide an illustration of the proposed framework showing three modules — the Interface, Adaptor and Coordination managers — that interact with each other. The Interface Manager inspects components as they enter the context of the system and keeps references to them in a repository. The Adaptor Manager keeps track of mappings between classes - these mappings are produced dynamically the first time there is any communication between a particular set of classes. The Coordination Manager manages the sessions that are created for each "conversation" between classes. The authors propose using aspects to achieve this functionality since such an approach would allow for the necessary functionality to be weaved in and out of the code execution as needed. This is an interesting approach to dynamic component evolution, but we will need to do more research into other options for achieving the behavior we are after for ISVis's future.

D. GUI Regression Testing

In Section VII-A we discussed a methodology for evolving a system into components - two key assumptions made in [25] are that (1) the system is coded in a language for which a code-profiling tool is available and (2) that the legacy system has regression test suites. ISVis satisfies (1) but not (2) and thus could not use the methodology described without first developing a reliable set of test cases. This raises an important issue: How do we reliably test a GUI based application? There is a technique described by White in [41] where automated regression tests can be generated to test both static and dynamic event interactions in a GUI. Static interactions are those that are contained within one GUI screen. Dynamic interactions involve decisions happening on multiple screens. White describes three solutions that could be used for generating the interaction test cases: (1) brute force enumeration of the elements of each possible path, (2) random generation of test cases, and (3) Mutually Orthogonal Latin Squares. White concludes that while using Latin Squares would generate the minimum number of tests, it is a more complex procedure to apply to larger systems. In such a case he suggests using the random generation technique since it is still more efficient than brute force. Relating this to ISVis and taking White's suggestion, we would probably want to apply the Latin Squares approach to generating test cases since ISVis is not a very large application.

However an entirely different approach could be to use the techniques described by Memon et. al in [42]. This

paper describes the use of planning, an AI technique, to automatically generate test cases for a GUI. The paper argues that instead of trying to design test cases for every possible state of a GUI, it is more effective and simpler to specify the goals a user might want to accomplish and have a tool that can automate the sequence of events that would need to occur to meet those goals. The test case generation system developed by the authors takes these goals as input in the form of initial and end states. Their system then generates sequences of actions to transition between the two states, which serve as the test cases for the GUI.

While Memon and White propose two different techniques to test case generation, they both agree that automation of test cases for GUI's is a necessary to accommodate the changes that can occur during its development and maintenance. Which technique to use for ISVis still remains to be seen. There are even decisions that must be made after the test cases are in place concerning how to maintain them. Memon & Soffa write about a possible solution to that dilemma in [43] describing a technique to repair unusable test cases after the components of a GUI have changed.

When a GUI changes it is inevitable that some tests will become unusable because the chain of events that occur for interactions with the GUI will change. Normally, these unusable tests would have to be regenerated, but GUI test case generation is expensive. Memon & Soffa's solution is based on modeling the GUI's events and components and then comparing the original and modified versions of the models. From that comparison they are able to automatically detect unusable test cases and repair the ones that can be repaired. ISVis isn't quite ready for these techniques since there are no test cases related to the GUI yet, and the GUI is not in a rapid state of change, but it may be wise to consider such things when choosing a method to generate the test cases that will be needed.

In [44], Memon et al. describe the use of an automated oracle to help test the expected behavior of a GUI. The automation occurs both in the derivation of what the expected states of the GUI are and in the comparison of those expected states to the actual state of the GUI as it is executed. The authors achieve this by first creating a model of the GUI's objects, properties, and actions. They then use this model to determine what the expected states of the GUI will be given any particular set of actions. The paper describes three different levels of testing that can be used to compare the expected states to the actual states of the GUI - determining what to use is left up to the test designer.

After the authors lay out how to achieve the automated oracles, they give some performance results showing that the extra time needed to generate the automated test cases is not substantial. Consequently, they argue that even though there is a reasonable amount of human effort involved in creating the oracle, there is definitely an amortized benefit for the work that is done. I am not totally convinced of that argument since it was based on metrics gathered by testing a custom Notepad editor. It would take substantial effort to produce this oracle for ISVis especially considering that it is directly tied to an automated test case generator produced previously by the authors. All of that would have to be done from scratch if ISVis was to take advantage of this method. That being said, if it were in place, it would certainly be beneficial to future development and refactoring.

Another work built on Memon's is that of Qing Xie in [45] where he gives an overview of a strategy he developed to study GUI faults, GUI event interactions, and the development of GUI fault detection techniques. The rationale behind this is that when testing a GUI an unexpected screen could be generated from an incorrect state of the GUI - in such a case, the tester would want to detect this fault so the test case can be terminated. Xie proposes a framework to help automate the testing process all built on an automatically generated model of the GUI. He then proposes a series of steps that would be needed for the framework to accomplish its purpose. One module of the framework that would prove useful for ISVis is the Regression Tester. The author does not go into details about how each of the modules are implemented, but just the idea of having a separate entity whose sole purpose is to regression test is a sound one. ISVis could definitely benefit from such a module to verify the refactoring changes are done correctly.

It seems that the art of testing a GUI is still somewhat specialized based on the needs of a particular application. While the above examples show the strides being made to generalize the art, we would argue that they would be too elaborate to implement at the present stage of ISVis's life cycle. We are not quite sure what type of testing would best be suited to ISVis so for the moment we are just logging tests in a spreadsheet until we settle on a more formal approach. Further research into this topic would certainly be beneficial in order to help develop a more reliable process in refactoring ISVis.

VIII. Conclusion

To adapt ISVis we had to handle moving to a new operating system, upgrading the language to a new version, dealing with third-party libraries, understanding GUI library functions, and discovering the design and usage of an undocumented code base. The research needed to conduct our adaptations was extensive and wide-ranging yielding results and experiences that are applicable to a variety of topics.

So why is this study important? From a business perspective, consider that ISVis took 648 hours to be adapted to a working state. If we translate that into a typical 40-hour work week, it would have taken one person working full-time approximately 16 weeks to complete this task. If managers are tasked with estimating the time it would take to adapt a system this study can help them get a sense of the issues that may be encountered along the way.

From an academic perspective, case studies are valuable since they help to bring real numbers and considerations to the ideas important to the research community. This study has touched on a number of those ideas including software migration difficulties, upgrading and debugging legacy code, and GUI regression testing. Adapting ISVis has shown us that there is a lot to cover under the umbrella of "software adaptation."

ACKNOWLEDGMENT

This study was conducted under the guidance of Spencer Rugaber. Contributions were made by Michele Biddy, Peter Calvert, Phil Norton, Chung-Yen Huang, and Angela Navarro. The research they contributed that was not referenced directly include: [46], [47], [48], [49], [50], [51], [52], [53], [54], [55], [56], [57], [58], [59], [60], [61], [62], [63], [64], [65], [66], [67], [68], [69], [70], [71], [72], [73], [74], [75], [76], [77], [78], [79], [80], [81], [82], [83], [84], [85], [86], [87], [88], [89], [90], [91], [92], [93], [94], [95], [96].

REFERENCES

[1] "Isvis original home page." [Online]. Available: http://www.cc.gatech.edu/morale/tools/isvis/original/isvis.html

[2] "Solaris home page." [Online]. Available: http://www.sun.com/software/solaris

[3] "Roguewave." [Online]. Available: http://www.roguewave.com

[4] S. Rugaber and D. Jerding, "Using visualization for architectural localization and extraction," 1997.

[5] D. R. Tribble, "Incompatibilities between iso c and iso c++," August 2001. [Online]. Available: http://david.tribble.com/text/cdiffs.htm#C99-vs-CPP98

[6] "Standard template library." [Online]. Available: http://www.sgi.com/tech/stl

[7] "Motif." [Online]. Available: http://www.opengroup.org/motif

[8] "Project bauhaus." [Online]. Available: http://www.bauhaus-stuttgart.de/bauhaus/index-english.html

[9] "Semantic designs." [Online]. Available: http://www.semanticdesigns.com

[10] "Understand." [Online]. Available: http://www.scitools.com/products/understand/features.php

[11] "Lattix." [Online]. Available: http://www.lattix.com

[12] "C++ standards committee." [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21

[13] "Boost." [Online]. Available: http://www.boost.org

[14] O. Jones, *Introduction to the X Window System*. Prentice Hall, 1989.

[15] "Eclipse." [Online]. Available: http://eclipse.org

[16] "Log4cxx." [Online]. Available: http://logging.apache.org/log4cxx/index.html

[17] J. Hannemann and G. Kiczales, "Overcoming the prevalent decomposition in legacy code," *ICSE 2001 Workshop on Advanced Separation of Concerns*, May 2001.

[18] W. F. Opdyke, "Refactoring object-oriented frameworks," Ph.D. dissertation, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1992.

[19] G. Visaggio, "Ageing of a data-intensive legacy system: symptoms and remedies," *Journal of Software Maintenance*, vol. 13, no. 5, pp. 281–308, 2001.

[20] R. A. Doblar and P. Newcombe, "Automated transformation of legacy systems," *CrossTalk*, December 2001. [Online]. Available: http://www.stsc.hill.af.mil/crosstalk/2001/12/newcomb.html

[21] A. Pols and C. Stevenson, "An agile approach to a legacy system," in *5th International Conference on Extreme Programming and Agile Processes in Software Engineering (XP 2004)*, 2004, pp. 123–129.

[22] D. Thomas, "Agile evolution towards the continuous improvement of legacy software," *Journal of Object Technology*, 2006. [Online]. Available: http://www.jot.fm/issues/issue_2006_09/column2

[23] J. L. Don Batory and C. Lengauer, "Feature oriented refactoring of legacy applications," in *ICSE '06: Proceedings of the 28th international conference on Software engineering*. New York, NY, USA: ACM, 2006, pp. 112–121.

[24] Y. Yu, Y. Wang, J. Mylopoulos, S. Liaskos, and A. Lapouchnian, "Reverse engineering goal models from legacy code," in *RE '05: Proceedings of the 13th IEEE International Conference on Requirements Engineering (RE'05)*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 363–372.

[25] A. Mehta and G. T. Heineman, "Evolving legacy system features into fine-grained components," in *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*. New York, NY, USA: ACM, 2002, pp. 417–427.

[26] M. Abi-Antoun and W. Coelho, "A case study in incremental architecture-based re-engineering of a legacy application," in *WICSA '05: Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 159–168.

[27] A. Bianchi, D. Caivano, V. Marengo, and G. Visaggio, "Iterative reengineering of legacy functions," in *ICSM*, 2001, pp. 632–641.

[28] M. L. Brodie and M. Stonebraker, *Migrating legacy systems: gateways, interfaces & the incremental approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995. [Online]. Available: http://portal.acm.org/citation.cfm?id=208444

[29] B. Wu, D. Lawless, J. Bisbal, R. Richardson, J. Grimson, V. Wade, and D. O'Sullivan, "The butterfly methodology: A gateway-free approach for migrating legacy information systems," in *ICECCS '97: Proceedings of the Third IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '97)*. Washington, DC, USA: IEEE Computer Society, 1997, p. 200.

[30] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[31] E. J. Chikofsky and J. H. C. II, "Reverse engineering and design recovery: A taxonomy," *IEEE Softw.*, vol. 7, no. 1, pp. 13–17, 1990.

[32] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Trans. Softw. Eng.*, vol. 30, no. 2, pp. 126–139, 2004.

[33] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen, "The structure and value of modularity in software design," in *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*. New York, NY, USA: ACM, 2001, pp. 99–108.

[34] C. Y. Baldwin and K. B. Clark, *Design Rules: The Power of Modularity*. MIT Press, 2000.

[35] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[36] R. Leitch and E. Stroulia, "Understanding the economics of refactoring," in *5th International Workshop on Economics-Driven Software Engineering Research (EDSER-5): The Search for Value in Engineering Decisions*, May 2003, pp. 44–49.

[37] S. Roock and M. Lippert, *Refactoring in Large Software Projects*. John Wiley and Sons, Ltd., 2003.

[38] K. Stirewalt, S. Rugaber, H.-Y. Hsu, and D. Zook, "Experience report: Using tools and domain expertise to remediate architectural violations in the logicblox software base," in *Proceedings of the International Conference on Software Engineering, (ICSE 2009)*, 2009.

[39] G. T. Heineman and H. M. Ohlenbusch, "An evaluation of component adaptation techniques," in *In 2nd ICSE Workshop on Component-Based Software Engineering*, 1999.

[40] J. C. Javier Camara, Carlos Canal and J. M. Murillo, "J.m.m.: An aspect-oriented adaptation framework for dynamic component evolution," in *RAM-SE, Fakultt fr Informatik, Universitt Magdeburg (2006) 5970*, 2006, pp. 59–71.

[41] L. White, "Regression testing of gui event interactions," *Software Maintenance 1996, Proceedings., International Conference on*, pp. 350–358, Nov 1996.

[42] A. M. Memon, M. E. Pollack, and M. L. Soffa, "Using a goal-driven approach to generate test cases for guis," in *ICSE '99: Proceedings of the 21st international conference on Software engineering*. New York, NY, USA: ACM, 1999, pp. 257–266.

[43] A. M. Memon and M. L. Soffa, "Regression testing of guis," in *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*. New York, NY, USA: ACM, 2003, pp. 118–127.

[44] M. E. P. Atif M. Memon and M. L. Soffa, "Automated test oracles for guis," *SIGSOFT Softw. Eng. Notes*, vol. 25, no. 6, pp. 30–39, 2000.

[45] Q. Xie, "Developing cost-effective model-based techniques for gui testing," in *ICSE '06: Proceedings of the 28th international conference on Software engineering*. New York, NY, USA: ACM, 2006, pp. 997–1000.

[46] A. Holub, "Roll your own persistence implementations to go beyond the mfc frontier," *Microsoft Systems Journal*, June 1996.

[47] W. F. Tichy and J. Heilig, "A generative and generic approach to persistence."

[48] F. Cao, B. R. Bryant, W. Zhao, C. C. Burt, R. R. Raje, A. M. Olson, and M. Auguston, "Marshaling and unmarshaling models using the entity-relationship model," in *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*. New York, NY, USA: ACM, 2005, pp. 1553–1557.

[49] S. Wang, S. R. Schach, and G. Z. Heller, "A case study in repeated maintenance," *Journal of Software Maintenance*, vol. 13, no. 2, pp. 127–141, 2001.

[50] J. Soukup, *Taming C++: Pattern classes and persistence for large projects*. Addison-Wesley, June 1994.

[51] D. Dig and R. Johnson, "How do api's evolve," *Journal of Software Maintenance and Evolution: Research and Practice*.

[52] G. Bastide, "A refactoring-based tool for software component adaptation," in *Proceedings of the Conference on Software Maintenance and Reengineering*.

[53] P. A. T. B. A. A. P. C. Pazel D, Varma P, "A framework and tool for porting assessment and remediation," in *Proceedings of the 20th IEEE International Conference on Software Maintenance(ICSM'04)*.

[54] R. M. and O. F., "An effective strategy for legacy systems evolution," *Journal of Software Maintenance and Evolution: Research and Practice*.

[55] T. M. Jim, J. Buckley, M. Zenger, and A. Rashid, "Towards a taxonomy of software evolution," 2003.

[56] Y. Yang, M. Munro, and Y. Kwon, "An improved method of selecting regression tests for c++ programs," *Journal of Software Maintenance and Evolution: Research and Practice*.

[57] T. Mens and M. Wermelinger, "Separation of concerns for software evolution," *Journal of Software Maintenance*, vol. 14, no. 5, pp. 311–315, 2002.

[58] R. K. S. J. Carriere, S. G. Woods, "Software architecture transformation," in *Proceedings of WCRE 99, (Atlanta, GA)*, October 1999, pp. 13–23.

[59] J. D. Ahrens and N. S. Prywes, "Transition to a legacy and reuse-based software life cycle," *Computer*, vol. 28, no. 10, pp. 27–36, 1995.

[60] H. Z. Hongji Yang, Xiaodong Liu, "Abstraction: a key notion for reverse engineering in a system reengineering approach," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 12, no. 4, pp. 197–228, 2000.

[61] G. W. Matthew Harrison, "Identifying high maintenance legacy software," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 14, no. 6, pp. 429–446, 2003.

[62] L. X. B. X. Jianjun Zhao, Hongji Yang, "Change impact analysis to support architectural evolution," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 14, no. 5.

[63] W. R., "Improving the maintainability of software," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 8, no. 8, pp. 345–356, 1996.

[64] D. Advani, Y. Hassoun, and S. Counsell, "Understanding the complexity of refactoring in software systems: a tool-based approach," *International Journal of General Systems*, vol. 35, no. 3, pp. 329–346, 2006.

[65] H. M. Sneed, "Measuring reusability of legacy software systems," *Software Process: Improvement and Practice*, vol. 4, no. 1, pp. 43–48, 1998.

[66] W. Scacchi, "Understanding software process redesign using modeling, analysis and simulation," *Software Process: Improvement and Practice*, vol. 5, no. 2-3, pp. 183–195, 2000.

[67] M. Olsem, "An incremental approach to software systems re-engineering," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 10, no. 3, pp. 181–202.

[68] S. Ramanujan, R. W. Scamell, and J. R. Shah, "An experimental investigation of the impact of individual, program, and organizational charactersitics on software," *The Journal of Systems and Software*, vol. 54, no. 2, p. 137, 2001.

[69] N. Chapin, J. E. Hale, K. M. Khan, J. F. Ramil, and W.-G. Tan, "Types of software evolution and software maintenance," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 13, no. 1, pp. 3–30, 2001.

[70] K. P. B. Webster, K. M. de Oliveira, and N. Anquetil, "A risk taxonomy proposal for software maintenance," in *ICSM*, 2005, pp. 453–461.

[71] K. Mens, A. Kellens, F. Pluquet, and R. Wuyts, "Co-evolving code and design with intensional views: A case study," *Computer Languages, Systems & Structures*, vol. 32, no. 2-3, pp. 140–156, 2006.

[72] I. N. O. M. Tanaka T., Hakuta M., "Approaches to making software porting more productive," in *Proceedings of the 12th TRON Project International Symposium*, December 1995, pp. 73–85.

[73] J. Lee and C. Stricker, "Function-based process analysis: A method for analyzing and re-engineering processes," *Knowledge and Process Management*, vol. 4, no. 2, pp. 126–138, 1997.

[74] M. E. L. Ackers, Baxter, "Automated restructuring of component-based software," *CrossTalk-The Journal of Defense Software Engineering*, 2005.

[75] H. Suenobu, K. Tsuruta, and U. Masuda, "Stepwise approach for introducing object-oriented technique at software maintenance stages," in *Systems, Man, and Cybernetics, 1999. IEEE SMC '99 Conference Proceedings*, October 1999.

[76] C. Verhoef, "Towards automated modification of legacy assets," *Ann. Software Eng.*, vol. 9, pp. 315–336, 2000.

[77] D. Wilkening and K. Littlejohn, "Legacy software reengineering technology," in *Digital Avionics Systems Conference*, October 1996, pp. 25–30.

[78] W. Bergey, Smith, "Dod legacy system migration guidelines," carnegie Mellon Software Engineering Institute - Technical Report CMU/SEI-99-TN-013.

[79] T. Mens, T. Tourwé, and F. Muñoz, "Beyond the refactoring browser: Advanced tool support for software refactoring," in *IWPSE*, 2003, pp. 39–44.

[80] H. M. Sneed, "Encapsulation of legacy software: A technique for reusing legacy software components," *Ann. Software Eng.*, vol. 9, pp. 293–313, 2000.

[81] W. W. Song, "Integration issues in information system reengineering," in *COMPSAC*, 1996, pp. 328–335.

[82] L. Tahvildari, K. Kontogiannis, and J. Mylopoulos, "Requirements-driven software re-engineering framework," in *WCRE*, 2001, pp. 71–80.

[83] R. Morrison, D. Balasubramaniam, R. M. Greenwood, G. N. C. Kirby, K. Mayes, D. S. Munro, and B. Warboys, "A compliant persistent architecture," *Softw., Pract. Exper.*, vol. 30, no. 4, pp. 363–386, 2000.

[84] D. Eichman, "Factors in reuse and reengineering of legacy software," repository Based Software Engineering Program Research Institute for Computing and Information Systems University of Houston Clear Lake.

[85] W. C. Chu, C.-W. Lu, C.-P. Shiu, and X. He, "Pattern-based software reengineering: a case study," *Journal of Software Maintenance*, vol. 12, no. 2, pp. 121–141, 2000.

[86] A. Dearle and D. Hulse, "Operating system support for persistent systems: past, present and future," *Softw., Pract. Exper.*, vol. 30, no. 4, pp. 295–324, 2000.

[87] R. Fanta and V. Rajlich, "Reengineering object-oriented code," in *ICSM*, 1998, pp. 238–246.

[88] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, "A study of the documentation essential to software maintenance," in *SIGDOC*, 2005, pp. 68–75.

[89] A. A. Terekhov, "Automating language conversion: A case study," in *ICSM*, 2001, pp. 654–658.

[90] R. Koschke, "Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey," *Journal of Software Maintenance*, vol. 15, no. 2, pp. 87–109, 2003.

[91] E. B. Buss and J. Henshaw, "A software reverse engineering experience," in *CASCON*, 1991, pp. 55–73.

[92] D. Gregor, J. Järvi, J. G. Siek, B. Stroustrup, G. D. Reis, and A. Lumsdaine, "Concepts: linguistic support for generic programming in c++," in *OOPSLA*, 2006, pp. 291–310.

[93] E. Fricke and A. P. Schulz, "Design for changeability (dfc): Principles to enable changes in systems throughout their entire lifecycle," *Systems Engineering*, vol. 8, no. 4, 2005.

[94] P. P. Tommi Mikkonen, "Practical perspectives on software architectures, high-level design, and evolution," in *International Conference on Software Engineering archive, Proceedings of the 4th International Workshop on Principles of Software Evolution*, 2001, pp. 122–125.

[95] K. H. Bennett and V. Rajlich, "Software maintenance and evolution: a roadmap," in *ICSE - Future of SE Track*, 2000, pp. 73–87.

[96] C. D. Rosso, "Continuous evolution through software architecture evaluation: a case study," *Journal of Software Maintenance*, vol. 18, no. 5, pp. 351–383, 2006.

[97] "Xmdrawingarea documentation." [Online]. Available: http://www.vaxination.ca/motif/XmDrawingAr_3X.html

[98] "Xtaddcallback documentation." [Online]. Available: http://linux.die.net/man/3/xtaddcallback

[99] "Xmnexposecallback documentation." [Online]. Available: http://www.vaxination.ca/motif/XmDrawingAr_3X.html

[100] "Expose events documentation." [Online]. Available: http://tronche.com/gui/x/xlib/events/exposure/expose.html

[101] "X event structures." [Online]. Available: http://tronche.com/gui/x/xlib/events/structures.html

[102] "Callback setup example." [Online]. Available: http://www-h.eng.cam.ac.uk/help/tpl/graphics/Motif/mt4

[103] "Xtappaddworkproc documentation." [Online]. Available: http://www.xfree86.org/4.4.0/XtAppAddWorkProc.3.html

[104] "Xtappaddworkproc example." [Online]. Available: http://www.ist.co.uk/motif/books/vol6A/ch-26.fm.html

APPENDIX A
MOTIF RESEARCH

A. Toolkit and Widget Changes

Listed below are the specific Toolkit and Widget class deprecations from Motif 1.2 to 2.1.

Renditions and RenderTables
The XmFontList data type and associated functions are now deprecated, replaced by a new entity called the XmRendition.

VendorShell
The VendorShell has the new resources XmNbuttonRenderTable, XmNlabelRenderTable, and XmNtextRenderTable that replace the deprecated XmNbuttonFontList, XmNlabelFontList, XmNtextFontList resources.

Label
The XmFontList resource is deprecated and is superseded by the XmNrenderTable resource. The same is true for LabelGadget.

MainWindow
The routine XmMainWindowSetAreas() is eprecated. The XmNcommandWindow, XmNmenuBar, XmNmessageWindow, XmNworkWindow, XmNhorizontalScrollBar, XmNverticalScrollBar resources should be set directly using the standard Xt mechanisms.

MenuShell
The XmNbuttonFontList and XmNlabelFontList resources are deprecated, and are replaced by the XmNbuttonRenderTable and XmNlabelRenderTable resources. Also deprecated is the XmNdefaultFontList resource, but there is no replacement XmNdefaultRenderTable resource.

Scale
The XmNfontList resource is deprecated, and replaced with the newer XmNrenderTable resource

Text and TextField
The XmNfontList resource is obsolete and is replaced by the XmNrenderTable resource.

B. Obsolete Symbols

Table VI gives a list of the symbols that were updated from Motif 1.2 to 2.1.

APPENDIX B
MURAL BUG ANALYSIS

The Mural constructor calls `Mural::Init()`. The `init` method initializes a `XmDrawingArea` variable (`draw_area`) which serves as the container for the mural drawing area (see [97]). Then a callback reference is set up on the `draw_area` widget via a call to `XtAddCallback` [98]. The callback event is set to `XmNexposeCallback` [99] which is triggered when the widget is exposed (i.e. when the widget is displayed, see [100]). The callback method is set to `Mural::ExposeCB` which is called when the event is triggered.

The `ExposeCB` method extracts the `XEvent` object [101] from the callback object that is passed to it. From that it has access to the `XExposeEvent` object which has the information it needs to perform its function. First, it checks the number of additional expose events that are queued to call itself - since it is going to tell the Mural to draw it only wants to do this on the last call to itself, ignoring all events until the last one. When the number of queued events is equal to zero, the method will then grab the x/y coordinates of the top left corner of the mural window along with its width and height and call `Mural::Draw`.

After some checks to make sure the drawing area window is set up properly, the `Draw` method checks if the mural is set to "dirty" - meaning the mural has received new values or a new layout and needs to be redrawn. The method then calls `Mural_Client::MuralRedrawNeededCB` (we will address this method later as it is at the center of the problem). The `Draw` method assumes that calling this method will set the appropriate values in a 2D array called `mural_array`. The two dimensions represent the x/y position of each pixel in the mural window. Each cell in the array contains a `mural_array_entry_t` struct defined as:

```
typedef struct
{
  float value;
  short r, o, y, g, c, b, p;
} mural_array_entry_t;
```

The `value` field stores a percentage value used for rendering in anti-aliasing mode (beyond the scope of this analysis). The letter fields stand for the colors red, orange, yellow, green, cyan, blue, purple (respectively). These are the colors available to the ISVis user via the color palette displayed on the main view. When another part of the application sets the color of a `mural_array_entry_t` struct for a particular index it is telling the `Mural` class to draw that color for that pixel. As mentioned above, the `Draw` method assumes these values are set after calling `Mural_Client::MuralRedrawNeededCB`. After calling that callback method[2], the `Draw` method then calls `Mural::DrawMural` which draws the mural based on the values in `mural_array`.

Diagnosis
The symptom of the problem is that all the colors for the mural array entries are still in their initial state after the callback method is called. `Mural_Client::MuralRedrawNeededCB` is a virtual method implemented by any extenders of `Mural_Client` - our extender of interest is the `Scenario_View` class. The `Scenario_View`'s implementation is as follows:

1) Initialize the mural, clearing out any previous values that

[2]Any method in the ISVis code ending in `CB` is usually named as such to denote it is used as a callback for some event.

TABLE VI
MOTIF OBSOLETE SYMBOLS

1.2 Symbol	2.1 Symbol
_XmOSPutenv()	-
_XmGetDefaultTime()	XmeGetDefaultTime()
_XmGetColors()	XmGetColors()
_XmDrawShadows()	XmeDrawShadows()
_XmEraseShadow()	XmeClearBorder()
_XmGetArrowDrawRects()	XmeDrawArrow()
_XmOffsetArrow()	XmeDrawArrow()
_XmDrawSquareButton()	-
_XmDrawDiamondButton()	XmeDrawDiamond()
_XmDrawShadowType()	XmDrawShadows()
_XmDrawBorder()	XmeDrawHighlight()
_XmMoveObject()	XmeConfigureObject()
_XmResizeObject()	XmeConfigureObject()
_XmVirtualToActualKeysym()	XmeVirtualToActualKeysyms()

may be stored in the mural array
2) Register a worker method as a background procedure to be run by X windows. This is done via a call to `XtAppAddWorkProc` (see [102], [103], [104])

The worker method registered to be run in the background is `ScenarioView::redrawMuralWorkProc`. This method increments an iterator to get the next interaction that needs to be drawn to the mural. If the iterator points to an interaction then the mural is updated with the appropriate values for the interaction being examined and the method returns False (0) indicating there are still interactions that need to be drawn. If there are no more interactions to iterate over, the method returns True (1). The iterator is held as a class level variable so after the method returns it can be called again and pick up at the next interaction. This looping is necessary in order to iterate over all the interactions and have them all be drawn in the mural.

The problem lies in the fact that this process never gets executed, so the mural array never gets populated with the values corresponding to the interactions being displayed by the scenario view. For some reason the background process is not calling this worker method. This explains why the mural is blank.

Solution

Our first test was to check if the drawing capabilities of the Mural were functioning properly. We tested the `Mural::MuralDrawHorizontalLine` method and successfully drew several different color lines in the mural window from the scenario view callback method. This was done by using hard coded values for the pixel range and the line color.

Once the drawing functionality was verified further analysis was conducted that led to the realization that `redrawMuralWorkProc` (the method that would update the mural) was not being called. Since the reason for using the background process to call the worker method is not entirely clear[3], it was decided to comment it out and just call the method directly. The following loop was set up in `MuralRedrawNeededCB`:

```
int done = redrawMuralWorkProc();
while(done == False)
{
  done = redrawMuralWorkProc();
}
```

This executed the method we needed but it also created an infinite loop because after it was done iterating over all the interactions it called `Mural::RequestDraw()` which eventually ended up calling `MuralRedrawNeededCB` again. To cater for this we introduced a class level boolean variable called `redrawInProgress`, initialized to false. The body of `MuralRedrawNeededCB` (including the above loop) was wrapped in the following `IF` clause:

```
if(!redrawInProgress)
{
  redrawInProgress = true;
  //code to iterate over the interactions
  ...
  Mural::RequestDraw();
  redrawInProgress = false;
}
```

So the first time `MuralRedrawNeededCB` is called, the body will get executed and the mural array will get updated allowing the mural to be drawn properly. When the interaction iteration is completed and the call is made to draw the mural, the body of `MuralRedrawNeededCB` will not get executed since `redrawInProgress` would be set to `true` at that point. Once the mural draw method returns, `redrawInProgress` is set back to `false` so the method will be executed on the next valid redraw request. This solution draws the mural correctly.

[3]We speculate it has to do with older machines not being able to handle the drawing function as it might have been an expensive process 10+ years ago.

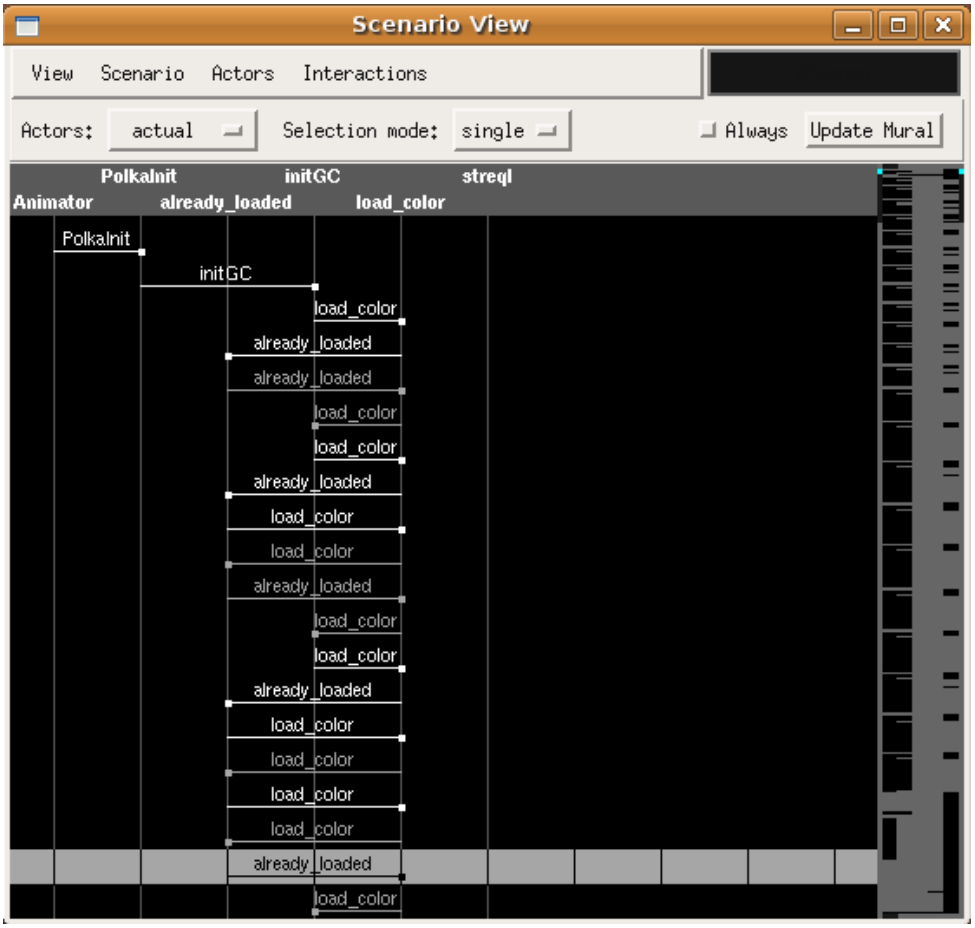

Fig. 7. Mural display for dfs.xml

Validation

Validating that the mural was being drawn correctly was somewhat of an interpretive exercise since we did not have access to the original developer of the program, nor any other user that could reliably say what it should look like for a given set of data. Consequently our judgment of correctness is based solely on our understanding of how the mural should work. The mural window should show a visualization of all of the actors being presented to the user in either the main or scenario view — from here on when we refer to the mural, we mean the mural shown in the scenario view.

After correcting the mural code and opening the scenario view for our test data set, we could see the mural as shown in Figure 7. The mural depicts a visualization of 14,608 interactions. What looks like blocks in the visualization are actually lines stacked on top of each other in succession to represent the interactions shown between the functions in the view. After assigning some colors to different interactions we can see in Figure 8 that the mural successfully recolors itself.[4] Figure 9 shows the result of resizing the mural. We can see that some of the lines are now more precise as to what region of the view they represent. Since the mural has a fixed space to draw in, it makes sense that some of the calculations to turn the position of an actor in its container to its corresponding Y-axis pixel position would overlap given a large data set.

To make one last confirmation we decided to reduce the data set. We cut down the number of interactions read into the scenario view to 20. To do this we had to modify the scenario file we were using. We copied `dfs.xml` and created `dfs-short.xml`. The XML structure was as follows:

```
<boost_serialization ... >
  <this class_id=''0'' ... >
    <programModel class_id=''2'' ... >
      ...
      <scenarios class_id=''15'' ... >
        ...
        <item class_id=''16''>
          <second class_id=''17''>
            ...
            <impl class_id=''19''>
              <trace class_id=''23''>
                <messageTrace>
                   14,608 item (interaction) nodes
                </messageTrace>
```

We deleted all the 14K `item` nodes except for 20. We then updated the two places where the count was hard-coded — the `count` and `numEvents` nodes — to reflect the new number of interactions. Note that the `dfs.trace` and `dfscopy` scenarios

[4]You must click the 'Update Mural' button or the 'Always' checkbox after picking a color for an actor

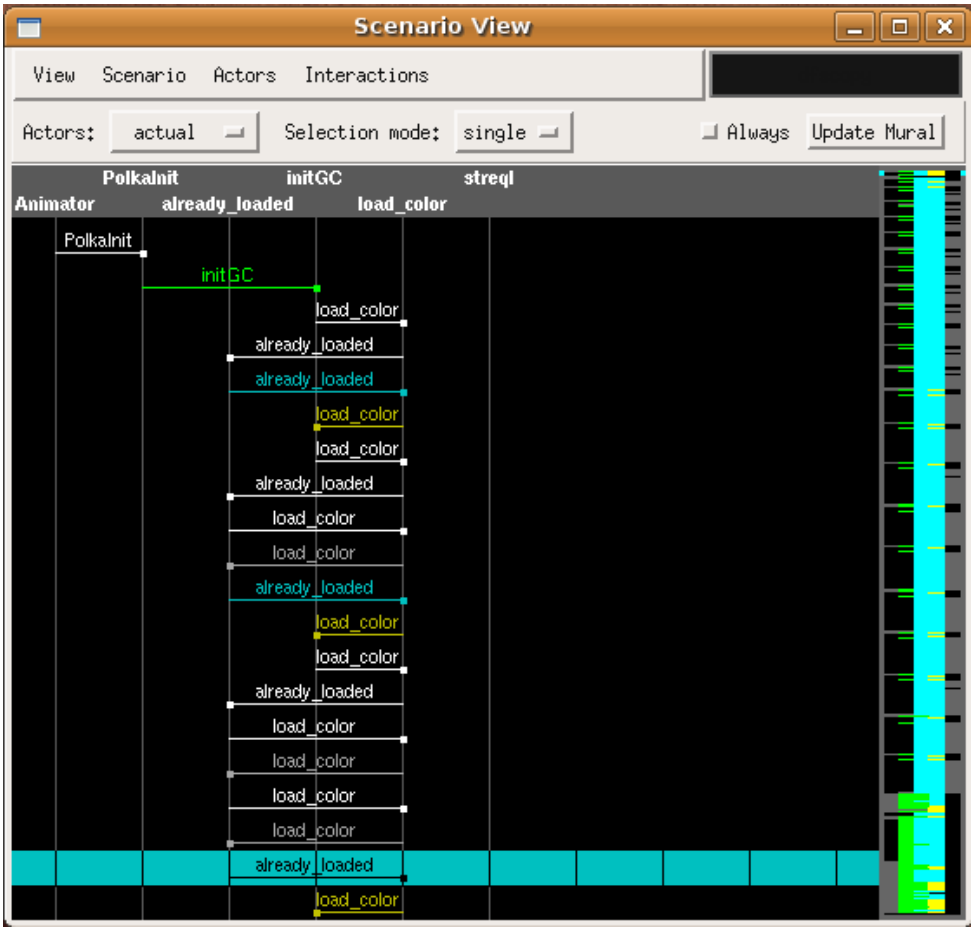

Fig. 8. Mural with colored interactions

are both contained in dfs.xml — we only made this change to `dfs.trace`. Consequently, you can use `dfs-short.xml` to test the shorter list of interactions via `dfs.trace` alongside the full list via `dfscopy`.

The resulting mural is shown in Figure 10 — the figure shows a shot with the same interactions coloring as in Figure 8. From this we were able to see a clearer representation of the interactions being shown. Since none of the lines overlap, we were able to verify that the colors were working on the lines we expected them to be on and that the correct number of lines were being drawn.

Appendix C
Refactoring Example

This section describes a simple example of a refactoring we implemented in ISVis to enhance it's readability. To start off with, we looked for a portion of the code that we knew executed properly and that could be easily tested. Another criteria was that it would be as close to "stand-alone" as possible - meaning the candidate would have minimal (if any) references to global variables or methods external to its class. The code pertaining to the color buttons on the main view fit these conditions perfectly. Additionally this code originally caught my attention for a refactoring upon recognizing the Duplicated Code bad smell [30]. We applied the Extract Method refactoring in the following steps:

Step 1:

```
XrmValue bg = X_Application::ColorRmValue(``white'');
Widget defaultButton =
  XtVaCreateManagedWidget(
    ``default'',
    xmPushButtonWidgetClass,
    canvas,
    XmNshadowThickness,
    2,
    XmNbackground,
    X_Application::ColorValue(``white''),
    XmNforeground,
    X_Application::ColorValue(``black''),
    NULL);

  XtAddCallback(
    defaultButton,
    XmNactivateCallback,
    (XtCallbackProc)(&MainView::setSelectedDefaultCB),
    (XtPointer)this);
```

We identified the block of code that was being repeated for each color button. The text highlighted red is the only code that varies per block. This code was originally in `MainView()` lines 241-322, repeated 6 times for each color button shown in the main view color palette.

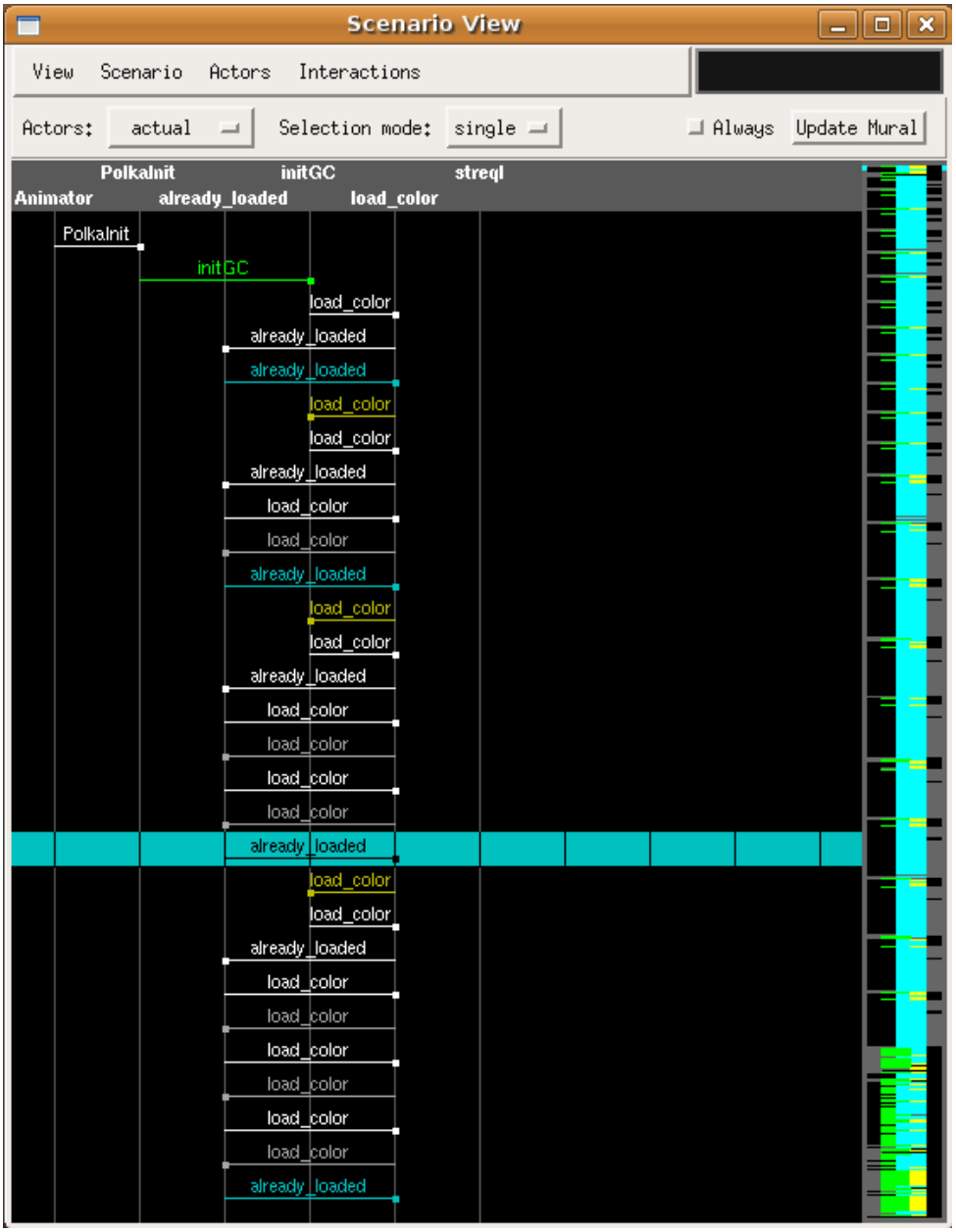

Fig. 9. Mural resized

Step 2:

```
Widget XMainView::CreateColorButton(
  const char * color,
  const char * buttonName,
  XtCallbackProc callBackProc,
  XtPointer callbackMatch)
{
  XrmValue bg = X_Application::ColorRmValue(``white'');
  Widget button =
    XtVaCreateManagedWidget(
      buttonName,
      xmPushButtonWidgetClass,
      canvas,
      XmNshadowThickness,
      2,
      XmNbackground,
      (Pixel) *((Pixel *)bg.addr),
      XmNforeground,
      X_Application::ColorValue(``white''),
      NULL);

  XtAddCallback(
    button,
    XmNactivateCallback,
    callBackProc,
    callbackMatch);

  return button;
}
```

We extracted the common block of code into a method called `CreateColorButton` and parameterized anything that needed to vary per method call (i.e. the red text from Step 1).

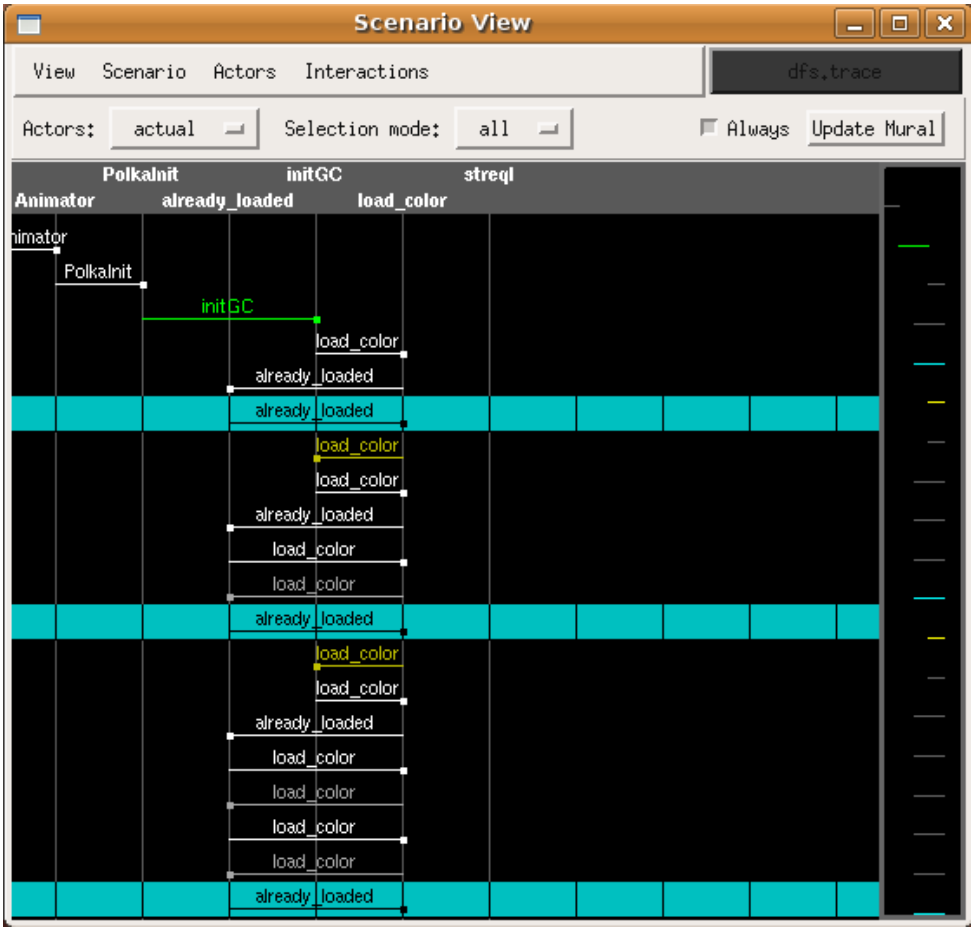

Fig. 10. Mural showing 20 interactions

Step 3:

```
Widget defaultButton =
  XMainView::CreateColorButton(
    ``white'',
    ``default'',
    &canvas,
    &MainView::setSelectedDefaultCB,
    this);
```

We replaced the 6 blocks of code from Step 1 with 6 method calls to the new method (only one is shown, but all the others are the same calls with different colors assigned to their corresponding buttons).

Step 4:

```
Widget XMainView::CreateColorButton(
  const char * color,
  const char * buttonName,
  Widget *parent,
  XtCallbackProc callBackProc,
  XtPointer callbackMatch)
{
  XrmValue bg =
    X_Application::ColorRmValue(``white'');
  Widget button =
    XtVaCreateManagedWidget(
      buttonName,
      xmPushButtonWidgetClass,
      *parent,
      XmNshadowThickness,
      2,
      XmNbackground,
      (Pixel) *((Pixel *)bg.addr),
      XmNforeground,
      X_Application::ColorValue(``white''),
      NULL);

  XtAddCallback(
    button,
    XmNactivateCallback,
    callBackProc,
    callbackMatch);

  return button;
}
```

We extracted the body of the new `CreateColorButton` method to a method of the same name in a new class called `XMainView`. To do this, we added one additional parameter (shown in blue) to the method: the parent (owner) of the button. Previously the parent was a global variable in the `Main_View` class (the `canvas` variable).

Step 5:

```
Widget MainView::CreateColorButton(
  const char * color,
  const char * buttonName,
  XtCallbackProc callBackProc,
  XtPointer callbackMatch)
{
  return XMainView::CreateColorButton(
    color, buttonName, &canvas, callBackProc, this);
}
```

We replaced the body of `MainView`'s `CreateColorButton` with a simple call to `XMainView`'s method of the same name.

APPENDIX D
TEST CASES

While fixing the display related issues, we kept track of the test cases we used to verify the GUI functionality as we made code changes. This was a completely manual process. For each test we tracked the following information: test name, test description, input to generate the condition being tested, expected output, and test status (pass/fail). A subset of that information is listed in Table VII. The full list can be found at http://www.cc.gatech.edu/morale/tools/isvis/2009/ISVisTestPlan.xls.

TABLE VII
CURRENT STATUS OF TEST CASES

Test	**Status**
Testing application startup	Pass
Opening a saved session	Pass
Open a scenario view	Pass
Open a second session	Fail
Yellow button	Pass
Green button	Pass
Cyan button	Pass
Blue button	Pass
Purple button	Pass
White button	Pass

APPENDIX E
X WINDOWS REFERENCES

Figures 11, 12, 13 and 14 list the frequencies of calls made to X Windows specific classes.

APPENDIX F
COMPILE ERRORS

Table VIII shows the initial errors/warnings encountered when trying to compile ISVis.

Library	Package	Class	ISVis Reference	Frequency
X11	Instrinsic	ArgList	xapp.C	1
			xapp.H	1
			Total	**2**
X11	X	Drawable	xapp.C	10
			xapp.H	10
			Total	**20**
X11	Instrinsic	Pixel	scenario_view.C	3
			xapp.C	6
			xapp.H	2
			Total	**11**
X11	Instrinsic	Widget	actor_list_view.H	2
			io_shell.C	3
			io_shell.H	4
			main_view.H	22
			mural.H	9
			scenario_view.C	43
			scenario_view.H	36
			view.H	5
			xapp.C	15
			xapp.H	14
			Total	**153**
X11	Instrinsic	WidgetClass	xapp.H	1
			Total	**1**
X11	X	Window	actor_list_view.H	1
			main_view.H	1
			mural.C	1
			mural.H	1
			scenario_view.H	1
			xapp.C	4
			xapp.H	3
			Total	**12**

Fig. 11. X Windows References

X11	Instrinsic	XtAppMainLoop	xapp.H	1
			Total	**1**
X11	Instrinsic	XtCallbackProc	actor_list_view.C	1
			io_shell.C	1
			scenario_view.C	39
			xapp.H	1
			Total	**42**
X11	Instrinsic	XtPointer	actor_list_view.C	2
			io_shell.C	2
			io_shell.H	2
			main_view.H	38
			mural.H	6
			scenario.H	69
			view.H	5
			xapp.C	3
			xapp.H	4
			Total	**131**
X11	Instrinsic	XtWorkProc	scenario_view.C	1
			xapp.C	1
			xapp.H	1
			Total	**3**
X11	Instrinsic	XtWorkProcId	scenario_view.H	1
			xapp.C	2
			xapp.H	2

Fig. 12. X Windows References - continued

			Total	**5**
X11	Xlib	XImage	xapp.C	2
			xapp.H	3
			Total	**5**
X11	Intrinsic	XtAppContext	xapp.C	1
			xapp.H	2
			Total	**3**
X11	Xlib	Display	xapp.C	3
			xapp.H	2
			Total	**5**
X11	Xlib	GC	xapp.C	1
			xapp.H	1
			Total	**2**
X11	Xlib	Visual	xapp.C	1
			xapp.H	1
			Total	**2**
X11	X	Colormap	xapp.C	2
			xapp.H	1
			Total	**3**
X11	X	Cursor	xapp.C	1
			xapp.H	1
			Total	**2**
X11	Xlib	XColor	xapp.C	8
			xapp.H	1
			Total	**9**
X11	Xlib	XFontStruct	xapp.C	1
			xapp.H	1
			Total	**2**
X11	Xresource	XrmValue	scenario_view.C	1
			xapp.C	4
			xapp.H	1
			Total	**6**

Fig. 13. X Windows References - continued

Xm	CascadeB.h	xmCascadeButtonWidgetClass	scenario_view.C	6
			xapp.C	1
			Total	**7**
Xm	ToggleB.h	xmToggleButtonWidgetClass	scenario_view.C	2
			xapp.C	2
			Total	**4**
X11	Xlib	XEvent	actor_list_view.C	1
			actor_list_view.H	1
			io_shell.C	1
			main_view.H	1
			mural.H	3
			scenario_view.C	1
			scenario_view.H	1
			view.C	2
			view.H	2
			xapp.C	1
			Total	**14**
			Grand Total	**445**

Fig. 14. X Windows References - continued

TABLE VIII
INITIAL COMPILE ERRORS WITH ISVIS

Source File	**Occurrences Found**
Actor.c	*line 51*: Error: A typedef name cannot be used in an elaborated type specifier..“actor.H” *line 402*: Error: Incorrect access of a member from const-qualified function. (2)
Actor_list-view.c	*line 51*: Error: A typedef name cannot be used in an elaborated type specifier..“view_manager.H” *line 122*: Error: Cannot return const DotActor* from a function that should return DotActor*.“program_model.H” *line 447*: Error: i is not defined.“actor_list_view.C”, (11)
Disk_file.c	Compilation aborted, too many RW Error messages
Event_writer.c	*line 59*: Error: Could not open include file “event_writer.H”. *line 85*: Error: The function “initTimestamp” must have a prototype
Main_view.c	*“program_model.H”, line 122*: Error: Cannot return const DotActor* from a function that should return DotActor *“main_view.C”, line 84*: Warning: String literal converted to char* in initialization. (23)
Program_model.c	*“program_model.H”, line 122*: Error: Cannot return const DotActor* from a function that should return DotActor*. *“program_model.C”, line 1028*: Error: Incorrect access of a member from const-qualified function. (12) Compilation aborted, too many Error messages.
Scenario.c	*“program_model.H”, line 122*: Error: Cannot return const DotActor* from a function that should return DotActor*. *Error*: Cannot use const Trace* to initialize Trace*. *Error*: Cannot use const Scenario* to initialize Scenario*. *Error*: Cannot use const TraceImpl* to initialize TraceImpl*. *Error*: Cannot use const InteractionImpl* to initialize InteractionImpl*.
Scenario_view.c	*“scenario_view.C”, line 85*: Warning: String literal converted to char* in initialization. (47) *“scenario_view.C”, line 329*: Error: i is not defined. (16) Compilation aborted, too many Error messages
Static_analyzer.c	*“static_analyzer.C”, line 87*: Error: Only a function may be called. (6) *“static_analyzer.C”, line 171*: Error: Incorrect access of a member from const-qualified function. (2) *Error*: Only a function may be called. (3) Compilation aborted, too many Error messages.
Trace.c	*“program_model.H”, line 122*: Error: Cannot return const DotActor* from a function that should return DotActor*. *Error*: i is not defined. (15) *Error*: Cannot use const Trace* to initialize Trace*.
Trace_analyzer.c	*Error*: Could not open include file <ctoken.h>. *“program_model.H”, line 122*: Error: Cannot return const DotActor* from a function that should return DotActor*. *Error*: Cannot use RWCString to initialize int. *Error*: Only a function may be called. (3)
View_manager.c	*Error*: Could not open include file <bstream.h> *Error*: Cannot return const DotActor* from a function that should return DotActor* *Error*: Cannot use std::basic_ifstream<char, std::char_traits<char>> to initialize int. *Error*: The operation “int >> ViewManager” is illegal. *Error*: Incorrect access of a member from const-qualified function. *Warning (Anachronism)*: Formal argument 2 of type extern “C” void(*)(int) in call to std::signal(int, extern “C” void(*)(int)) is being passed void(*)(int). *Warning (Anachronism)*: Formal argument 2 of type extern “C” void(*)(int) in call to std::signal(int, extern “C” void(*)(int)) is being passed void(*)(int).
Xapp.c	*Warning*: String literal converted to char* in initialization. (32) *Error*: best is not defined. (4) *Error*: Cannot assign const char* to char*.