Assessing the impact of changes at the architectural level: A case study on graphical debuggers

D. Scott McCrickard Gregory D. Abowd
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
{mccricks,abowd}@cc.gatech.edu

Abstract—**Understanding the high-level structure of a software system is useful for determining the impact of proposed changes. In this paper, we investigate techniques for recovering the high-level structure, or software architecture, for cases in which no trusted architectural description exists. We then demonstrate how a scenario-based approach to architectural evaluation can be used to document the impact of proposed changes on a software system. Architectural recovery and impact analysis are demonstrated using a case study comparing two graphical debuggers, `xdbx` and `ups`.**

Keywords—**Software architecture, architectural recovery, impact analysis, scenario-based evaluation**

I. Introduction

When faced with the responsibility of maintaining an unfamiliar piece of software, it is often useful to have an understanding of that software's overall structure. Armed with such overview information, it is then desirable to assess the impact that proposed changes to the system will have on that structure. This impact analysis will help in determining the expected cost of the change and will inform decisions as to whether a restructuring is in order.

Designers have known for a long time that it is the structure of a software system that matters. Recently, this implicit understanding has emerged explicitly as the study of *software architecture* [9], [3]. A software architecture consists of high-level computational elements (called components) and the interactions between them (called connections). By presenting the software at a high level, an architectural description communicates the general behavior of a software system in an understandable and informative manner, even for those not familiar with the software.

While it is usually the case that behind a successfully deployed software system there lurks a well-defined architecture, the definition is often in the heads of a select few and not always available to those who must maintain the system. It is this ironic circumstance that we wish to explore in this paper. We will demonstrate by means of a case study how an architectural representation can be the basis for analyzing the impact of changes on a software system. When no architectural representation is available, support must be provided to help produce the representation.

In this paper, we will present an evaluation of two graphical debuggers using a method for evaluation called the Software Architecture Analysis Method (SAAM). The emphasis in this paper is not on the definition of the SAAM method. Rather, we are focusing on two aspects of SAAM that are essential in software maintenance. The first is a way to determine the overall architecture of a system using visualization techniques and static analysis tools. The second focus is to demonstrate how the scenario-based approach to SAAM can provide documented impact analysis of proposed changes to a software architecture. For concreteness, we provide a simple but realistic example of the application of SAAM to compare two graphical debuggers, `xdbx` and `ups`. The results of this case study show not only how SAAM can be used to do impact analysis on a single system but also how it enables a comparative evaluation of systems in a common domain.

In Section II, we provide a brief overview and history of SAAM. In Section III, we briefly describe the case study on graphical debuggers before discussing techniques for determining the architectures of our candidate systems in Section IV. In Section V, we introduce scenarios that are the basis for the impact analysis of the two debuggers. In Section VI we summarize the results of the SAAM evaluation exercise, and in Section VII we discuss the contributions of this work.

II. Overview of SAAM

The Software Architecture Analysis Method (SAAM) provides a systematic scenario-based approach to analyzing programs at the software architecture level [6], [5], [1]. A SAAM evaluation involves the following activities.

1. **Describe candidate architectures**: The architectural description must indicate the high-level components that comprise the system as well as the connections between the components. The components include active components which manipulate data and control the program execution flow, and passive components that only store data. The interactions include the flow of control and the flow of data.
2. **Develop scenarios**: Scenarios are well-defined tasks that reflect both expected and anticipated uses of a system. We can view scenarios as concrete descriptions of otherwise abstract qualities of a system. For example, a scenario that describes porting a system to a new hardware platform is a concrete realization of the abstract quality of portability. The number and variety of scenarios determined for a SAAM evaluation depends entirely on the goals and resources of the evaluation team. In this paper, we will provide a small number of scenarios for evaluating graphical debuggers.

3. **Perform scenario evaluations**: Evaluations determine if each scenario is supported directly or indirectly. A scenario is directly supported if it can be executed by the system without any further modification. If modifications to the system are required in order to execute the scenario, then the scenario is indirectly supported. The required modifications are then described in terms of the previously defined architectures. That is, the impact of the scenario on the architecture is described by indicating which components are affected and the level of effort involved in making that change. A summary of all scenario evaluations then provides a documented impact analysis on the architecture and can be used to compare two candidate architectures against each other or to inform any decision to restructure an individual system to better support anticipated changes.

III. Case study overview

Debugging is an important part of the software development process. Many debuggers take advantage of the windowing and interface capabilities of graphical workstations. It may be difficult for a programmer starting a project to determine which graphical debugger is best for the current job. A SAAM evaluation can help in this process by providing evaluations for a number of scenarios in different domains of interest. By selecting the scenarios that are most relevant, a user can determine the debugger that best suits the job at hand. If an important scenario is not directly supported, the evaluation results should dictate which components of the system must be altered, and the user should be able to go into the code and change them. Thus, it is important to show which code belongs in each component.

In this study, we perform a SAAM evaluation on two popular public-domain graphical debuggers: `xdbx` and `ups`. `xdbx`[1] provides a graphical front end to the dbx debugger found on most Unix systems. It allows mouse input for many dbx debugging commands and supplies scrollable display windows for the dbx output and the program source code. `ups`[2] is a self-contained graphical debugger that relies on direct manipulation and graphical feedback. The user can annotate the existing code with interpreted code to set breakpoints, manipulate variables, and print data.

IV. Architectural description

First in our evaluation we must describe the debuggers in common syntactic terms to a common level of detail. To accomplish this, it is necessary to establish component and connection boundaries within the code. Since we expect that our scenarios will address fairly high-level issues, the architecture should be described at a high level as well.

A SAAM analysis reveals the components in an architecture that require changes in order to accommodate new functionality. Since users of our analysis may wish to implement these changes, we felt it necessary to provide an accurate architecture-to-code mapping to facilitate the implementation task. To obtain the desired mapping we used reverse engineering tools.

Reverse engineering explores methods for extracting information from software systems. Some reverse engineering tools can be used to derive high-level architectural information from the source code. Of particular interest is RMTool, the Reflexion Model Tool [8]. RMTool resolves differences between the user's high-level concept of a function call graph and the actual model resulting from calls in the code. To obtain an outline of the high-level model, we use RunView, a program understanding and debugging tool that we are developing here at Georgia Tech. RunView provides a high-level view of the source code and allows a user to create architectural components based on execution information. Together, RunView and RMTool can facilitate the discovery and verification of a software architecture.

The remainder of this section describes in detail the tools and methods used to obtain and verify the architectural descriptions for `xdbx` and `ups`.

A. Identifying boundaries with RunView

The size of the programs and our lack of knowledge about them made identifying the architectural boundaries difficult. While the documentation that came with the `xdbx` and `ups` distributions was helpful, it did not contain the complete architectural description at our desired level of detail. We needed tools that would help us map the large amounts of code to a small number of architectural components.

RunView is a program visualization tool designed for understanding and debugging large programs. It provides the user with an overview of the programs by replacing program elements with graphical objects (see Figure 1). The code display window shows each file in the program as a column and each function in a file as a block within the column. The size of the block corresponds to the size of the function. By highlighting files or functions of interest with different colors, the user can run the program and visually trace their execution. The execution display window shows the current call stack at each function call with the right portion of the window providing a zoom of the left. Each horizontal line in the display represents the call stack at some point in time, with each function in the call stack appearing as a line segment. To semantically connect the windows, each line segment has the same color as the corresponding function in the code display. At any time, the user can highlight a function representation in either view to determine when the function was called, which functions called it, and which functions it called.

To use RunView to help identify an architecture, we assigned each module a color and set out to highlight each function with the color of its module. Our initial assignments were based on similarities in file name, but as we noted certain patterns in the execution, we altered the color

[1] The source and documentation are available from ftp://ftp.x.org/R5contrib/

[2] The source and documentation are available from ftp://unix.hensa.ac.uk/pub/misc/unix/ups/

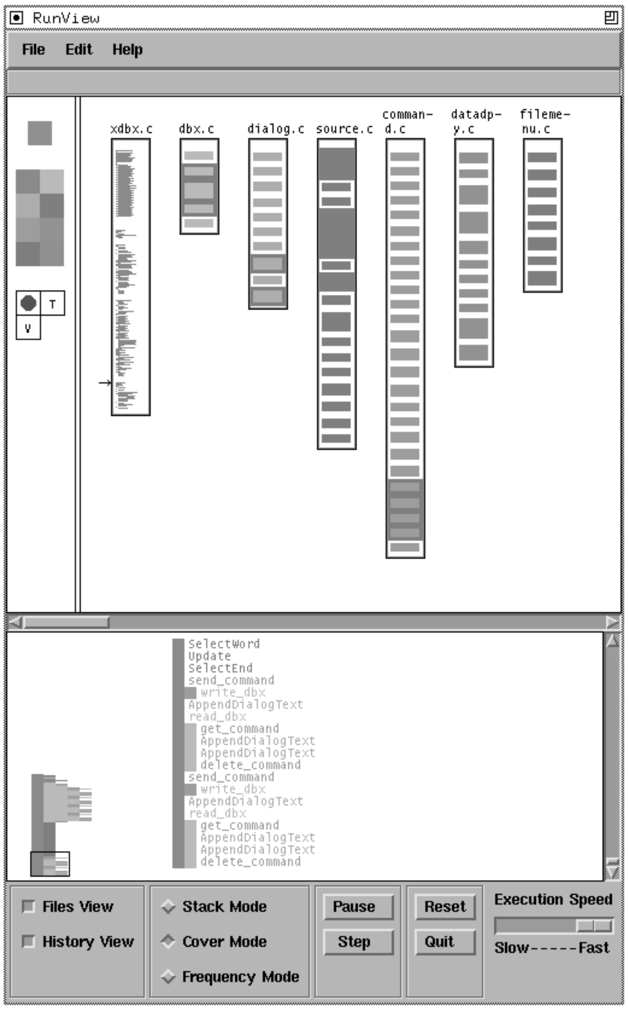

Fig. 1. Screenshot of RunView executing on a portion of `xdbx`. The code display view shows the files that make up `xdbx`, and the execution display view shows the call stack history. The functions in the xdbx.c file are expanded to trace each line, while the other files only trace by function call.

mapping. For example, we combined files into one module if they are call each other often. After limiting the number of components, we used RMTool to verify our assignments.

B. Verifying boundaries with RMTool

RunView facilitates the definition of an architecture by focusing on a single execution. However, to verify the correctness of the boundaries defined by RunView requires additional testing. No matter how many times the program is executed, we cannot be sure that all possible executions have been included. To verify the correctness of our model we used RMTool.

RMTool, the Reflexion Model Tool, compares the high-level model provided by the user to the actual calls made in the code. The resulting Reflexion model is a box-and-arrow view of the components and connections (see Figure 2). The solid lines show connections that are verified as correct. Dashed lines show connections in the high-level model with no corresponding function calls, while dotted lines show existing connections that are not reflected in the high-level model. The number next to each connection line is the number of function calls between the components.

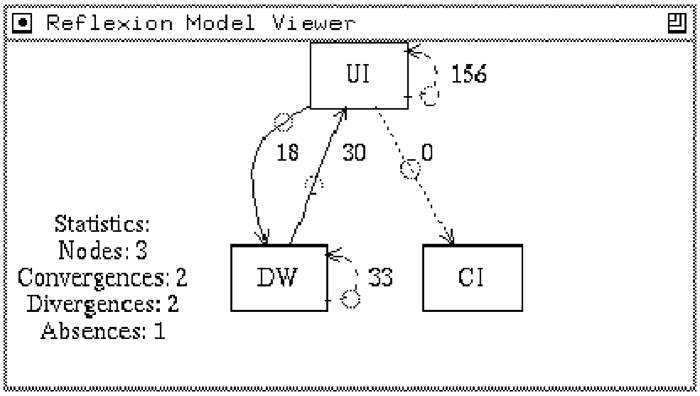

Fig. 2. RMTool display of the `xdbx` code modules. The nodes represent modules and the lines represent connections between the modules.

To verify the architecture established using RunView, we fed into RMTool the file-to-component mapping, the connections between components, and the source code information. When RMTool showed differences between our model and the actual function calls, we altered our model (testing it using RunView) and ran RMTool again, repeating this process until the model was correct.

Several problem arose when using these tools. Since RunView and RMTool only analyze function calls and not data flow, they are only useful for determining active components and connections. We had to deduce the passive ones on our own by examining the code for `xdbx` and `ups`. In addition, it is unclear how graphical widget callbacks should be handled since they are not called directly by another function. Neither tool makes the distinction between callbacks and other function calls; thus, this sort of connection information had to be determined by examining the source code. However, these tools saved time and increased our confidence in the architectural models.

C. `xdbx` *modules*

The active modules in `xdbx` are the user interface, the debugger wrapper, the dbx debugger, the code interpretation module, and the executing process. The passive modules are the code, the executable, and the dbx configuration file.

See Figure 3 for an overview of the modules and their interactions.

- UI: This module controls all aspects of the `xdbx` user interface. UI lays out buttons, windows, and scrollbars in the appropriate regions, distributes user actions to the DW and CI modules, and displays information obtained from these modules about code, execution, and variables.
- DW: The debugger wrapper module interacts with the text debugger (usually dbx) via a Unix two-way pseudo-terminal device. DW sends commands to dbx based on requests from the UI module, and it parses the dbx output and translates it into information that the UI can understand.

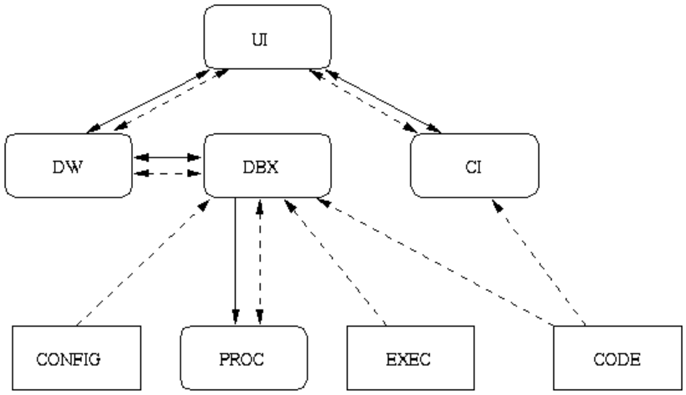

Fig. 3. `xdbx` modules and connections. The solid lines are control connections, and the dashed are data. Modules with rounded corners are active, and the others are passive.

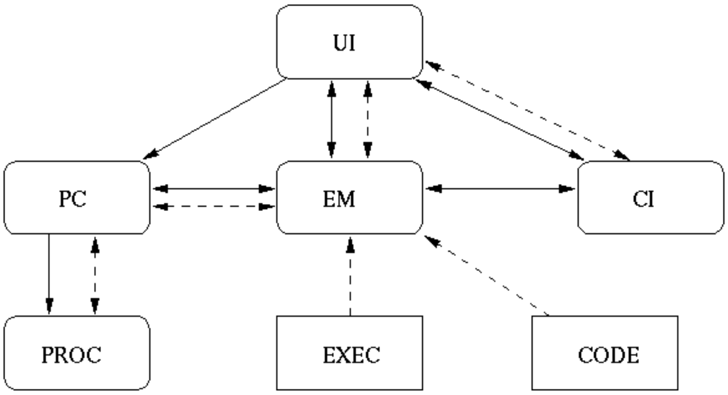

Fig. 4. `ups` modules and connections. The solid lines are control connections, and the dashed are data. Modules with rounded corners are active, and the others are passive.

- CI: The code interpretation module controls the listing and loading of code for `xdbx`.
- DBX: The dbx debugger performs text debugging on the selected executable. Dbx can read core files provided by the user or monitor executing programs. For the purposes of this analysis, we assume that the dbx source code is unavailable, but that the entire module may be replaced with another debugger. Since the ability to replace one version of dbx with another is considered one of `xdbx`'s greatest assets, we will treat it as a module that can be replaced but not modified out of fairness to `ups`.
- CONF: The dbx configuration file contains a list of initial commands to be run by the debugger before the debugging process begins.
- PROC, EXEC, and CODE: These modules are specified by user either at run time or via the UI module. Only the code files are accessed by any module other than dbx. Otherwise, they interact with dbx (and each other) as they did in `ups` with the process controller. PROC can be either an active process or a core file (which is essentially a snapshot of a process at some point in time). EXEC is the executable file.

Interactions of modules are as follows:

The user interacts with the program through the user interface module.

The UI module determines the command corresponding to the button presses and mouse actions. It sends the command to the debugger wrapper.

The debugger wrapper receives events from the user interface and translates them into debugger commands for dbx. It sends these commands to the dbx module via a pseudo-terminal. It receives the response from the same terminal, translates the received information into the form desired by the user interface, and sends the translated information to the user interface.

When the dbx module receives a command from the dbx wrapper, it performs the command and access PROC (either the active process or core file, whichever is available). When an executable is first accessed, dbx loads in the necessary information from the executable file. At times, dbx accesses information from the code files as well.

The code interpreter performs all directory and file accesses for the user interface. When the code from a file is requested, the user interface passes the file name to the code interpreter, which accesses the file and passes the code to the user interface.

D. `ups` *modules*

The active modules in the `ups` debugger are the user interface (UI), process controller (PC), execution monitor (EM), code interpreter (CI), and active process (PROC). The passive modules are the executable file (EXEC) and the code files (CODE).

See Figure 4 for the outline of the module interactions.

- UI: The user interface module lays out the windows, buttons, menus, and scrollbars for `ups`, monitors user actions from the keyboard and the mouse, and displays information about code, execution, and variables to the user in the display window regions.
- PC: The process controller starts and stops the executing process, examines the existing process state for information such as current function, stack state, and variable values, and passes requested information to the UI module. When available, the process controller reads and interprets core files.
- EM: The execution monitor reads the code and executable files, scanning the symbol table and building a partial list of files, functions, and variables. At run time, the execution monitor loads in-depth information as desired by the user or when needed by other modules.
- CI: The code interpreter maintains a list of breakpoints and interpreted code entered by the user for each breakpoint. When a breakpoint is reached, the code interpreter executes the interpreted code and sends the output to the UI.
- PROC, EXEC, and CODE: These modules are specified by user either at run time or via the UI module. Only the code files are accessed by any module other than dbx. Otherwise, they interact with dbx (and each other) as they did in `ups` with the process controller.

PROC can be either an active process or a core file (which is essentially a snapshot of a process at some point in time). EXEC is the executable file.

Interactions of modules are as follows:

The user interacts with the program through the user interface module.

The user interface module determines which function corresponds to each user action and activates the appropriate module. If buttons pertaining to the process execution (like start, step, or stop) are pressed, the process controller is called. When information about the executable or the current execution (such as the files used in compilation of the executable) is desired, the execution monitor is queried. When breakpoints are added or edited, the code interpreter is instructed to store the information. The user interface is instructed by the execution monitor and code interpreter to display information to the user in one of the display windows.

Upon queries from the user interface or execution monitor, or upon arriving at a breakpoint, the process controller reads information from the PROC module (either the executing process or the core file, whichever is present) and passes it on to the execution monitor.

The code interpreter interprets code inserted by the user. At each breakpoint, the execution monitor queries the code interpreter to see if any code needs to be executed. If the interpreted code has output, the code interpreter passes it to the user interface for display. When breakpoints are set, the code interpreter must inform the execution monitor of the location in the code so it can translate it into a location in the executable and pass the information to the process controller.

The execution monitor reads information from the executable file and code files whenever it is needed by the user interface, process controller, or code interpreter. It acts as an intermediary for all communication between the process controller (which knows about locations in the executable) and the code interpreter (which knows about code line numbers). The execution monitor determines the information that the user wants to see and pass this information to the user interface.

The executable and code files are static objects that are read by the execution monitor. In addition, the executing process uses the information in the executable. If a core file is present, the user may choose to use the information present in it instead of an executing process.

V. Scenarios to consider

The next step in our evaluation is to identify important characteristics for debuggers and to instantiate these characteristics into testable scenarios. This section outlines scenarios for three classes of debugger characteristics: portability, debugging support, and usability.

A. Portability

Often programmers utilize a number of different hardware platforms but still desire a consistency in debugger interfaces. For these users, it is important for a debugger to work in a variety of situations. We summarize these concerns with the following two scenarios.

- **P1** Change to a platform that has dbx: Programmers often port their software to new hardware platforms. Their debugger should port as well to provide interface consistency. In this scenario, we will assume that the target platform has all of the required windowing libraries as well as a copy of the textual debugger dbx.
- **P2** Change to a platform with no textual debugger: This scenario is identical to the previous one except that we assume no textual debuggers are available on the platform.

B. Debugging support

The main purpose of a debugger is to locate execution-time errors; however, errors that are more likely to occur may be of greater concern than others. For example, if a program was known to contain a large number of array data structures, the user may be particularly concerned with identifying writes outside of array bounds. Our SAAM evaluation should help users choose a debugger that will assist in the identification of certain classes of bugs.

From existing literature on the debugging of programs [4], [7], we gleaned three classes of errors that are common in many programming languages: assignment, memory, and termination errors.

- **D1** Assignment: This class of bugs results in an incorrect value assigned to a variable. These bugs occur when initialization statements are missing and when incorrect formulas are used. A debugger should allow the user to monitor selected variables as the program executes.
- **D2** Memory: These errors result from improper use of the memory by the application. Examples include failure to allocate memory for a data structure, illegal writes outside of allotted memory, and allocated space with no pointers referencing it. In this scenario, we focus on checking for illegal writes outside of allocated memory. We assume that the version of dbx used by `xdbx` does not support memory checking, though some commercial versions of dbx can do this checking.
- **D3** Termination: We use termination errors to include loops and recursion that terminates incorrectly or does not terminate at all; for example, an infinite number of iterations through a loop or an incorrectly terminating series of recursive calls. In this scenario, we wish to use a debugger to discover a non-terminating series of recursive calls.

C. Usability

Usability reflects the efficiency and effectiveness of an interface. The ideal interface should combine power with ease of use; the user should be able to perform any desired command with minimal effort. The following scenarios echo two widely-accepted interface design strategies.

- **U1** Immediate availability of common commands: Commands that are frequently used should be accessible to the user with minimal effort. An analysis of debugger command usage [2] revealed several commands that were by far the most frequently used when debugging an executing program: stop at a given source line, stop in a particular function, execute the next source line, step into the next procedure to execute, continue the current execution, print a value, and redo the last command. The availability and use of these commands should be obvious at all times to encourage and facilitate their use.

- **U2** Accelerators for user-selected commands: While it is important to offer easy and obvious access to widely used commands, depending on the task at hand and the user's preferences, other commands may prove to be more usefully and necessary. The debugger should provide the ability to create shortcuts to these commands.

VI. Scenario evaluations

After completing the scenario development and architectural descriptions, the next step is to evaluate the scenarios with respect to the architectures. To reflect the difficulty of the changes, we established the following scenario impact factors.

1. Change a configuration file or variable and restart: These changes require no coding.
2. Modify data, data types, or data structures within the code and recompile: These changes require only minor modifications within the existing code. The general structure and functionality of the code remains the same.
3. Change the functionality of the source code for the modules: These changes impact large amounts of code in one or more modules. The implementer must have an understanding of the existing code and a plan for implementing the new code.
4. Change the software architecture: These changes require additions and deletions to the architecture's modules and connections. One or more modules must be written from scratch.

Each indirectly supported scenario receives the appropriate impact factor. Table 1 summarizes the results of our evaluation; the rest of the section explains how the results were derived.

A. Portability

P1 Change to a platform that has dbx

While `ups` has been ported to a large number of hardware platforms, including the SPARC, the Sun 3, the Sun 386i, the DECstation, and the Intel 386, the use of tracing mechanisms leads to platform dependency and, according to documentation included with the `ups` distribution, a port to a new platform is likely to take several weeks. The modules that would require change are the process controller and the execution monitor.

Since `xdbx` runs on top of the dbx debugger, it can run on any platform that supports the X11 windowing system and has a working copy of dbx. Slight modifications to the dbx wrapper may be required to account for differences in syntax between the versions of dbx.

#	scenario name	xdbx	ups
P1	platform with dbx	2	3
P2	platform w/o a debugger	4	3
D1	assignment errors	-	-
D2	memory errors	4	3
D3	termination errors	-	-
U1	available commands	-	-
U2	accelerators	1	3

TABLE I
A summary of the SAAM scenario evaluations.

`ups` has an impact factor of 3 for this scenario, while `xdbx` has an impact factor of 2.

P2 Change to a platform with no textual debugger

The `ups` situation is unchanged, and the impact factor remains at 3. `xdbx` will require either a restructuring of the architecture to more closely match that of `ups`, or the creation of a textual debugger for the new platform. The impact factor for `xdbx` is 4.

B. Debugging support

D1 Assignment errors

In `ups`, the user can maintain a list of variables of interest. `ups` updates the values of the variables at every stopping point, allowing the user to step through the code and visually monitor the variable.

`xdbx` provides a "display" function that prints the value of selected variables at every stopping point. By stepping through the code, the user can monitor the variable during execution.

Since `ups` and `xdbx` both allow the user to print the values of variables at breakpoints, both are said to directly support this scenario.

D2 Memory errors

`ups` only discovers memory errors if the program crashes because of it; for example, if a value were assigned to an unallocated memory location and a bus error or segmentation violation occurred. To isolate these errors, the user would have to augment the execution monitoring module to check the location of each memory write and ensure that it is correct.

`xdbx` relies on dbx to monitor of the executing process and check for events such as memory errors. Most available versions of dbx only discover these errors if the program crashes because of it. Isolating these errors would require an additional module designed to monitor the executable and verify memory write locations.

Since `ups` requires changes to the source code in a module, it receives a 3. Since `xdbx` requires the addition of a module, it receives a 4.

module	P1	P2	D1	D2	D3	U1	U2	Level
UI	-	-	-	-	-	-	-	-
DW	X	-	-	-	-	-	-	1
DBX	-	-	-	-	-	-	-	-
CI	-	-	-	-	-	-	-	-
CONF	-	-	-	-	-	-	X	1
PROC	-	-	-	-	-	-	-	-
EXEC	-	-	-	-	-	-	-	-
CODE	-	-	-	-	-	-	-	-
new	-	X	-	X	-	-	-	2

TABLE II
A SUMMARY OF THE INTERACTION LEVELS FOR `xdbx`.

module	P1	P2	D1	D2	D3	U1	U2	Level
UI	-	-	-	-	-	-	X	1
PC	X	X	-	-	-	-	-	2
EM	X	X	-	X	-	-	-	3
CI	-	-	-	-	-	-	-	-
PROC	-	-	-	-	-	-	-	-
EXEC	-	-	-	-	-	-	-	-
CODE	-	-	-	-	-	-	-	-
new	-	-	-	-	-	-	-	-

TABLE III
A SUMMARY OF THE INTERACTION LEVELS FOR `ups`.

D3 Termination errors

If the program output gives no hint of the runaway recursion, it is expected that the user will eventually send an interrupt to the executing program. When the program halts, `ups` displays the current call stack. Upon careful observation of the stack, the user would notice the repeated pattern of recursive function calls.

In `xdbx`, the user must request to see the stack when the program is interrupted. Again, upon viewing the stack, the repeated pattern of function calls would indicate a problem with the recursion.

`ups` and `xdbx` both directly support this scenario.

C. Usability

U1 Immediate availability of common commands

Both `ups` and `xdbx` provide all but one of the most-used commands in their button menus. The only command not provided is the redo command. Since each button is visible in the same location at all times, repeating one of these commands would require clicking the same button multiple times. Since both interfaces provide easy and highly visible access the most-used command, we say that both interfaces directly support this scenario.

U2 Accelerators for user-selected commands

`xdbx` allows the user to create aliases for a frequently used sequence of commands. By adding the alias to the configuration file and restarting `xdbx`, the user can execute the new command with only a few keystrokes.

While `ups` provides easy access to a large number of commands, there is no mechanism for creating new commands. To add a new command would require changes to the user interface module. The user must add a new interface device and appropriate callbacks that execute the command and process the resulting data when the device is accessed.

Since `xdbx` requires only a change to the dbx configuration file, it receives a 1. `ups` requires changes to the UI module, so it receives a 3.

D. Assessment of Interactions

Two or more indirect scenarios interact if they require changes to the same component. Scenario interaction measures the separation of concerns within the architecture and between the scenarios. A high degree of interaction merits a closer look at the architectures and scenarios: perhaps the architectural descriptions need to be more fine-grained, or perhaps overlapping scenarios need to be combined.

Table 2 summarizes the interaction levels for `xdbx`. Although few changes are required to the modules, several scenarios called for the creation of new modules. This results from our decision to consider DBX to be a module unavailable for modification: we stick by that decision for reasons outlined previously.

Table 3 summarizes the interaction levels for `ups`. Recall that scenarios P1 and P2 are identical for `ups`, but these scenarios differ drastically for `xdbx`. We consider it important to emphasize this difference by including both scenarios.

Note that scenarios P1, P2, and D2 all require changes to the execution monitor. This module is central in the reading, storing, and processing of data. In future evaluations, these multiple functionalities of the EM module could be divided among several smaller modules.

VII. CONCLUSIONS

This paper investigated techniques for recovering the software architecture of a system when no trusted architectural description is available. We showed how a scenario-based approach to architectural evaluation can be used to document the impact of proposed changes to a software system.

Rather than focus solely on the traditional static box-and-arrow display, we used alternative visualizations to provide different perspectives on software systems. A dynamic display of the executing program provides an animated view of limited portions of the program at a time. Graphical representations of files and functions compress large systems in a relatively small space. These techniques, combined with the Reflexion model comparison of user models to source models, proved useful in recovering unfamiliar architectures.

We showed the usefulness of architectural discovery as part of two scenario-based architectural evaluations. Tabulating the results from these evaluations demonstrated the impact on the systems from proposed changes.

References

[1] Gregory Abowd, Jonathan Engelsma, Luigi Guadagno, and Okokon Okon. Architectural analysis of object request brokers. *Object Magazine*, March 1996. In press.

[2] Evan Adams and Steven S. Muchnick. Dbxtool: A window-based symbolic debugger for sun workstations. *Software – Practice and Experience*, 16(7):653–669, 1986.

[3] David Garlan and Mary Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, pages 1–39. World Scientific Publishing Company, 1993.

[4] John D. Gould. Some psychological evidence on how people debug computer programs. *International Journal of Man-Machine Studies*, 7:151–182, 1975.

[5] Rick Kazman, Gregory Abowd, Len Bass, and Paul Clements. Scenario-based analysis of software architecture. *IEEE Software*, 1996. In press.

[6] Rick Kazman, Len Bass, Gregory Abowd, and Mike Webb. Saam: A method for analyzing the properties of software architectures. In *Proceedings of the 16th International Conference on Software Engineering*, pages 81–90, 1994.

[7] F. J. Lukey. Understanding and debugging programs. *International Journal of Man-Machine Studies*, 12:189–202, 1980.

[8] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proceedings of the ACM SIGSOFT '95 Conference*, pages 18–28, 1995.

[9] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

Appendix

This section contains the module-to-source mappings for `xdbx` and `ups`. The mappings are in the form used by RMTool: brackets enclose a function or file and the module to which it is mapped.

First we give the mapping for `xdbx`.

```
[ file=dialog.c mapTo=UI ]
[ file=command.c mapTo=UI ]
[ file=windows.c mapTo=UI ]
[ file=signs.c mapTo=UI ]
[ file=datadpy.c mapTo=UI ]
[ file=source.c mapTo=UI ]
[ file=utils.c mapTo=UI ]
[ file=handler.c mapTo=UI ]
[ file=filemenu.c mapTo=UI ]
[ file=xdbx.c mapTo=UI ]

[ file=dbx.c mapTo=DW ]
[ file=calldbx.c mapTo=DW ]
[ file=signals.c mapTo=DW ]
[ file=parser.c mapTo=DW ]
[ file=regex.c mapTo=DW ]
[ file=command.c function=send_command mapTo=DW ]
[ file=command.c function=get_command mapTo=DW ]
[ file=command.c function=delete_command mapTo=DW ]
[ file=command.c function=insert_command mapTo=DW ]

[ file=source.c function=LoadFile mapTo=CI ]
[ file=source.c function=LookUpFileTable mapTo=CI ]
[ file=source.c function=BuildLinePos mapTo=CI ]
[ file=filemenu.c function=changeDir mapTo=CI ]
[ file=filemenu.c function=InList mapTo=CI ]
[ file=filemenu.c function=ScanDir mapTo=CI ]
[ file=filemenu.c function=DisplayMenuFile mapTo=CI ]
```

Next we give the mapping for `ups`.

```
[ file=ui_*.c.c mapTo=UI ]
[ file=va_*.c.c mapTo=UI ]
[ file=obj_*.c.c mapTo=UI ]
[ file=reg.c.c mapTo=UI ]
[ file=text*.c.c mapTo=UI ]
[ file=output.c.c mapTo=UI ]
[ file=printf.c.c mapTo=UI ]

[ file=proc.c mapTo=PC ]
[ file=core.c mapTo=PC ]
[ file=stack.c mapTo=PC ]
[ file=trun_*.c mapTo=PC ]
[ file=exec.c mapTo=PC ]

[ file=ci_*.c mapTo=CI ]
[ file=src*.c mapTo=CI ]

[ file=st_*.c mapTo=EM ]
[ file=as_*.c mapTo=EM ]
[ file=var_*.c mapTo=EM ]
```