

Representing Unit Test Data for Large Scale Software Development

Joseph A. Cottam, Joshua Hursey, Andrew Lumsdaine
Indiana University
Open System Laboratory
Bloomington, IN USA
{jcottam, jjhursey, lums}@osl.iu.edu

Abstract

Large scale software projects rely on routine, automated testing to gauge progress towards its goals. The diversity and quantity of these tests grow as time and project scope increase. This is as a consequence of both experience and expanding audience. It becomes increasingly difficult to interpret testing results as the testing suites multiply and diversify. If interpretation becomes too difficult, testings results could become ignored all together. Visualization has proven to be an effective tool to aid the interpretation of large amounts of data. We have adapted visualization techniques based on small multiples to communicate the health of the software project across several levels of abstraction. The collective set of techniques we refer to as the SeeTest visualization schema. We applied this visualization technique to the Open MPI test results in order to assist developers in the software release cycle. Through the visualizations, developers found a variety of surprising mismatches between their data and their intuitions. This exploration did not involve collecting any data not already being collected, merely presenting it in manner that better supported their needs. In this paper, we detail the development of the representation we used and give more particular analysis of the insights gained by the Open MPI community. The techniques presented in this paper can be applied to other software projects.

CR Categories: D.2.9 [Software Engineering]: Management—Life Cycle; D.2.9 [Software Engineering]: Management—Software Quality Assurance (SQA); H.5.2 [Information Systems]: Information Interfaces and Presentation—Screen Design; H.5.3 [Information Systems]: Group and Organization Interfaces—Web-based Interaction; I.3.8 [Computing Methodologies]: Computer Graphics—Applications

Keywords: project management, testing, visualization, MPI

1 Introduction

Software projects require routine testing to assess progress towards their goals. Directly analyzing the raw testing data is sufficient for relatively small projects with simple testing requirements. Large software projects require more extensive testing due to software complexity and a more expansive set of project goals. Due to the resultant size of the testing data it becomes difficult (some may argue impossible) to directly analyze the raw testing data.

To effectively handle the considerable amount of testing data, large software projects must rely on automated testing toolkits that provide data gathering, storage and reporting features. Most testing toolkits provide effective solutions for data gathering by providing a full featured testing harness, and data storage by providing an archival medium (e.g., database). Unfortunately it is rare to find a testing toolkit that provides effective solutions for data reporting. Solutions provided by many toolkits settle on the ability to list test results overlaid with a basic color coding. These solutions represent only a minor improvement from directly interpreting the raw testing data.

Effective reporting solutions must minimally address issues of timeliness, interactivity and variable specificity. If reporting takes too much time, then results will be outdated before they can be acted upon, therefore results must be timely. Interactivity is required in large data sets because one view rarely satisfies all needs. The ability to navigate between views based on current issues or the specific concerns of a person is covered by interactivity. Limiting data to just that of concern to a particular person or problem is the purview of specificity. A good reporting system needs to incorporate representations to communicate both overall status as details of particular areas.

Many test reporting systems for large-scale projects address timeliness and interactivity to a sufficient degree, but fail to provide effective ways to represent the overall status of the project. They instead provide only detail views. This has serious implications for the stability and validity of large-scale software.

In this paper, we present a visualization schema to address issues of interactivity and variable specificity. This schema is referred to as the SeeTest schema, which we believe is applicable to any large-scale software development situation. We have implemented the SeeTest schema in support of an independent software project, Open MPI [Gabriel and et al. 2004], as they approached a major release. The resulting extension to their testing environment provided timely assessment of the overall health of their software library, helping them improve their testing and address issues before finishing their release cycle. We present an informal evaluation of the schema and its implementation conducted with the Open MPI developers. We deem our design successful based on its ability to assist those developers in discovering new information in their existing test data.

The structure of this paper is as follows. Section 2 describes work related to this work. Section 3 describes the MPI Testing Tool (MTT), our data source, including data collection and historical reporting methods. Section 5 describes the visualization schema designed with goal refinements for the pr. The paper then continues with a discussion of the integration of the developed schema as the MTT Visualization Extension. Section 6 describes the results of our informal evaluation. Section 8 discusses our findings and describes our future development goals extending from this work.

2 Related Work

There exists a large field of research for improving testing techniques, but only a few of these ideas seem to have an impact on the software engineering process used to develop and maintain large software systems [Osterweil 1996; Harrold 2000].

The Open MPI project was chosen as the target for this visualization effort because of its active development community, commitment to project goals, and large amount of testing data. The Open MPI community assessed many different testing toolkits before creating the MPI Testing Tool (MTT) [Hursey et al. 2007]. Due to the loyalty of the Open MPI community to the MTT project we used MTT as a foundation for our visualization efforts. There are other testing toolkits available; for a discussion of these toolkits see the MTT project web site [Open MPI 2008].

The roots of many software visualization tools can be drawn back to SeeSoft [Eick et al. 1992]. SeeSoft provided a line-oriented program visualization environment suitable for combining many types of statistics with a representation of the program source-code. SeeSoft was developed as a general purpose software visualization tool for both developers and managers of developers. Its flexibility, coupled with a target audience well versed in the structure of source code, influenced the decision to use a highly literal representation. Lines of code are never abstracted farther than lines on the screen, and often the literal text is still used. Similarly, the file and directory divisions are preserved as grouping structures.

Since SeeSoft, tools more tightly targeted on metrics-based evaluation [Pinzger et al. 2005], debugging [Panas et al. 2005; Reiss and Renieris 2005], software structure [North 1998; Voinea et al. 2005] or program state [Reiss and Renieris 2003; Meyer et al. 2006] have been created. These more focused tools employ more abstract representations of the software than SeeSoft, focusing more on the particular goals of their restricted audiences and tasks.

As mentioned, SeeSoft provides the conceptual roots for many software visualization tools. SeeSoft predates the formalization of unit testing strategies. It is straightforward to adapt the SeeSoft strategy to unit testing results. The TARANTULA system [Jones et al. 2002] is one implementation employing this strategy. TARANTULA uses line-level color coding to show either test results or testing coverage. They demonstrate that visual representations of testing data aids in isolating particular code lines that contribute to faults. This highly detailed technique is best applied in a focused debugging cycle, but is ill suited to understanding the overall health of a software system across multiple environments or through time. Therefore it is too narrowly focused for the tasks that this paper discusses.

More general, are the standard visualizations provided by many unit testing frameworks [JUnit 2008; NUnit 2008]. Rather than focusing on the line-level contributions they represent results in terms of test conditions passed, failed and aborted. Such tools typically provide a hierarchical arrangement of test results grouped into lists. Such a representation does not scale to groups with large numbers of unit tests which often cause the loss of the hierarchical arrangement leaving the user with a long list representing testing results. This is not conducive to global assessments of the overall state of the software.

Toolkits designed to manage software distributions often some include support for test visualization. KitWare's cDash [Kitware Inc. 2008a] is typical of the reporting options available, many of which are based either conceptually or technologically on the Dart toolkit [Kitware Inc. 2008b]. The textual options form the main communication medium of this family of test information management toolkits. These textual displays are based in tabular data summarizing test results at various levels, with annotations for test

types, platforms and configurations. Such tables often include basic color encoding of individual cells to provide at-a-glance status reports, with textual values providing details. Graphical options provide the limited ability to view single test changes through time (usually on a variation on a line graph). These representations represent a good start to understanding the state of a complex distribution. However, their representation techniques are often too shallow, displaying only a single level of detail at a time (e.g. tests accumulated or each unit test separately). Aggregation is only shown as text, which has implications for the degree of data density that can be achieved.

3 Data Set and Graphics Environment

Our design and evaluation was done in conjunction with the Open MPI development team. Critical scientific applications (e.g., molecular dynamics, weather and fusion simulations) push the boundaries of the most capable High Performance Computing (HPC) systems and support software. These applications trust in the validity and stability of HPC support software, such as communication libraries. The de-facto standard communication library for HPC is the Message Passing Interface (MPI) standard [Message Passing Interface Forum 1993]. Open MPI is an implementation of the MPI standard committed to the goals of software portability, stability, and high performance at a variety of scales. The MPI standard specifies an API for programming parallel applications on large scale HPC systems. The Open MPI development community consists of 24 contributing organizations from around the world, and a sizable user community. Open MPI is a large software project with a code base measured in hundreds of thousands of lines of code, and an extensive set of tests.

The Open MPI project is dedicated to the goal of portability over a wide range of architectures, operating systems, runtime environments, networks, compilers, and system sizes. The resulting testing space requires the Open MPI project to distribute testing tasks into the dispersed development community. The Open MPI developers can only provide validity and stability assurances to applications by acting on accurate and insightful interpretation of the testing data. The MPI Testing Tool (MTT) [Hursey et al. 2007] is a flexible testing toolkit specifically designed for testing MPI implementations across multiple organizations and multiple environments. The MTT supports this effort by providing the development community with the tools to test the software, report the results, and present aggregate and detailed views of these results. The MTT also provides a basic solution to the data reporting task through the MTT Reporter. The MTT Reporter provides both high level summaries and low level details, but lacks effective representations for aggregate queries used for project goal assessment. The Open MPI community's assessment of its progress towards development goals is through the resultant testing data. The visualization presented in this paper aims to assist the Open MPI developers to understand their test data, thereby improving the support they provide to critical applications.

3.1 MTT Architecture

The MTT is comprised of three logical components, namely the MTT Client, MTT Database, and MTT Reporter. The MTT Client is a testing harness deployed to a testing location. It is used to coordinate testing activities at that location. This coordination is driven by a testing specification provided by the developer. After receiving a specification, the MTT Client downloads appropriate software, builds the software, builds the tests with that software, runs the tests, and reports detailed information back to the MTT Database.

The MTT Database is the central repository and archive of all testing data. It is realized as a PostgreSQL database hosted by Indiana University. The testing data is normalized on a field by field basis and indexed for quick access and aggregation. The MTT Database accumulates about 2.8 million testing results a month, and has been archiving testing data since November 2006.

The MTT Reporter is a web application that allows the testing community to interactively explore the accumulated testing data stored in the MTT Database. The primary task of the MTT Reporter is to aid a developer in characterizing problem areas highlighted by the testing. Figure 1 shows a typical summary level view of the MTT Reporter.

The primary mode of interaction with the MTT reporter is to “drill-down” into data by specifying continually more restrictive query constraints until only the desired testing data remains. The MTT Reporter provides the developer with a detailed view of the how the test was built and the environment in which it was run (i.e., the ‘testing characteristics’). The testing characteristics assist the developer in recreating the failure conditions for a specific test or a group of tests.

Even though the MTT Reporter can be used to display aggregate testing data, users have found it difficult to interpret these aggregate results in the tabular format in which they are presented. The goal of the MTT Visualization Extension is to improve the aggregate reporting aspect of the MTT Reporter by applying visualization techniques to this aggregate data. The success of the visualizations will be discussed in Section 6 and it is evaluated relative to the developers ability to discern previously obfuscated information in the existing data.

3.2 MTT Reporter User Roles

The MTT Reporter provides answers to general questions about the state of the Open MPI code base, and helps developers recreate failure conditions when problems occur. As a result the MTT Reporter attracts a variety of users each with a set question to answer or task to achieve. Classifying these users into a set of roles allows us to better assess the applicability of the MTT Visualization Extension. There are three main user roles that interact with the MTT Reporter:

- **Release Manager:** These users are concerned with assessing the state of the project as a whole. They typically wish to see all of the test data in order to extract knowledge about testing coverage and the stability of the code base (e.g. what fails that once passed). Release Managers are responsible for driving the software release schedule of the Open MPI project.
- **Organization Manager:** These users are concerned with the contributions of one or more platforms from a specific organization. Even though they constrain their view to a single organization, they wish to answer the same questions as the Release Manager with respect to that organization. Additionally these users may prioritize certain combinations of testing elements or testing dimensions over others (e.g., Operating system, network, compiler, bitness) to ensure sufficient testing where critical to their organization.
- **Platform Manager:** These users are concerned with the stability of a single platform within an organization. Though these users share the same general goals as the Organization Manager, they tend to be more focused on tracking down individual problems highlighted by the testing.

The MTT Reporter is arguably most useful for Platform Managers, since it has been successful in assisting users with describing in detail the testing characteristics under which problems occurred.

Submission Information	
Username Organization Time of submission	Host Platform Text Description
System Information	
Operating System Name Operating System Version Bitness	Architecture Endianness
Build Information	
Compiler Name Compiler Version Compiler Options Configuration Options Test Name Build Output	Compiled Bitness MPI Branch MPI Branch Revision Test Suite Build Success/Fail
Runtime Information	
Environment Variables Resource Manager Number of Processes (np) Success/Fail Test Run Output Performance Data	Command line arguments Process Launcher Networks Used Fail mode (Time Out/Skipped) Test Duration

Table 1: A selection of the testing characteristics tracked by the MTT Client and used as aggregation dimensions for the MTT Reporter.

The MTT Reporter has proven slightly less assistive to the other user roles, as the representation of the high level aggregations tend to be hard to interpret.

The goal of the MTT Visualization Extension is to provide a useful representation for each of the above listed user roles. However, we focused more on the currently under served Release and Organization Managers than on the Platform Managers. The MTT Visualization Extension is not intended as a replacement for the existing MTT Reporter, as the MTT Reporter has proven most useful for the Platform Managers. As part of our analysis of the visualization in Section 6 we will include discussion on how individuals in each of the above roles perceived the visualization.

3.3 MTT Testing Data

For every test, the MTT Client must take care to accurately report all of the testing characteristics (e.g., environment variables, software used to build the test, compile time and runtime options). These testing characteristics are used by developers to recreate the environment when debugging problem areas. Each testing characteristic detail represents a dimension of aggregation for the MTT Reporter. This allows a user to ask questions about the testing of a specific aspect (e.g., 64 bit GNU compiler builds) across the entire data set. A selection of these dimensions of aggregation are provided in Table 1.

3.3.1 MTT Test Suites

Open MPI developers have collected and maintained an extensive repository of test suites used by the MTT for routine testing. Each test suite contains one or more individual tests. The test suite may focus on tests for MPI standard compliance, or application support. Application support test suites allow the routine testing of typical real world applications, which may expose subtle problems in even the most standard compliant MPI implementation.

MTT Reporter

All phases
 MPI install
 Test build
 Test run

Date range:
 Hardware: Show ▾

Org: Show ▾
 OS: Show ▾

Local username: Hide ▾
 MPI name: Show ▾

Platform name: Show ▾
 MPI version: Show ▾

[\[Reset form\]](#) [\[Start over\]](#)
Summary
[Detail](#)
[Performance](#)
[\[Preferences\]](#)
[\[Advanced\]](#)

Current time (GMT): 2008-04-01 20:40:06

Date range (GMT): 2008-03-31 20:40:06 - 2008-04-01 20:40:06

Absolute date range: [Create permalink](#)

Phase(s): MPI install, Test build, and Test run

Relative date range: [Create permalink](#)

Number of rows: 5

#	▲Org▼	▲Platform name▼	▲Hardware▼	▲OS▼	▲MPI name▼	▲MPI version▼	MPI install		Test build		Test run				
							▲Pass▼	▲Fail▼	▲Pass▼	▲Fail▼	▲Pass▼	▲Fail▼	▲Skip▼	▲Timed▼	▲Perf▼
1	iu	IU BigRed	ppc64	Linux	ompi-nightly-trunk	1.3a1r18049	4	0	13	0	1981	753	18	266	0
2	iu	IU BigRed	ppc64	Linux	ompi-nightly-v1.2	1.2.6rc3r17884	4	0	10	0	2968	25	18	13	0
3	iu	IU Odin	x86_64	Linux	ompi-nightly-trunk	1.3a1r18049	15	0	54	0	13679	47	24	0	0
4	iu	IU Odin	x86_64	Linux	ompi-nightly-v1.2	1.2.6rc3r17884	4	0	16	0	5232	8	24	0	0
5	iu	IU Sif	x86_64	Linux	ompi-nightly-trunk	1.3a1r18049	1	0	6	0	0	0	0	0	0
Totals							28	0	99	0	23860	833	84	279	0

Figure 1: MTT Reporter summary level view example.

Individual tests within a test suite have expected outcomes, and any deviation from these expectations are considered failures. A unit test may be considered failing in a variety of ways including:

- The crash of one or more processes (i.e., segmentation fault)
- Timeout exceeded (e.g., deadlock state)
- Incorrect result (e.g., incorrect API implementation)
- Appreciable performance degradation

Additionally the MTT may decide to skip a test if the requirements for running that test are not met by the testing characteristics. For example if a test requires a working MPI-I/O interface, and the MPI implementation was compiled without this portion of the API then these tests would be skipped in routine testing as the requisite API is not present.

At a high level users typically group process crash, timeout, and incorrect result failures together to determine the pass/fail ratios in the aggregate data. Distinguishing between these three types of failure is important for Platform Managers, as it is an indicator to the type of problem they must investigate. However, the purposes of our visualization we do not distinguish between these failure types, better serving those least served by the existing MTT Reporter (i.e., Release and Organizational Managers).

4 Stencil

We employed the Stencil visualization environment as our graphics engine. Stencil is a declarative domain-specific language focused on transforming streams of data tuples into graphic representations (the abstract ideas are presented as a Tuple-Space-Mapper in [Cottam and Lumsdaine 2007]). In its current iteration, the Stencil environment is developed over the Piccolo scene-graph library [Bederson et al. 2004]. Stencil is in an early development phase.

The Stencil visualization system operates as a declarative visualization environment. Visualizations are created by describing a collection of rules composed of ‘layers’ that group elements with a common visual derivation and ‘legends’ that transform incoming data tuples into visual elements. Such a rule collection written in the Stencil language is called a ‘stencil’. Layers may be inter-related, and tuples from multiple data sources can be combined. The Stencil language focuses on describing the transformation of the raw data into the desired visual elements. Where the Stencil language itself does not provide sufficient tools to describe the transformation, calls to arbitrary Python code may be made. To generate an image in batch-mode, the Stencil program is invoked from the command line, with the desired stencil and a listing of data sources. Output options are specified as command-line switches.

We chose the Stencil visualization environment because of its declarative definition format and the ability to output both a visual and a textual description of the graphic. The declarative definition allows us to rapidly prototype and refine the visualizations. Separating the definition of the visualization from the visualization engine allows us to maintain a history of idea branches without the incumbencies of a regular visualization library. This is similar to how SQL is separate from the database engine, allowing the user to modify and maintain historical queries. The textual representation makes it possible to computationally reason about the image without needing complex image processing.

5 Visualization Design and Integration

The goal of the MTT Visualization Extension is to aid the Open MPI developers in accurately assessing the testing results and degree of testing coverage on a continual basis. Such assessments are required by large software projects to assess how well they are meeting their goals. Historically, the Open MPI development community communicated the testing coverage needs for a software re-

lease with hand-made graphics on a whiteboard. When employed, these graphics explored the intersection of multivariate testing characteristics. Typically the testing characteristics of platform, compiler, and test suite were made significant.

A platform was represented by the architecture, native bitness, and operating system. A compiler was identified by the vendor name, compiled bitness (e.g., compiling the library in 32 bit on a 64 bit machine), and compiler version. Unit tests are referenced by their enclosing test suite. The degree of passing is determined at the test suite level. In the original graphic schema, the foundation of the image was the intersection of these three variables with platform and compiler forming the major axes and the cells heavily annotated with additional testing information. Additional testing information explored in the grid cells expressed test suites, configuration options, runtime options, network interfaces, compiler versions and number of processes. Since the graphic space was severely constrained, only a subset of the value combinations were selected for representation. Further, since the image was static in time and hard to update, important high-level assessments were hard to approach such as, “How did the changes from yesterday effect the stability of the project?” or, “What compilers have we tested with in the last week?”

5.1 Visualization Schema Design

Moving beyond these hand-made graphics, the MTT Visualization Extension is designed to improve timeliness, interactivity, and specificity. Improved timeliness is achieved since graphics are generated from the most current testing data. The hand-made approach took substantial effort to generate and was thus constantly out of date. Our automated system can receive data from database queries and takes seconds to update. This is advantageous because graphics can be generated for a development meeting with the most up-to-date data. Large batches of graphics are often generated the night before a meeting. These batches include a variety of comparisons at various levels of aggregation. Even though these graphics can be generated dynamically the nightly batches give developers time for analysis before the meeting.

Specificity and interactivity go hand-in-hand. Typically only one hand-made graphic was created when preparing for a software release which represented all testing configurations. As such, all of the desired testing configuration data was layered onto this single graphic. This resulted in a general purpose representation that was hard to interpret. We are able to improve the readability of the graphics and keep the desired graphics findable by simplifying the visual representation to just a few variables (increasing specificity of the graphics), but linking them together (providing interactivity). The remainder of this section details the development of the graphic elements that comprise the visualization.

5.1.1 Reference System

We were free to create a new set of graphic metaphors when we began creating the visual schema. To harness the Open MPI developers familiarity with the previously described hand-made graphics, we use elements from the hand-made graphics as a basis of our automated ones. We retain the grid composition, representing the test results in the grid cells. An example of the final visualization can be seen in Figure 2 with corresponding raw data in Table 2.

We are able to simplify the data encoded in any single grid cell because we can efficiently construct multiple versions of the graphic. We identified the test suite pass/fail rate as the variable of primary interest. Rather than combining all test suites, we split the test suites and give each its own graphic in the shared intersection. We fo-

cused our experimentation on the test suite pass/fail encoding since it needs to be prominent and is the only variable being encoded in the grid space.

5.1.2 Background

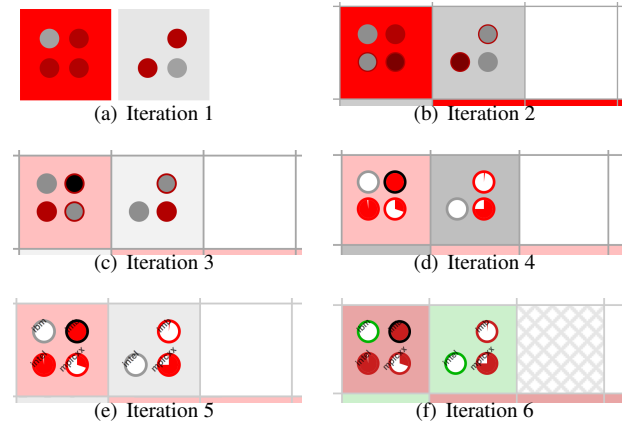


Figure 3: Iterations of the encoding of test suite in grid cells over synthetic data. Each image above shows the same data. From left to right, each image shows a **Trivial not complete-pass**, **Trivial complete-pass**, and a **not-applicable (no-data was always encoded as white)**. Other suite markers were chosen to show the evolution of the central glyphs.



Figure 4: Example states of the central glyph in the final iteration. The border is used to disambiguate the near-complete pass/fail combinations from the complete combinations.

Early in the discussions of the graphic content, it was apparent that one collection of tests was a precondition for all other suites. The **Trivial** test suite was designed so that failure in it would essentially guarantee failure in all other tests. Other test suites were designed with more orthogonality in mind, where the **Trivial** test suite was designed as a pre-condition for other tests. Failure in the **Trivial** test suite undermines the validity of the results in all other suites. Its special status led us to give it a unique representation relative to the other test suites, encoding it in the background color. The fundamental question, at this level of aggregation, was “Did any of the tests in the **Trivial** test suite fail?” instead of the more common “How many of the tests in this test suite fail?” As such, we used a binary encoding where gray represents all tests passed (i.e., *complete-pass*), and red indicates that at least one test failed (i.e., *not complete-pass*).

Our initial prototypes used high saturation colors, but we eventually moved to a much lower saturation values to keep the background subdued (approximately 75% transparency on a white background). The relative proportion of background color to foreground glyphs and the proximity of background color regions to each other keeps the background visible, despite these relatively low saturation values (see Figure 3, especially 3(a) and 3(e) for examples of background values used). Since the Stencil framework is entirely

ompi-nightly-trunk : 1.3a1r18049

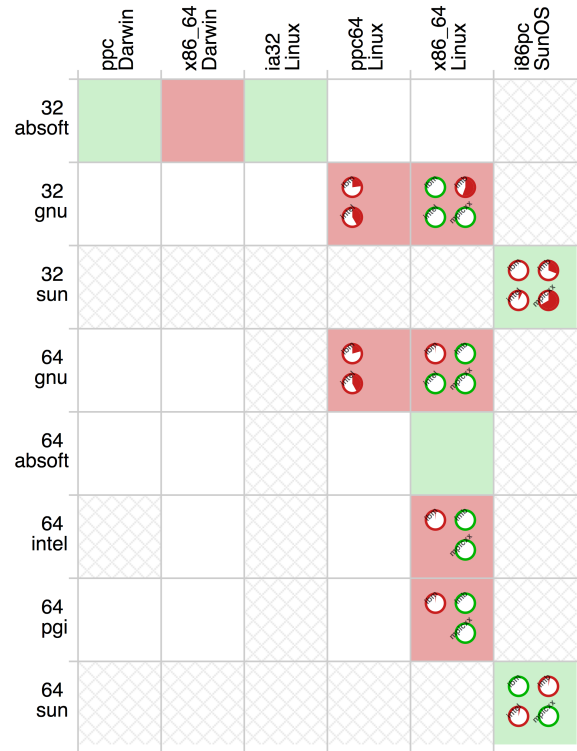
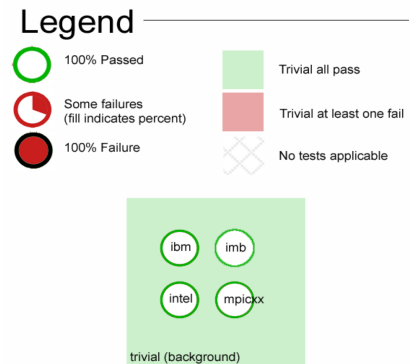


Figure 2: Visualization of raw data seen in Table 2. The reference system is a grid of multi-valued axes. The X-axis encode architecture and operating systems. The Y-axis encode compiled bitness and compiler family. The grid cells are used to create a small-multiples effect for comparison between values and in constructing a general overview using pre-attentive grouping of regions with similar backgrounds.

data-driven, the background color (white in this case) became the de-facto representation for *no-data*.

Our original conception of the data was that the majority of grid-cells would be occupied by testing data. The initial testing data sets supported this assumption; however, as we refined our data selection, the number of *no-data* cells grew substantially. This made the color coding effectively a three-state instead of its original two-state (pass/fail/no-data color encoded by gray/red/white respectively). Furthermore, as developers became familiar with the visualization, they indicated a fourth category of importance: *not-applicable*. Certain cells in the grid are untestable as they represented conflicting concept combinations (e.g. using a Sun compiler on the Darwin operating system). Our de-facto three-state encoding needed to be revised to explicitly handle the, now apparent, four states (pass/fail/no-data/not-applicable).

The existing MTT reporter uses a red/green color encoding scheme to represent pass/fail. Taking this into consideration we modified corresponding values, keeping white as *no-data*. Initial experiments coded *not-applicable* as solid gray. This was later modified to a gray pattern-fill in order to keep the four states distinct in a wider variety of circumstances (see Figure 3(f)). To avoid problems surrounding color-vision deficiency and grayscale printing, we carefully selected red and green values that remain distinct in a variety of circumstances.

5.1.3 Foreground

The foreground of each grid cell is used to represent the results of the non-**Trivial** test suites. Each additional test suite is given a stable

location in the grid cell. For example, each graphic put the **IBM** test suite in the top-left corner, regardless of other test suites present. The visualizations focus on the five most popular test suites: **IBM**, **IMB**, **Intel**, **mpicxx**, **Trivial**. Even though we scale down the testing data set by constraining the test suites considered, the methodology presented is generalizable to the inclusion of more test suites. Our initial prototype represented each test suite as a circle, filled according to the same rules as the background of the cell (see Figures 3(a) and 3(b)). We modified the background scheme, using a higher saturation red and gray color to improve the visibility of the relatively small glyphs. If a test suite is not present in a given cell, no placeholder is used, its position is left alone. We retain these basic elements of spatial alignment and color correspondence throughout the development of the MTT Visualization Extension. However, the precise foreground graphic representation went through several iterations as seen in Figure 3.

In response to the original scheme, feedback indicated a desire for presentation of more intermediate test suite results. Our first attempt was to place a ring around the circular glyph used the red/gray color coded to represent the presence/absence of failures. The center of the glyph represents the majority item (gray if 50% or more passed, red otherwise (Figure 3(c))). This representation, though richer, still provided insufficient details. We looked at various glyph encodings to allow approximate percentages to be represented, but eventually settled on direct representation of the actual percentage. As such, we converted our basic representation to a pie chart with two regions: pass and fail (Figure 3(d)). The percent of tests that fail in a test suite are represented by a red pie slice (based at the 12:00 position). The remainder of the pie is left empty (white). The red/white scheme provides high contrast and easy comparison

Hardware	OS	Bitness	Compiler Name	Test Suite	Pass	Fail
ppc	Darwin	32	absoft	trivial	18	0
x86_64	Darwin	32	absoft	trivial	0	24
ia32	Linux	32	absoft	trivial	24	0
ppc64	Linux	32	gnu	ibm	412	122
ppc64	Linux	32	gnu	intel	548	388
ppc64	Linux	32	gnu	trivial	27	3
x86_64	Linux	32	gnu	ibm	1428	0
x86_64	Linux	32	gnu	imb	8	10
x86_64	Linux	32	gnu	intel	3744	0
x86_64	Linux	32	gnu	mpicxx	16	0
x86_64	Linux	32	gnu	trivial	96	160
i86pc	SunOS	32	sun	ibm	515	1
i86pc	SunOS	32	sun	imb	27	12
i86pc	SunOS	32	sun	intel	1147	89
i86pc	SunOS	32	sun	mpicxx	2	4
i86pc	SunOS	32	sun	trivial	24	0
ppc64	Linux	64	gnu	ibm	424	110
ppc64	Linux	64	gnu	intel	543	393
ppc64	Linux	64	gnu	trivial	27	3
x86_64	Linux	64	absoft	trivial	24	0
x86_64	Linux	64	gnu	ibm	14059	185
x86_64	Linux	64	gnu	imb	44	0
x86_64	Linux	64	gnu	intel	3744	0
x86_64	Linux	64	gnu	mpicxx	40	0
x86_64	Linux	64	gnu	trivial	669	1
x86_64	Linux	64	intel	ibm	12618	198
x86_64	Linux	64	intel	imb	36	0
x86_64	Linux	64	intel	mpicxx	72	0
x86_64	Linux	64	intel	trivial	575	1
x86_64	Linux	64	pgi	ibm	2088	36
x86_64	Linux	64	pgi	imb	12	0
x86_64	Linux	64	pgi	mpicxx	24	0
x86_64	Linux	64	pgi	trivial	189	3
i86pc	SunOS	64	sun	ibm	516	0
i86pc	SunOS	64	sun	imb	38	1
i86pc	SunOS	64	sun	intel	1193	43
i86pc	SunOS	64	sun	mpicxx	6	0
i86pc	SunOS	64	sun	trivial	24	0

Table 2: The raw data used to generate the visualization in Figure 2. The data set contains 38 records and is fed into the Stencil system to produce the visualization. An additional listing of ‘Not Applicable’ table cells is fed in as well, but not included here.

between graphics. The pie-chart encoding was appreciated as it allows a direct comparison of values, without needing to revert to a textual representation of the raw pass/fail data.

In its basic form there were still problems with the pie-based encoding. The *near-complete* cases, which are of particular interest to developers, were difficult to visually distinguish from the 100% cases (i.e., *complete-pass* and *complete-fail*). *Near-complete* cases occur when the pie chart is either nearly completely empty (i.e., *most-pass*) or nearly completely full (i.e., *most-fail*). The presence of at least one failure (i.e., *most-pass*) or of zero passes (i.e., *complete-fail*) is considered particularly significant to developers. Encoding the raw presence of fails as a red border improves the situation where only a small percentage of tests failed (see Figure 5.1.2 for examples of all border/pie combinations). Instead of a small sliver, *near-complete* passes appear as a red, *nearly* empty ring enclosure. Indicating a *complete-fail* case by a black ring around a filled red circle quickly distinguishes a *complete-fail* from a *most-fail* case. *Complete-pass* is represented as a green ring. This extra encoding aids the developers in prioritizing their tasks by letting them quickly identify corner cases. Often, *most-pass* cases are easily fixed and *complete-fail* cases indicate a need to re-examine assumptions.

A side effect of using the additional ring for double-encoding presence/absence of failures was a richer graphic. Glancing at it revealed where test suites failed as the rings were visually prominent, regardless of their contents. When desired, a deeper inspection could be made for approximate percentages or comparison between related groups. Since test suites were consistently encoded both within a graphic, between graphics and through time (i.e., the

positions never changed), a small-multiples effect was realized for the overall appearance of the graphic.

The positional encoding had mixed utility. The consistency made it possible to confidently grid cells. However, there was no indication in the graphic of which test case appeared in which position (aside from the legend) and no natural ordering to fall back on that would yield long-term stability. Though there were few test suites represented, Open MPI developers expressed concern about needing to memorize the positions of the test suites and additional concerns about comparing graphics should different test suits be used in the future. Test suite positions are kept stable in all iterations of the graphic (i.e. the upper-right hand glyph always represents the same suite) and a legend clearly identifies each position. However, browsing the generated graphic often moves the legend off the screen. To resolve these concerns, we placed a semi-transparent (20% opaque black) textual identifier for each suite directly on their respective graphic (See Figure 3(e) and 3(f)). Using a highly transparent value reduced the visual prominence of this element when zoomed out, while preserving legibility when zoomed in.

5.2 Visualization Integration

The design of the visual schema for the MTT Visualization Extension is only part of the problem of delivering a testing visualization to the broader Open MPI development community. Additionally, we have to construct an infrastructure that can keep the visualizations up-to-date, that will be accessible from a variety of locations, and allows us to quickly change the visualization as its use and the needs of the Open MPI developers became more refined.

The international development community participating in the Open MPI project provided invaluable feedback during the development of the visual schema. To provide this development community with the most useful visualizations for the various user roles, we needed to make sure it interacted well with the existing MTT Reporter environment. It is natural to join these two systems since the Reporter acts as a browser for testing details, and the Visualization Extension acts as a browser for high level aggregations of the overall testing. The resultant goals of the integrated visualization environment are to:

1. Move data quickly through the visualization process
2. Quickly update the visual representations used
3. Link generated graphics to the MTT Reporter

The MTT Reporter is an interactive PHP script with a link to a PostgreSQL database. This web site collects query requests through the HTTP POST mechanism. Queries are sent to the Postgres database through the MTT Reporter that also formats the results for display. The MTT Reporter is well suited for the construction of tabular outputs. However, we found no visualization technologies that could support similar ad-hoc queries, quickly update the visual representation, and maintain links back to the source data (items 2 and 3 above).

Since we could not find an on-line system to perform the required custom diagramming and interaction tasks, we elected to generate a series of progressively more focused data graphics statically, and interlink them by image-maps and html links. This trail of links eventually allow the user to step down into the MTT Reporter for more detailed information. In this way, testing data can be visually browsed in an interactive fashion, making the visualization highly responsive and broadly accessible. This relaxes the restrictions on the visualization engine, requiring only a batch mode of operation in contrast to a fully interactive generation environment. However, by doing so we sacrifice the ability to specify arbitrary comparisons on the X and Y axes of the graphic (the glyph values are already agreed to be fixed to test suite pass percentage). However, the set of initial requested comparisons is quite small: architecture versus compiler family, architecture versus scale, and architecture versus compiler version.

5.2.1 Final System

The MTT Visualization Extension is composed of the MTT Database, stencils, HTML page generation scripts, and the web server. Integrating the various components of the system ended up involving a pipe-line setup where each successive step consumed the output of the prior and produced its own as shown in Figure 5. The ability to include the visualization system directly in with other data processing tools is a major benefit to using the Stencil system.

The first stage in the visualization pipeline is the Aggregate Data Collection. This is implemented as a Perl program that queries the MTT Database with SQL queries for a range of predetermined aggregations over a specified date range. This stage generates files containing both the results of the SQL queries and the SQL queries themselves. An XML file is generated describing each set of files, and some details regarding the query constraints for reference later in the pipeline.

The Stencil Rendering stage is passed the SQL output from the Aggregate Data Collection stage along with a stencil specification which is generated by hand. This stage calls the Stencil package that is implemented in Java. The Stencil package generates a PNG representation of the aggregate data as specified by the stencil, and

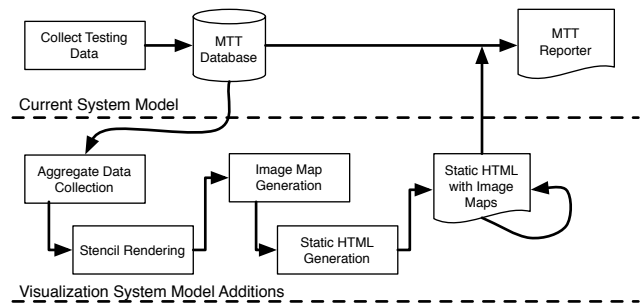


Figure 5: A flow diagram illustrating the MTT visualization system as it relates to the existing MTT Reporter.

a textual representation of the visualization describing the placements of glyphs in the PNG file. The tuple-based textual representation that Stencil produces makes it simple to create image maps separate from the image generation. Textual representations make it possible to computationally reason about the image without needing complex image processing (e.g., OCR), computer vision algorithms (i.e., region identification) or detailed understanding of the internal structures the Stencil system employed (e.g. traversal of its internal graphic representation).

The Image Map Generation stage takes the textual representation from the Stencil Rendering stage. This stage is implemented as a Perl program using the textual representation to combine the PNG with hypertext linking between one image and either other images or the MTT Reporter. Both the Stencil Rendering stage and this stage append information to the XML file.

The Static HTML Generation stage is implemented by a Perl script that takes all of the accumulated data from all stages and produces a series of statically linked web pages. The Open MPI developers view the visualization and navigate through the previously determined drill-down varieties in these web pages. Additionally at any particular web page the user can jump into the MTT Reporter for a more detailed analysis of the data.

Modifying the visualization typically only requires changes to the stencil, though occasionally the query must be changed to reflect different data requirements. Using this setup, we are able to move our data efficiently through the visualization pipeline. We can update our visualizations by modifying the stencils and/or SQL queries that are held externally from the logic of the scripts. As a side effect of using the Stencil system, we can also service requests for views of custom data in the standard fashion by using existing stencils with non-standard queries. Alternatively, we can also prepare slightly modified graphic views without modifying our mainline code by providing custom stencils.

6 Visualization Analysis

The Open MPI project is dedicated to the goal of portability over a wide range of architectures, operating systems, runtime environments, networks, compilers, and system sizes. The resulting testing space requires the Open MPI developers to interpret thousands of testing results on a daily basis. The only effective way to interpret these results in through a well reasoned representation of the data.

The goal of the SeeTest schema, and more particularly the MTT Visualization Extension, is to provide the Open MPI developers with an accurate high level assessment of the overall stability and degree of testing of the code base in an interpretable manner. The MTT Reporter is a sufficient tool for tracking specific sets of failures in

detail. However, users have reported that the MTT Reporter lacks the ability to express high level aggregations useful when assessing the overall health of the project. This visualization is designed to augment, but not replace the existing MTT Reporter, as each serve different user roles.

To assess the effectiveness of the visualization we made our prototype extension results available to the Open MPI developers through web-pages for informal evaluation. The group of respondents represented all roles described in Section 3.2, and each respondent had previous experience with the existing MTT Reporter. Our schema development began with requirements gathering from OpenMPI developers. We continued that relationship by asking a series of open-ended questions after giving them a few days to explore the visualization schema and interact with the extension. These questions are intended to elicit information regarding the effectiveness of the effectiveness of the visualization and directions for improvement. Specifically we were interested in responses the following:

- Is the visualization useful in assessing the overall goals of the project?
- Is the visualization easier to interpret than the current MTT Reporter?
- Did the visualization lead the developers to discover “new” information that was previously obfuscated by the current MTT Reporter interface?
- In what ways could the visualization be improved to better represent the data and/or assist the user in evaluating the data?

The Open MPI developers unanimously expressed that the MTT Visualization Extension was a valuable addition to the MTT Reporter environment. Additionally they all expressed that, once fully deployed, the visualization will be a critical asset to the community for evaluating the degree of testing, and achievement of the overall goals of the project.

The respondents indicated that the holistic view of the testing data, along with focused versions supported their respective roles. The respondents, mostly in the Platform Manager role, expressed that having the ability to “step-down” into the MTT Reporter for a finer granularity of testing data analysis is a useful bridge between the MTT Reporter and Visualization Extension.

We asked the Open MPI developers to take note of any “new” information about the testing data that they discovered by using the visualization that they previously did not notice using the MTT Reporter. This was to assess the effectiveness of the visualization as a method of expressing the aggregated information in an easily interpretable manner. Among these discoveries the respondents noticed that:

- The community is not sufficiently testing many platform and compiler combinations to confidently assess the stability goals of the project.
- The scale of testing (measured by number of processes) is relatively small (less than 16 processes), and routine testing should include a few runs at larger scales to consistently assess the scalability goals of the project.
- There were a few reporting inconsistencies that had gone undetected (i.e., ‘gcc’ instead of ‘gnu’). Such inconsistencies cause problems for aggregation. Identified instances have since been addressed.

Overall, feedback was positive, but we pressed them for criticisms of the MTT Visualization Extension. The respondents expressed

that it would be useful to indicate the “depth” of testing in the visualization. Since each visualization represents the aggregation of many testing results from different testing configurations (a compression of a N dimensional space into a 2 dimensional space) it would be useful to express this as part of the visualization. Respondents noted that such additional information would be useful when determining if a failure profile from a test suite is consistent across the aggregated testing configurations.

The Open MPI developers also desired the ability to scale one of the dimensions of the visualization by time. This alternative visualization would allow the developers to determine if the code base is becoming more or less stable over time. This would assist the Platform Manager user role in determining when a problem started to occur. It is currently difficult to extract this data from the current MTT Reporter and the Visualization Extension.

Most of the respondents were eager for a more interactive environment in which they could dynamically specify the aggregation dimensions and axis parameters. In this prototype, we provided the developers with three of the most commonly asked for aggregations: architecture versus compiler family, architecture versus scale, and architecture versus compiler version. We did not provide an interactive environment in the visualization prototype since our primary goal was on developing and evaluating the visualization representation. We intend to develop such an interactive environment for the development community as part of our future work.

Concurrently using a red/white/absent color coding in the foreground and a red/gray/white coding in the background to represent pass/fail was concerning to us. After the addition of the ring to the pie-chart, there was sufficient contrast between the encodings that it did not seem to present any difficulty to interpretation to the development team. Even when directly questioned on the potential cognitive dissonance, there were no negative comments. Their comments mirrored our own feelings that using a missing glyph to represent *no-data* was more intuitive than an empty circle. Furthermore, when the white circle with gray border was employed to represent 100% passed, it created a suitably distinct graphic that did not command foreground space. This condition was retained with the border was moved to green.

7 Future Work

The informal evaluation presented in Section 6 has helped develop a strong basis for visualization of large-scale testing data. More formal evaluation methods are being pursued to provide more rigorous evaluation and guide future development of the SeeTest Schema pursuant to resolving the issues highlighted in Section 6. This extended evaluation will likely include other software projects and a more rigorous recruitment process.

The described graphic schema is used for several different comparisons over the data. It was successfully employed for examining questions like “What has been tested and how well did it fare?” Other questions of interest to the development team included “How much has this been tested?” and “How have these values changed over time?”

Expanding on the suggestions from the developer feedback, we intend to add the ability to explore a time-base visualization based on the design presented in this paper. We have established, through similar user solicitation, that the representation described in this paper is insufficient to support serious time-based evaluations of testing data. We believe that allowing developers to track the health of the project over time, and investigate development trends that appear, will be a valuable extension to the MTT.

Similarly, we are developing prototypes of a visualization to represent the over and under testing of individual testing configurations. The idea of “how much” a particular test is run (and under what circumstances) was a question of details not available in percentage form (after all 1 is 1% of 100 to the same degree that 100 is 1% of 100,000, but the two are by no means equal from a “how much” perspective). This will assist Open MPI developers in determining the most effective use of the limited resources available for testing.

Finally we are developing an interactive online viewer that will allow the Open MPI developers to customize the data set being rendered, and the visual elements of the visualization in real time. This will mark the next major step for this visualization allowing a customized investigatory process for developers.

8 Conclusion

The ability of a large software project to assess the degree to which it is meeting its goals is critical to the success and longevity of the project. The Open MPI project relies on the MPI Testing Tool (MTT) to provide routine testing, archiving, and reporting with the overall mission to assess the projects portability, stability, and scalability goals. The MTT Reporter only provides a tabular representation of the testing data, making it difficult to investigate questions involving aggregation. This deficiency is addressed by the MTT Visualization Extension.

The MTT Visualization Extension proved to be effective in answering questions about the overall health of Open MPI. This extension was deemed successful by Open MPI developer feedback. We believe that testing toolkit visualization extensions, similar to the one described here, represent an advancement in the communication of the health of a large software project.

Acknowledgments

This work was supported by a grant from the Lilly Endowment and National Science Foundation grants NSF ANI-0330620, CDA-0116050, and EIA-0202048. The authors would like to thank the Open MPI developer community for their participation in the development and assessment of this project.

References

BEDERSON, B. B., GROSJEAN, J., AND MEYER, J. 2004. Toolkit design for interactive structures and graphics. *IEEE Transactions on Software Engineering* 30, 8, 535–546.

COTTAM, J. A., AND LUMSDAINE, A. 2007. Tuple Space Mapper: Design, challenges and goals. Tech. Rep. TR648, Indiana University, Bloomington, IN, June.

EICK, S. G., STEFFEN, J. L., AND SUMMER, E. E. 1992. SeeSoft: A tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering* 18, 11 (November), 956–968.

GABRIEL, E., AND ET AL. 2004. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, 97–104.

HARROLD, M. J. 2000. Testing: A roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, ACM Press, New York, NY, USA, 61–72.

HURSEY, J., MALLOVE, E., SQUYRES, J. M., AND LUMSDAINE, A. 2007. An extensible framework for distributed testing of MPI implementations. In *Proceedings, Euro PVM/MPI*.

JONES, J. A., HARROLD, M. J., AND STASKO, J. 2002. Visualization of test information to assist fault localization. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, ACM, New York, NY, USA, 467–477.

JUNIT, 2008. JUnit: Resources for Test Driven Development. <http://www.junit.org/>.

KITWARE INC., 2008. CDash: Open Source, Distributed, Software Quality System. <http://www.cdash.org/>.

KITWARE INC., 2008. Dart: Tests, Reports and Dashboards. <http://public.kitware.com/Dart/>.

MESSAGE PASSING INTERFACE FORUM. 1993. MPI: A Message Passing Interface. In *Proc. of Supercomputing '93*, IEEE Computer Society Press, 878–883.

MEYER, M., GİRBA, T., AND LUNGU, M. 2006. Mondrian: An agile information visualization framework. In *Proceedings of the 2006 ACM Symposium on Software Visualization (SOFTVIS'06)*, ACM, New York, NY, USA, 135–144.

NORTH, S. 1998. Visualizing graph models of software. In *Software Visualization: Programming as a Multimedia Experience*, J. Stasko, J. Domingue, M. H. Brown, and B. A. Price, Eds. The MIT Press.

NUNIT, 2008. nUnit. <http://www.nunit.org/>.

OPEN MPI, 2008. MTT: MPI Testing Tool. <http://www.open-mpi.org/projects/mtt/>.

OSTERWEIL, L. 1996. Strategic directions in software quality. *ACM Comput. Surv.* 28, 4, 738–750.

PANAS, T., LINCKE, R., AND LÖWE, W. 2005. Online-configuration of software visualizations with Vizz3D. In *Proceedings of the 2005 ACM Symposium on Software Visualization (SoftVis'05)*, ACM Press, New York, NY, USA, 173–182.

PINZGER, M., GALL, H., FISCHER, M., AND LANZA, M. 2005. Visualizing multiple evolution metrics. In *IEEE International Workshop of Software Visualization (SoftVis2005)*.

REISS, S., AND RENIERIS, M. 2003. The Bloom software visualization system. In *Software Visualization: From Theory to Practice*, J. Cao, Z. Ren, A. Chan, L. Fang, L. Xie, and D. Chen, Eds. Kluwer Academic Publishers, Norwell, Massachusetts, 243–284.

REISS, S., AND RENIERIS, M. 2005. Jove: Java as it happens. In *Proceedings of the 2005 ACM Symposium on Software Visualization*, ACM Press, New York, New York, 115–124.

VOINEA, L., TELEA, A., AND VON WIJK, J. 2005. CVSscan: Visualization of code evolution. In *IEEE International Workshop of Software Visualization (SoftVis2005)*.