Efficient Integration Testing using Dependency Analysis

Amitabh Srivastava
Microsoft Research
One Microsoft Way
Redmond, WA
amitabhs@microsoft.com

Jay Thiagarajan
Microsoft Research
One Microsoft Way
Redmond, WA
jaythia@microsoft.com

Craig Schertz
Microsoft Research
One Microsoft Way
Redmond, WA
cschertz@microsoft.com

ABSTRACT

Although testing starts with individual programs, programs are rarely self-contained in real software environments. They depend on external subsystems like language run time and operating system libraries for various functionalities. These subsystems are developed externally to any given program, with their own test processes. Of course, an uncoordinated change in one of the external subsystems may affect the program's correctness. Test teams therefore add an integration testing step to their process to ensure that programs will continue to operate with different versions of the external subsystems. As full testing may take days or weeks to run, it is useful to understand how to prioritize these tests.

We present an integration testing system to understand and quantify the impact of a change, so test teams can focus their testing efforts on the most likely affected parts of the program. Detecting the impact of a change is a hard problem due to the size and complexity of the control and data dependencies involved. Our new approach is based on a binary dependency framework, MaX, that determines control and data dependencies in a system and represents them in a dependency graph. MaX is designed to work on systems consisting of thousands of binaries and millions of procedures. It constructs the graph in multiple steps to allow the analysis of individual binaries to proceed in parallel. MaX provides simple abstractions for defining systems, and provides a simple programming interface to tools for analysis of the graph.

The integration testing system also contains two tools that use MaX to advise test teams. MaxCift quantifies the effect of a change to guide how much testing is likely to be needed. MaxScout prioritizes an existing set of tests based on changes made to external subsystems. All of the tools use a binary code based approach that does not require source code for external subsystems, an important requirement for practical use.

MaX runs under the Windows environment and is used by Microsoft product teams. Early results show that the system scales to production software and is effective in guiding testing.

Keywords

Software testing, integration testing, test prioritization.

1. INTRODUCTION

Programs continuously evolve to address new requirements, changing environments, marketplace demands and existing deficiencies. Software testing has to ensure that no new defects are introduced throughout the program's evolution. Testing of software, therefore, occurs continuously throughout the development cycle. Over the past decade a number of techniques have been proposed to reduce the cost of testing (we discuss these in Section 2). However, all of the proposed techniques focus *internally* on a single program; that is, they consider only internal parameters such as changes made to the program itself, the rate of faults in the program, and test coverage of the program.

Programs, however, are rarely self-contained in real software environments; they depend on a number of external subsystems[1] like applications, third party dynamically linked libraries (dll), language run time dlls, and operating system dlls for various functionalities. These subsystems are *external* to the program and are often maintained by different teams with independent testing process and release schedules. A change in any external subsystem may affect the program.

As production software is complex, it is difficult to manually track all of its dependencies (the transitive closure of all immediate dependencies), yet this is needed to assure continued operation at customer sites. To handle this challenge, test teams have added an integration testing step in their development process to ensure that their applications will continue to operate with different versions of external subsystems. The external subsystems are often available to test teams in binary form, and a test team generally does not know the changes that have been made in these external subsystems from the previous version. For example, an application which uses the Windows XP operating system libraries may have access to the Windows XP and Windows XP SP1 (Service Pack 1) binaries but will not know the exact details of the changes that were made in Windows XP to produce the Windows service pack SP1.

The complexity of integration testing increases with the number of external subsystems. As full testing of a program may take days or weeks to run, an effective strategy is to focus testing effort on parts of the program that may have been affected by the changes in the external subsystems. In time-constrained situations, an appropriate set of tests, subject to the specified time limit, can be selected to best use the available time.

[1] A subsystem is a logical collection of shared dynamically linked libraries (.dlls); for example, the subsystem for the windows operating system may include dlls such as kernel32.dll, gdi32.dll, and user32.dll.

To improve the efficiency of integration testing, we have built a test integration system consisting of the following tools:

1. MaX – a dependency framework for an entire system.
   - MaX works with systems consisting of native x86 and .NET managed CIL [31] binaries.
   - MaX determines control and data dependencies by examining the system binaries. It builds the dependence graph with a uniform representation for control and data dependencies.
   - MaX provides a simple set of hierarchical abstractions for defining a system and representing its dependence graph.
   - MaX provides a simple programming interface for navigating and querying the dependence graph.
   - MaX scales to large production systems and can be used in various real time scenarios.
2. MaxCift – quantifying a change. MaxCift uses the dependence graph to quantify the effect of a change. It computes the change impact factor (CIF) by measuring the fraction of binaries and procedures that have been affected in the system. It helps test teams understand how much testing may be needed and where to focus.
3. MaxScout - a test prioritization system, which presents a new approach for prioritizing an existing set of tests for a program based on changes made in its external subsystems.

All the tools use a binary code based approach that works well for integration testing, as source code for external subsystems is generally not available to the program's test team.

This paper describes the design and implementation of the integration testing system. It discusses the performance of MaX in building dependence graphs of large production applications. The paper also analyzes the impact of change in Microsoft product service packs.

2. RELATED WORK

Many approaches have been proposed to address the cost of testing, including test selection [1][2][4][17][26], test prioritization [5] [6][7][14][15], and a hybrid approach [29] that consists of test selection using source code changes [12] followed by test prioritization to schedule the selected tests. Test minimization techniques that permanently discard tests have been proposed in [3][11][16][28].

Computing program change has been proposed [1][2][3][17][26] for test selection, and for test prioritization [5][7][14][29]. Techniques used to compute program change are source code differencing [5][7][26], data and control flow analysis [1][17], and coarse grained modified code entities [4] to identify which parts of the program might be affected by the changes.

Echelon [20] proposed a binary code based approach for test prioritization. Echelon used binary matching [27] to compute change in a program at basic block granularity. Instead of using expensive techniques like data flow analysis, Echelon utilized a fast and simple heuristic to predict which tests will cover the affected basic blocks. By doing prioritization, Echelon can use a non-precise algorithm that works well in practice. Echelon's technique is fast and scales well to large programs, making it suitable for use in development environments.

Echelon and all the previous systems are designed for testing a single changed program. They use different techniques to compute changes in the program at different levels of granularity and prioritize the test cases based on it.

None of the previous systems, including Echelon, propagate the effect of external changes into a program. MaX, on the other hand, uses a dependency framework to find relevant affected parts of the system outside the program boundaries.

MaxCift computes a coefficient that serves as a simple heuristic that quantifies change. The relative value of the coefficient reflects the impact of a change.

MaxScout may mark a program as affected or changed even though there were no changes in any of its own code. All the previous systems including Echelon were not designed for such cases.

3. ENVIRONMENT FOR THE MaX TOOLS

The MaX tools are part of the Magellan test effectiveness tool set. They leverage the Magellan test effectiveness infrastructure and the Vulcan [21] binary modification infrastructure for information and analysis. In this section, we briefly describe Magellan and Vulcan. We also discuss Echelon [20], which is part of the Magellan tool set; MaxScout uses a modified form of Echelon for prioritizing tests in its last step.

The Magellan Test Effectiveness Infrastructure

The Magellan tool set provides an infrastructure for collecting, storing, analyzing, and reporting information about a test process. The core of Magellan is a SQL Server-based repository that stores test coverage information for each test. The coverage information can be mapped to the static structure of the program: its procedures, files, directories, binaries etc. All the program binaries that were tested and their corresponding symbol files are stored in a separate symbol repository and cross-linked. The Magellan infrastructure is designed to be extensible; Magellan provides a well-defined interface for accessing and storing information.

Although new tools are easily added, Magellan also provides a basic set of tools that are commonly needed during the test process. This includes a test coverage collection tool that uses binary instrumentation to collect block coverage and arc coverage information both in user and kernel mode. The coverage information is collected for each test and stored in the repository. For easy presentation of coverage data, Magellan also provides reporting tools with graphical user interfaces that can map the data to the source code. Blender, a test migration tool, migrates coverage data from older versions of Magellan to newer versions.

Magellan includes Echelon [20], a test prioritization system that uses a practical binary code based approach for test prioritization based on program change. Echelon takes as input two versions of the program in binary form along with the test coverage information of the older version, detailing which tests covered which parts of the program. Echelon outputs a prioritized list of tests, starting with a minimal sequence of tests, drawn from the given tests that cover as many of the affected parts of the program as possible. This is followed by another minimal sequence of tests drawn from the remaining tests, and so on; it ends with a sequence of tests that do not cover any of the affected parts of the program. Echelon operates on a single binary and computes the impacted blocks of the binary by determining which blocks in the binary have been modified or added, using binary matching [27]. Echelon's technique is fast and scales well to large programs, making it suitable for use in large-scale development environments.

Vulcan Binary Modification Infrastructure

MaX utilizes a rich binary modification infrastructure called Vulcan [21]. Vulcan is a second-generation toolkit that provides both static and dynamic binary code modification and provides a framework for analysis and optimization [22][23][24]. Vulcan provides a uniform abstraction for binaries with a simple API for inspection, instrumentation and optimization. Vulcan works in the Win32 environment and can process x86, IA64, and CIL [31] binaries. Vulcan has been used to improve the performance and reliability of many Microsoft products.

To compute the changes between two versions of a binary, MaXScout utilizes BMAT [27], a binary matching tool built using Vulcan. BMAT is a fast and effective tool that matches two versions of a binary program without knowledge of source code changes. This tool uses a hashing-based matching algorithm and a series of heuristic methods, with the goal of matching as much of the program as possible, thus identifying the changes. The algorithm first matches procedures, then blocks within each procedure. Several levels of matching are attempted with varying degrees of precision. This process allows correct matches to be found even in the presence of shifted addresses, different register allocation, and various small insignificant program modifications. The success rate of matching of code blocks is often higher than 99% [27].

4. MaX: DEPENDENCY FRAMEWORK

This section describes how MaX computes the control and data dependencies of a system, and constructs the dependency graph. It also discusses the programming interface MaX provides for analysis, and its performance on large systems.

4.1 Constructing the Dependency Graph

MaX constructs the dependency graph of a system in three steps. In the first step, a hierarchical definition of the system is created using the MaX abstractions. In the second step, MaX analyzes each binary individually to compute the binary's data and control dependencies. In the last step, the information from each binary is consolidated to build the system dependence graph. This section discusses each of the steps in detail.

Defining the System Definition File

MaX provides four levels of abstraction: *system*, *subsystem*, *binary, and procedure*. A system consists of one or more subsystems, a subsystem contains one or more binaries, and a binary consists of one or more procedures. The system definition file uses these MaX abstractions to describe the system in a hierarchical manner.

```
<system       name = "magsys">

  <subsystem  name = "magellan"             file = "mag.xml">

    <binary    name = "coverage.dll"     file = "coverage.xml"/>
    <binary    name = "covercmd.exe"     file = "covercmd.xml"/>
    <binary    name = "magcore.dll"      file = "magcore.xml"/>
    <binary    name = "magtraces.dll"    file = "magtraces.xml"/>
  </subsystem>

  <subsystem  name = "vulcan"           file = "vulcan.xml">
    <binary    name = "vulcan23.dll"     file = "vulcan23.xml" />
    <binary    name = "vuldyn.exe"       file = "vuldyn.xml" />
    <binary    name = "vuldynpxy.dll"    file = "vuldynpxy.xml" />
    <binary    name = "vulutil.dll"      file = "vulutil.xml" />
  </subsystem>

  <subsystem  name = "vc"               file = "vc.xml">
    <binary    name = "mspdb71.dll"      file = "mspdb71.xml" />
     <binary   name = "msvcr71.dll"      file = "msvcr71.xml" />
    <binary    name = "msvcp71.dll"      file = "msvcp71.xml" />
    <binary    name = "msobj71.dll"      file = "msobj71.xml" />
  </subsystem>

  <subsystem  name = "windows"          file = "windows.xml">
    <binary    name = "kernel32.dll"     file = "kernel32.xml" />
    <binary    name = "nt.dll"           file = "nt.xml" />
    <binary    name = "user32.dll"       file = "user32.xml" />
    <binary    name = "gdi32.dll"        file = "gdi32.xml" />
 </subsystem>

</system>
```

Figure 1. System Definition File

Figure 1 shows the XML file containing the definition of the *magsys* system, which has four subsystems: *magellan*, *vulcan*, *vc*, and *windows*. The binaries comprising each subsystem are enumerated in the system file. For example, the Vulcan subsystem contains four binaries: vulcan23.dll, vuldyn.exe, vuldynpxy.dll, vulutil.dll. The system definition file also contains the names for each of the abstractions and their file locations.

Creating the Binary Dependency Information File

MaX analyzes the binaries defined in the system definition file, one binary at a time, to produce the dependency information file. This file summarizes the dependency information for that binary. The binaries in the system may be processed in parallel. The processing time thus can be controlled by adding more machines or processors. Section 4.3 discusses the performance of MaX.

All the binaries in a system have to be processed at least once by MaX at the start of the process. However, as software development progresses and parts of the program are changed, only the modified binaries have to be reprocessed by MaX. This tremendously cuts the processing costs as not all binaries are normally modified at each step of the development process.

To express the dependencies in a binary, MaX adds two abstractions to the binary abstractions: *call-in* points and *call-out* points. Control enters a binary through one of its call-in points and it continues out of the binary through one of its call-out points. (We do not consider returns, exceptions, etc., as call-outs.) A binary may have a number of call-in and call-out points. Call-in points are procedures in the binary while call-out points link to procedures in other binaries it may call.

Many call-in and call-out points, such as the exports and imports in a Win32 PE binary, are easily found. Binaries also contain relocation records to mark instructions that can be a target of indirect procedure call, and MaX marks these as call-in points. MaX uses Vulcan to determine the control transfer instructions that might transfer control outside the binary.

While many control transfer targets can be determined statically, others require more analysis. MaX uses static analysis and several heuristics to determine these dependencies. For example, MaX uses constant propagation to determine the names of dlls that are explicitly loaded at runtime. Indirect calls through a virtual table require finding the virtual table and using the index to determine the target of the call. These methods may not identify the targets of some calls. If MaX cannot resolve an indirect transfer of control, it flags the call, which is then resolved by providing manual feedback to MaX or by refining MaX's heuristics, which work quite well in practice. MaX lists all of the call-in and call-out points in the binary dependency information file.

In the next step, MaX determines dependencies. A call-in point is dependent on a call-out point if there is a control-flow path from the call-in point to the call-out point. Using reachability analysis, MaX determines the dependency relationships between call-in and call-out points in the binary. These call-in/call-out dependencies are also listed in the binary information file. Similarly, for each procedure in the binary, MaX lists the dependent call-in points in the binary dependency information file.

All the dependencies that we have described so far are control dependencies. To handle data dependencies, MaX introduces one more abstraction, *named object,* which is a data element in the program with a name. Two code segments, which may be residing in different binaries, are *data dependent* if they read or write to the same named object. For example, two procedures are data dependent if one procedure writes to a global variable while the other procedure reads the same global. Named objects can also be registry keys, semaphores, mutexes, etc.

As data dependent procedures may reside in different binaries, MaX, in this step, outputs the procedures in the binary that read or write to named objects that are being tracked. This information is consolidated in the next step to determine data dependencies.

Building the Dependence Graph

MaX aggregates the dependency information files of each binary to build the dependence graph. MaX maintains the same hierarchy of system, subsystem, binary, and procedures in the dependency graph. For performance, MaX initially considers only procedures that are call-in or call-out points for binaries, and creates directed edges between dependent call-in and call-out points across binaries. Other data such as procedure dependencies is read only when needed.

MaX creates a named object data structure for each named object that is being tracked. For each procedure that references a named object, MaX creates a directed edge between the procedure and named object, whose direction depends on whether the procedure reads the named object or writes to it.

The reason for constructing the graph in two steps is to enable MaX to exploit the parallelism in the first step; to take advantage of the fact that only changed binaries need to be reprocessed, and to allow construction of data dependencies in a scalable manner that makes this approach practical for large production systems.

4.2 MaX Programming Interface

MaX provides a simple programming interface for tools to analyze the dependency graph. MaX models a system as lists of subsystems, binaries and procedures. MaX provides query primitives to inspect various properties like names, locations etc, and navigation primitives to move through the respective lists. A node abstraction represents the procedures in the dependency graph, and MaX provides primitives to traverse its callers and callee nodes.

4.3 Status and Performance

MaX has been implemented in C# for x86 platforms running Windows. It handles both native x86 and .NET managed CIL binaries. MaX is under active development. It can currently detect static dependencies, indirect dependencies from loadlib, CreateProcess GetProcAddress, and CoCreateInstance in COM. New dependencies, including registry keys, are being added.

Table 1. System Information

Dependency Graph Objects	Count
Binaries	3,308
Procedures	1,495,295
Call-in points	197,534
Call-in/Call-out Dependency edges	2,206,972
Named Objects	18,898

To measure the performance of MaX, we constructed the dependency graph of a large production system with 3120 binaries and 1.2 million procedures. MaX added 763950 dependency edges between the call-in and call-out nodes. It also created 2406 named objects. The details are summarized in Table 1.

MaX took 287.4 minutes to collect the dependency data for all 3308 binaries on a dual processor 2.18GHz XEON P4 with 2GB of memory running Windows Server 2003. Note that this step is needed only once; subsequently, only changed binaries need to be reprocessed. Further, since the binaries can be processed in parallel, this processing cost can be reduced by adding more machines; for example, it can be cut approximately in half by adding a second machine. Most of the time is spent in writing the binary dependency information files.

MaX takes only 58.6 seconds, less than a minute, to analyze the dependency information files of the 3308 binaries and build the control and dependency graph. As MaX only reads the relevant information needed to build the graph, it can process all the files in less than a minute; it reads other data such as procedure dependencies, only when needed. Once the graph is constructed, reachability analysis can be performed interactively. By quickly reprocessing the changed binaries, the updated graph can be quickly created and used interactively.

5. MaxCift: QUANTIFYING CHANGE

The effects of different changes are not equal; a change in a core binary can have a much larger impact on a system than a change in a binary on the periphery. Test teams can make better decisions when they understand how a change affects the system. For example, the impact of a change can be used to decide whether to accept change late in the development process. It can also be used to prompt a code review if the impact of a change is above a certain threshold. Further, in time-constrained situations, such as when issuing a security fix, a test team can allocate necessary testing resources and focus its testing efforts to relevant parts of the system by understanding the impact of the proposed changes.

MaxCift uses MaX to compute the effect of a change on a system. It uses the reachability analysis to find which procedures have been affected. If a procedure in a binary has been affected, the binary is also marked as affected. The fraction of affected binaries measures the "spread" of the change. The fraction of affected procedures measures the "depth" of the change. Both depth and spread are used to balance the size of a binary versus the number of binaries affected. MaxCift combines these measures to compute the change impact factor, CIF, as follows:

$$\text{CIF} = \log_{10}(\%\text{affected binaries} * \%\text{affected procedures} + 1)$$

CIF is on a log scale and can range from 0 to 4; therefore, a CIF of 2.0 is ten times more serious than a CIF of 1.0.

We analyzed a service pack of a Microsoft product by computing the CIF values of each binary that was modified, as shown in Figure 2.

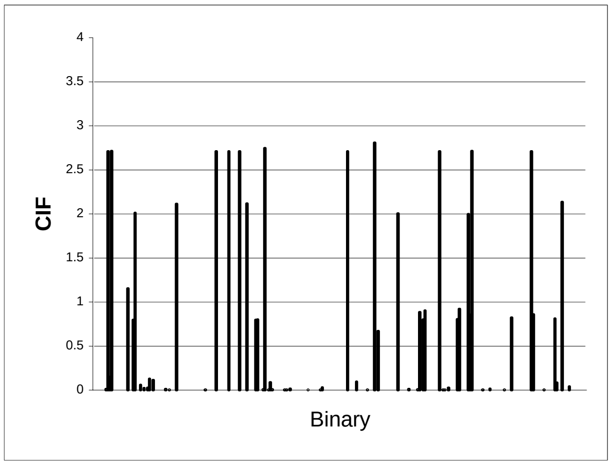

Figure 2. CIF for changed binaries in Service Pack

In the 101 binaries that were modified in Figure 1, only 11 binaries had a change impact factor or CIF greater than 2.5. A test team can focus on those binaries. Our manual analysis confirmed that these were critical binaries.

6. MaxScout: PRIORITIZING TESTS

MaxScout takes as input a system definition file, the binaries that make up the system, new versions of binaries for subsystems that have changed, and test coverage information of a specified program. It produces a prioritized list of tests, as well as a list of subsystems, binaries and specific call-in points that have been affected by the change.

MaxScout computes the changes between two versions of the binaries for the subsystems that have a newer version available. It then uses MaX to propagate the changes to find the affected parts of the system by performing reachability and dependency analysis using the MaX dependency graph, as described in section 4.1.

MaxScout starts by computing changes at block granularity using BMAT [27]. For each updated binary it marks the *affected* blocks that have either been modified or added. MaxScout first uses MaX dependency information for each binary to mark call-in points that have a path to the affected block. Next, MaxScout uses MaX to conduct reachability analysis to determine the call-in points in other binaries that can reach an affected call-in point. If a call-in point of a binary has been affected, MaxScout marks the binary as affected.

MaxScout prioritizes the available tests for each affected binary. (Note that a binary may be affected even if not a single block was changed.) MaxScout uses Echelon [20] to prioritize tests. MaxScout, however, uses a different algorithm to compute the impacted blocks. In Echelon [20], the impacted blocks were computed by finding the set of blocks that were either modified or were added. MaxScout, on the other hand, defines the impacted blocks as a set of call-out blocks of the binary that are connected to affected call-in points. If a call-out point is affected, all its dependent call-in points are affected. We thus prioritize tests that cover an affected call-in and call-out point over others; a test which covers more call-in and call-out points will get a higher priority. Unlike Echelon's measure, MaxScout's measure addresses tests for binaries that have been affected even though they were not modified.

MaxScout uses Echelon to prioritize the tests based on the computed impacted blocks. Echelon uses an iterative, greedy algorithm to first find a short sequence of tests taken from the available tests that covers as many of the impacted blocks as possible. It starts by assigning a priority weight to each test, equal to the number of impacted blocks it covers. The test with the maximum weight is first selected, and in case of a tie, the test with the maximum overall coverage is selected. The selected test is removed from the list, and the impacted blocks covered by it are removed from the impacted block set. The priority weight for each test is recalculated based on the updated impacted block set. At each step, Echelon picks a test from the available tests to cover the maximum number of the remaining impacted blocks; this repeats until no remaining test can cover any remaining impacted block. The tests thus selected form a first sequence to provide the maximum coverage of the impacted blocks. This process is repeated starting again with the full set of impacted blocks to generate the next test sequence, till no remaining test can cover any of the impacted blocks. Remaining tests are added to a separate sequence in the order of their overall coverage.

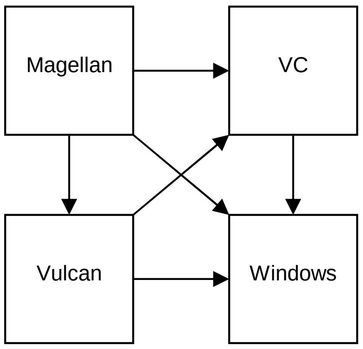

Figure 3. System Dependencies

6.1 Results and Analysis

MaxScout has been tested on a number of binaries from Microsoft's internal development environment. We used the system defined in Figure 1 to study MaxScout in more detail. shows the dependencies of each subsystem. Table 2 shows the details of this system: the size of each binary and of its associated symbol file, and the number of call-in and call-out points of each binary.

Table 2. System Information

Sub system	Binary	Binary Size (bytes)	Symbol Size (bytes)	Call-in	Call-out
Magellan	coverage	9,216	76,800	11	7
	covercmd	27,136	142,336	1	49
	magcore	436,224	3,804,160	93	286
	magtraces	95,232	1,846,272	15	101
Vulcan	vulcan23	1,847,296	7,957,504	2295	153
	vuldyn	36,864	273,408	1	95
	vuldynpxy	28,672	93,184	4	31
	vulutil	57,344	355,328	2	60
VC	mspdb71	241,664	2,575,360	172	148
	msvcr71	344,064	2,345,984	761	143
	msvcp71	499,712	2,952,192	1088	106
	msobj71	73,728	1,469,440	20	62
Windows	kernel32	904,192	3,785,128	900	375
	ntdll	654,848	2,589,696	1104	0
	user32	528,896	2,507,776	719	310
	gdi32	235,008	2,098,176	592	109

We next took an updated version of the kernel32.dll binary (size = 903,680, symbol size = 3,785,128) in the Windows subsystem from a build about a month later. In the new kernel32, 195 (167 modified, 28 added) blocks were impacted, and 80 call-in points out of 900 kernel32 call-in points were affected by the changes. All four subsystems were affected by the change, but only 11 binaries were affected. Table 3 shows the number of affected call-in points in each binary.

MaxScout prioritized the 104 tests into 22 sequences for the Magellan subsystem. The first sequence, which provides a maximum coverage of impacted blocks, contained 25 tests. Each subsequent sequence contained fewer tests. Figure 4 shows the number of tests in each sequence. The number of tests in each sequence falls sharply.

Figure 5 shows how many impacted blocks are covered by each minimal sequence. As expected the first sequence covers the maximum number possible. However, the graph shows a sharp decline after the first few sequences. This information can be very useful for a test team when deciding on how many test sequences to run.

Table 3. Call-in points affected by Windows change

Sub system	Binary	Call-in Points	Call-in Points affected
Magellan	coverage	11	0
	covercmd	1	0
	magcore	93	11
	magtraces	15	2
Vulcan	vulcan23	2295	1183
	vuldyn	1	0
	vuldynpxy	4	0
	vulutil	2	2
VC	mspdb71	172	37
	msvcr71	761	545
	msvcp71	1088	674
	msobj71	20	5
Windows	kernel32	900	159
	ntdll	1104	0
	user32	719	141
	gtdi32	592	196

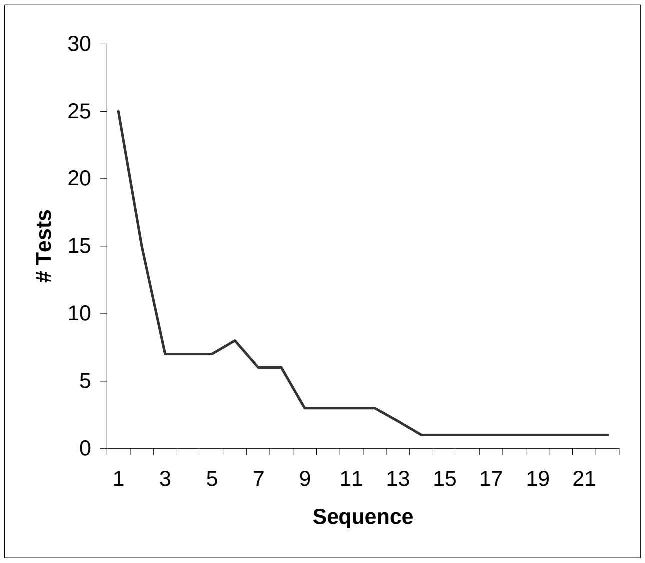

Figure 4. Number of tests in each sequence

As kernel32 is a critical component in the Windows operating system, it is not surprising that all the subsystems were affected. One of the changes to kernel32 was in a commonly used string print function, which was called directly or indirectly by Magellan from 103 call sites. This change required 25 tests to provide the maximal coverage of the impacted blocks. Note that some impacted blocks are not covered by any test. New tests will be needed to cover those blocks.

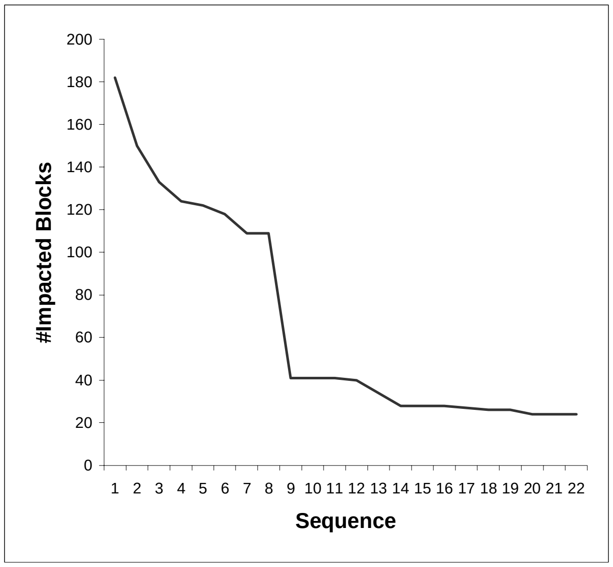

Figure 5. Number of impacted blocks in each sequence

We repeated the study with a change in the Vulcan subsystem instead of the Windows subsystem. We took a new version of the vulcan23.dll binary (size = 1,847,296, symbol size = 7,957,504) in the Vulcan subsystem from a build about a month later. Here, 303 (159 modified, 144 new) blocks in the new version of vulcan23 were impacted. Only 30 call-in points out of 2295 call-in points of vulcan23 were affected by the changes, and only the Magellan and Vulcan subsystems were affected, including only 2 binaries. Table 4 shows the number of affected call-in points in each binary.

MaxScout prioritized the 104 tests into 104 sequences for the Magellan subsystem, as shown in Figure 6. As the impact on Magellan was minimal, each sequence contains only one test.

Figure 7 shows how many impacted blocks are covered by each minimal sequence. As expected, the first sequence covers the one impacted block, as does the second, but the rest of tests do not cover any impacted block. As MaxScout does not eliminate any tests, it places all of these tests at the end of the list.

It is interesting to contrast the two cases. In the first, a change to a critical function in a subsystem caused many dependencies. In the second case, although the change was larger, most of these changes were found to have no impact on the other subsystems, resulting in very different results.

Table 4. Call-in points affected by Vulcan change

Sub system	Binary	Call-in Points	Call-in Points affected
Magellan	coverage	11	0
	covercmd	1	0
	magcore	93	1
	magtraces	15	0
Vulcan	vulcan23	2295	30
	vuldyn	1	0
	vuldynpxy	4	0
	vulutil	2	0
VC	mspdb71	172	0
	msvcr71	761	0
	msvcp71	1088	0
	msobj71	20	0
Windows	kernel32	900	0
	ntdll	1104	0
	user32	719	0
	gtdi32	592	0

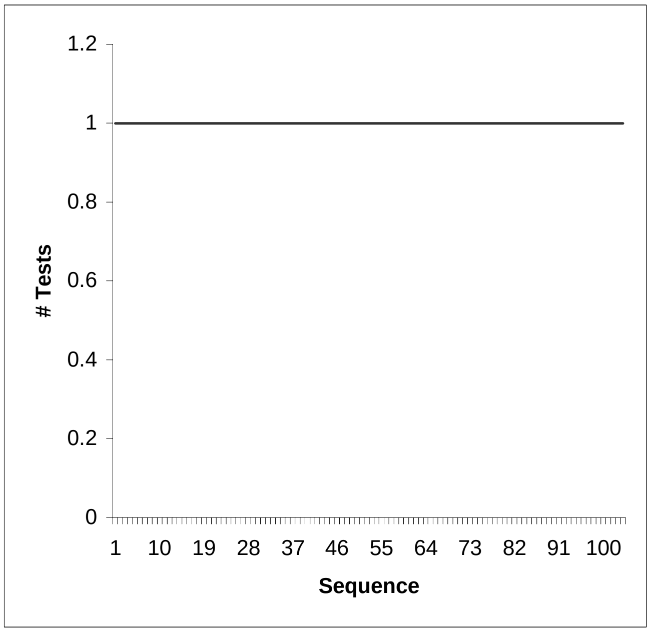

Figure 6. Number of tests in each sequence

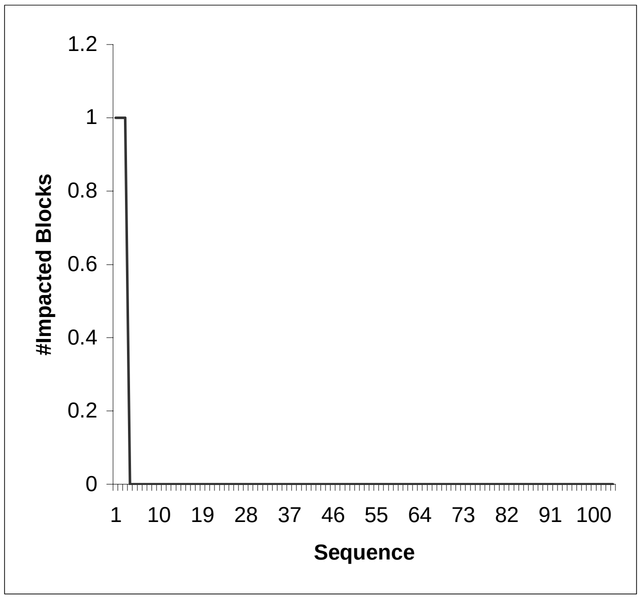

Figure 7. Number of impacted blocks in each sequence

7. APPLICATIONS OF MaX

As MaX can propagate the effect of changes to various call-in and call-out points in a system, it can be used to prioritize the number of configurations in which a program needs to be retested. For example, a program that prints reports may normally need to be tested with a wide variety of printers, with the printers' downstream code connected to a particular call-out point in the program. If that call-out point is not affected by the change, we can prioritize lower the retest of the program across the large number of available printers. Thus MaX can reduce the number of variables in a configuration matrix, where the configuration is defined as a function of call-in and call-out points.

MaX can also be used for what-if analysis. Using the dependency graph as a query engine, teams can study what procedures and binaries will be impacted if a change were to be made to a particular procedure. For example, if a previously published interface is to be deprecated, a team can find the affected owners and inform them. MaX is used very often in what-if mode.

8. CONCLUSIONS

Test teams constantly deal with program change. As the impact of different changes is not equal, better understanding of their impact will enable test teams to allocate their resources and focus testing effort on the affected parts of the program. Understanding the impact of change in a complex production system is hard for testers due to the system's sheer size, and because of its complex control and data dependencies. The MaX tools build a data and control dependence graph in a scalable manner, at various granularities, allowing the analysis of the impact of change in a large system. Using the dependence graph, these tools can help test teams by quantifying the impact of change and prioritizing appropriate tests. Our early results on production systems are very encouraging. We are continuing to improve the MaX tools as we learn from their usage in production environment.

9. ACKNOWLEDGMENTS

Our work leverages the great work of many people, especially those in the Magellan and Vulcan teams at Microsoft Research. David Notkin, Jim Larus, Marne Staples, Sriram Rajamani, Tom Ball, and GS Rana reviewed early versions of the paper. John DeTreville gave very insightful and perceptive comments that helped us improve the paper. Our sincere thanks to all.

REFERENCES

[1] T. Ball, "On the Limit of Control Flow Analysis for Regression Test Selection". Proc. ACM Int'l Symposium. Software Testing and Analysis, pp. 134-142, Mar. 1998.

[2] D. Binkley, "Semantics guided Regression Test Cost Reduction", IEEE Trans. Software Eng., vol. 23, no. 8, pp. 498-516, Aug. 1997.

[3] T.Y. Chen and M.F. Lau, "Dividing Strategies for the Optimization of a Test Suite", Information Processing Letters, vol. 60, no. 3, pp. 135-141, Mar. 1996.

[4] Y.F. Chen, D.S. Rosenblum, and K.P. Vo, "TestTube: A System for Selective Regression Testing," Proc. 16th Int'l Conf. Software Eng., pp. 211-222, May 1994.

[5] S. Elbaum, .A. Malishevsky and G. Rothermel, "Test case prioritization: A family of empirical studies", IEEE Trans. Software Engg. , vol. 28, no. 2, pp. 159-182, Feb. 2002.

[6] S. Elbaum, .A. Malishevsky and G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization", Proc. 23rd Int'l Conf. Software Engg., pp. 329-338, May 2001.

[7] S. Elbaum, .A. Malishevsky and G. Rothermel, "Prioritizing test cases for regression testing", Proc. Int'l Symp. Software Testing and Analysis, pp. 102-112, Aug. 2000.

[8] T. L. Graves, M.J. Harrold, J-M. Kim, A. Porter and G. Rothermel, "An empirical study of regression test selection techniques", 20th Int'l Conference on Software Engineering, Apr. 1998.

[9] M.J. Harrold and G. Rothermel, "Empirical Studies of a Prediction Model for Regression Test Selection", IEEE Trans. On Software Eng., vol. 27, no. 3, Mar. 2001.

[10] M. J. Harrold, "Testing Evolving Software", Journal of Systems and Software, vol. 47, no. 2-3, pp. 173-181, Jul. 1999.

[11] M.J. Harrold, R. Gupta and M.L. Soffa, "A Methodology for Controlling the Size of a Test Suite", ACM Trans. Software Eng. And Methodology, vol. 2, no. 3, pp. 270-285, July 1993.

[12] J.R. Horgan, and S.A. London, "ATAC: A data flow coverage testing tool for C", Proc. of Symp. On Assessment of Quality Software Development Tools, pp. 2-10, 1992.

[13] D. Rosenblum and G. Rothermel, "An Empirical comparison of regression test selection techniques", Proceedings of the Int'l Workshop for Empirical Studies of Software Maintenance, Oct. 1997.

[14] G. Rothermel, R.H. Untch and M.J. Harrold, "Prioritizing Test Cases For Regression Testing", IEEE trans. On Software Engineering, vol. 27, no. 10, Oct. 2001

[15] G. Rothermel, R.H. Untch, C. Chu, and M.J. Harrold, "Test Case Prioritization: An Empirical Study", Proc. Int'l Conf. Software Maintenance, pp. 179-188, Aug. 1999.

[16] G. Rothermel, M.J. Harrold, J. Ostrin and C. Hong, "An Empirical Study of the Effects of Minimization on the Fault Detection Capabilities of Test Suites", Proc. Int'l Conf. Software Maintenance, pp. 34-43, Nov. 1998.

[17] G. Rothermel and M.J. Harrold, "A Safe, Efficient Regression Test Selection Technique", ACM Trans. Software Eng. And Methodology, vol. 6, no. 2, pp. 173-210, Apr. 1997.

[18] G. Rothermel and M. J. Harrold, "Experience with Regression Test Selection", Proc. of the Int'l Workshop for Empirical Studies of Software Maintenance, Monterrey, CA, Nov. 1996.

[19] G. Rothermel and M.J. Harrold, "Analyzing Regression Test Selection Techniques", IEEE Trans. Software Eng., vol. 22, no. 8, pp. 529-551, Aug. 1996.

[20] A. Srivastava and J. Thiagarajan, "Effectively Prioritizing Tests in Development Environment", Proc. Int'l Symp. Software Testing and Analysis, July. 2002.

[21] A. Srivastava, A. Edwards, and H. Vo, "Vulcan: Binary Transformation in a Distributed Environment", Microsoft Research Technical Report, MSR-TR-2001-50.

[22] A. Srivastava and D. Wall. Link-Time Optimization of Address Calculation on a 64-bit Architecture. Symposium on Programming Language Design and Implementation, 1994, pp 49-60.

[23] A. Srivastava and A. Eustace, "ATOM – A System for Building Customized Program Analysis Tools", Symposium on Programming Language Design and Implementation, 1994, pp. 196-205, 1994.

[24] A. Srivastava and D. Wall. A Practical System for Intermodule Code Optimization at Link Time. Journal of Programming Language, 1(1):1-18, March 93.

[25] F. Vokolos and P. Frankl, "Empirical evaluation of the textual differencing regression testing techniques", Int'l conference on Software Maintenance, Nov. 1998.

[26] F. Vokolos and P. Frankl, "Pythia: a regression test selection tool based on text differencing", Int'l conference on reliability, Quality and Safety of Software Intensive Systems, May 1997.

[27] Z. Wang, K. Pierce, and S. McFarling, "BMAT: A Binary Matching Tool for Stale Profile Propagation", The Journal of Instruction-Level Parallelism, vol. 2, May 2000.

[28] W.E. Wong, J.R. Horgan, S. London, and A.P. Mathur, "Effect of Test Set Minimization on Fault Detection Effectiveness", Software-Practice and Experience, vol. 28. no. 4, pp. 347-369, Apr. 1998.

[29] W.E. Wong, J.R. Horgan, S. London, and H. Agrawal, "A Study of Effective Regression Testing in Practice", Proc. Eighth Int'l Symposium Software Reliability Eng., pp. 230-238, Nov. 1997.

[30] W.E. Wong, J.R. Horgan, A.P. Mathur, and A. Pasquini, "Test Set Size Minimization and Fault Detection Effectiveness: A Case Study in a Space Application", Proc. 21st Ann. Int'l Computer Software & Applications Conf., pp. 522-528, Aug. 1997.

[31] ISO/IEC 23271, ISO/IEC 23272, ECMA-335, ECMA TR84,