Mining Software History to Improve Software Maintenance Quality: A Case Study

Alexander Tarvo, *Microsoft*

Errors in software updates can cause regressions—failures in stable parts of the system. The Binary Change Tracer collects data on software projects and helps predict regressions in software projects.

Software maintenance includes correcting discovered faults, adapting the system to changes in the environment, and improving the system's reliability and performance.[1] These activities result in system modifications that are distributed to customers as updates. Microsoft Windows is no exception to this process. After Microsoft releases a new version of Windows, the Windows Serviceability team makes post-release changes (fixes) to the system. However, any incorrect changes will cause *software regressions*—failures in already stable features and parts of the system. Regressions are exceptionally painful for customers: imagine what will happen if your computer suddenly stops working after you install an update.

The best method to avoid regressions' negative consequences is extensive testing of all fixes. However, because of the large user base and the growing number of Windows versions we must support, this method results in a constantly increasing amount of testing for the fixes. This requires us to distribute our limited resources in the most efficient way: we must detect the riskiest software updates and concentrate on testing them. This will let us detect and fix possible regressions early, before releasing the updates to customers. So, we need a tool or a method that can predict the amount of risk for each fix.

To meet this need, I developed the Binary Change Tracer (BCT). BCT extracts information on all changes that have happened to Microsoft Windows. Data mined with this tool let us build a statistical model that predicts each fix's risk of regression. This approach exploits features of a standard engineering process, so it can used for a variety of software projects.

Risk Prediction in Software Projects

Considerable research exists on risk prediction in software projects. In particular, researchers have concentrated on *fault proneness prediction* (FPP) models.[2–4] These models tend to predict which parts of the newly developed software system will be most prone to failures. According to FPP models, a system consists of components described by metrics (numeric properties) such as code complexity or the number of prerelease changes. Some statistical model, such as a decision tree[2] or logistic regression,[3] uses these metrics to predict the components' fault proneness. Initially, you train the

0740-7459/08/$25.00 © 2009 IEEE

statistical model on data about a similar software system for which you know the components' metrics and fault proneness (which components have failed). Once you've trained the model, you can use it to predict the new system's fault proneness.

Unfortunately, FPP models aren't designed to predict the risk for a particular software update. Nevertheless, you can build a model to predict software regressions that works similarly to FPP models: a set of metrics will describe each code change. On the basis of these metrics, some statistical method will predict each change's regression risk (see Figure 1).

Predicting software regressions is a much less explored area; very few research papers address this topic.[5,6] This lack of research might be due to the difficulties of mining data on fixes. To be trained, a statistical model requires detailed information on hundreds of prior fixes. Unfortunately, such information is scattered throughout multiple data sources: version control systems (VCSs), bug-tracking databases (BTDs), mailing lists, and even the program's source code. When the data is in these discrete sources, it's hardly accessible and so is of little value. To be useful, we must extract it, clean it, and convert it into an appropriate format. So, to build a regression prediction model, we need to develop a special program called a *mining tool* that performs these tasks.

Researchers have recently discussed several designs for mining tools. Usually, this has occurred in the context of some concrete problem, such as discovering fix-introducing changes,[6] helping new programmers understand a large project,[7] or discovering patterns in a project's history.[8] Instead of developing a specialized tool just for extracting fix metrics, we created a more flexible tool that we can use for other purposes—BCT.

Evolution of a Software System

When any change in Windows is necessary, a program manager, test engineer, or support specialist creates a bug record in the BTD. Each bug record has a unique identifier (bug ID) and contains fields describing the change: its type (whether it's a bug fix or new functionality), the date the record was created, and other information. If the bug is a fix for a regression, the record includes a reference to the bug that caused the regression.

Once initial analysis of the bug is done, a developer downloads the current version of the source code from the VCS and makes the change. The VCS stores a complete history of changes to the product's code at the source file level. When a developer adds a new file to the VCS, the file's version is set to 1.

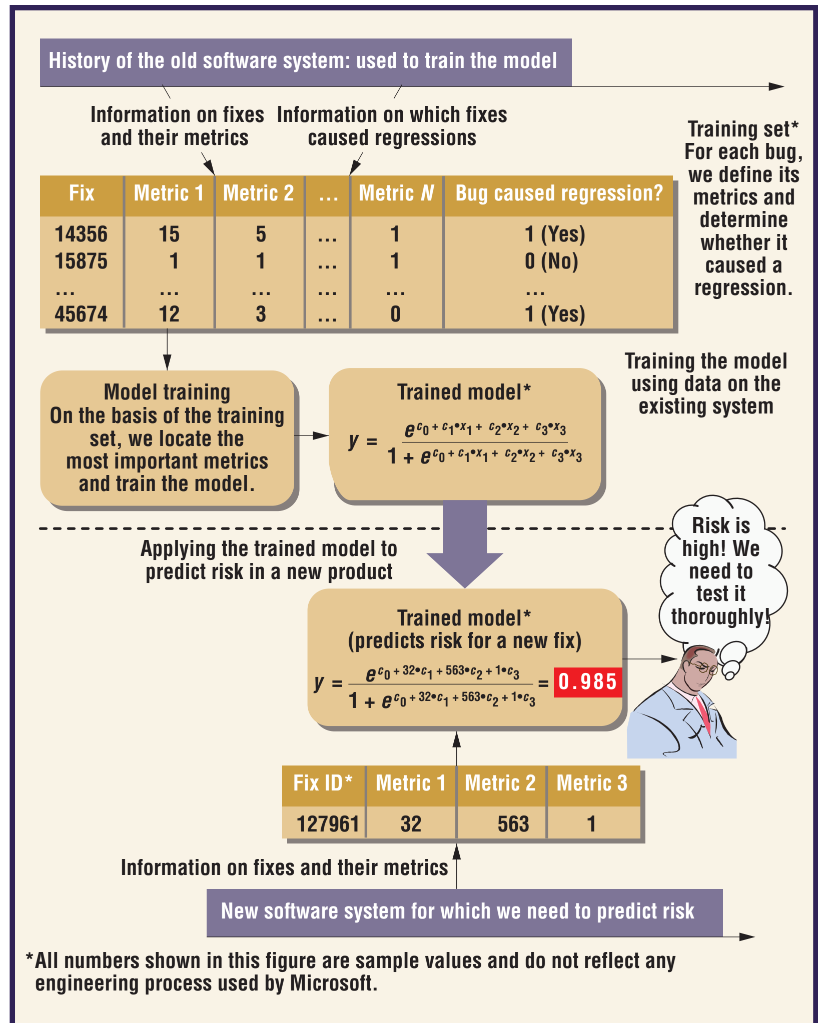

Figure 1. Building the regression prediction model. The model is trained using the known data and then used to predict risk for a new fix.

Each time the file changes, its version number increases by one.

After the developer implements a change and the change passes some basic functionality testing, he or she integrates the changed source files back into the VCS. The integration takes the form of an atomic transaction, or *check-in*. Each check-in has a unique number (check-in ID) and contains a list of changed source files, developer comments (description), the change's date, and the developer's name.

Once the fix's development is complete, the test engineer must test the change. On the basis of the information about the change and the test engineer's knowledge of the system, he or she estimates the necessary testing for that fix—that is, which tests to run and what level of testing is required. Many factors influence the amount of testing, but the most obvious is the regression risk: the higher the risk, the more thorough the testing should be.

If testing reveals any problems with the change, the test engineer asks the developer to fix them. Otherwise, the test engineer marks the bug record

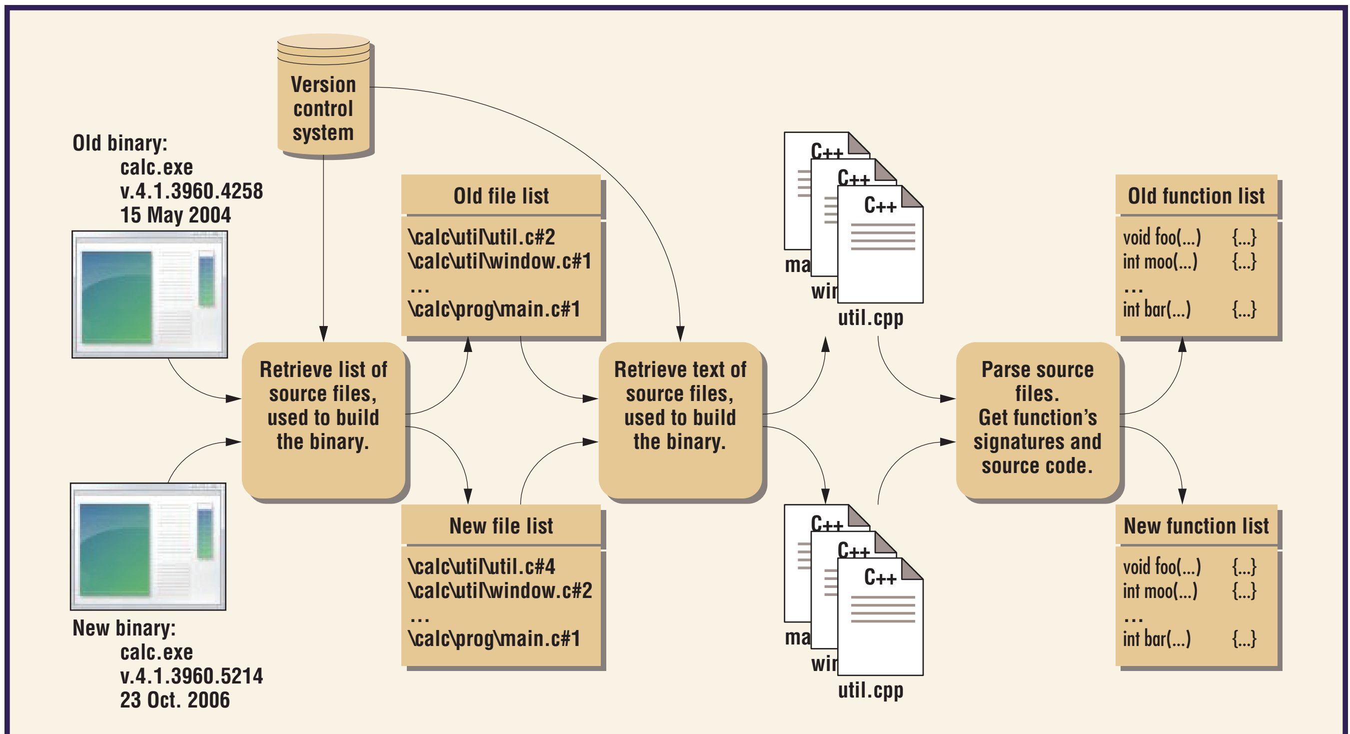

Figure 2. Dividing the binary into components. The further analysis will be carried out in the context of these smaller components.

"closed," and a software update can be built. This update is a set of new binary modules (*binaries*) whose code was changed owing to the fix. These binaries are "wrapped" into a package and sent to the customer.

So, by looking at the software system and the history of its changes, we find several different entities: bug records, developers, check-ins, source files, functions, and binaries. These entities are related. For example, each check-in consists of source files, and each source file can be divided into source functions. Each entity is uniquely identified by its ID and described by its metrics. For example, a source file can be identified by the combination of its name and version number and is characterized by size, complexity, and the number of prerelease changes. BCT's goal is to identify these entities, retrieve their metrics, and restore the relations between them.

Mining Information about the Software System

While analyzing the system, BCT uses a decomposition of the system into binary modules. The tool analyzes them one by one, accumulating the results until it has retrieved a complete history of changes in the system's source code. For each binary, BCT collects information on changes that occurred only during a certain time interval. This interval can be of an arbitrary length and is specified by time stamps that the user passes to the tool. Finally, BCT collects information on all changes in the system that occurred during the specified time interval.

Analysis of a binary takes four steps:

1. Dividing the binary into components.
2. Extracting the history of code changes.
3. Associating code changes with bugs.
4. Storing the extracted data.

Each step uses information obtained during the previous one, so the steps execute in strict order.

Dividing the Binary into Components

When the compiler builds the software system, in addition to binaries, it generates symbol files. These files let us determine which chunk of a binary corresponds to a particular line in the source file. BCT uses this information to retrieve a complete list of the names of the source files used to build old and new versions of the binary. Given these names, the tool locates corresponding source files in the VCS and, by knowing when the binary was built, accurately determines exact versions of these source files.

Next, BCT obtains the text of these source files from the VCS (see Figure 2). The tool processes each source file with a parser that locates all functions in the file and then reports each function's source code and signature.

Eventually, BCT restores componentization of the software system. It divides the binary logi-

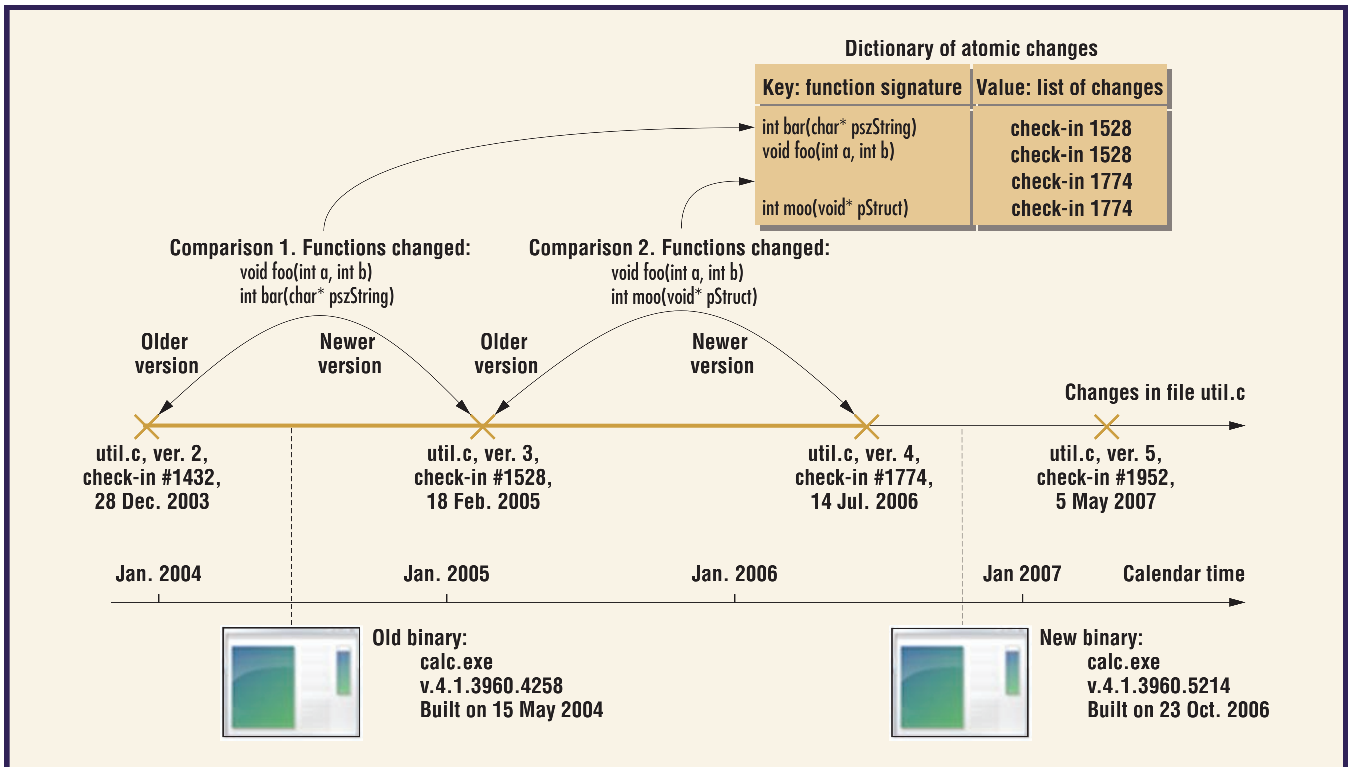

Figure 3. Extracting the history of code changes. For each source file, BCT scans the history of its changes and retrieves a list of changed functions.

cally into source files and divides the source files into functions (or methods, if the program has been developed using an object-oriented methodology). BCT stores each component in one of the internal lists. The *old file list* contains all the source files for building the binary's old version, and the *new file list* contains the source files for building the new version. Correspondingly, BCT stores functions in the *old functions list* and *new functions list*. Each function contains a link to its source file, and each source file has a list of its functions.

Extracting the History of Code Changes

For each source file of the binary, BCT retrieves all the versions newer than that source file's oldest version (used to build the old version of the binary) but older than or the same age as the source file's newest version (used to build a new version of the binary). Then, the tool performs a pairwise comparison of these versions in ascending order. That is, it compares the oldest version of the source file with the next one and so on, until it reaches the newest version (see Figure 3).

BCT compares each source-file pair at the function level. It parses the source code of each of the file's two versions (older and newer) and retrieves their lists of functions. For each function in the newer version, BCT tries to locate a function with the same signature in the older version. If it can't find a match, it considers the function to have been added (it's in the newer version of the file but not the older one) or deleted (it's in only the older version). If it finds the corresponding function in the older version of the source file, it compares the function's source code with that of the newer version. If the source code doesn't match, BCT considers the function changed.

BCT adds these changes to the dictionary of atomic changes that happened to the binary. This dictionary's keys are the changed functions' signatures, and its values are the lists of check-ins that caused these changes. The resulting dictionary contains all the changes that happened to all the functions between the old and new versions of the binary. Because every function has a link to the source file, the history of source file changes is retrieved automatically.

Associating Code Changes with Bugs

The VCS tells us what has been changed and when, but it doesn't say why; such information is in the BTD. To retrieve this information, we must relate the code changes (check-ins) in the VCS to the corresponding bug records in the BTD. Usually either the check-in or the bug record has a link to its counterpart.

Regarding the VCS, developers can put IDs of check-ins associated with a bug in the bug record's special field. BCT creates a list of check-in IDs and scans the BTD to find all bug records pointing to

To extract metrics such as code complexity, coupling, and object-oriented metrics, we use the MaX framework, another tool developed at Microsoft.

these check-ins. This is the most reliable way to link check-ins with bugs, but these check-in IDs aren't always present in the BTD.

Regarding the BTD, developers can also leave numeric IDs of corresponding bug records in the check-in description field. Unfortunately, analysis of this field is more difficult for the tool. The check-in description is just a text field, and any number in it could be considered an ID of the corresponding bug. So, BCT retrieves all numbers from the check-in description. If the BTD contains bugs with such IDs, the tool considers them related to the check-in. To remove all incorrect matches, BCT analyzes the closure dates of all bugs linked to the check-in. Normally, the test engineer should close a bug shortly after the developer registered the corresponding check-in in the VCS. Otherwise, BCT discards that relation between the bug record and the check-in.

This approach is similar to one that Jacek Sliwerski, Thomas Zimmerman, and Andreas Zeller developed;[6] however, our approach doesn't perform lexical analysis of check-in descriptions. Nevertheless, the combination of both data sources (the VCS and BTD) lets BCT accurately detect corresponding check-ins for the vast majority of bug records.

Storing the Extracted Data

This final step is straightforward: BCT stores all mined entities (bugs, check-ins, developers, binaries, source files, and functions), their metrics, and their links in the SQL database.

BCT itself doesn't mine all code metrics for changed functions and binaries. To extract metrics such as code complexity, coupling, and object-oriented metrics, we use the MaX framework, another tool developed at Microsoft.[9] Thanks to BCT's relational model, it easily incorporates information retrieved by MaX. For example, to get code metrics for some function, BCT uses the function's signature to query MaX databases. In this light, you can view BCT as a "metaminer": in addition to its own ability to mine the history of changes, it uses external information sources to obtain code metrics for changed components. This feature, as well as the ability to collect data on different levels of a system's componentization (binaries, source files, and functions), distinguishes BCT from other tools.[7,8]

Using BCT to Predict Software Regressions

To predict a fix's regression risk, we developed the Fix Regression Prediction (FRP) model. The FRP model is based on logistic regression, a statistical method.[10] The model predicts the regression risk $\hat{y}$ for a fix on the basis of the fix metrics $x_1, \ldots, x_n$ and model coefficients $c_1, \ldots, c_n$:

$$\hat{y} = \frac{e^{c_0 + c_1 x_1 + \ldots + c_n x_n}}{1 + e^{c_0 + c_1 x_1 + \ldots + c_n x_n}}$$

The fix metrics are the metrics of the corresponding bug record and all its related entities:

- the number of changes the fix caused (the number of components it changed),
- the experience of the developer who made the fix (the number of fixes he or she has done during the past year),
- the code metrics of the binaries, functions, and source files that the fix affected (their size, their complexity, and other metrics), and
- general fix metrics (whether the change introduced a new functionality, whether the fix was for a previously discovered regression, and other similar metrics).

The coefficients' values are inherent to the model; they contain that "hidden knowledge" regarding which fix metrics are the best indicators of the possible regression. Coefficients are calculated by Matlab when we train the model such that, for any given fix, the predicted regression risk $\hat{y}$ will be as close as possible to the real (observed) risk y.

The model $\hat{y}$ outputs a real number greater than 0 but less than 1; the higher $\hat{y}$ is, the higher the regression risk. To distinguish between high- and low-risk fixes, we use a numeric threshold t. If $\hat{y}$ exceeds t, the fix is risky (the probability of a regression is high); if $\hat{y}$ is less than t, the fix isn't risky (the probability of a regression is low).

Measuring Accuracy

The ideal classifier would detect all high-risk fixes without making mistakes. However, in reality, some mistakes are unavoidable. Each fix prediction has four possible outcomes:

- *True positive.* The fix was classified as risky and caused a regression.
- *False positive.* The fix was classified as risky but didn't cause a regression.
- *True negative.* The fix was classified as not risky and didn't cause a regression.
- *False negative.* The fix was classified as not risky but caused a regression.

These outcomes form the basis for several metrics for classifier accuracy. Two of the most widely

used metrics are the *true positive rate* (TPR) and *false positive rate* (FPR):

$$TPR = \frac{number\ of\ true\ positives}{number\ of\ all\ fixes\ that\ caused\ regression}$$

$$FPR = \frac{number\ of\ false\ positives}{number\ of\ all\ fixes\ that\ didn't\ cause\ regression}$$

Both the TPR and FPR have values between 0 and 1. A high-accuracy classifier will have a high TPR (it detects most high-risk fixes) and a low FPR (it makes few mistakes).

Obviously, the TPR and FPR values depend on t. If t is high, the classifier will detect fewer regressions (low TPR) but will make fewer mistakes (low FPR). Correspondingly, low values of t will result in high TPR and FPR. So, an indefinite number of (TPR, FPR) pairs is possible. These pairs form a graph with TPR values on the y-axis and FPR values on the x-axis. Such a graph is called a *receiver operating characteristic* (ROC) curve.

ROC curves are widely used to estimate classifier performance. The ideal classifier's ROC curve would be a straight line from (0, 1) to (1, 1). The worst classifier's ROC curve would be a line from (0, 0) to (1, 1). The *area under the ROC curve* (AUC) is a number characterizing the classifier's performance. AUC = 1 for the ideal classifier; AUC = 0.5 for the worst one.

Building the Model

To train the model and estimate its accuracy, we collected data on Windows XP and Server 2003 post-release fixes made from 2004 through 2006. For each fix, BCT reported the metrics and determined whether the fix caused any regressions (see Figure 1).

To measure the model's accuracy, we used a data-splitting technique.[11] During each split, we randomly divided the whole data set into a training set and test set. The training set contained 70 percent of all the fixes; the test set contained the remaining 30 percent. We performed 50 splits. For each split, we created an ROC curve and calculated the AUC. We then averaged the ROC curves to measure the model's accuracy.

To determine which metrics best predict risk, we used a stepwise procedure.[10] We started without any predictors in the model. During the forward step, we evaluated the available metrics one by one. We added a metric to the model if it resulted in a statistically significant improvement of accuracy. Once we tried all the metrics, we started the backward step. We removed a metric from the model if its accuracy didn't significantly deteriorate. We repeated this entire procedure until we couldn't add or remove any metrics.

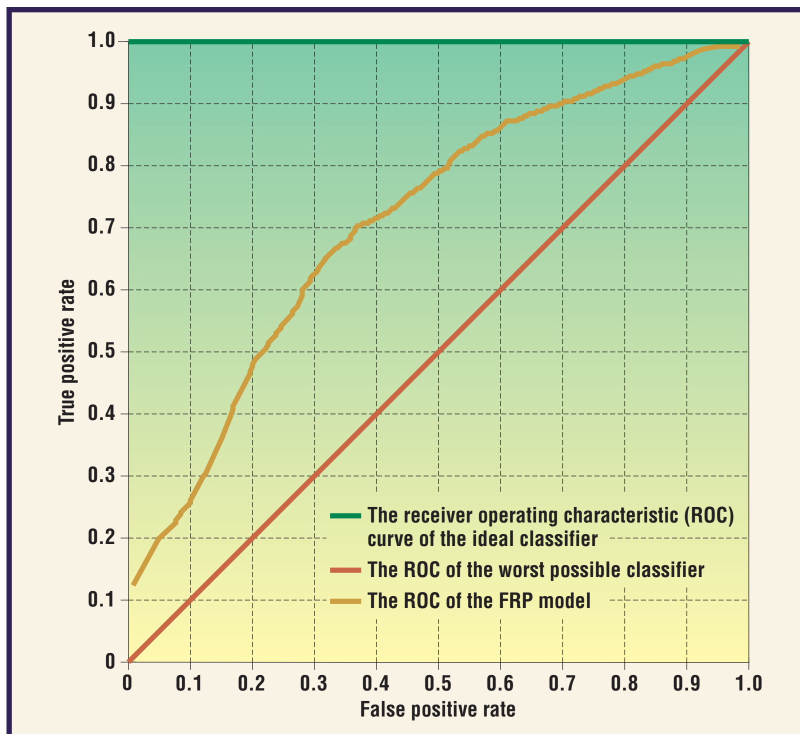

Figure 4. The Fix Regression Prediction (FRP) model's accuracy. We use the receiver operating characteristic (ROC) as an accuracy measure.

We found three metrics to be the best indicators of regression risk. The first is the number of source files affected by the fix ($p = 0.006$). The second is the summary change in the size of all functions changed by the fix ($p < 0.001$). The third is whether the fix introduced a new functionality ($p = 0.025$).

The final model's AUC was 0.71 (see Figure 4), which is significantly higher than the random draw (AUC = 0.5).

Deploying BCT at Microsoft

Since the FRP model's deployment in March 2008, the Windows Serviceability team has used it to predict regressions in upcoming Windows fixes.

When the new fix is available, BCT automatically extracts its metrics and passes them to the FRP model, which calculates the regression risk. On the basis of this risk, we assign the fix to one of these categories:

- *Very high risk*. The probability of regression is approximately X percent.
- *High risk*. The probability of regression is twice the average.
- *Average risk*. The probability of regression is about average.
- *Low risk*. The probability of regression is 50 percent of the system's average probability of regression.

About the Author

Alexander Tarvo is a software development engineer in test on Microsoft's Windows Serviceability team. He works on developing tools to mine software archives and on statistical models for predicting risk in software systems. His professional interests include data mining, machine learning, simulation, and software reliability. Tarvo received his MS in computer science from Chernigov State Technological University. Contact him at alexta@microsoft.com.

- *Very low risk.* The probability of regression is negligibly low.

Test engineers use this information to plan fix testing; fixes with high or very high risk should undergo rigorous testing, whereas fixes with low or very low risk can undergo reduced testing. The model has quickly gained popularity. Analysis of the usage logs shows that more than 70 percent of the test engineers on the team regularly use it.

To estimate the FRP model's utility, we compared its results with those from manual estimation of regression risk. Before deploying the model, the test engineer analyzed the fix and classified its risk as high, medium, or low. The results of such manual estimation were available for approximately 30 percent of all fixes in our data set. Tests revealed that the FRP model has significantly higher accuracy than test engineers. So, FRP proved useful for risk prediction.

We also use BCT to analyze daily builds. The practice of daily builds supports proper project management, especially when you're developing a large, complex software system.[12] However, if you look at the daily build (which is just a set of binaries), you'll have trouble finding which bugs were fixed. To get such information, we use BCT to compare the latest build with a previous one. The resulting report contains a complete list of bugs fixed in the latest build. For each bug, BCT lists the affected components, related check-ins, and the names of the persons who made the change.

Program managers have found daily-build reports useful; the reports provide a complete list of changes in the software project and help the managers track its progress. We're integrating build reports with the FRP model. We'll use the model to predict regression risk for each bug fixed in a build, which will help leverage testing of the daily builds.

BCT is a multipurpose tool that can mine vital information on a software system from structured data sources such as VCSs, BTDs, and a system's code. Its distinctive feature is an explicit representation of the mined data as a set of related entities. Such an approach allows the parallelization of data collection between multiple machines or CPUs. In addition, it lets us easily add new types of entities or metrics.

BCT proved a stable, scalable platform for mining data, but we're constantly improving it. We're trying to increase its scalability by reducing the number of calls to the VCS and BTD. In addition, we're taking steps to make it more flexible, so that any development team can easily adopt it with minimal effort. We're also looking for other uses of information mined with BCT, such as developing new statistical models. Finally, we're working on a BCT user interface that will let us visualize all processes occurring in the software system.

We hope that adoption of BCT and statistical models created with its help will further increase our organization's efficiency without decreasing software updates' quality.

Acknowledgments

I thank Jacek Czerwonka, John Erickson, Kanika Nema, and Brendan Murphy for reviewing earlier drafts of this article. I also thank the reviewers and guest editors of the special issue for their insightful comments.

References

1. *IEEE Std. 1219-1998, IEEE Std. for Software Maintenance*, IEEE, 1998.
2. T. Menzies, J. Greenwald, and A. Frank, "Data Mining Static Code Attributes to Learn Defect Predictors," *IEEE Trans. Software Eng.*, vol. 33, no. 1, 2007, pp. 2–13.
3. N. Nagappan, T. Ball, and A. Zeller, "Mining Metrics to Predict Component Failures," *Proc. 28th Int'l Conf. Software Eng.* (ICSE 06), IEEE CS Press, 2006, pp. 452–461.
4. N. Nagappan and T. Ball, "Use of Relative Code Churn Measures to Predict System Defect Density," *Proc. 27th Int'l Conf. Software Eng.* (ICSE 05), IEEE CS Press, 2005, pp. 284–292.
5. A. Mockus and D. Weiss, "Predicting Risk of Software Changes," *Bell Labs Tech J.*, vol. 5, no. 2, 2000, pp. 169–180.
6. J. Sliwerski, T. Zimmermann, and A. Zeller, "When Do Changes Induce Fixes?" *Proc. 2nd Int'l Workshop Mining Software Repositories*, ACM Press, 2005, pp. 24–28.
7. M. Fischer, M. Pinzger, and H. Gall, "Populating a Release History Database from Version Control and Bug Tracking Systems," *Proc. Int'l Conf. Software Maintenance* (ICSM 03), IEEE CS Press, 2003, pp. 23–32.
8. D. Ubrani and G.C. Murphy, "Hipikat: Recommending Pertinent Software Development Artifacts," *Proc. 25th Int'l Conf. Software Eng.* (ICSE 03), IEEE CS Press, 2003, pp. 408–418.
9. A. Srivastava, J. Thiagarajan, and C. Schertz, *Efficient Integration Testing Using Dependency Analysis*, tech. report MSR-TR-2005-94, Microsoft Research, 2005.
10. D.T. Larose, *Data Mining Methods and Models*, Wiley-Interscience, 2006, p. 322.
11. D. Hand, P. Smyth, and H. Mannila, *Principles of Data Mining*, MIT Press, 2001, p. 546.
12. S. McConnell, "Daily Build and Smoke Test," *IEEE Software*, vol. 13, no. 4, 1996, pp. 143–144.