Finding Dependencies from Defect History

Rajiv Das, Wipro Technologies
Jacek Czerwonka, Microsoft Corporation
Nachiappan Nagappan, Microsoft Corporation

Context – Windows Development

- Size and scope
  - 40+ MLOC
  - Development team spread all over the world
  - 1B+ users
  - 400,000 supported devices
  - 6,000,000 apps running on Windows
  - Up to 10 years of servicing
- Challenges
  - Large, complex codebase with millions of tests
  - Diverse customer base
  - Time and resource constraints
  - Diverse test execution
  - Very low tolerance to failure

Problems

- Unknown Dependencies
  - Static, Dynamic Analysis does not find everything
- Large number of Dependencies
  - How to prioritize Integration Testing when changes are routine and costs involved are high?

Motivation

Graphics Driver crashes whenever user 'Pastes' image to *Photo Editor*

- Internal defect in driver, exposed by unknown dependency between editor and driver

Defect Id	2125
Title	Graphics driver crashes on Paste
Status	Closed
Opened By	Alice
Opened On	7-November-2005
Affected Component	Drivers\Video\Driver.sys
Resolution	Fixed
Resolved By	Bob
Resolved On	1-December-2005

Sample Defect Record

* Source Component Photo Editor, not recorded explicitly

Definitions

- **Dependency:** *If defects are found frequently in component C_1 when component C_2 is tested, then C_2 may be dependent on C_1*
- **Source Component:** *The component containing the defect*
- **Affected Component:** *The component affected due to a defect*

Frequent Itemset Mining

If X and Y are items in the transaction dataset:

- ***Support*(*X*)**: probability of occurrence of *X*, *p(X)*.
- ***Confidence*(*X*=>*Y*)** : how frequently *Y* occurs when *X* occurs, *p (Y|X).*
- ***Importance* (*X*=>*Y*)**: The log likelihood of *Y* occurring with *X*, than without it i.e. $log \frac{p(Y \mid X)}{p(Y \mid \text{not } X)}$

In a transaction dataset, frequent itemsets X and Y can be found using **Dependence Rules Mining**:

1. *Support(X)*, *Support (Y)* and *Support(X and Y)* above threshold
2. *Confidence(X=>Y)* above threshold

How to Identify Dependencies

- Let C_S be source component
- Let C_A be affected component
- Find frequent pairs of source and affected components in the component map using Dependence Rules, C_S => C_A where
  1. *Support*(C_A), *Support*(C_S), *Support*(C_A *and* C_S) >= support cutoff
  2. *Confidence*(C_S => C_A) >= confidence cutoff
  3. *Importance*(C_S => C_A) is positive, meaning that the affected component is positively statistically dependent on the source component

How to Rank Dependencies

- Rank dependencies first by confidence and then by importance.
  - Higher confidence has higher rank
  - Higher importance has higher rank
- For *k* topmost dependencies,
  - First chose all dependencies greater than confidence cutoff
  - Then choose *k* dependencies out of them with largest importance

Example

Affected	Source
C1	C2
C3	C2
C4	**C2**
C1	C3
C2	C5
C1	C2
C4	**C2**
C1	C2

Component Map

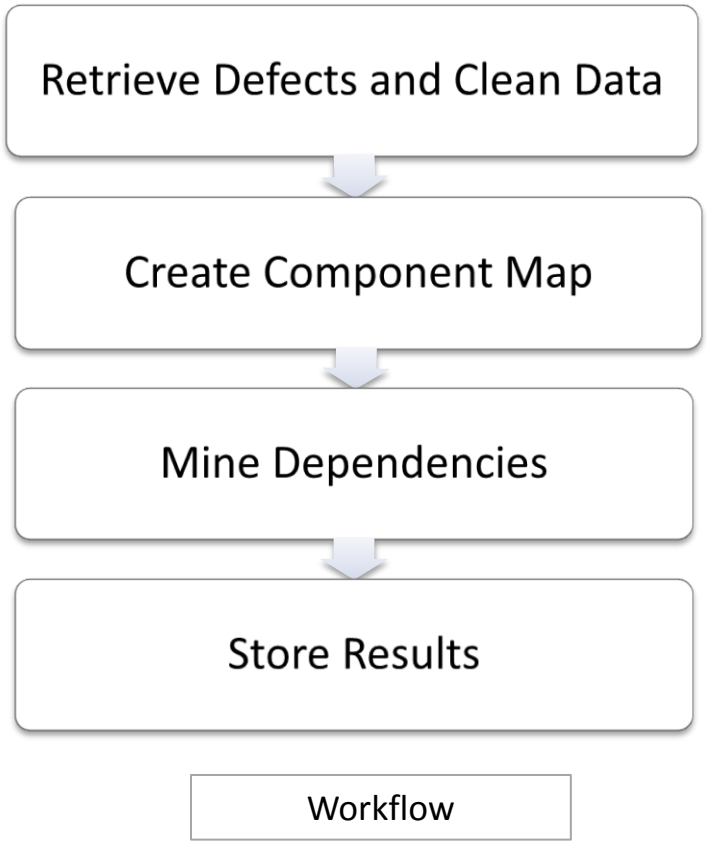
- **Support** 0.25
- **Confidence** 0.1
- **Importance** > 0

Rule	Support	Confidence	Importance
C2 => C4	2	2/6 = 0.33	0.176

Dependencies Found

Ladybug Tool

- Automates dependency discovery
- Easily Customizable
- Built on SQL Server Platform – Analysis Services, Integration Services, SQL Server



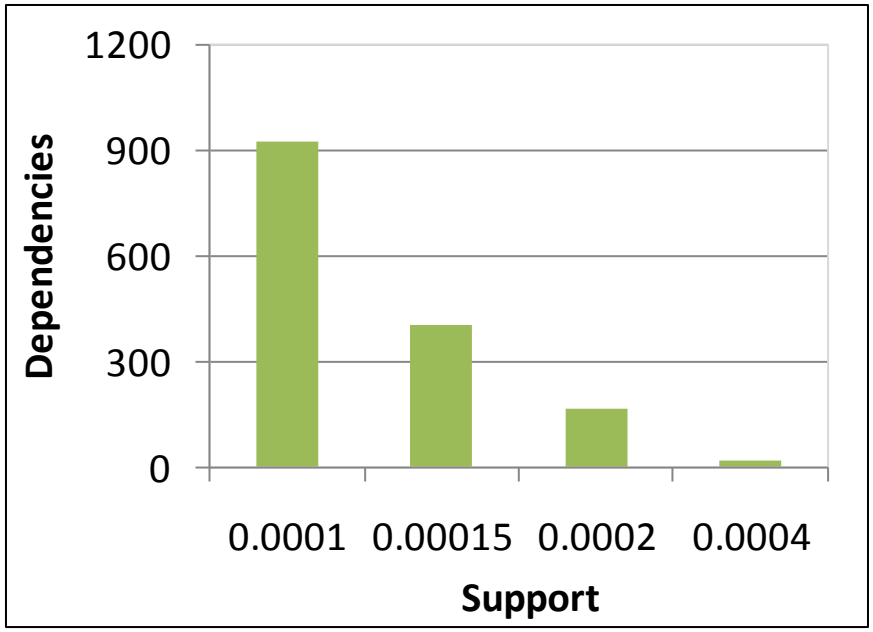
Experiment

- Pre-release Defects for Windows Vista and Windows Server 2008

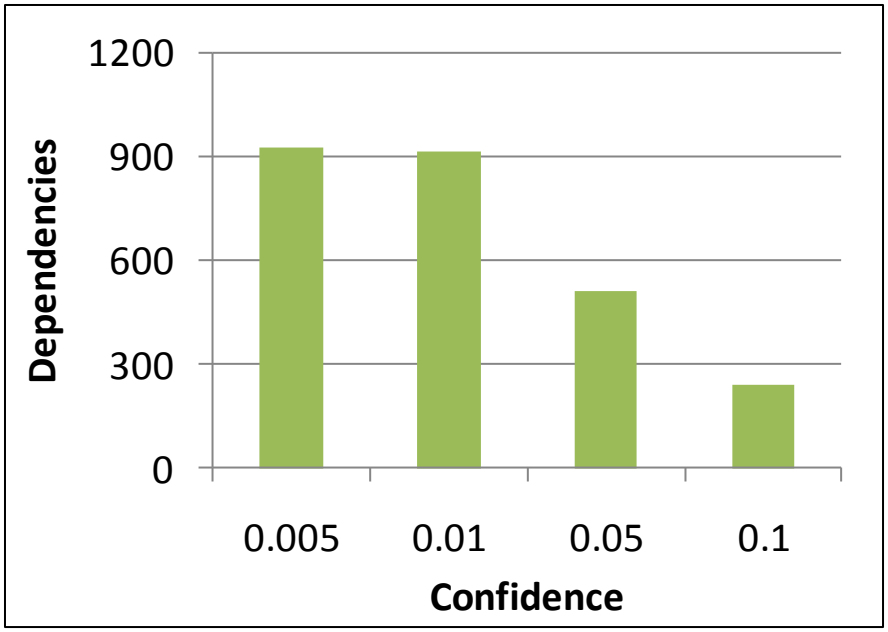
Size	92,976
Defects Included	28,762
Affected Components	1,649
Source Components	1,480

Input Component Map

Dependencies Found



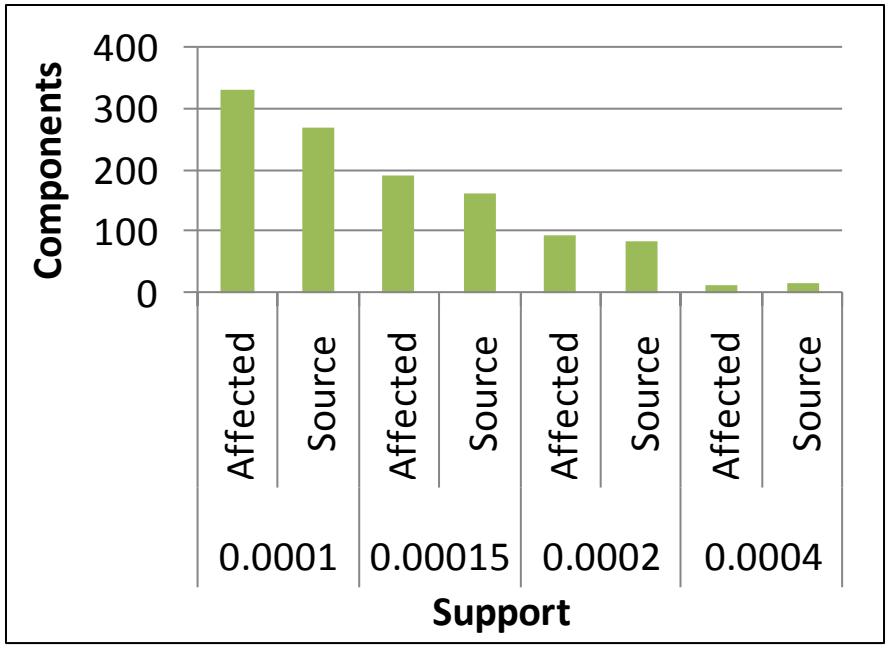
Dependencies found by varying Support with Confidence 0.005



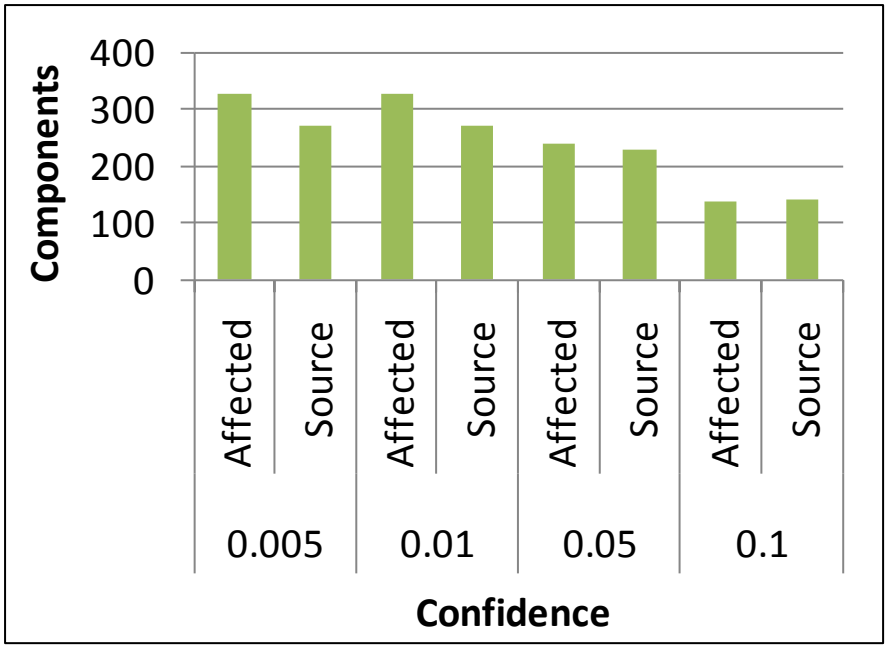
Dependencies found by varying Confidence with Support 0.0001

Outcome indicates that dependencies can be found for various combinations of support and confidence thresholds

Source and Affected Components



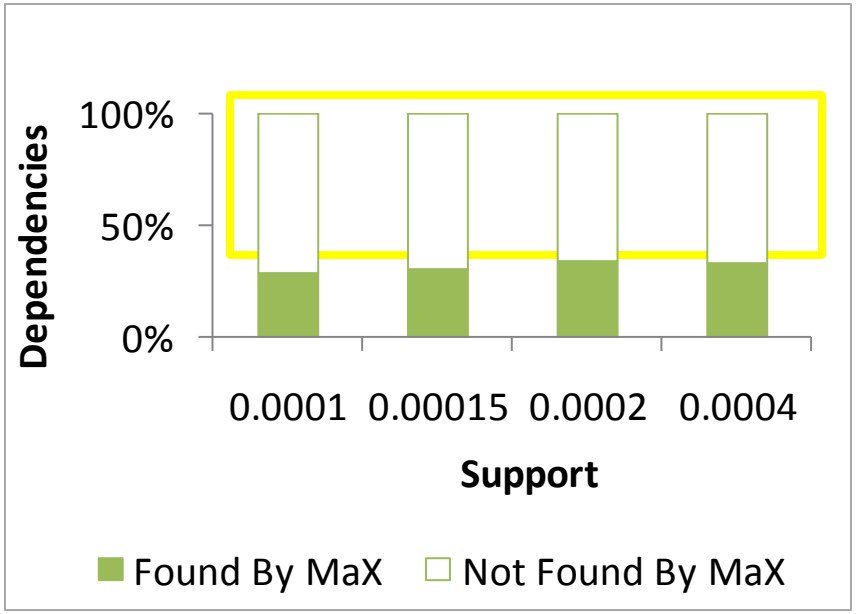
Source and Affected Components included in dependencies found by varying Support with Confidence 0.005



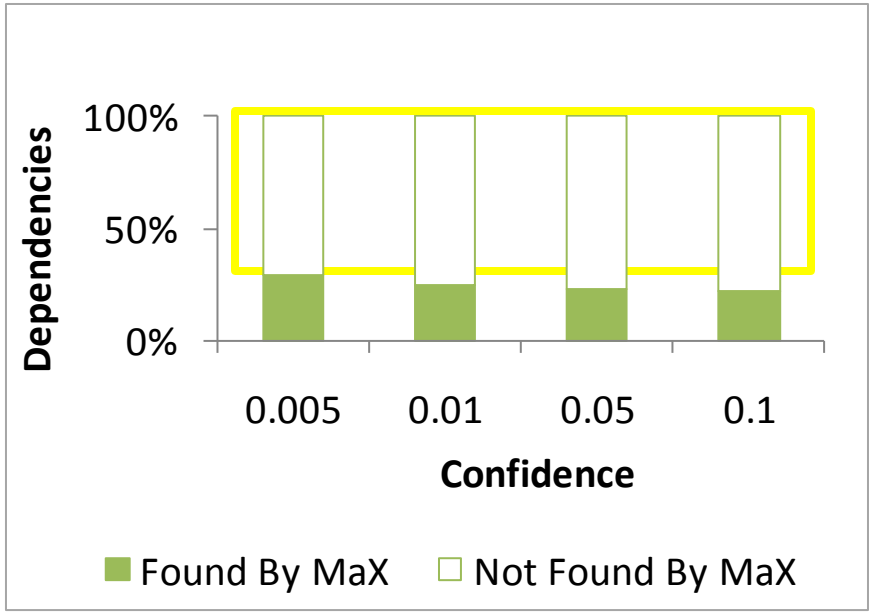
Source and Affected Components included in dependencies found by varying Confidence with Support 0.0001

Fewer components get included in the results as the thresholds are raised

Effectiveness



Comparison with MaX for dependencies found with different Support values with Confidence 0.005



Comparison with MaX for dependencies found different Confidence values with Support 0.0001

Outcome indicates our approach can possibly find new dependencies.

Manual Validation

Total dependencies	276
Votes Cast	211
Dependencies with Votes	182
Dependencies with Multiple Votes	29
Experts Invited	127
Experts Participated	**70**

Dependencies found by our method with support 0.0001 and confidence 0.005

Vote tally for different buckets of dependencies

In general, owners seemed to confirm the dependencies considerably more often than reject them

Manual Validation (2)

	Found by MaX	Not Found by Max	Row Total
Confirmed	27	**115**	142
Rejected	12	28	40
Total	39	143	182

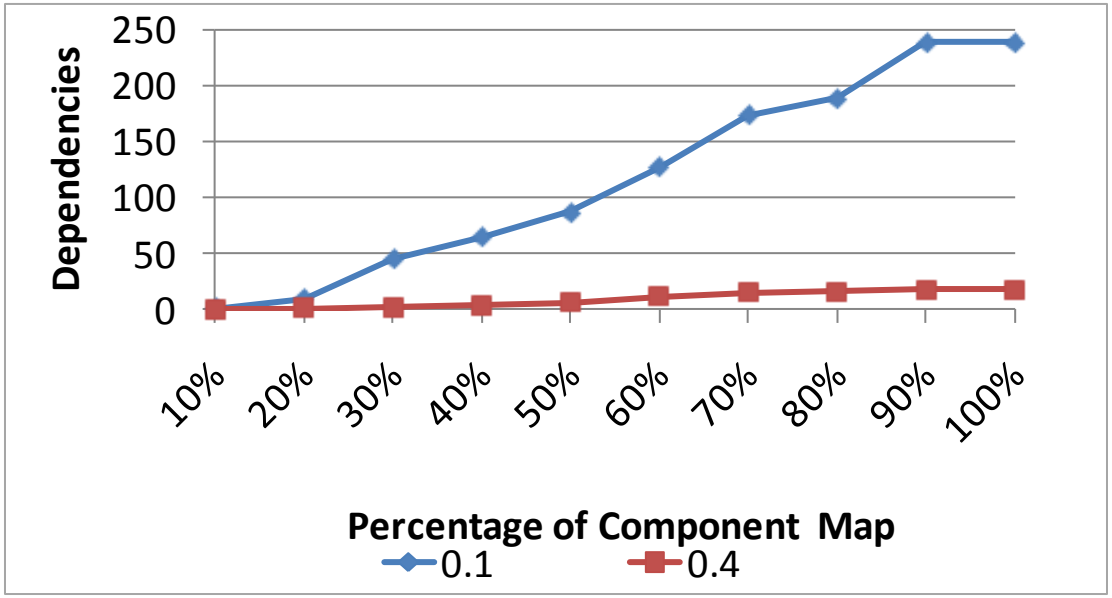
Contingency table showing votes versus detection by MaX

H_0: Experts voted based on the rule content and their background system knowledge independently of what the MaX data says, which they may have seen before

We do not reject H_0 at 95% confidence level using Chi-Square analysis.

It is possible to discover additional new important dependencies using our method

Applicability



Dependencies found as a function of defect reports, for different confidence values and support count 25, indicates more defect reports yielded more dependencies

We can start mining at early phase of software development and keep refining model over time as more data becomes available.

Alternative Input Component Maps

	Component Map 1	Component Map 2
Ownership Error (%)	0	5
Map Size	89,075	92,976
Defects	28,028	28,762
Affected Components	1,637	1,649
Source Components	1,470	1,480

- Alternative Component Maps were extracted using different values of Ownership Errors
- No noticeable difference in the results

* Detailed description available in paper

Threats to Validity

- Dependencies cannot be found for Components that have not been part of significant number of defects in the past
- Our study is on a well-componentized, large-scale software system with a stable development process and considerable number of defect reports.
- For practical applications, it may be useful to use higher thresholds to restrict the outcome to the most significant rules only.

Conclusions

- New Approach to *identified* software dependencies
- An approach to *rank dependencies* using defect history
- *Ladybug tool* to mine defect history for new dependencies
- Possible to *start mining* at any phase of development and refine models over time
- *Found a large number of dependencies* confirmed by experts but are not found by static analysis tools
- Ladybug analysis has been incorporated in a larger change analysis and test targeting system used in Windows Serviceability and recommendations are used by hundreds of engineers every month

Future Work

- Apply Ladybug to defect datasets of other software
- Look at ways of incorporating user judgment to generate better dependency recommendations

Thank You

Finding Dependencies from Defect History

Rajiv Das
Wipro Technologies,
Bellevue, Washington, USA
v-rdas@microsoft.com

Jacek Czerwonka
Microsoft Corporation,
Redmond, Washington, USA
jacekcz@microsoft.com

Nachiappan Nagappan
Microsoft Corporation,
Redmond, Washington, USA
nachin@microsoft.com

ABSTRACT

Dependency analysis is an essential part of various software engineering activities like integration testing, reliability analysis and defect prediction. In this paper, we propose a new approach to identify dependencies between components and associate a notion of "importance" with each dependency by mining the defect history of the system, which can be used to complement traditional dependency detection approaches like static analysis. By using our tool Ladybug that implements the approach, we have been able to identify important dependencies for Microsoft Windows Vista and Microsoft Windows Server 2008 and rank them for prioritizing testing especially when the number of dependent components was large. We have validated the accuracy of the results with domain experts who have worked on designing, implementing or maintaining the components involved.

Categories and Subject Descriptors

D.2.8 **[Software Engineering]**: Metrics—Process metrics, Product metrics. D.2.9 **[Software Engineering]:** Management—Software quality assurance (SQA)

General Terms

Measurement, Reliability, Experimentation

Keywords

Empirical study, defect database, dependency, dependence rules

1 INTRODUCTION

Understanding software dependencies is essential to performing various software engineering activities including integration testing [17], reliability analysis for code churn [11], defect prediction based on complexity of the dependency graph [20] and aiding developers [12]. Any dependency missed during integration testing can cause a costly new regression. For a large and continually changing software system like Microsoft Windows, a reliable dependency detection mechanism is necessary. Complete integration testing of the entire system is unlikely to be cost effective given the volume of changes that happen every day and the complex setup and hardware requirement for so many scenarios.

Dependencies in a system are usually identified by static analysis. Tools like MaX can determine various control flow and resource dependencies such as imports, exports and runtime dependencies [17]. Data flow dependencies along with control flow dependencies can be identified with CodeSurfer [3]. Dynamic tracing by using Magellan/Echelon [16] can identify for us control flows at runtime and possibly reveal new paths not found by static analysis. However, for example, new resource dependencies will not be found by MaX if adapters for the resource type are not available. Other static analysis tools may have similar limitations. Moreover, in order to use dynamic analysis we need to have tests available to hit new paths and the ability to associate tests reliably with the component, for which they were originally written. Clearly, using only one approach may not be practical. While multiple approaches can be used together to find more dependencies, dependencies originating from semantic behavior of systems are not usually detected in open process architectures like Microsoft Windows that does not enforce high-level semantic contracts [8]. Nevertheless, we must strive to find as many dependencies as possible.

In addition, when considerable number of dependent components is found for a change (e.g. an operating system kernel binary that is used by almost every other binary), it is necessary to have a notion of importance for each dependency so that we can rank them and prioritize testing. For integration testing, those dependencies that are more likely to find defects could be prioritized higher.

In this paper, while we build our dependency identification approach around integration testing, we believe dependencies thus identified may be useful in other software engineering activities as well.

The defect history has been a useful source of information in various studies like determining implementation expertise [4], Feature Tracking [7], Component Failure Prediction [10], and Defect Correction Effort Prediction [15]. The defect record contains the circumstance in which it occurred, its causes as well as the ensuing changes in the system. For large software systems, the defect database would contain many defects. We propose to look at the defect history of the system and see which components frequently found defects in a particular changed component. In this context, we define dependency in the following way:

Definition 1: *If defects are found frequently in component C_1 when component C_2 is tested, then C_2 may be dependent on C_1.*

In the usual sense of software dependency, C_2 being dependent on C_1 does not imply that C_1 depends on C_2. Further, when C_2 is dependent on C_1 and C_1 is dependent on another component C_0, it does not imply that C_2 depends on C_0. A component may have multiple dependencies on itself and conversely, it can itself depend on many other components.

We applied our approach to the defect database of Microsoft Windows Vista and Windows Server 2008 and found several important dependencies. These results have been validated with domain experts and are being used for recommending dependencies as part of a Change Analysis system for Windows.

1.1 CONTRIBUTIONS

The significant contributions of the work are:

- A *new approach for identifying dependencies* in software systems, to our knowledge
- An approach to *rank dependencies* using defect history that can be used to *prioritize higher the frequently defect-finding components* when a large number of dependencies are known
- A tool to mine dependencies from the defect history – Ladybug that automates the entire mining process from defect retrieval to results storage using Dependence Rules mining [14] to identify frequent component pairs and measure the statistical dependency between them.
- We show that *defect mining can be started at any phase of software development and the models can be refined* as new defects are found.
- A *detailed experimental study* of our techniques applied to a large software system with numerous end-users. In this study, we discovered a large number of confirmed dependencies that were not found by static analysis.

The remainder of the paper is organized as follows. First, we review related work. In Section 2, we describe the main principles behind our work. The Ladybug tool is described in Section 3. We discuss implementation details and results of our study in Section 4. Finally, in Section 5 we make concluding remarks.

1.2 RELATED WORK

Various approaches have been used to define software dependencies and develop methods and tools to identify them. Some of the early works in this field include the formal dependency model defined by Podgurski et al [13], and the Dependency Analysis Toolset for the dependency graph created by Wilde et al [18]. In general, static analysis is used for dependency identification. Some of the systems built on this approach are CodeSurfer developed by Anderson et al to aid software inspection using the data and control flow dependency graph of the entire program [3] and the MaX toolkit developed by Srivastava et al to guide integration testing that analyzes the compiled program rather than raw source code [17]. NDepend is another dependency discovery tool primarily targeted to assist software development on the Microsoft.Net platform [12].

Dynamic analysis was used alongside static analysis by Eisenbarth et al to map features to software components and hence aid program comprehension [6]. The Echelon tool developed by Srivastava et al can identify the most important tests for the change and, therefore, dependent components for integration testing [16].

Apart from this, we found that a data mining approach was used by Zimmermann et al in the eRose tool to discover program coupling, undetectable by program analysis, by mining version histories for co-occurring changes [19].

2 MINING DEPENDENCIES

Defects in a software component surface when other components use it, sometimes in unanticipated ways.

For example, a Graphics Driver is observed to crash whenever the Photo Editor tries to process a *Paste* command from the user. On investigation, we find an internal defect in the driver that was exposed by a dependency between the editor and the driver.

We call components like the Graphics Driver that contain the defect as *source* and components like the Photo Editor that expose the defect as *affected.* Note that though multiple components may be affected by the same defect, as the Photo Editor found this defect first it is considered the affected component.

To find dependencies, we look at the defect history for defects that were found in a component when another component was being primarily used (or targeted with testing), as in the example above. If activity on a component C_a frequently exposed defects on component C_s, we consider that component C_a is probably dependent on component C_s. We can find such frequent pairs in the dataset using Dependence Rule Mining [14].

2.1 DEFECT RECORD

Table 1 shows a fictional defect in **Drivers\Video\GDriver.sys** component that was found by **Alice** on **7-November-2005** and fixed by **Bob** on **1-December-2005.** Note that the defect record does not include the affected component, usage or deliberate testing of which triggered the failure. In fact, defect repository software is not usually configured to capture the affected component. This information was not explicitly stored in the Windows defect repositories we perused nor does it appear in the default configuration of other known software for defect tracking like Bugzilla for Mozilla [5].

Defect Id	2125
Title	Graphics driver crashes on Paste
Status	Closed
Opened By	Alice
Opened On	7-November-2005
Affected Component	Drivers\Video\GDriver.sys
Resolution	Fixed
Resolved By	Bob
Resolved On	1-December-2005

Table 1: Sample defect

However, it is *possible* to identify affected components, from information usually available in defect reports, as required in our analysis. We describe one such approach in section 4.1

2.2 DEPENDENCE RULES

Dependence Rules learning is a generalized form of Association Rules learning that can be used to identify frequent itemsets from a transaction dataset and measure the statistical dependence between frequently co-occurring itemsets [14]. Dependence Rules can be mined using the *Apriori* algorithm with minimum cutoffs for *support* and *confidence* [1, 2] and *importance* [9].

If X and Y are items in the transaction dataset, these measures are defined as follows:

Definition 2: Support(X) is the probability of occurrence of X, $p(X)$.

Definition 3: Confidence(X=>Y) is the measure of how frequently Y occurs when X occurs, $p(Y|X)$.

Definition 4: Importance (X=>Y) is the log likelihood of Y occurring with X, than without it i.e. $log \frac{p(Y \mid X)}{p(Y \mid \text{not } X)}$

Positive importance implies that *Y* occurs when *X* does while a negative value implies *Y* will not occur when *X* occurs. For zero importance, *X* does not influence the occurrence of *Y*. In other words, importance determines the statistical dependence between the itemsets.

For our problem, the defect dataset forms the transaction dataset and each defect represents a transaction. If C_A denotes the itemset containing the affected component and C_S the itemset containing the source component, our goal is to find rules of the form $C_S => C_A$ such that the following conditions are satisfied:

1. *Support*(C_A), *Support*(C_S), *Support*(C_A *and* C_S) are not less than support cutoff
2. *Confidence*($C_S => C_A$) is not less than confidence cutoff
3. *Importance*($C_S => C_A$) is positive

Each rule found after mining is one of the most likely affected components for a given source or, in other words, a dependency.

2.3 RANKING DEPENDENCIES

All the three parameters used in Dependence Rules mining – support, confidence and importance – have been used for rule ranking in literature.

In our experiments, we ranked dependencies first by confidence and then by importance. When ranking by confidence, a dependency with higher confidence than another is ranked higher. Similarly, to rank by importance a dependency with higher importance is ranked higher. To select *k* topmost dependencies, we first chose all dependencies greater than the confidence cutoff and then choose *k* dependencies with the largest importance values.

2.4 EXAMPLE

We now attempt to discover dependencies from a sample component map shown in Table 2. Such a map can be derived from the defect data as described in Section 3.1

Affected	Source
C1	C2
C3	C2
C4	C2
C1	C3
C2	C5
C1	C2
C4	C2
C1	C2

Table 2: Sample component map

On mining for Dependence Rules with thresholds – minimum support 0.25, minimum confidence 0.1 and positive importance, we get a single dependency - C4 depends on C2 with confidence 0.333 and importance 0.176.

Rule	Support	Confidence	Importance
C2 => C4	2	0.333	0.176

Table 3: Dependence rules found from table 2

3 LADYBUG

Ladybug tool has been developed to automate our dependency discovery method. It is built using Microsoft SQL Server 2005 Integration Services and Microsoft SQL Server 2005 Analysis Services and Microsoft SQL Server 2005.

Currently Ladybug is implemented as an Integration Services package that has separate tasks for each of the workflow stages described in Section 3.1. Besides, there are setup scripts for SQL Server database initialization and Analysis Services mining model creation. This design allows Ladybug to be easily configurable for specific applications. For example, we can customize the defect retrieval task to the specific schema of the defect store or various types of stores including SQL databases, XML files and web services, apply custom filters for different noise patterns and experiment with various thresholds for the mining parameters. Moreover, new tasks can be easily added or existing ones modified.

3.1 Workflow

Figure 1 shows the different phases in the Ladybug workflow.

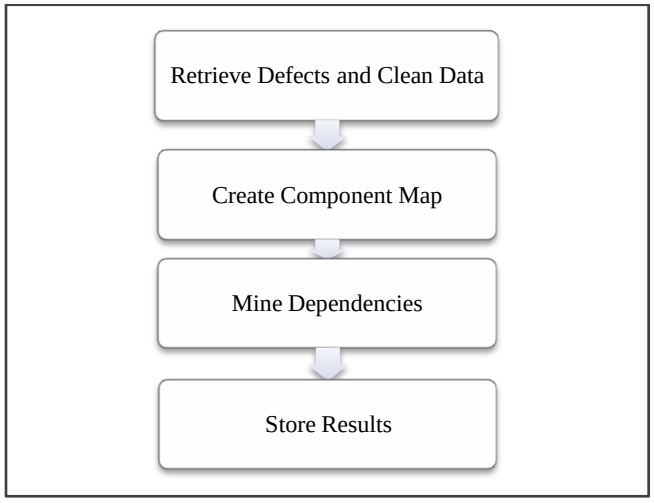


Figure 1: Ladybug workflow

1. **Retrieve Defects and Clean Data:** In this stage, we retrieve defects from the data store and clean the data if necessary. The attributes retrieved for the defects are usually the ones shown in Table 1.

2. **Create Component Map: I**f the defect data contains the affected component for each defect a component map as shown in Table 2 is readily available and we can proceed to stage 3.

   However, as we have noted in Section 2.1, a defect record does not usually contain the affected component and it is necessary to implement an application-specific approach to determine the same, before a component map can be created.

3. **Mine Dependencies:** Ladybug uses the component map as input and creates models for mining Dependence Rules using the Microsoft Association Rules algorithm in SQL Analysis

Services. It uses manual thresholds for support and confidence and *zero* for importance.

4. **Store Results:** Dependencies found by the mining step above are retrieved from the Analysis Services data store, transformed, and stored in an easily retrievable form as required by the application.

4 EXPERIMENTAL RESULTS

We applied Ladybug to mine the pre-release defect reports of Microsoft Windows Vista and Windows Server 2008.

4.1 EXPERIMENTAL SETUP

Customization

In order to use Ladybug for the current dataset, we customized the various stages of the workflow as follows:

1. **Retrieve Defects and Clean Data:** We configured the extraction routine for the target data store schema. Further, we added a cleanup sub-task to remove defects on components that no longer exist (placeholder components used during development).

2. **Create Component Map:** We mapped all components owned by the defect opener on the defect opening date as affected components, provided she did not own the source component. If the opener owned no components, no associations were made. This approach complies with the necessary condition in Definition 1 that the defect in the source component was found while *primarily* using the affected component. Further, more than one component can be mapped as a *potential* affected component.

   In order to get the components owned by a person on a date, we used a *temporal ownership map*.

   Create Temporal Ownership Map: Defects in the data store were typically resolved by Component owners. Therefore, for each (Component, ResolvedBy) pair in the retrieved dataset we get an *OwnershipItem* as follows:

   OwnershipItem
   {
   Component,
   ResolvedBy AS **Owner**,
   MIN (ResolvedOn) – ε/2 * Range AS **Start**,
   MAX (ResolvedOn) + ε/2 * Range AS **End**
   }

   Range denotes the interval between MAX (ResolvedOn) and MIN (ResolvedOn) in days.

   The *ownership error* ε is used to model for the possibility that a person had owned a component beyond the *range* observed. In practice, a larger value of this parameter allows us to associate more potential affected components with a defect. This value will usually be a small fraction of the observed ownership period. For example, if the maximum observed ownership over all component-owner pairs is 500 days and we want to allow at most 10 additional days of ownership for any pair, we may use the value of the parameter ε as 10/500 or 2%. Note that we did not account for discontinuities in component ownership, as that was deemed very unlikely in this case.

3. **Mine Dependencies:** We experimented with various thresholds for support and confidence as described in Section 4.2

Dataset

From around 28,000 defects, we created component maps with different ownership errors. Table 4 shows two maps created with ownership errors 0% and 5% respectively. In our dataset, the maximum observed ownership period was around 2400 days and the minimum observed was 1 day. Therefore, 5% error means we add at most 120 days to the ownership periods.

	Component Map 1	**Component Map 2**
Ownership Error (%)	0	5
Map Size	89075	92976
Defects	28028	28762
Affected Components	1637	1649
Source Components	1470	1480

Table 4: Component maps used in experiment

We can see that Component Map 2 includes more defects, source and affected components due to greater ownership error. Using these maps, we created various mining models by varying support and confidence thresholds.

4.2 DISCUSSION

Figure 2 and 3 show the outcome of eight mining models created with Component map 1 and Component map 2 using various support values and confidence 0.005. In figure 2, we can see that the number of dependencies discovered decreased gradually with increase in support. In figure 3, we can see at most around 22% of the affected and around 21% of source components got included in the Dependencies, for either map. Further, for the same support, corresponding models for Component Map 1 and Component Map 2 had similar number of Dependencies and included components.

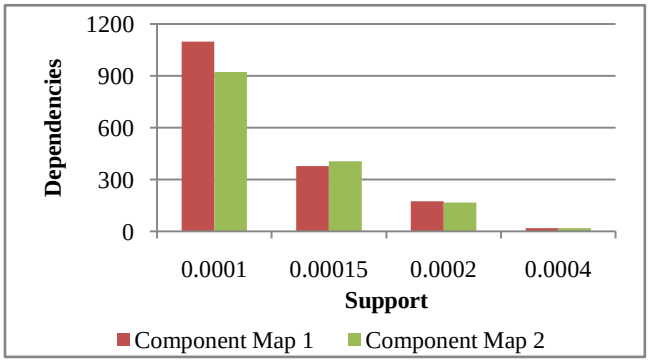


Figure 2: Dependencies found with confidence 0.005 and different support cutoffs

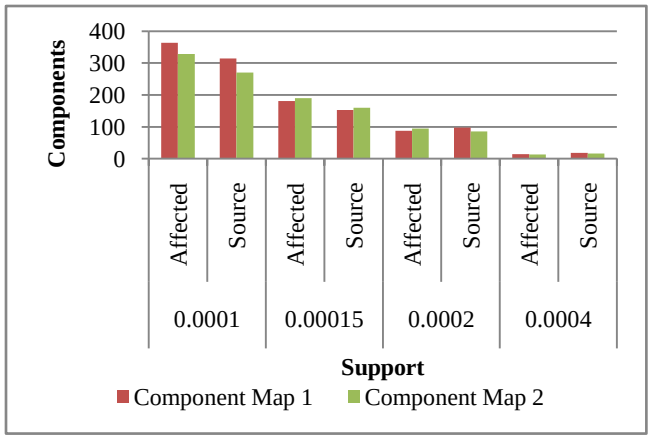


Figure 3: Components included in dependencies found with confidence 0.005 and different support cutoffs

In figure 4 and figure 5, we can see the characteristics of eight more models with various confidence values and support 0.0001. From Figure 5, we see that the number of affected and source components included in the dependencies is at most 22% and 21% of the input respectively. Even here, corresponding models for Component Map 1 and Component Map 2 had similar outcomes.

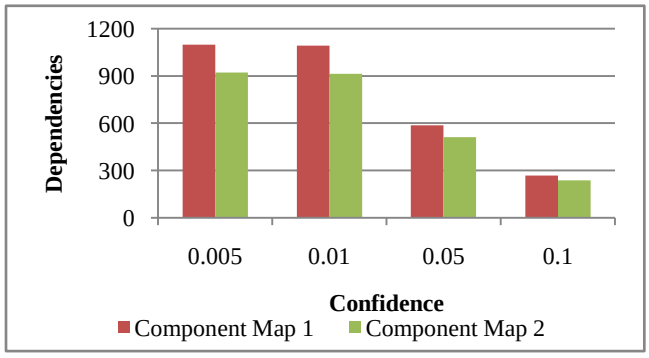


Figure 4: Dependencies found with support 0.0001 and different confidence cutoffs

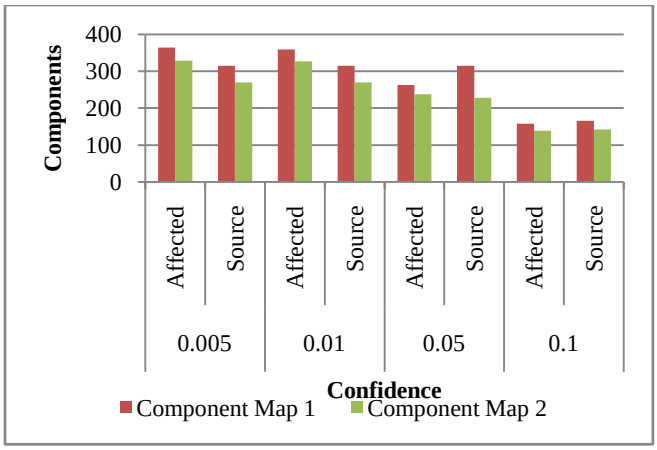


Figure 5: Components included in dependencies found with support 0.0001 and different confidence cutoffs

4.2.1 EFFECTIVENESS

In order to determine Ladybug's effectiveness, we compared the dependencies found from the different models above with dependencies determined by MaX for the same systems. We wanted to see how many dependencies were in common and whether our approach finds new dependencies, at least for this application.

Approach

Dependencies found by Ladybug are in terms of components. However, dependencies found by MaX are in terms of functions - (Binary1, Function1) *depends on* (Binary2, Function1). We substituted the (Binary, Function) pairs with the component the binary belongs to so that the MaX dependencies were identified in terms of components. Then, for each dependency found by Ladybug, if we could locate a path from the affected component to the source component in the derived graph. We considered it to be found by MaX as well, otherwise not.

Results

As we can see in Figures 6 and 7, the number of dependencies in common with MaX for the eight models in Figure 2 is around 20-30%. Similarly, for the eight models in Figure 4, 20-30% dependencies are found in common with MaX as shown in Figures 8 and 9.

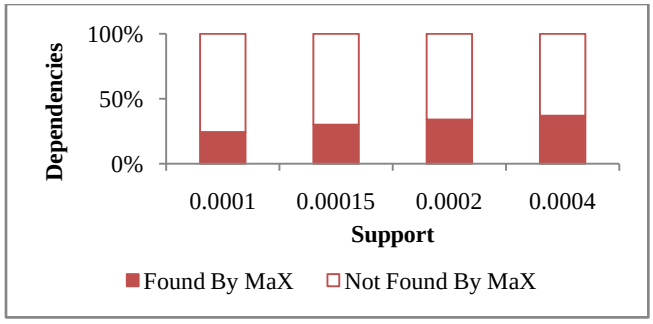


Figure 6: Comparison with MaX for dependencies found for models using Component Map 1, confidence 0.005 and different support cutoffs

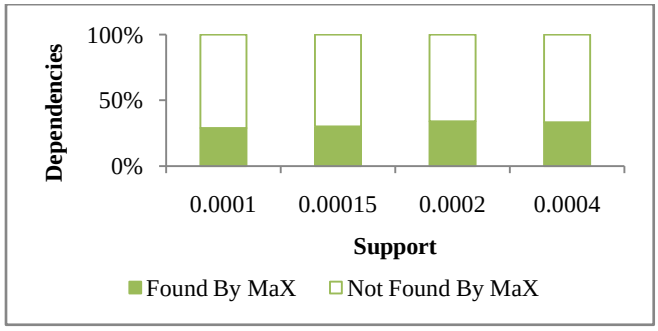


Figure 7: Comparison with MaX for dependencies found for models using Component Map 2, confidence 0.005 and different support cutoffs

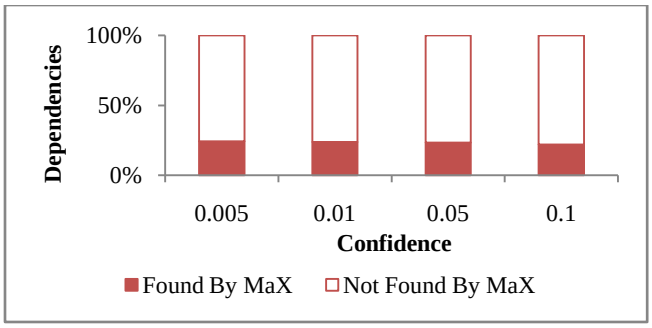


Figure 8: Comparison with MaX for dependencies found for models using Component Map 1, support 0.0001 and different confidence cutoffs

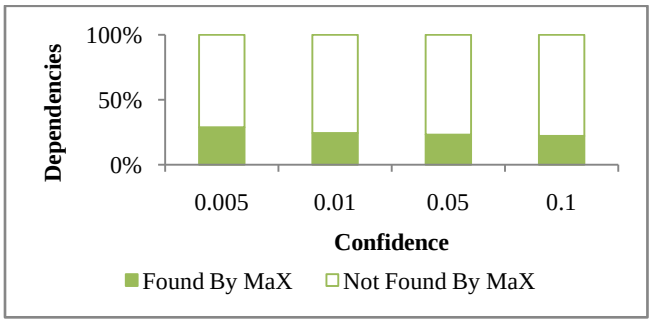


Figure 9: Comparison with MaX for dependencies found for models using Component Map 2, support 0.0001 and different confidence cutoffs

This outcome shows that our approach can possibly find new dependencies.

4.2.2 VALIDATION

In order to validate our findings and to find out whether the new dependencies we detected were accurate, we engaged with 127 domain experts (70 participated) who have worked on designing, implementing or maintaining the components involved and were the then component owners. We created an online survey for the top 30% dependencies (276 in all) found by mining Component Map 2 with support 0.0001 and confidence 0.005 and asked the experts to confirm or reject the dependencies using their knowledge of system architecture and interactions between the components included in a rule. Along with a dependency, its strength (support, confidence and importance values) was revealed without explaining the significance of these metrics.

Total Dependencies	276
Votes Cast	211
Dependencies with Votes	182
Dependencies with Multiple Votes	29 (20 all agree + 3 all disagree + 6 conflict)
Experts Invited	127
Experts Participated	70

Table 5: Summary of survey to measure effectiveness of Ladybug results for the top 30% dependencies found by mining Component Map 2 with support 0.0001 and confidence 0.005

Table 5 shows a summary of the survey. Of the dependencies with votes, 29 received multiple votes as the components involved had multiple owners. To summarize the votes and have the *effective vote tally* we treated a dependency with multiple votes as follows: if all votes were positive the dependency was considered as accepted otherwise it was considered rejected.

As we can see in Figure 10, out of all 182 dependencies that received votes 78% were accepted. If we consider the 18 dependencies with highest confidence (0.4 or greater), acceptance ratio was 83%. Again, for the 27 dependencies with top 10% importance, the acceptance was 90%. *In general, we conclude that owners seemed to confirm the dependencies considerably more often than reject them.*

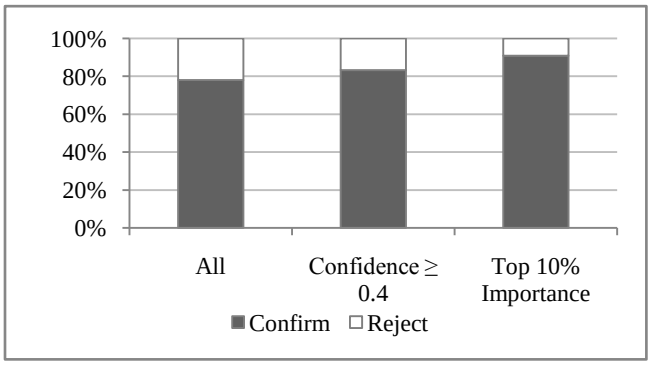


Figure 10: Effective vote tally for different buckets of dependencies

The contingency table in Table 6 shows for dependencies with votes, whether they were discovered by MaX. We can see that component owners confirmed as accurate 115 out of 143 dependencies found by Ladybug that were not detected by MaX while rejecting the other 28. Note that participants of our survey might have used MaX previously on these systems. Further, whether a dependency was found by MaX too was not revealed to them.

	Found by MaX	**Not Found by Max**	**Row Total**
Confirmed	27	115	142
Rejected	12	28	40
Total	39	143	182

Table 6: Contingency table showing votes versus detection by MaX

We performed chi-square test on Table 6 for hypothesis H_0,

H_0: Experts voted based on the rule and background system knowledge independently of the MaX data, which they could have seen before

For a probability value of 0.5 with one degree of freedom, the chi-square cutoff value is 3.84 while the chi-square value for this table is 0.134731. Hence, we do not reject H_0 at 95% confidence level.

Therefore, we believe that *by using our approach it is possible to discover additional important dependencies* that may not be found by, for example, static analysis of the system.

4.2.3 APPLICABILITY

In a sub-experiment, we created ten smaller maps from Component Map 2 through *percentage random sampling* using 10% to 100% of the entries. For each of these ten derived maps, we built three mining models with confidence 0.1, 0.4, 0.7. A support count of 25 was used (the support percentage value varied based on the size of the input dataset). In figure 11, we can see the number of dependencies found for the 30 models.

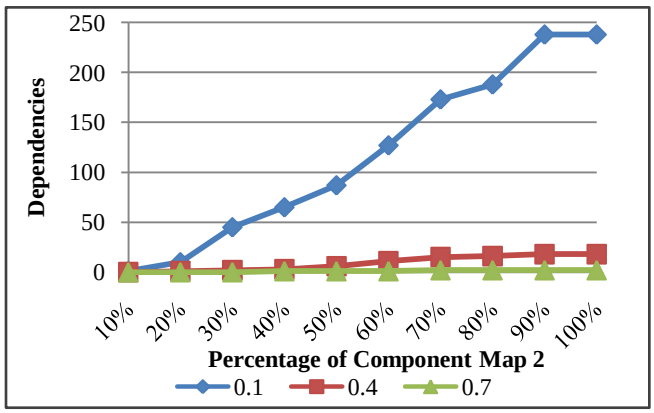


Figure 11: Dependencies found as a function of number of defect reports, for different confidence values and support count 25, indicates more defect reports yielded more dependencies

The graph shows that more defect reports yielded more dependencies. Moreover, the graph suggests a linear relationship between the number of component map items and the number of dependencies found. Therefore, we can start *mining for identifying dependencies at early phase of software development and keep refining the model over time as more data becomes available*.

4.3 THREATS TO VALIDITY

It is not possible to discover all dependencies for a system through defect history mining due to the scope of the data. Components that have not been part of significant number defects in the past will not benefit from this analysis. Therefore, we have to use the method in conjunction with other dependency analysis methods (static and run-time) to derive a comprehensive view of dependencies.

Further, we have applied our method on a well-componentized, large-scale software system with a stable development process. The number of defect reports available to us was considerable. It is possible the same method would not work on software systems with smaller scale of development or with less componentization in the architecture.

In our experiments, we have used for some models lower values for support and confidence that helped find a larger number of dependencies. For practical applications, it may be useful to have higher thresholds to find the most significant dependencies only.

5 CONCLUSION AND FUTURE WORK

For large software systems, a reliable method of discovering and ranking dependencies is important for understanding and prioritizing the testing required. We have used defect history to discover and rank dependencies in a software system. From our experiments, it appears as a novel way of discovering new information about dependencies in software.

We have shown that we can start the defect mining at any phase of software development and refine the model gradually as more defects are found.

We introduced a tool Ladybug that automates the mining process for Windows Vista and Windows Server 2008. We have shown that Ladybug discovers a number of dependencies that are not found by MaX. We intend to adapt Ladybug for other systems and study the dependencies in them. We also intend to see if we can find patterns in dependencies that were not found by MaX and try to trace then to a systemic deficiency in MaX.

In our study, we engaged with the component owners to validate our results against their expert knowledge and they generally confirmed dependencies provided by Ladybug as real. In the future, we will look at other ways of incorporating expert knowledge for creating better dependency recommendations.

Ladybug analysis has been incorporated in a larger change analysis and test targeting system used in Windows Serviceability and recommendations are used by hundreds of engineers every month.

6 ACKNOWLEDGEMENTS

We would like to thank Vivek Mandava and Alex Tarvo for their inputs and Pratima Kolan for doing an initial investigation. We also thank everyone who participated in the validation survey.

7 REFERENCES

[1] R. Agrawal, T. Imielinski, A. Swami, “Mining Association Rules between Sets of Items in Large Databases,” SIGMOD Conference 1993: pp. 207-216, 1993

[2] R. Agrawal and R. Srikant, “Fast Algorithms for Mining Association Rules,” Proceedings of the 20th VLDB Conference 1994, pp. 487-49, 1994

[3] P. Reps Anderson, T. Teitelbaum, M. T. Zarins, “Tool Support for Fine-Grained Software Inspection”, IEEE Software, 2003, VOL 20; PART 4, pages 42-56, 2003

[4] J. Anvik, G.C. Murphy, “Determining Implementation Expertise from Defect Reports,” Mining Software Repositories, 2007ICSE Workshops MSR '07. Fourth International Workshop on, vol., no., pp.2-2, 20-26 May 2007

[5] Bugzilla, Bug Tracking System for Mozilla, https://bugzilla.mozilla.org

[6] T. Eisenbarth, R. Koschke, D. Simon, “Aiding program comprehension by static and dynamic feature analysis,” Proceedings of IEEE International Conference on Software Maintenance, pp 602-611, 2001

[7] M. Fischer, M. Pinzger, and H. Gall, “Analyzing and Relating Defect Report Data for Feature Tracking,” Proceeding Working Conf. ReverseEng., pp. 90-99, 2003

[8] G. Hunt, M. Aiken, M. Fähndrich, C. Hawblitzel, O. Hodson, J. Larus, S. Levi, B. Steensgaard, D. Tarditi, T. Wobber, Sealing OS processes to improve dependability and safety, Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, March 21-23, 2007, Lisbon, Portugal

[9] Microsoft Association Algorithm Technical Reference (Analysis Services - Data mining), online resource at http://msdn.microsoft.com/en-us/library/cc280428(SQL.100).aspx

[10] N. Nagappan, T. Ball, A. Zeller, “Mining metrics to predict component failures”, Proceeding of the 28th international conference on Software engineering, May 20-28, 2006, Shanghai, China

[11] N. Nagappan, Ball, T., “Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study,” Proceedings of International Symposium on Empirical Software Engineering, pp. 364-373, 2007

[12] NDepend tool, http://ndepend.com

[13] A. Podgurski, L. A. Clarke, “A formal model of program dependences and its implications for software testing, debugging, and maintenance”, IEEE Transactions on Software Engineering, Volume 16, Issue 9, Sept. 1990 Page(s):965 - 979

[14] C. Silverstein, S. Brin, R. Motwani, “Beyond Market Baskets: Generalizing Association Rules to Dependence Rules”, Data Mining and Knowledge Discovery, v.2 n.1, p.39-68, January 1998

[15] Q. Song, M. Shepperd, M. Cartwright, C. Mair, "Software Defect Association Mining and Defect Correction Effort Prediction," IEEE Transactions on Software Engineering, vol. 32, no. 2, pp. 69-82, Feb., 2006

[16] A. Srivastava, J. Thiagarajan, “Effectively prioritizing tests in development environment,” Proceedings of the 2002 ACM SIGSOFT international Symposium on Software Testing and Analysis, pp. 97-106, 2002

[17] A. Srivastava, J. Thiagarajan, and C. Schertz, “Efficient Integration Testing using Dependency Analysis,” Microsoft Research-Technical Report, MSR-TR-2005-94, 2005

[18] N. Wilde, R. Huitt, “A reusable toolset for software dependency analysis”, Journal of Systems and Software, v.14 n.2, p.97-102, Feb. 1991

[19] T. Zimmermann, P. Weißgerber, S. Diehl, A. Zeller, “Mining Version Histories to Guide Software Changes,” Saarland University, September 2003. Proc. 26th International Conference on Software Engineering (ICSE), Edinburgh, UK, May 2004

[20] T. Zimmermann, N. Nagappan, “Predicting Component Failures using Dependency Graph Complexities”, Proceedings of International Symposium on Software Reliability Engineering, pp. 227-236, 2007