Using Decision Trees to Predict the Certification Result of a Build

Ahmed E. Hassan
Dept. of Electrical and Computer Engineering
University of Victoria
Victoria, Canada
ahmed@ece.uvic.ca

Ken Zhang
Toronto Software Lab
IBM Canada
Toronto, Canada
kenzhang@ca.ibm.com

ABSTRACT
Large teams of practitioners (developers, testers, etc.) usually work in parallel on the same code base. A major concern when working in parallel is the introduction of integration bugs in the latest shared code. These latent bugs are likely to slow down the project unless they are discovered as soon as possible. Many companies have adopted daily or weekly processes which build the latest source code and certify it by executing simple manual smoke/sanity tests or extensive automated integration test suites. Other members of a team can then use the certified build to develop new features or to perform additional analysis, such as performance or usability testing.

For large projects the certification process may take a few days. This long certification process forces team members to either use outdated or uncertified (possibly buggy) versions of the code. In this paper, we create decision trees to predict ahead of time the certification result of a build. By accurately predicting the outcome of the certification process, members of large software teams can work more effectively in parallel. Members can start using the latest code without waiting for the certification process to be completed. To perform our study, we mine historical information (code changes and certification results) for a large software project which is being developed at the IBM Toronto Labs. Our study shows that using a combination of project attributes (such as the number of modified subsystems in a build and certification results of previous builds), we can correctly predict 69% of the time that a build will fail certification. We can as well correctly predict 95% of the time if a build will pass certification.

1 INTRODUCTION

To ensure that products are delivered on time in a fast paced competitive environments, large teams of practitioners must work nowadays in parallel on the same code base [5]. Many large software projects use version control systems like CVS [7], Perforce [22], and ClearCase [2], to ease and coordinate parallel development activities. These systems permit each developer to have his or her local snapshot of the code base. Developers work independently on their local snapshot: adding features and fixing bugs. Version control systems are then used to coordinate changes between these developers, and to merge and integrate changes in different local snapshots back to the main branch of the code base.

Such integration work is rather cumbersome and time consuming. The most obvious challenge is merging the source code text between a developer's local snapshot and the latest code which resides in the version control's main branch. A rich area of research focuses on source code merging and conflict resolution techniques. These techniques merge source code changes from different developers to create a new copy of the program text which contains a conflict free version of the latest code changes. The most notable tools used for this work are the Unix `diff` tool and various commercial code merging tool.

Another challenging matter which faces practitioners is the introduction of undiscovered integration bugs in the latest source code. These bugs are usually due to unexpected interactions between prior code integrations by different developers. Whereas each developer may independently test his or her code changes, the code in the main branch, which contains the combined work of several developers, is not tested and may contain bugs. These undiscovered bugs slow down the progress of a project since team members would need to redo their work or may waste time investigating unrelated bugs. For example, performance engineers, who run longevity tests on the latest code, may discover the existence of a bug and would need to rerun their tests. Similarly, developers, who refresh their local snapshot, could waste a few hours tracking a bug to later discover that the bug is not due to their local changes interacting with the refreshed main branch instead the bug already resided in the main branch due to prior integrations.

To speed up the process of locating integration bugs, many organizations have instilled build certification processes. These processes are performed by integration testing teams which certify the quality of the latest code by building the code and executing a suite of tests. These tests could be simple smoke/sanity or complex test suites that are run continuously after each code integration into the version control [8, 26].

For medium size project, this certification process may require around eight hours of testing (a full workday). Larger projects may require considerably more time [4]. While the

21st IEEE International Conference on Automated Software Engineering (ASE'06)
0-7695-2579-2/06 $20.00 © 2006 IEEE

build is being certified, the status of the latest shared code for a project is undefined. The latest code may contain bugs that the certification process would uncover within the following hours or days. This long certification process reduces the ability of development team members to work in parallel as they need to wait till a certified build is ready. Alternatively, team members could either use outdated or uncertified (possibly buggy) versions of the code. Both options are not optimal since the first option may not be feasible since the outdated copy may not have needed features. The second option is time consuming since the team members have to redo their work if a build later fails certification.

Under tight schedules (in particular for emergency and security fixes builds), members of large software projects (e.g. developers, project managers, and testers) need to use in parallel the most up to date copy of the code, which contains the latest fixes and features, in order to release the product as soon as possible. For example to speed up the release cycle, in particular for hot fixes, a build is sent to the software quality assurance teams before the build certification process is completed. Moreover if project managers can determine ahead of time if a build is fault-free or if recent changes integrated in a build are likely to fix a previously buggy build then they can speed up the development and release cycles by sending notifications to other teams so they would either start using a build before it is fully certified (i.e. tested), or they would at least start planning and allocating resources to pickup the build as soon as it is certified.

In this paper, we use decision tree techniques to derive a set of rules which practitioners could use to predict the likelihood of a build failing or passing the certification process. To derive these rules we mine historical information (code changes, build certification results) for a large software project under development at the IBM Toronto Labs. Developers could use these rules to decide if it is worth refreshing their local code snapshot or if it is more prudent for them to wait for a better (safer) time. By better time we mean a time where the chances of the refreshed code being buggy are low. Similarly, other team members, like quality assurance and performance engineers, could use the same rules to determine if they should pick the latest code build for further testing or if it is more prudent for them to allocate their resources elsewhere till that build passes the certification process.

Organization of the Paper

The organization of the paper is as follows. Section 2 discusses various attributes and factors that are used nowadays by practitioners to guess the outcome of the certification process. Section 3 presents decision trees and motivates their use to predict the certification result of a build instead of depending on ad hoc factors. Section 4 presents the software system analyzed in our case study and details the data used in our analysis. We generate various decision trees using this historical data and we discuss the accuracy of the generated decision trees in predicting the certification result of a build. Section 5 presents related work. Section 6 concludes the paper and outlines some future work.

2 FACTORS USED TO PREDICT THE CERTIFICATION RESULT OF A BUILD

The certification process of a build usually consists of a variety of tests such as simple manual smoke/sanity tests or more elaborate suites of automated integration tests. The failure of a test is likely due to the introduction of a bug. For some software projects, a formal certification process may not exist, instead a build is considered problem free if no one complained about it for the last few days. In either cases (formal or informal certification), practitioners must spending some time waiting in order to determine the status of a build.

To speed up development and avoid wasting time using a buggy build, practitioners tend to employ a variety of ad hoc methods to decide if the latest code is likely to fail testing and certification. For example, many developers avoid refreshing their local snapshot on Friday afternoon and prefer to refresh their snapshot mid week to ensure that any new integration problems are uncovered over the weekend and are solved at the beginning of the work week. Other developers may avoid refreshing from the main branch once a junior developer performs a large integration or if an error prone subsystem is modified.

In this section we present several factors which practitioners may use to predict the certification result of a build. We as well explore the intuition behind using various attributes related to these factors. We present below a list of possible factors:

1. Social Factors (such as work habits and team organization).
2. Technical Factors (such as software structure and complexity).
3. Coordination Factors (such as parallel changes to files or subsystems by different developers).
4. Prior-Certifications Factors (such as the results of the last certification, or the number of days since the last failed certification).

In our case study, presented later in the paper, we will empirically examine each of these factors separately to determine the most promising ones. We later combine these factors to derive an optimal method to assist practitioners in accurately predicting the certification results of the latest code build.

Social Factors

Conway's hypothesis [3], formulated by Brooks [9] as Conway's law, states that the structure of a software system will match the organization of the group that designed the system. Bowman and Holt have demonstrated the effect of team structure and collaboration on the structure of the software

Attribute Name	Explanation and Rationale
Time	Time of Day (0-24). Do morning vs. afternoon code changes increase the likelihood of introducing an integration bug?
WeekDay	Day of Week (Mon, Tue, Wed, Thu, Fri, Sat, Sun). Are changes done on Mondays more likely to introduce bugs than changes done during other weekdays or over the weekend?
Month Day	Day in Month (1-31). Are changes earlier in the month more likely to introduce bugs than changes towards the end of the month?

Table 1: Work Habits Attributes.

system for large software systems [1]. These findings encourage us to investigate the effects of work habits and team structure on the introduction of integration bugs. For example, in many projects, developers usually avoid refreshing their local snapshot at the end of the week. The rationale being that there is a higher chance that integration bugs may have been introduced towards the end of the week and are not yet resolved. Table 1 lists a few work habits attributes which a practitioner may consider.

A recent survey of software practitioners has shown that senior developers are likely to review code changes produced by specific team members [12]. These team members tend to introduce bugs into the code more often in comparison to other team members. Hence if a particular developer integrates his or her changes, other team members may prefer to wait till the latest code in the main branch is certified before they refresh their local snapshot. Khoshgoftaar $et\ al.$ [18] have shown that the experience of a developer contributes to his or her potential to introduce a bug. After the initial mentoring period, new developer are more likely to introduce bugs than other developers as their changes are no longer closely reviewed by senior mentors. Table 2 lists a few team attributes which practitioners can consider when attempting to predict the certification result of a build.

Technical Factors

Practitioners also consider technical factors when deciding whether to use the latest uncertified code or an outdated version of the code. Practitioners may examine recent code changes and base their decision on the location of changes relative the structure of the software system. For example, some subsystems in a software system may be more error prone than others and changes to these subsystems are likely to introduce integration bugs. For example, changes to common subsystems, such as libraries, are likely to introduce bugs since they are likely more complex as they are used by several subsystems. Changes that are spread out across a large number of files or subsystems may introduce integration bugs as well [11]. Table 3 lists several technical attributes which practitioners are likely to consider. In addition to these factors, the complexity of the changed code (e.g. Mccabe complexity is a possible attribute [20]), the intuition being that changes to complex code are trickier and more likely to introduce integration bugs.

Attribute Name	Explanation and Rationale
Developer Count	The number of developers who integrated changes to the main branch. The more developers integrating changes, the more likely an integration bug will be introduced.
Developer Name	The name of each developer who integrated changes to the main branch. Are particular developers more prone to introducing integration bugs?
Experience	Experience of the developers who submitted code changes to the main branch. Experience is measured by counting the number of prior code changes submitted to the version control system by a developer. The following are a few metrics that capture developers' experience: 1. **Max Exp**. measures the experience of the most senior developer out of all the developers who did recent changes since the last certified build. 2. **Min Exp**. measures the experience of the most junior developer who performed changes since the last certified build. 3. **Weighted Exp**. measures the weighted experience of developers since the last certified build. It is calculated by multiplying for each changed file the experience of the developer who changed the file then dividing the sum with the number of changed files.

Table 2: Team Structure Attributes.

Coordination Factors

Intuitively, a critical yet simple factor in deciding whether to use the latest build is if the latest build has files which were modified several times by different developers during a short time span. Developers are likely to miss some integration issues due to rapid consecutive changes to these files. We refer to these files as *collision files*. Table 4 lists several coordination attributes which practitioners may examine when deciding if they should use the latest build or await the result of its certification.

Prior-Certifications Factors

Certification factors are considered as well by many practitioners. For example, developers may check the status of the previous certification. If the last certification failed, in turn indicating that the main branch is buggy, then developers would avoid refreshing their local snapshot. Similarly, software assurance engineers may avoid immediately pick-

Attribute Name	Explanation and Rationale
Number of Files	The number of modified files. The intuition being that changes to a large number of files in a build increases the likelihood of introducing an integration bug, since developers may have missed considering some interactions between the changed files.
Number of Subsystems	The number of modified subsystems since the last certified build. Instead of simply considering the number of files, considering the number of subsystems takes into account the spread of changes throughout the architecture of the software system. Parallel changes spread across many subsystems are more likely to result in integration bugs, in contrast to changes localized to a single subsystem.
Subsystem Change Entropy	Instead of simply tracking the number of modified subsystems, we could track the distribution of changes across all subsystems. Research has shown that measuring the entropy of change is a good predictor of future bugs [11], therefore we expect that it may be a good predictor of integration bugs.
Subsystem Name	The name of each changed subsystem, since some subsystems may be more error prone than others.
File Change History	The count of prior changes to modified files. We can define metrics such as: smallest change history, oldest change history, weighted change history. Developers consider the change history since they fear that integration bugs could be introduced in rapidly changing files in comparison to stable files.

Table 3: Technical Attributes.

ing up a build which follows a build that failed certification. Developers may as well consider the time since the last failed certification. The longer it has been since the last failure, the more likely the certification will fail soon.

In the case study, presented later in the paper, we examine these various factors and consider the best combination of factors to assist practitioners in predicting the certification result of a build. If practitioners can correctly predict certification results then they can work more effectively in parallel since they can start using and examining builds as soon as possible instead of waiting for certification results.

3 DECISION TREES

A build may belong to one of two classes: Fail or Pass certification. To determine ahead of time the certification result of a build, we use historical project information (prior code changes and certification results) to define several attributes for each build. We then use machine learning techniques,

Attribute Name	Explanation and Rationale
Number of Collision Files	The number of files modified multiple times in parallel since the last certified build.
Number of Collision Subsystems	The number of subsystems which contain files that have been modified multiple times in parallel.
Name of Collision Subsystems	The names of subsystems which contain files that have been modified several times in parallel. By considering the names of the subsystems, we may uncover particular subsystems collisions that introduce integration bugs.
Number of Collision Developers	The number of developers who modified files in parallel.
Name of Collision Developers	The names of developers who modified files in parallel.

Table 4: Coordination Attributes.

like *Decision Trees*, to learn different rules based on our attributes to predict the certification result (i.e. class) of a build before its certification is completed.

To perform our analysis, we divide historical project information into two sets: a training and a testing data set. For each build in either of the sets we record all new code changes integrated into that build and define several attributes, based on the factors defined in Section 2. The number of changed files and the number of developers who performed changes are two example attributes. We as well track the certification result for each build: Fail or Pass.

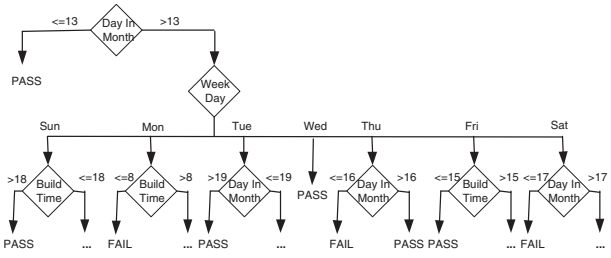

Figure 1: A Sample Decision Tree Using Social Factors (Only the top 3 levels are shown). Each diamond node represents a decision and each edge represents a possible result of the decision. Each leaf classifies the build into one of two classes: Pass or Fail. The root of the tree (level 0) is the node at the top.

We used the C4.5 algorithm by Quinlan [13] to build our decision trees. The algorithm creates a tree that captures relations in the training data. For example, the algorithm may determine that if a build under certification has changes by more than 10 developers who changed more than 20 files, then the build will likely fail the certification and practition-

ers should avoid using this build.

The C4.5 algorithm starts with an empty tree and adds decision nodes or leaf nodes as it grows. The algorithm determines the information gain for each attribute and picks the attribute with the highest information gain. It then performs additional analysis to determine the cut-point for the values of the picked attribute based on the training data set. For example, the algorithm could determine to split the attribute "number of developers" into larger than 5 developers and smaller than 5 developers. To avoid overfitting the data, the algorithm prunes the tree at the end to remove branches for which there does not exist enough support in the training data. A more detailed explanation of the algorithm is presented in [13]. An example of a produced decision tree from our case study is shown in Figure 1. The decision tree indicates that builds in the first half of a month and builds on Wednesday are likely to pass certification. Whereas builds on Monday before 8 AM in the second half of a month are likely to fail certification.

On the Use of Decision Trees

We decided to use decision trees in our analysis since they offer an explainable model. This model could be studied and verified by domain experts. Newer relations may be uncovered when studying the tree. On the other hand other classification methods are usually treated as black box models where the reason for the classifications (a case being assigned to a particular class) is not explained. The tree shown in Figure 1 supports developer's intuition of avoiding to refresh their local code snapshots late in the week instead opting to refresh their local snapshot at the middle of the week to ensure that any integration problems are fixed at the beginning of the week.

We believe that the fact that the classifications are explainable is very valuable to team members who must justify their decisions. Upper management is likely to require justification from practitioners who recommend avoiding or encourage using the latest code/build. Simply stating that their recommendation is based on a a black box classifier without offering a good rationale for such a recommendation is not sufficient and managers would likely ignore the recommendation. Furthermore, the explainable model permits practitioners to examine the tree in order to determine if they should override its classifications/recommendations. For example, if a tree were to indicate that a build is likely to fail certification because it contains a large code integration, a developer may decide to ignore this classification if she or he knows that the big change was simply to update the copyright notices at the top of each file and did not involve any actual code changes.

Evaluating a Decision Tree

To evaluate the predictive power of a derived decision tree, we use the tree to classify the builds in the testing data set and we compare the classified certifications against the actual certification results. Table 5 shows the confusion matrix for classifications done using a decision tree. The evaluation of the decision tree's accuracy is calculated in terms of the percentage of misclassified certifications in the testing data. We desire a classifier with low misclassification rates. We measure three rates:

	Classified As	
True Class	Fail	Pass
Fail	a	b
Pass	c	d

Table 5: Confusion matrix – The entries in the matrix are the classification made by a tree classifier versus the actual class in a two-class (Fail and Pass) setting.

1. **Overall misclassification rate:** This captures the overall performance of the decision tree for both classes (Fail and Pass). It is defined as: $(b+c)/(a+b+c+d)$.
2. **Fail misclassification rate:** This captures the performance of the decision tree for failed certifications. It is defined as: $b/(a+b)$.
3. **Pass misclassifications rate:** This captures the performance of the decision tree for passed certifications. It is defined as: $c/(c+d)$.

If the Pass misclassification rate is high, then the derived decision tree would slow down practitioners who would incorrectly avoid using the latest code for fear it is buggy. If the Fail misclassification rate is high then practitioners are likely to waste time since they will need to redo their work once the buggy build is fixed and it passes certification.

4 CASE STUDY

We conducted a case study to empirically examine the affect of the different factors in building a decision tree to predict the certification result of a build. We examined a large software system developed primarily at the IBM Toronto Labs. The system is sold to large and medium size organizations to automate manual tasks for provisioning and configuring of physical and virtual servers, operating systems, middleware, applications, storage and network devices (such as routers, switches, firewalls, and load balancers). The system allows users to create, customize, and quickly utilize best-practice automated provisioning packages.

The system is developed in Java. It has a strict 4 layered architecture which contains 13 subsystems. A set of automated tests exist for each subsystem. There are approximately 2,500 tests in total for the whole software system. Developers are expected to run the automated tests for the subsystems they modified before they integrate their changes back into the main code branch.

More than 200 practitioners (such as testers, developers,

and project managers), which reside in different time zones worldwide, are involved with this project. Around 40 developers work in parallel on the code for the software system. Research has suggested that frequently integrating code to the main branch is desirable since it eases the final integration work prior to a release [4, 16]. To ensure that these practitioners can work in parallel, that they can easily integrate their changes, and that no integration bugs are introduced, continuous integration and testing processes were put in place [8, 26]. These processes are run several times a day. They are run twenty minutes after every code integration into the main branch, unless the process is already running. These two processes represent the build certification process for this project. Once the build certification is completed, a certification report is published on a website. The report contains information about changes to the code since the last certified build and the results of tests. If the certification were to fail, then a certification failure email is sent out to all developers who integrated changes into the main branch since the last build which passed certification.

This automated certification process (continuous integration and testing processes) has been in place since April 2005. In our case study, we are interested in examining the ability of various factors in correctly predicting the certification result of a build. We built decision trees using attributes for the different factors and measured the misclassification rates for the generated trees.

For our analysis, we use the data from June 2005 to February 2006. We do not use the data for the first two months (April and May) in building or evaluating the decision trees, instead the data for these two months is used to calculate some of the historical attributes such as the file change history (*see* Table 3). For the studied period (June to February), we have 1,429 certification builds: 1,220 of these builds passed the certification and 209 failed.

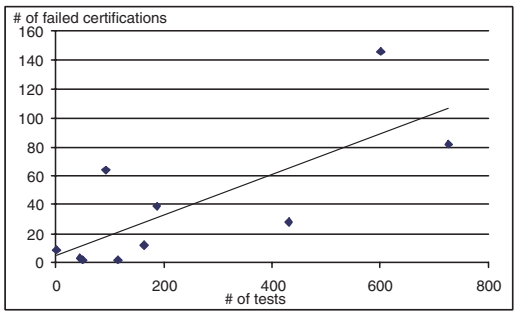

Figure 2: Number of Tests associated with a Subsystem versus the Number of Failed Certification Builds Due to a Failed Test in the Corresponding Subsystem.

Most subsystems in the software system contain a large number of tests. Each subsystem has on average over 240 tests with the largest subsystem having over 700 tests. Figure 2 correlates for the 13 subsystems the number of tests associated with a subsystem against the number of failed certification builds due to the failure of a test associated with that subsystem.

Study Procedure

In our case study, we conducted several experiments. For each experiment, we examined separately each of the factors explored in Section 2. For our last experiment we examined the combination of all factors. For each experiment, we created decision trees using the C4.5 algorithm to predict the certification result of a build using attributes derived from one or several factors. For example, in our first experiment we used the work habits attributes (in Table 1) to create decision trees.

To evaluate the accuracy of the created decision tree, we used a ten fold cross validation procedure [6]. The ten fold cross validation procedure randomly breaks down the data into ten mutually exclusive sets of approximately the same size. We combine nine of the ten sets into a large training set. The training set is used to create a decision tree. One of the ten sets is kept aside to test the built decision tree against it. This process is repeated ten times. Each time a different set is used for testing the decision tree. The misclassification rates for the ten created decision trees is averaged to derive the mean performance of the decision trees for a studied factor.

To compare the performance of trees generated for the studied factors, we measure the statistical significance of the difference. Our statistical analysis assumes a 5% level of significance (i.e. $\alpha = 0.05$). We formulate the following test hypotheses:

$$\mathcal{H}_0 : \mu(Perf_A - Perf_B) = 0$$

$$\mathcal{H}_\mathcal{A} : \mu(Perf_A - Perf_B) \neq 0$$

$\mu(Perf_A - Perf_B)$ is the population mean of the difference between the performance of each of the ten cross validation trees generated for a factor in an experiment. If the null hypothesis H_0 holds (i.e. the derived p-value $> \alpha = 0.05$), then the difference in mean is not significant. If H_0 is rejected with a high probability (i.e. the derived p-value $\leq \alpha = 0.05$), then we can be confident about the performance improvements of a particular factor. We test the significance of the difference in the mean using a non-parametric paired test. We used a paired Wilcoxon signed rank test which is resilient to strong departures from the assumptions of the t-test [25].

Experiment #1: Social Factors – Work Habits

Figure 1 shows one of the generated decision trees using the work habits attributes. Analyzing one of the generated tree for the work habits attributes, we can derive, for example, the following rules:

1. IF (Day == Mon) AND (Time <= 17) AND (Month Day > 16) AND (Month Day <= 18) THEN the build will

fail certification. This rule is correct 89.1% of the time in the testing data.

2. IF (Month Day $<=$ 13) THEN the build will pass certification. This rule is correct 90.7% of the time in the testing data.

The tree in Figure 1 shows that builds usually earlier in the month tend to pass certification whereas the certification results of builds in the later half of the month depend on several additional attributes such as the day and time of the changes integrated into the build. Moreover, builds on Wednesday (middle of the work week) have a high chance of passing the certification. Practitioners, examining these results, can derive simple rules of thumb to follow, such as the fact that picking a Wednesday build is a good and safe bet.

The results of the ten fold validation procedure show that on average the Fail misclassification rate is 67%, the Pass misclassification rate is 3%, and the overall misclassification rate is 12%. These results indicate that if a decision tree were to predict that a build will fail certification then it is correct 33% of the time. These results are disappointing but given the limited information that is used (only the work habits attributes), they are promising. If a decision tree indicates that a build will Pass, then it is on average correct 97% of the time which is a very high rate and practically should be followed instead of simply following the intuition and gut feelings of senior practitioners (such as senior developers or project managers).

Experiment #2: Social Factors – Team Structure
Instead of considering only the time of changes in a build, we can as well consider attributes based on the structure of the team. For example, we can consider the names of the developers who integrated changes and their experience. We re-ran our analysis using the attributed defined in Table 2. Our cross validation results show a large statistically significant improvement in the Fail misclassification. The average Fail misclassification rate for a build is 42% (a 37% improvement), the Pass misclassification rate is 3%, and the overall misclassification rate is 9%.

Level	Frequency	Attribute
0	9	Developer Name
	1	Maximum Developer Experience
1	9	Maximum Developer Experience
	3	Minimum Developer Experience
	1	Developer Name
2	1	Weighted Developer Experience

Table 6: Top Nodes in Decision Trees Created Using Team Structure Attributes.

We examined the ten decision trees created by our cross validation procedure to determine which attributes are considered more important by the decision tree algorithm. The closer an attribute is to the root node (i.e. level 0), the more important that attribute is considered [27]. Table 6 summarizes the result of our Top Nodes analysis. The table shows that the names of developers who recently integrated code into a build are very influential in predicting the results of certification. These results correlate well with recent surveys of software practitioners, who believe that particular developers are more prone to introducing bugs than others [12]. The experience of the developers who integrated code, in particular the experience of the most senior developer, is an important attribute as well.

It is surprising that the number of developers who integrated changes in a build is not considered an important attribute. We would have expected that more developers integrating changes in a build may result in less communication between them and would increase the chances of introducing a bug. These results match earlier results by Graves $et\ al.$ [10] who examined the development history of a large telephony system and noted that there exists no evidence to support the "too many cooks" effect: the number of developers who had changed a module (file) did not help in predicting the incidence of faults in that module.

We re-ran our tree creation algorithm and added the work habits attributes, from our first experiment, along with the team structure attributes but the performance of the generated trees was no better than the performance of the trees generated using only the team structure attributes.

Experiment #3: Technical Factors
In addition to considering social factors, we examined how technical factors would help predict the certification result of a build. We re-ran our experiment using the technical attributes outlined in Table 3. The average misclassification rate for a build is 39%, the Pass misclassification rate is 5%, and the overall misclassification rate is 10%. These results are statistically better than the results produced using the social factors (work habits and team attributes). On average a decision tree generated using technical factors will predict incorrectly 39% of the time that a build will fail certification.

We performed our Top Nodes analysis to determine which attributes are considered by the generated decision trees to be the most influential. Table 7 summarizes the result of our Top Nodes analysis. Our analysis reveals that the name of the subsystem is not an influential attribute in contrast to the name of the developer in our previous experiment. Furthermore, the number of changed subsystems is the most important attribute, in contrast the number of developers who performed changes was not considered an important attribute in our previous experiment.

Experiment #4: Coordination Factors
In our fourth experiment, we examined the effect of coordination factors in predicting the certification result of a build. The coordination attributes try to capture importance for in-

Level	Frequency	Attribute
0	10	Number of Subsystems
1	9	Number of Files
	6	Weighted File Change History
2	8	Maximum File Change History
	3	Number of Files
	1	Subsystem Change Entropy
3	9	Maximum File Change History
	7	Subsystem Name
	2	Subsystem Change Entropy
	1	Weighted File Change History

Table 7: Top Nodes in Decision Trees Created Using Technical Attributes.

formation hiding and encapsulation [21], and the need for developers to communicate effectively when working in parallel since they are modifying the same files or subsystems within a short time span. We had expected that a tree using the coordination attributes would perform well. We re-ran our experiment using the coordination attributes outlined in Table 4. The average Fail misclassification rate is 51%, the Pass misclassification rate is 3%, and the overall misclassification rate is 10%. These results are statistically better than the results produced using the work habits attributes but are unfortunately worse than the results of all our other experiments. Given we only analyzed one software system, we cannot rule out yet the importance of coordination factors in predicting the results of certification.

Experiment #5: Prior-Certifications Factors
In experiment 5, we considered certification factors. In particular, we built decision trees using the following attributes:

1. Days since the last failed certification.
2. Number of builds which passed certification prior to this build.
3. Certification result of the previous build.

The cross validation procedure shows that the average Fail misclassification rate is 39%, the Pass misclassification rate is 4%, and the overall misclassification rate is 9%. These results are similar to the results produced using the technical factors.

Experiment #6: All Factors
In our final experiment we combined the attributes from all studied factors in an effort to derive the best decision tree. The average Fail misclassification rate is 31%, the Pass misclassification rate is 5%, and the overall misclassification rate is 9%. These results are statistically better than the results produced using other factors separately. We performed our Top Nodes analysis to determine the attributes that are considered by the generated decision trees to be the most influential. Table 8 summarizes the result of our Top Nodes analysis. Our analysis reveals that the certification attributes are the most influential followed by team structure attributes.

Level	Frequency	Attribute
0	10	Status of the Previous Build
1	9	Developer Name
	2	Name of Collision Subsystem
	2	Subsystem Name
2	9	Name of Collision Subsystems
	2	Developer Name
	1	Weighted Developer Experience
	1	Time
3	4	Minimum File Change History
	2	Subsystem Name
	1	Weighted Developer Experience

Table 8: Top Nodes in Decision Trees Created Using All the Factors.

Discussion and Limitation Of Our Results
Table 9 summarizes the performance of the decision trees generated in the six experiments conducted in our case study. Our results reveal the following interesting points:

1. In contrast to other attributes which require developers to manually examine changes integrated into a build and previous certification results, work habits attributes are the simplest and easiest attributes to measure (only the date of the build is needed). Our results show that a simple rule such as picking mid-week builds is highly effective. Nevertheless, other factors produce better prediction in particular for builds that will fail certification.
2. Decision trees built using technical and prior certifications factors outperform all other factors. Prior-certification factors require an automated certification process to be in place. This may not be the case for many projects. Therefore a technical factor may be a better choice. It is interesting to note that if an automated certification process is not in place, then we would not be able to measure the prediction accuracy of trees built using technical factors. Nevertheless, managers could use the trees' classifications (predictions) as a warning flag. For a build that is classified to fail certification, managers may consider running a manual certification process or performing additional testing.

Empirical research studies should be evaluated to determine whether they were able to measure what they were designed to assess. In particular, we should examine if our findings that a particular combination of attributes is more effective than others are valid and applicable in general or if they are due to flaws in our experimental design. Four types of tests are used [28]: construct validity, internal validity, external validity, and reliability.

EXP#	Description	Fail	Pass	Overall
1	Social Factors – Work Habits	67	3	12
2	Social Factors – Team Structure	42	3	9
3	Technical Factors	39	5	10
4	Coordination Factors	55	3	10
5	Historical Factors	39	4	9
6	All Factors	31	5	9

Table 9: Misclassification Rates for the Conducted Experiments.

Construct Validity Construct validity is concerned to the meaningfulness of the measurements – Do the measurements quantify what we want them to? Practitioners do not want to waste time using buggy builds and working on buggy source code. It is clear that we cannot rule out if the latest source code is fault-free, instead we use a practical definition of buggy. A build is buggy if it fails one of the currently available tests. Once a build fails a test, practitioners (e.g. performance engineers) using it will likely have to redo their work. This rework will result in wasted time and effort. Our definition is highly dependant on the number of tests that exist for a software system. For example, a software system with no tests or certification process will never be considered buggy. Also a build that may have been considered not buggy may be considered buggy in the future if additional newly developed tests were executed on it. For our purposes, the fact that a build is considered buggy later in the future is not relevant since practitioners did not have a way to recognize this at the appropriate time and did not need to perform any rework. We draw an analogy to research in bug prediction which attempts to predict bugs using bug reporting databases. A system may have many bugs in it but if it is never used by customers or used under particular conditions then these bugs won't show up. This does not imply that the system is fault-free, it just implies that given the current environment these bugs are not visible and likely not a concern for a practitioner.

Internal Validity Internal validity deals with the concern that there may be other plausible rival hypotheses to explain our findings – Can we show that there is a cause and effect relation between changes in certain attributes and the certification result of a build? Software development in large teams is a complex process. In our experiments we pick a limited number of attributes which clearly are not sufficient to capture this complex process. Nevertheless, these attributes help us gain a better understanding of some of the main factors that affect the certification result of a build.

We also note that subsystems with a large number of tests have a higher chances to fail tests (as shown in Figure 2). This may indicate that builds fail certifications due to the number of tests and not due to studied attributes. Unfortunately, our currently collected data does not capture the date of the introduction of each test into the certification process. We only know the number of current tests. Therefore we do not have a good idea of when tests were introduced. To investigate this concern, we re-ran our last experiment with all the attributes and added an additional "test count" attribute. This test count attribute captures the number of available tests for all modified subsystems in a built. Our Top Nodes analysis showed that this attribute was not considered influential and it did not show up in the final pruned decision tree. Although we used the current count of tests for each subsystem, we believe that the ratio of tests between the different subsystems has been constant over time (i.e. subsystems with low number of tests have always had low number of tests relative to other subsystems). We have as well started capturing the information for newly added tests so we can later study this concern in more detail.

External Validity External validity tackles the issue of the generalization of the results of our study – Can we generalize our results to other software systems and projects? Although we looked at a large commercial software system developed by a large number of practitioners, we only looked at one system at one company. We need to consider other software systems. We hope to contact other teams at IBM which have deployed a similar build certification process. We plan to perform similar analysis on their historical project information and certification results and to study if our results can generalize across projects.

Reliability Reliability refers to the degree to which someone analyzing the data would reach the same conclusions or results. We believe that the reliability of our study is high. The data used in our study is derived from historical project information. Other projects which track such historical information can easily run the same experiments on their project to produce findings specific to their projects and to compare them to our findings.

Summary of Limitations Although our study has limitations, we believe that practitioners could use our findings to avoid wasting time working and deploying builds that will be considered buggy. Our results have currently been applied to a single project due to the limited access to such rich historical information (historical build certification results are not widely available for many projects). We hope in the future to study other projects to ensure the validity and generality of our findings.

5 RELATED WORK

We use classification trees to predict the certification result of a build. Most prior work has focused on predicting defects in software systems. Early work by Porter and Selby used decision trees to identify fault-prone modules based on attributes derived from software metrics [23]. Work by Khoshgoftaar *et al.* explored other types of classification trees such as regression trees to predict as well fault-prone modules [18, 17]. Similarly, work which uses classification

trees for software quality predictions has been conducted by Troster and Tian [15], and Takahashi $et\ al.$ [24]. Shirabad used decision trees to mine historical code changes to give recommendation for other files to propagate changes to when performing code changes [27]. Recently Knab $et\ al.$ used classification trees to investigate defect densities in open source projects [19].

6 CONCLUSION

Integration testing is an essential and vital part of modern code development. Through integration testing software development teams can certify the quality of builds and can detect bugs early before they affect other team members and slow down the progress of a project. Unfortunately, integration testing is a time consuming process which prevents development team from working effectively in parallel. For example, performance and longevity testing of a build cannot commence till integration testing is completed successfully and the build is certified to be of good quality. To speed up development cycles (in particular for emergency fixes and security updates), development teams would like to start using a build as soon as it ready even before it is certified.

In this paper, we explored building decision trees to help practitioners decide if it is prudent to start using uncertified builds or if they should await the certification to be completed. By using uncertified builds practitioners could perform a variety of tasks in parallel instead of waiting for the certification process to be completed. Our best decision tree is accurate on average 69% of a time in predicting that a build will fail certification. Using this tree practitioners can weigh the risks of using the latest uncertified build to speed up the release cycle, versus waiting for the certification process in order to avoid wasting resources. In future work, we will explore our findings using additional software projects. We as well would like to consider recent enhancement to the C4.5 algorithm (e.g. Boosting [14]).

Acknowledgments

We are grateful to the IBM Toronto Lab for providing us access to the historical project information used in this study. The findings and opinions in this paper belong solely to the authors, and are not necessarily those of the IBM and RIM. Moreover, our results do not in any way reflect the quality of the IBMs software products.

REFERENCES

[1] I. T. Bowman, R. C. Holt, and N. V. Brewster. Reconstructing Ownership Architectures To Help Understand Software Systems. In *Proceedings of the 7th International Workshop on Program Comprehension*, Pittsburgh, USA, May 1999.

[2] Rational ClearCase - Product Overview. Available online at `http://www-306.ibm.com/software/awdtools/clearcase/`.

[3] M. E. Conway. How do comittees invent? 14(4):28–31, 1968.

[4] M. A. Cusumano and R. W. Selby. *Microsoft Secrets*. The Free Press, 1995.

[5] Dewayne E. Perry and Harvey P. Siy and Lawrence G. Votta. Parallel Changes in Large Scale Software Development: An Observational Case Study. In *Proceedings of the 20th International Conference on Software Engineering*, pages 251–260, May 1998.

[6] B. Efron. Estimating the error rate of a prediction rule: improvement on cross-validation. *Journal of American Statistical Association*, 78(382):316–331, Dec. 1983.

[7] K. Fogel. *Open Source Development with CVS*. Coriolos Open Press, Scottsdale, AZ, 1999.

[8] M. Fowler. Continuous Integration. Available online at `http://www.martinfowler.com/articles/continuousIntegration.html`.

[9] J. Frederick P. Brooks. *The Mythical Man-Month*. Addison Wesley Professional, 1974.

[10] T. L. Graves, A. F. Karr, J. S. Marron, and H. P. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661, 2000.

[11] A. E. Hassan and R. C. Holt. Studying the chaos of code development. In *Proceedings of the 10th Working Conference on Reverse Engineering*, Victoria, British Columbia, Canada, Nov. 2003.

[12] A. E. Hassan and R. C. Holt. Source Control Change Messages: How are they used? What do they mean? 2004. Draft Available Online.

[13] J. Quinlan. *Programs for Machine Learning*. Morgan Kaufmann, 1993.

[14] J. Quinlan. Boosting and C4.5. In *Thirteenth National Conference on Artificial Intelligence*, pages 725–730, Cambridge, MA, USA, 1996.

[15] J. Troster and J. Tian. Measurement and defect modeling for a legacy software system. *Annals of Software Engineering*, 1:95–118, 1995.

[16] Jacky Estublier and Rubby Casallas. The Adele Configuration Manager. In W. Tichy, editor, *Configuration Management*, pages 99–133. John Wiley and Sons, Ltd., Baffins Lane, Chichester, West Sussex PO19 1UD, England, 1994.

[17] T. M. Khoshgoftaar, E. B. Allen, R. Halstead, G. P. Trio, and R. M. Flass. Using Process History to Predict Software Quality. *Computer*, 31(4), 1998.

[18] T. M. Khoshgoftaar, E. B. Allen, W. D. Jones, and J. P. Hudepohl. Data Mining for Predictors of Software Quality. *International Journal of Software Engineering and Knowledge Engineering*, 9(5), 1999.

[19] P. Knab, M. Pinzger, and A. Bernstein. Predicting Defect Densities in Source Code Files With Decision Tree Learners. In *Proceedings of the 3rd International Workshop on Mining Software Repositories*, Shanghai, China, May 2006.

[20] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(6):308–320, 1976.

[21] D. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053 – 1058, 1972.

[22] Perforce - The Fastest Software Configuration Management System. Available online at `http://www.perforce.com`.

[23] Porter, A.A. and Selby, R.W. Empirically guided software development using metric-based classification trees. *IEEE Software*, 7(2):46–54, 1990.

[24] R. Takahashi and Y. Muraoka and and Y. Nakamura. Building software quality classification trees: Approach, experimentation, evaluation. In *Eighth International Symposium on Software Reliability Engineering*, pages 222–233, Albuquerque, NM, USA, 1997.

[25] J. Rice. *Mathematical Statisitcs and Data Analysis*. Duxbury press, 1995.

[26] D. Saff. Continuous Testing. Available online at `http://pag.csail.mit.edu/continuoustesting/`.

[27] J. S. Shirabad. *Supporting Software Maintenance by Mining Software Update Records*. PhD thesis, University of Ottawa, 2003.

[28] R. K. Yin. *Case Study Research: Design and Methods*. Sage Publications, Thousand Oaks, CA, 1994.