Automated Identification of Breaking Changes in Continuous Integration Systems Using Under Uncertainty Reasoning

mgr inż. Stanisław Świerc

Silesian University of Technology

Faculty Of Automatic Control, Electronics And Computer Science

Institute of Computer Science

Supervisor:
Dr hab. inż. Krzysztof Cyran, prof. nzw. w Pol. Śl.

July 7, 2015

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

The theory of probabilities is at bottom nothing but common sense reduced to calculus; it enables us to appreciate with exactness that which accurate minds feel with a sort of instinct for which ofttimes they are unable to account.

Pierre Simon Laplace, 1819

Contents

List of Figures

List of Tables

1 Introduction

Many software engineering Researches have argued for more use of data and data mining algorithms in Software Engineering [BZ14; HX10; BZ10]. One of the obvious benefits is the opportunity to make more informative decisions about the project during its development phase. However, the data can also be used to enhance the capabilities of tools used every day by all contributors. In particular, advanced automation and decision support systems can take the burden of many manual tasks and let people focus on though-provoking, creative and more valuable work.

One of the most popular software development practice is Continuous Integration. It became almost a standard in the industry worldwide. At its core it encourages developers to integrate their changes often, even several times a day. Each change can be considered as integrated only after the product is successfully rebuilt and it passes a set of predefined tests. In order to help people follow these guidelines many supporting software tools have been created. They automate many steps in the integration process, which would have to otherwise be performed by the developer. However, there are some tasks that even today still have to be done manually.

When integration builds fail they are typically diagnosed by developers who know the project source code well enough to find defects and fix them. This task is very challenging to automate because of a few reasons. First, the diagnosis requires good understanding of the project structure and the technologies it uses. Second, the fix might involve modifying source code to a level only a human can handle. There are other strategies for managing broken builds which do not suffer from these problems, but they have their own limitations.

Once a project reaches a certain size and the integration builds start getting long, with compilation phase reaching more than several hours on modern hardware, CI becomes very challenging to practice. It gets even more problematic if teams are distributed geographically in different time zones and all require both the CI system to be available and the project

source code to be in a healthy state. One solution to this problem is to follow a post-integration verification strategy with backward-fix policy as described in detail in the following chapters. It assumes that changes are checked in batches and if the subsequent integration build fails due to some detects then culprit changes get reverted from the main Version Control System. This task can be handled by designated developers or by a dedicated team depending on the project size.

1.1 Problem statement

By analyzing different strategies for managing broken builds in large-scale Continuous Integration systems we identified certain manual tasks which can be avoided by delegating them to a dedicated expert system. We recognized that the task of fault diagnosis can be automated. Additionally, with backward-fix policy it is possible not only to find the problem but also take some actions to automatically resolve it.

In this dissertation we argue that:

> *It is possible to create an autonomous software agent capable of diagnosing faults in integration builds and automatically fixing them by reverting changesets which have introduced defects to the project source code.*

We also state that this thesis can be proved by designing an autonomous software agent with the listed capabilities, enabling it in a commercial Continuous Integration system, and showing its utility in that environment.

1.2 Dissertation organization

This dissertation described interdisciplinary research on Software Engineering and Bayesian modelling of reasoning systems, and it is reflected in its structure. Chapter 2 introduces the practice of Continuous Integration and explains the associated concepts. It lists all the core elements and extensions that have been proposed over the last years. Then, it focuses on different strategies for managing broken builds because this part is of special interest to this research.

A completely different subject is started in Chapter 3. It is about building systems for reasoning under uncertainty and Bayesian networks in particular. It covers in details different representations and local probability models, which are later used to describe the design of the diagnosis model proposed in this dissertation. The remaining part provides an overview of inference procedures and Bayesian learning, subjects too broad to cover well in this short book.

Chapter 4 begins the part of the dissertation which presents contributions of the research. It contains novel design of sample format capable of storing enough information about failures in Continuous Integration systems to build a data set which can be used to train and evaluate diagnosis models. Then, it lists several real-world scenarios and shows how they can be modified to enable manual and fully automated data collection process. Finally, it proposes several steps to improve data set quality.

Chapter 5 describes the most important contributions. It starts with a list of requirements for the successful fault diagnosis system. Then, it covers each step of the diagnosis procedure with a strong focus on the structure of the Bayesian network used to calculate probability distributions for random variables of special interest. Its last but one section it assumes fixed model structure and lists different options to train the model from the available data.

Usefulness and the overall efficiency of the proposed model is discussed in Chapter 6. In the first section there is information about where the data set used in the research is coming from and about its basic characteristics. Because the proposed diagnosis agent and the data set itself have some unique properties it was necessary to plan the research carefully to make sure estimated quality measures are not strongly biased. With well defined approach the remaining part of this chapter contains many quality measurement plots.

Finally, Chapter 7 mentions significant discoveries of the research and gathers all of the main contributions in a single place. It ends with the final conclusions from the research.

2 Continuous Integration

There have always been many different software development methodologies. In 1999 Kent Beck, who at the time was working as a project leader, collected practices that he found particularly helpful and published them in his thought provoking book "Extreme Programming Explained" [BA99]. The *Extreme Programming* methodology (XP) that he introduced was designed to help teams develop software in the face of vague or rapidly changing requirements. The word "extreme" in the name refers to the amplification to extreme levels of what the author considers "commonsense principles and practices".

There are several elements of the methodology but despite the promise of synergy not all of them were broadly adopted by the industry. By far the most popular practice is the *Continuous Integration* [DMG07]. It originates from *nightly builds* which has been considered as a best practice for many years [CS98]. According to the author he decided to extend it after making the following observations:

- Integration testing is important and should be done frequently,
- Costs of integrating source code are growing with time.

In order to solve these problems Beck formulated and published the following rules [BA99]:

> *No code sits unintegrated for more than a couple of hours. At the end of every development episode, the code is integrated with the latest release and all the tests must run at 100%.*

It is critical to understand that the term "continuous" is used in somewhat misleading way. Continuous typically implies that something starts and never stops. This suggest that the process is constantly integrating which is not the case here. On the contrary the integration process is triggered every time a development episode is completed. In that sense

the process could be better described as "continual integration", but that term never became a standard.

Another important aspect is the presence of the integration tests in the process. It is not enough for a change in source code to compile to be considered correct. It has to also result in a product that passes a set of predefined tests. Otherwise, there is a clear indication that either the feature implementation or the tests do not follow the specification. In both scenarios the problem should be surfaced to the development team and fixed.

Since it was introduced for the first time Continuous Integration has been evolving. There was a lot of research made to evaluate the impact the CI has on the organizations that practice it [Mor+10; Kim+08; DPH12] as well as on improving the model [Rob04; Sto08; GS12]. Moreover, many Independent Software Vendors (ISV) created custom solutions and systems that support this practice [Rog03; Sto07].

2.1 Principles and practices

In order for CI to work effectively on a project, developers must change their habits and follow some of the practices. In his work, Kent explained the core principles of the methodology [BA99], which were then broadened by the subsequent authors. Fowler and Foemmel [FF06] in their article described extended set of rules that led to the higher productivity in the projects they studied. Later, Duvall, Matyas, and Glover [DMG07] gathered them all again in the fist book focused exclusively on CI systems, where they explain in details every single one of them.

There are ten practices that are considered to make up an effective CI environment. They are described in the following sections.

Maintain a single source repository

Software projects can involve a lot of input files that are used to build a product. In order to keep track of all the changes that are happening in the project, teams typically use a Version Control System (VSC). It is a system whose main responsibility is to store files together with all their previous versions. Because most of the interactions with such system are related to storing and retrieving files, they are typically referred to

as *repositories*, a term will be used interchangeably with version control system further on. A unit of change in the repository is often called a *changeset* because it groups a set of changes to multiple files and save them time together as an unit with additional metadata such as the author identifier, timestamps and comments.

Although many teams use repositories a common mistake is that not all the files that are required to build the product are kept there [FF06]. Not only does it include source files but also test scripts, configuration files, database schemas, install scripts and other scripts related to the project. They should all be stored in a single place. Doing so makes it easy to setup a new development environment and avoids problems related to the product building successfully on one machine and failing on another.

Automate the build

The goal of CI system is to encourage frequent integrations of the changes in the project. One of the most important steps of this process is a build, which is much more than a compilation. A build may consists of the compilation, testing, code inspection, deployment and any other tasks required to consider a change as correctly integrated. Typically this is a complicated process which consists of multiple tasks.

Although the automation is not required, writing automated build scripts can reduce the number of manual, repetitive and error-prone tasks performed on a software project and is strongly encouraged. There are some dedicated scripting languages available for defining the build integration process [FF06; Rog03].

Make the build self-testing

CI was introduced to increase the frequency of the integration tests run in the project. Naturally tests should be executed as part of the process. There can be different types of tests included: unit tests, component integration tests, system integration tests.

There are many different ways to organize tests. Duvall, Matyas, and Glover [DMG07] recommends grouping and ordering tests by the time it takes to execute them. This gives developers flexibility of selecting the scope of testing they want to perform locally during development.

Czerwonka et al. [Cze+11] on the other hand discussed the advantages of failure prediction models for test prioritization. By running tests that are most likely to be effective in finding defects in the changed code it is possible to decrease the resource utilization, thus decreasing the overall costs.

Everyone commits to the mainline frequently

Contemporary version control systems allow to create multiple branches to handle different, concurrent streams of development. Although helpful, this feature can be overused. There is non-negligible cost associated with maintaining branches and moving code between them.

Fowler and Foemmel [FF06] recommends using single *mainline* branch which represents the current state of the project under development. Developers should work of it most of the time. The only reasonable branches other than the mainline can be created for maintenance of prior version of the project, features which are disruptive to the rest of the product and require extra isolation or temporary experiments.

This practice encourages developers to communicate with each other about the change they are working on. Prior to the integration they are forced to update their working copy to match the mainline, resolve any conflicts and execute local build. If the build is successful they are free to commit to the mainline. By repeating this process frequently developers are able to detect conflicts quickly and fix them before they grow to a point when they are hard to resolve.

Every changeset should build the mainline on an integration machine

When many developers are working directly of the mainline branch it has to be maintained in an eventually correct state. Changesets which fail to integrate are inherent part of the process. However they should be fixed in a timely fashion before somebody updates the working copy to a bad version and loses time diagnosing integration problems introduced by someone else [FF06].

There are many possible reasons for the integration to fail. Developers can forget to update working copy and execute local build prior to committing changes. Some integration tests may be nondeterministic and

failing in a random fashion. Finally, there may be some environmental differences between developer's machines.

The last problem can be solved by introducing a dedicated integration machine. Such machine should not be used for any development. Its sole purpose should be running integration builds. Only changesets that pass such build should be considered as correctly integrated.

Although it is not a requirement, the process can be improved with a Continuous Integration Server [Rog03]. This is a software that monitors the version control system for new changes. Once they are detected it initiates a new build on an integration machine. When the build is done it notifies authors about the outcome.

Keep the build fast

An important aspect of the CI is that it provides a rapid feedback mechanism. The rate of this feedback depends on the time it takes to execute an integration build. The longer it takes to complete the build the longer developers have to wait not knowing if their changes are correct or not. This can significantly reduce their ability to practice CI. Therefore, it is crucial to keep the build time reasonably short.

Rasmusson [Ras04] describes the negative impact of long running build on the team morale, productivity and consequently the project return of investment. He pointed out that when developers are waiting for a result of integration build they are distracted and cannot reach their full potential. Moreover, even if they start working on the new functionality they may be using code that has not been verified, thus increasing the amount of integration to be done later.

Rogers [Rog04] points out that long builds are typically caused by including too many verification tasks. It is common to include tasks such as compilation, unit tests, acceptance tests and building deployment packages. They all should be executed these frequently, however, not necessary at the same time. Instead of having one serial integration process it can be split into a set of independent consecutive or concurrent processes which can run with at different frequency. This solution is typically referred to as *staged builds*.

When grouping tasks, it is important to consider different parties that rely on the integration build and their individual requirements. For de-

velopers it is essential to have the code base that can be compiled and a suite of unit tests that are passing all the time. A dedicated build process consisting of only these tasks can be much shorter, thus encouraging developers to integrate their code more often.

On the other hand, feedback on acceptance tests, although also important, are not critical for a day to day work. They can be included in a different type of build triggered less often.

Test in clone of the production environment

The core reasons why CI was created was to increase the frequency of integration testing. To make the tests outcome trustworthy they should be executed in an environment similar to the production environment. Every difference results in a higher risk of missing a faults sensitive to the setup.

Fowler and Foemmel [FF06] explained that the goal should be to test in an environment which is an exact clone of the production environment. They admit that sometimes the costs of full duplication can be prohibitively high. Then, it is necessary to introduce some differences and acknowledge the risk they entail.

Stolberg [Sto09] in his case-study talks about the advantages of executing tests on a virtual machines (VM). In his setup he introduced a test controller which launches tests in a clean VM with an environment similar to production. This mitigates the problem of files left from a previous installation of the project interfering with the new files and makes the process more deterministic.

Make latest artifacts easily accessible

Apart from an information about the state of the project integration builds can also produce artifacts. Depending on the project type they can be executable, installers that can be run for demonstrations or exploratory testing, but also detailed reports regarding the code base like coding style violations. Because of the great diversity in the functions of artifacts produced it is critical to make them easily accessible to people involved in the project.

This practice ties nicely into the iterative approach for the software

development, where at the end of each phase the project despite having incomplete functionality can be demonstrated to stakeholders [Sch04]. In order to make this possible the artifacts from the last integration build in the phase should be saved in a distinctive location and be marked accordingly.

Keep the process transparent

With CI the integration status of the project is updated regularly. This information together should be easily accessible to everyone in the team. It is particularly important for developers to be aware of this status, so that any problems in the code base can be fixed in a timely fashion. Moreover, there is also behavioral aspect. If the integration builds are successful people experience positive reinforcement which encourages them to sustain this situation. Strength of this stimulus depends on the notification mechanism used in the project.

Ablett et al. [Abl+08] studied the effectiveness of different types of mechanism of notifying developers about successful and unsuccessful integration builds. They compared e-mail based solutions, ambient awareness devices and a robotic device in an academic environment. Their results indicate that the most effective solution is a combination of openly visible, but unobtrusive ambient device with a virtual communication such as e-mail.

Similar experiment was repeated by Downs, Plimmer, and Hosking [DPH12], but this time it was performed in an industrial environment with a bigger team. Contrary to Ablett, they focused on checking different hypotheses regarding project development metrics. They showed that the proportion of failed builds which were fixed the same day increased substantially above the baseline measured prior to the experiment. The transparent process gave the team members sense of awareness and responsibility for the failed builds.

Automate deployment

If the tests are being executed in an environment similar to the production environment there are probably some scripts which automatically install the software. A natural progression is to make them flexible enough to

deploy to any environment including production.

Of course the product should reach a certain quality bar before it is presented to users, but having automation can make the deployment process cheaper, thus allows the team to do it more often. Moreover, since all the steps are well defined in the script there is lower risk of errors related to some steps being executed in different order and the whole process become more repetitive.

As with any deployment there can be some hard to predict issues. Therefore, it is also recommended to prepare a set of scripts that can revert to the last known good state. This reduces a lot of tension of the deployment and encourages people to do it more often, thus deliver business value quicker [HF10].

2.2 Elements

Continuous Integration is a software development practice. It is all about how people behave and not what tools they are using. Nevertheless, there is a broad range of software solutions that can help developers follow the guidelines and make them more successful.

There are different variations of CI systems designed to work well in projects of different types and scales. The most popular environment is depicted in the Figure 2.1. It consists of just a handful of elements.

Developers work on the functionality on their workstations. When they complete a task they update their working copy to match the mainline and resolve any conflicts detected by the version control system. At this stage only conflicts between overlapping textual regions in the same source file are detected, but not between concurrent changes to different files. To make sure that there are no obvious issues they perform local build which is much faster than full integration build but still contains some verification steps.

If the local build is successful they commit changes to the Version Control Repository. It is a place where all the changes to the source code and other software assets such as documentation are stored together with the metadata, which describes the circumstances of when they were added. Although single repository can keep track of concurrent development efforts in branches there is one mainline branch which represents the current

state of the project.

Change to mainline is what triggers full integration build in the Continuous Integration Server. There can be different strategies for detecting changes. The most popular is simple polling. Server periodically connects to the repository and checks if there are any new changes. When they are detected it fetches them and triggers a full integration build.

On the completion of the build the results are published and communicated back to the developers through a feedback mechanism. There can be different solutions used, e-mail being one of them.

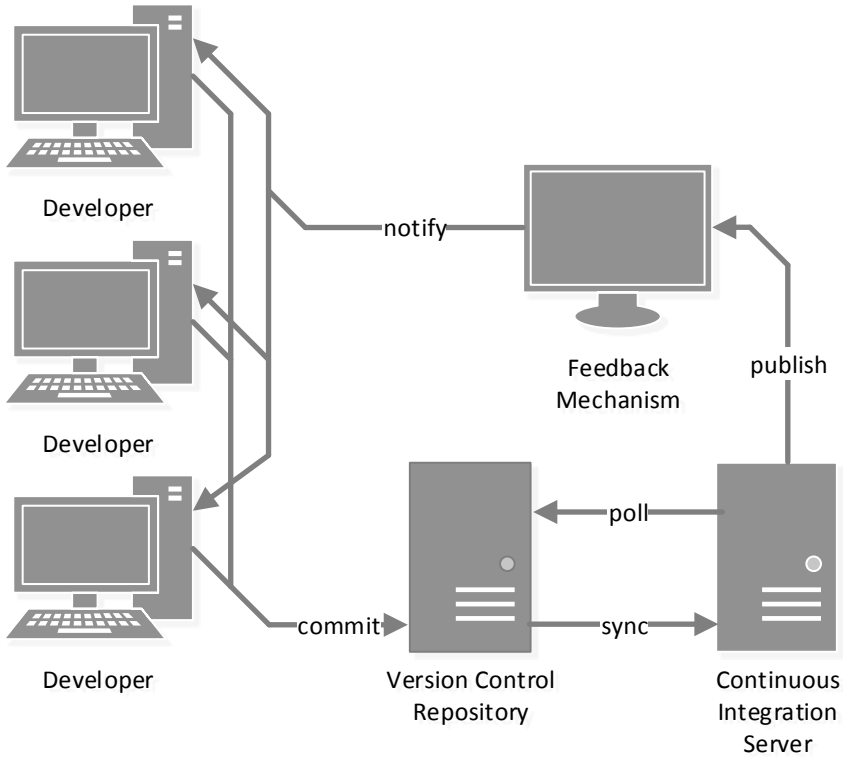


Figure 2.1 Elements of a Continuous Integration system

2.3 Integration build

Integration build is the most important element of the whole system. It is a process of transforming source code and other software assets into components, combining them into a software system, verifying that it is coherent and possibly deploying. It is much more than just a compilation or its dynamic language variations.

The build consists of multiple interconnected activities. Depending on the project they may vary but typically they can be grouped into the following phases.

Preparation

Preparation is the first phase of the process. Its goal is to acquire resources necessary for a build and make sure they are in the correct, clean state. The definition of what it means may vary depending on the project, but there are some common tasks. In particular, the integration machine should be reserved and its working copy of the project should be updated to match the mainline. Additionally, the intermediate and output locations used during the build should be cleared.

Compilation

Continuous source code compilation is one of the most basic and common features of CI systems. In this phase source files and other assets are transformed to create an executable. Other than producing artifacts the tools that are executed also perform basic verification and inspection. Compilers check if the source code adheres to the rules of a selected programming language and can also report warnings related to some constructs known to be problematic.

Verification

Once the executable files are available it is possible to use them to verify that the system matches expectations. This phase contains all the steps that are required to pass before the change is considered as correctly integrated. Typically this is a time when the automated tests are being

executed. There is no single right answer to what is the appropriate scope of testing, but it is recommended to run at least unit and integration tests.

Some projects require special verification steps. Lier, Schulz, and Lütkebohle [LSL12] proposed a CI system for verification of simulated robot models. In their work they presented a pipeline where correctness of each change was checked by deploying it both to simulation environment and to a real robot for comparison.

Inspection

The inspection phase focuses on the Quality of Code (QoC) which is defined as a compliance of the set of source code construct to pre-defined design and coding rules [FP98]. In contrast to the quality of product, which is the compliance of existing functionality with the specification, QoC is not clearly visible characteristic of the system. It is typically hidden and may become visible much later, after the system is deployed and when the project enters maintenance stage [Mis05]. By including inspection in the integration build the metrics not only become visible but the trends become available, thus the quality can be managed better.

Although it is possible to configure the system to fail the integration build if the change violates any of the coding rules, it is more productive to have a thresholds on the quality metrics instead [Bug09]. The reason is that most of the fixes to coding rule violations are small, thus it makes sense to make them in batches.

Design rule violations are more complicated and may require refactoring, which is the process of restructuring existing computer code without changing its external behavior [Fow99]. Since changing design may be expensive, not only should it be justified with quality metrics but also planned. This is another reason why the integration build should not fail immediately during inspection.

The outputs of this phase are detailed reports regarding the Quality of Code metrics. They should be stored either with other build artifacts or in external systems, and should be available to development team as well as to the stakeholders.

Deployment

The last phase focuses on the deployment aspects of the software. Its goal is to generate the bundled software artifacts with the latest changes and make them available to a staging environment. The format of the bundle depends on the target environment.

The deployment process can be included in the integration build, but that is rare practice. Typically Quality Assurance team will coordinate carefully where and when the software is being installed not to interfere with any ongoing test efforts in the staging environment. Whereas the deployment to production should be controlled to minimize the impact on the users.

2.4 Benefits

Setting up a continuous integration system can be expensive in terms of the hardware required to run the builds and the time it takes to configure everything. However the following benefits typically outweigh these costs.

Integration costs reduction

The most obvious benefit and the one which was a core motivation behind CI practice is the reduction of integration costs. They are growing in time so the best way to keep them low is to encourage developers to integrate their changes often.

Risk reduction

Because integration tests can run many times a day defects are discovered when they are introduced instead of during late-cycle testing or even after the project is deployed. Moreover, for most of the problems the sooner they are detected the easier it is to fix them, and the lower the costs of making the fix is [SE04].

By executing all the builds on a separate integration machine the process becomes more reproducible. The risk of not being able to rebuild the project due to missing third-party libraries or special environmental variables practically goes away.

Repetitive process reduction

Practicing CI encourages to improve automation used in the system and eliminate repetitive tasks which would otherwise be expensive in terms of time, effort and possible errors. This frees people to do more thought-provoking and valuable work. With the solid base of existing automation scripts it is much easier to extend them to cover more tasks as the opportunities emerge.

Availability of deployable software

This is the most important benefit from a perspective of clients or users. With CI system in place the product is always in a state where it can be deployed. When there are any problems reported, they can be fixed almost immediately and delivered to users. In projects where the integration is a separate phase in the development process it might be underestimated in time and even delay the release of the product [HF10].

Improved project visibility

Integration builds which include inspection steps constantly produce information about the health of the project and make them visible both to the team members and project owners. This data is always related to a specific version of the project fixed in time so it is possible to notice trends and respond to them. This also creates a natural setup for experimentation on the project because there is available data which can be used to check if a change to the process had the desired outcome.

Increased project quality

Improved project visibility has additional advantage that deserves to be listed out separately. When developers are aware that Quality of Code measures are monitored with every change they make, they put more effort into adhering to coding and design standards. Bugayenko [Bug09] in a case study of five commercial projects observed that the post-delivery defects rate, which is the relation between the number of defects discovered during the twelve months after project completion to the total number of

defects discovered, was much lower in projects where Quality of Code was monitored and sustained at a high level.

Greater product confidence

With every successful integration build the team and project owners know that the product, although incomplete, is in a healthy state. This gives them confidence to make even risky changes because they know that if something goes wrong they will immediately know about it and they will be able to resolve all the issues.

2.5 Broken build management

Integration builds are used to detect issues in the code base and notify development team about it. The most severe type of notification is a failed build which is commonly referred to as broken build. The set of issues that might be considered as breakage can be different for different projects but it usually covers compilation errors and integration test failures.

When a break happens it is important to get it resolved in a timely fashion. Otherwise it might negatively impact all the developers who are working on the common code base. That is because when the project's state is invalid it is hard or even impossible to detect new problems. Therefore, what typically happens is that developers are prohibited to check in new code until the correct state is restored. This interferes with their regular workflow and impedes productivity.

Moreover, there is also a risk of pulling down broken code from the repository. Developers might learn about it after the fact when they are unable to rebuild the code locally and spend time diagnosing defects introduced by someone else. This can be avoided by checking the state at the integration server before updating local workspace, however when developers really need to work with the latest version of the project it no longer is an option.

Based on the last two paragraphs one could infer that broken builds are so harmful that they should be avoided at all costs. However, that is the kind of feedback the CI systems were created for in the first place. It needs to be acceptable to break a build. What makes a difference between

a bad system and the one which is delightful to work with, is how often builds fail and what happens when they do.

Several different strategies have been proposed to solve the problem of a broken build management and they have been shown to work well in different environments. They range from behavioral patterns to more systematic approaches supported by the integration servers. The rest of this section explores them and discusses their strengths and weaknesses.

2.5.1 Related work

Continuous Integration systems have been studied by many researchers and practitioners in the last years. This section gathers some of the publications that were explicitly dealing with broken build management.

Duvall, Matyas, and Glover [DMG07] point out that it is dangerous to assume that everyone knows not to commit the code that does not work to the version control system. In order to foster good behaviors there should be well-factored scripts that developers can run locally to verify their changes before they reach the mainline branch of the project. Of course this practice does not eliminate broken integration builds completely. Therefore it is also important for the project culture to convey that fixing broken build is a top priority.

Fowler and Foemmel [FF06] explain that when developers commit to the mainline they become responsible for the next integration build. They should monitor the process for any failures and fix them immediately. They even suggest a policy that no one should leave the workplace after committing changes until all of them are proved to be correct in the next integration build.

Humble and Farley [HF10, p. 68] also recommend behavioral pattern mentioned above. This discipline is said to be particularly important in projects that are distributed geographically, where a team in a one time zone can be completely blocked if they discover that they do not have the expertise to fix a defect left by a developer in another time zone. This problem can be mitigated by introducing a dedicated team of build engineers, who look after the process.

Martin [Mar11, p. 111] stresses the importance of keeping CI tests running at all times. He suggests that when they fail the whole team should stop what they are doing and focus on getting the broken tests to pass

again. A broken build in should be viewed as an emergency, a "stop the process" event that everyone responds to. This of course is applicable to small projects where the coordination is not an issue.

Downs, Plimmer, and Hosking [DPH12] in their case study observed that developers sometimes are unable to immediately forward-fix all the issues observed in the system due to high complexity of the task or because the problem is outside of the area their control. Therefore it is necessary to either accept the fact that the project will stay in the unintegrated state for longer or implement backward-fix solution.

Miller [Mil08] studied a project run by *Microsoft Patterns & Practices* group during its nine month development cycle. The goal of his research was to estimate the time spent by the team on tasks related to project integration. The conclusion he came up with was that teams moving to CI driven process can expect at least 40% reduction of time it takes to commit and verify a change. Moreover, he noticed that 20% of time spent on maintaining the CI system was devoted to diagnosing and fixing broken integration builds.

Ablett et al. [Abl+08] studied the effectiveness of different types of mechanisms used to notify developers about successful and unsuccessful integration builds. The motivation of their research was to find the best notification device to make the developers aware of the issues so that they can fix them shortly after they are detected. They also highlight that the cost of fixing integration problems grows with time.

Rogers [Rog03] described the design of *CruiseControl* CI server developed by *ThoughtWorks*. This system solved the problem of broken integration diagnosis by allowing only single developer to trigger the integration build at a time. Although very simple, this solution is said to work well in small projects. It is always clear who is responsible for a defect, thus developers are discouraged from adopting practices that are likely to cause the integration build to fail. If a developer does not fix the problem in a timely fashion the whole team is repeatedly notified about it so that other people can react and help to solve it.

In his next article Rogers [Rog04] talked about the need to accept certain failure rate because when developers avoid breaking the build at all costs they decrease the frequency of integration, what makes it harder and more error prone process. In order to decrease the cost of a failure one can modularize the code base and organize developers into small, focused

teams. With this level of isolation it is much easier to find the defect because instead of analyzing the whole project one can focus on a single module.

Rasmusson [Ras04] in his long build trouble shooting guide talked about the impact of a long running builds on the team morale, productivity and presented some steps to overcome this problem. He admitted that sometimes the duration of an integration cannot go down. Then, the efforts should be focused on decreasing the failure rate of the build. Author presented a serialized commit process as a solution where only one person is allowed to commit code at a time. It can be enforced by the system or be a part of the project culture. The drawback of this solution is that it decreases the development velocity of the project, but is said to outweighs the loss of time spend diagnosing and fixing broken builds.

Similar solution was presented by Lacoste [Lac09] who described the development process used for *Launchpad* project run by *Canonical*. Initially the team was using the "feature branch" model where each developer worked on a bug in a separate branch which was merged with the mainline branch only after the test suit passed. This is a variation of the pre-integration verification described in the next section. This model led to a behavior where a lot of developers were submitting branches to be merged at the end of the development phase. When this was happening the integration queue was congesting making the system effectively unusable. This problem was solved by deferring execution of long running tests to the time when more branches are merged, thus batching the revisions to be tested. Additionally a policy enforced by the system was established to prevent anyone from committing the code other than a fix on top of a broken integration build.

Brooks [Bro08] discussed the effect of the time of the integration build and its failure rate on team's behavior. He compared two projects developed in similar technologies but with completely different scope of mandatory verification and consequently with different build times. He observed that with long builds there was a spike in the number of changes committed shortly after the first successful build. Unfortunately this tendency led to higher failure rate because instead of waiting for the code to have the right quality developers were committing code earlier to make it before the integration is broken again. Once a team found itself in this situation it was challenging to get back to the normal workflow. In a project with

a short build time developers did not need to rush, thus their error rate was much lower.

Holck and Jørgensen [HJ07] carried out a case study of two large-scale open source projects: *FreeBSD* and *Mozilla*. They pointed out that build break management is an important aspect. In general, failures are kept in the mainline branch and are fixed there to keep contributors engaged. When a change is reverted it is uncertain if author it going to fix and submit it again. This is particularly important for changes which succeed on one platform and fail on another. Contributors may not have access to all the platforms targeted by the project. In practice, *Mozilla* in general accepts a large part of the responsibility for correcting this type of failures to keep the parity between the supported platforms.

2.5.2 Strategies

Most systematic strategies of managing broken builds fall into two main categories. They can either deal in issues after they are discovered in the mainline in the code that has just been modified (post-integration verification) or they might try to detect problems in source code even before the changes are applied to the mainline (pre-integration verification). The latter category can be further divided depending on who is responsible for executing quality checks and where they are running.

Post-integration verification

One of the most natural way of dealing with problems in CI systems is post-integration verification. From developers' perspective it does not change their workflow in a significant way. Everyone can integrate changes into mainline branch whenever they are ready. These changes are then verified in an integration build and are considered correct only if it passes. However, if the build fails it means that the project got into a bad state and it has to be diagnosed.

While the bad state persists, developers are advised not to check-in new changes as they might make it harder to find the root cause of the problem. Depending on the size of the project the task of diagnosing broken build can be handled by people who made the most recent changes or by a dedicated team of build engineers, who are responsible for the

health of the whole CI system. Whoever starts investigation has to know programming languages and technologies used in the project to understand the relations between different components, find the one which has a defect and correlate it back to the recent changes in the source code.

Once the problem is identified there are two main options of how to resolve it. It might be *fixed forward* with a new changeset checked-in on top of the mainline branch. Alternatively the culprit can be completely reverted from the repository, which is commonly referred to as a *backward fix* because the project is reset back to the known good state.

Both actions lead to a positive outcome because the mainline branch is correct again and the subsequent integration build will most likely succeed, but they have slightly different characteristics and implications. In order to forward-fix an issue it is necessary to understand the source code around the part of the project that failed. Therefore, it is only feasible if developers are looking after their own builds. If the system is managed by a dedicated team who might lack the expertise required to make changes in the source code then backward-fix solution is preferred. It is easier to find a defect than to fix it.

One factor that also needs to be taken into consideration is the time it takes to resolve a problem. In backward-fix scenario the resolution is almost instantaneous because version control systems have great support for resetting projects to a previous state. When issues are forward-fixed it takes considerably more time because the change, which is supposed to fix the problem, not only has to be prepared but also tested before it is applied to the mainline branch. Without local testing there is a risk that developers responsible for creating a fix will try to verify it in the main CI system and effectively hijack it as their private test environment while everyone else is waiting for the project to be correct again.

Local pre-integration verification

One of the main critique of post-integration verification is that mistake of one developer can negatively impact many people who are working on the same project. A great way to mitigate this problem it to enforce some quality gates each change has to meet before it can be applied to the mainline branch.

This scenario is commonly referred to as *pre-integration verification* or

gated builds and there are two main variations depending on where the verification procedure is run: local and remote. Developers might be responsible for executing it on their *local* machines prior to checking in their changes or sending requests to a *remote* server to get their changes automatically integrated only after they are confirmed to be correct.

The scope of verification included in the quality gate can vary between projects. On the one extreme it might be necessary only to compile sources, whereas on the other the process can match exactly what is running in full integration builds. In practice one of the most popular solution is to find a balance where people are asked to compile the project and execute basic suite of very fast unit tests. Then, they can check-in the changes for the CI system to pick them up and execute integration tests.

When developers are making trivial changes they might be tempted to skip the verification and check them in directly. Although most of the time they might be successful, the fraction where they are not can have severe impact because in contrary to post-integration verification people do not expect the project to be in a broken state. Therefore, it is best to enforce the policy to make sure all changes get checked.

Remote pre-integration verification

In this scenario all changes get verified by executing a set of predefined steps on a remote server. Only when do all of them pass the changes are automatically integrated to the mainline branch. This guarantees that the source code will always match baseline quality requirements.

In order to enable this scenario it is necessary to prepare a dedicated machine where the verification steps will run. It is possible to repurpose the integration machine for this task and in fact many commercial CI servers support this mode of operation. It is important for this machine to have a good performance so that developers do not have to wait long to see if their changes are correct or not.

Case studies showed that long queues for pre-integration verification and slow feedback in general leads to a situation where developer integrate their changes less often and loose the real benefits of CI systems [Bro08; Rog04; DPH12]. Therefore, this strategy is applicable only to projects where builds are relatively fast and integration machines can cope with the rate of new changes coming in. Otherwise costs of hardware required

to maintain sufficiently high availability might be too high.

Other strategies and extensions

Apart from the most popular strategies there were other proposals, which have not been adopted by the industry. Their utility was mainly demonstrated in a controlled environments, but hardly any solutions reached a stage where they could be used in a commercial CI systems.

Guimarães and Silva [GS12] introduced an idea of continuous merging of the uncommitted changes in real-time. In their research they performed controlled user experiments on groups of graduate software engineers. During the experiment engineers had to solve a predefined programming problem in a small team with a confederate, who was adding conflicting source code at certain points in time. This empirical evaluation demonstrated that continuous merging makes developers aware of certain classes of conflicts earlier and fosters early resolution before the defective changes are picked up in a full integration build.

Hassan and Zhang [HZ06] presented how decision trees can be used to predict results of integration builds before their completion. Their solution is intended to help developers select the right time of a day to update their working copy of the project to minimize the risk of syncing to a bad state of the project. Additionally this solution claims to be beneficial for managers who can use it plan resources allocation should the integration fail.

Finally Storm [Sto07] proposed extension to the common post-integration verification where projects are split into a set of dependent components versioned independently. In case of a failure in one component it sometimes can be replaced with its previous version to form a complete product. This approach limits the scope of a failure to a boundary of a single component and provides clear backtracking procedure to restore the correct state. However, it requires complex versioning scheme which makes it hard to maintain in practice.

2.6 Challenges of large-scale CI systems

Success of Continuous Integration practice depends strongly on the discipline of the team. The tools are there to help developers and encourage the right behaviors. As the project scales up in terms of the size of its

code base or the size of the team the more elaborate solutions are required to sustain high productivity.

There are several factors that undermine the discipline of team members making the Continuous Integration increasingly hard to practice. Understanding these obstacles and their symptoms is essential in preparing strategies to overcome them by both improving tools and evolving behavioral patterns.

Code base size

When the size of the code base starts to grow, the time it takes to run a full integration build is increasing as well. The increase rate may depend on a specific characteristics of a project. While some phases of integration build such as compilation tends to increase slowly during development, the test phase is less predictable and can grow rapidly dominating the overall build time. This phenomenon is particularly important in projects with high number of *integration tests* which check interactions between different components. If not managed properly, the build time can increase exponentially with the number of tests executed [Rog04].

When the build time grows developers need to wait longer to receive feedback on the results of their integration. Even if they have a new task to work on the lack of closure on the previous task and the context switching decrease their productivity. A natural reaction is to reduce the frequency of check-ins in order to minimize this futile time. However, this creates new problems by itself because it increases the likelihood of merge conflicts and makes each integration much harder task.

There is no simple solution to this problem and the only way to improve experience of people working on the project is to drive the build time down by following some recommendations, which were showed to work in practice [Rog04; Ras04]. Of course not all recommendations are applicable to all projects, but there is a high chance that some of them will work and by combining them together one can increase the gain even further.

Development velocity

Another problem is the development velocity expressed in the rate of new changes that are made to source code. Although the relation is not pro-

portional the more people are working on a project the more changes they can prepare in a given time frame. Apart from that it obviously exacerbates the problem of rapidly growing code base discussed in the previous section it brings some unique challenges.

With high number of concurrent changes there is a high chance of developers running into merge conflicts which need to be resolved prior to check-in. Existing tools, which support three-way merges can make this task easier, but it is still a manual process. The best way to mitigate the risk of conflicts is to split the project into well-defined modules and carefully plan the work to decrease the work concurrency level on any given source code artifact to preferably one.

Team size

Another challenge, which is closely related to development velocity, is how to design a CI system when it is going to be used by many very large teams. Here the biggest problem is that there are many people who depend on working build and a break in a compilation phase has potential to affect all of them. In such environments broken build management becomes critical.

One of the most natural solution is to organize people into small development teams and to divide project into self-contained components, which most of the time can be modified independently. Then each team can gets its own local integration server and the impact of the broken build is limited to the small group of people who work directly on the affected component.

Sometimes, however, it might be impossible to define components with sufficiently clear dependencies and the only way to make sure the whole project is in a correct state is to run a full integration build. In order to decrease likelihood that it will fail there can be local pre-integration verification put in place. The remote variation would not be very practical because it would require many machines to match the high rate of new changes.

Geographical distribution

Designing CI system for large teams is challenging, but it gets even harder if these team ad distributed geographically. This increases the impact of

a broken build because not only it can affect many people but also it can stay in this bad state much longer if the defect was detected outside of business hours of the team who owns the broken component. In worst case developers might be forced to work around the issue until it gets fixed the other day.

Humble and Farley [HF10] recognized this problem and pointed out that when working in a geographically distributed project it is absolutely necessary to fix a broken build before the end of the work day. Violating this rule can not only block a remote team for day but also undermine trust and relationship between teams.

However, such policy can only be enforced if the integration builds are fast enough to run multiple times a day. As soon as their duration reaches one hour developers it might be extremely hard or even impossible for developers to fix issues the same day. One could think about extending policy to force people to connect and resolve problems outside of business hours, but that would lead to the culture where people are afraid to make any changes and the productivity would drop.

A better option is to create a dedicated team of build engineers who are responsible for the health of the whole CI system. Its members should be distributed globally similar to how normal teams are distributed to provide full coverage during relevant business hours. They should be trained in technologies used in the project so that they can diagnose and possibly fix issues in any component.

Of course there are certain types of problems that build engineers will not be able to forward-fix either because the design is not clear or because they do not have the right experience. In such situations it is important to give them rights to revert a changeset, which was identified as culprit even if the attribution is not absolutely certain. It will generate extra work for developers who are the authors of reverted changesets because they will need to verify and check them in again, but the gain of having a whole remote team unblocked for a day justifies this effort.

2.7 Summary

Continuous Integration is a software development practice, which originally proposed as part of *Extreme Programming* methodology. It is the

only element which was broadly adopted by the industry. It consists of a set of rules which encourage developers to integrate and verify their changes frequently by checking them in into mainline branch and executing a predefined suite of integration tests. There are many tools and systems which help teams to be successful in following CI guidelines.

There are many challenges one may face when working with a CI system and they increase when the project grows in terms of code base size and the number of people involved in the project. Moreover these challenges typically come out together. With that regards the most complicated CI systems are the ones which have to serve large teams with hundreds developers working from locations distributed geographically in different time zones, where everyone contributes to the same project with large code base and slow integration builds.

In such projects broken build management becomes a critical aspect. In smaller projects the most effective solution is remote pre-integration verification, but it becomes prohibitively expensive when the number of developers is high. Another solution, which scales much better with development velocity, is the post-integration verification where the integration builds check a batch of changes at the same time.

Although post-integration verification solves scale problems it has its own challenges. The most significant is the impact of broken builds which can potentially affect everyone working on the project. It is critical that all the issues are detected and resolved in a timely fashion. This can be achieved by creating a dedicated team of build engineers who look after this process.

The goal of this research is to design and implement an agent capable of helping build engineers by automatically diagnosing failures, finding changesets which introduced defects, and possibly taking care of them by reverting them from the source code repository. This addition to existing CI system aims to improve productivity of developers by driving down the time to resolution for issues that would otherwise get into the normal development workflow.

3 Reasoning under uncertainty

When we talk about reasoning in the context of real world applications we typically refer to a task where the system has access to available information and it has to reach conclusions about what might be true and how to act [Rus+95]. When designing such system one inevitably has to deal with uncertainty. This is a consequence of several factors. We might be uncertain about the true state of the system because we cannot make all the necessary observations and have to work with partial data. For example in medical diagnosis problems patients true disease is often not directly observable, and his future prognosis is never directly available.

Even if we can make the observations oftentimes they are subjected to errors that have to be accounted for. They are a consequence of limited precision of measuring devices or the measuring process itself. Continuing example from the previous section one can imagine that even as simple and common observation as body temperature measurement has limited precision and depends on such aspects as the time of day and physical activity of the patient.

Finally, uncertainty might be an effect of complex relationships in the real world which are not modelled accurately because they are not well understood or because they are simply not deterministic, at least relative to our ability to model them. For example there are hardly any diseases where we have a clear, universally true relationship between them and their symptoms.

In summary, uncertainty is inherent to real world problems and has to be accounted for. Probability theory provides mathematically consistent framework to quantify and operate with uncertainty. In principle probabilistic model assign probability value to each of the possible state of the system. However in real world application, the number of states can be very high and sparse model representation is necessary to keep it manageable. Probabilistic graphical models are a general-purpose framework for modelling joint probability distribution over many random variables

and their possible assignments or distributions [KF09]. This chapter talks about Bayesian networks which are one of the most popular class of such models.

3.1 Previous work

In the early days of expert systems it was common to model uncertainty with *Certainty Factors (CF)*. This approach was used and popularized by *MYCIN* - a rule-based expert system created in the early 1970's by Buchanan, Shortliffe, et al. [BS+84]. The knowledge in this system was stored as rules of the form "if *evidence* then *hypothesis*" with certainty factor expressed as number in the range between -1 and 1 which indicates if the *evidence* increases or decreases belief in the *hypothesis*.

Although the system was never used in practice it was very successful in controlled studies by outperforming infectious disease experts who were judged using the same criteria [SB75]. This gave rise to research on certainty factors trying to find precise probabilistic interpretation [Ada84; Hec90]. Sensitivity analysis revealed that the system performance did not depend strongly on the change in the rule certainty factors and that the real value of the model was in the rules themselves.

Later Heckerman and Horvitz [HH87] showed that certain classes of probability dependencies, which occur commonly in real-world domains, cannot be represented in a natural or efficient manner within rule-based framework and certainty factors. In particular they identified mutual exclusivity and multiple causation as the main problems. Each of them require either rules with very long list of propositions or many related rules which quickly becomes unmanageable.

Several other methods to handle uncertainty in expert system have been proposed in the literature including *Confidence Factors*, *Dampster-Shafer Belief Functions*, *Rough Sets Theory* and *Fuzzy Logic*. They were demonstrated to perform well in certain domains and thanks to the more recent modifications they are applicable to modern problems. They are also a subject of an active research both in theoretical modelling and practical applications [Woz04; Cyr08; Woz11; KW12].

For example, *Rough Set Theory* is a great option for building medical diagnosis systems [IWD05; PWD07]. Such models make little assumption

about the data what makes them suitable also to the problem of knowledge discovery [IWD07]. It is possible because they operate on a set of uncertain rules, which can be both processed programatically and also interpreted by domain experts.

One of the earliest comparison of popular formalisms for representing uncertainty was created by Wise and Henrion [WH86]. In their research they focused on: *Bayesian Decision Theory*, *Confidence Factors*, *Dampster-Shafer Belief Functions*, and *Fuzzy Logic*. They showed that all the options they took into consideration exhibit significant problems in some key reasoning patterns. This was a clear sign that there is a room for improvement in this space.

In the late 1980's Judea Pearl and his team at *UCLA Cognitive Systems Laboratory* made a significant progress when they created *Bayesian Network* framework and described it in a series of papers that culminated in Pearl's highly influential textbook "*Probabilistic Reasoning in Intelligent Systems*" [Pea88]. Although it mainly focused on Bayesian networks, it draw attention to probabilistic graphical models in general and helped to drive development in this space.

This new approach was demonstrated to work in practice when Heckerman, Horvitz, and Nathwani [HHN91] constructed their highly successful, large-scale expert system for diagnosing pathology samples - *Pathfinder*. It was built on Bayesian network and avoided the unrealistic strong assumptions made by early probabilistic expert systems. This allowed them to reach diagnosis performance equal to human experts [HN92].

Over the years probabilistic graphical models were improving mainly in terms of inference algorithms which were applicable to broader ranges of problems. Some of the most significant advances were the belief propagation algorithm proposed by Pearl [Pea86]. It can calculate probability in any acyclic discrete network. When they were working on *Munin* expert system Olesen et al. [Ole+89] developed a *Hugin* algorithm also known as a *Clique Tree* algorithm. It has better performance and the subsequent iterations improved it even further.

More recent research was focused on effective inference in networks which make use of different families of continuous distributions. Winn and Bishop [WB05] proposed a *Variational Message Passing* algorithm which can perform approximate inference using a factorized variational distribution in any conjugate-exponential model, and in a range of non-

conjugate models. They also demonstrated its utility solving problems in the domain of machine vision and bioinformatics.

At present, although there is active work on other approaches to uncertain reasoning, probabilistic methods in general, and probabilistic graphical models in specific, have gained almost universal acceptance and are applied in many different domains [KF09]. They are particularly successful in problems where there is available prior expert knowledge like medical diagnosis, analysis of genetic and genomic data and fault diagnosis.

Researchers who want to include graphical models in their work can select from many software implementations available today [PHF10; KHH01]. One of the most interesting options is the *Infer.NET* library created by Minka et al. [Min+14] from the Machine Learning and Perception group at Microsoft Research Cambridge. It extends a general purpose programming language *C#* with probabilistic constructs. Its license is open for any non-commertial use.

3.2 Representation

The goal of Bayesian network is to represent a joint distribution P over a finite set of random variables $\boldsymbol{X} = \{X_1, ..., X_n\}$. Even in the simplest case where we deal only with binary random variables, there are 2^n possible assignments of values $x_1, ..., x_n$, and we would need $2^n - 1$ parameters to specify a joint distribution over this space. Of course for any reasonable number of variables this quickly becomes unmanageable for many reasons.

Computationally it would be very expensive to store and manipulate such big model. Cognitively, it would be impossible to acquire so many parameters in the process of expert elicitation and even if somebody could specify their values these numbers would have to be very small and imprecise. Finally, if we were to learn the distribution from data, we would need extremely large number of samples to have statistically significant estimates. These were the main challenges in probabilistic methods before graphical models were proposed to overcome them.

3.2.1 Bayesian network representation

Bayesian networks solve the combinatorial explosion problem of model parameters by exploiting conditional independence properties of the dis-

tribution. Instead of explicitly encoding single high-dimensional distributions they allow to factor it into many related local probability models, which can be combined together only when it is necessary to reason about the original joint distribution.

This model consists of a finite directed acyclic graph, where each node i represents a random variable X_i together with is prior probability $P(X_i)$ or conditional probability distribution (CPD) $P(X_i|Parents(X_i))$, where $Parents$ denotes variables in the graph whose nodes are directly linked by edges pointing at the node i. These edges not only encode information about all dependencies between variables, but also tell a lot about the independence assumptions. In particular, at each random variable is conditionally independent of its non-descendants ($NonDescendants(X_i)$) given its parents.

$$(X_i \perp NonDescendants(X_i)|Parents(X_i)) \tag{3.1}$$

This is a fundamental property which makes it possible to represent the joint distribution in a compact way. By following the graph structure it is possible to combine all the local probabilities stored in nodes to recreate a single global joint distribution as presented. Example of this composition for graph in Figure 3.1 was presented in Equation 3.2.

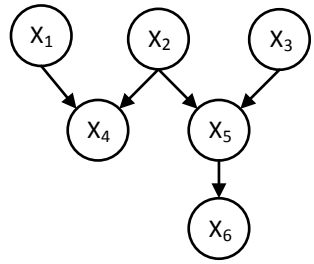


Figure 3.1 Simple Bayesian network with six random variables

$$P(X_1, X_2, X_3, X_4, X_5, X_6) = \\ P(X_6|X_5)P(X_5|X_2, X_3)P(X_3)P(X_4|X_1, X_2)P(X_2)P(X_1) \tag{3.2}$$

Please notice that this factorization of the joint distribution makes no assumption on the form of random variables. It works equally well with both discrete and continuous variables. However the latter are more challenging when it comes to specifying conditional probability.

3.2.2 Local probabilistic models

So far the discussion was focused on the graphs and their relation to concepts from the probability theory. This section is more specific and explains how the local probabilistic models are represented.

Tabular models

When dealing with graphs composed solely of discrete-valued random variables one can always represent conditional probability distributions $P(X|Parents(X))$ as a table that contains entry for each joint assignment to $Parents(X)$. For this table to be a proper CPD it is required that all the values are nonnegative and satisfy following Equation:

$$\sum_{x \in Val(X)} P(x|Parents(X)) = 1 \tag{3.3}$$

where $Val(X)$ stands for a set of all the possible values of X random variable. This model is very general and can encode any discrete CPD. An example of distribution $P(X_1|X_2, X_3)$ with binary random variables, which is represented in this way is presented in Table 3.1.

		X_1	
X_2	X_3	0	1
0	0	0.1	0.9
0	1	0.2	0.8
1	0	0.6	0.4
1	1	0.7	0.3

Table 3.1 Example of tabular conditional probability distribution

This representation has some significant limitations. First of all it cannot model random variables with infinite domains (e.g. continuous random variables) because it is impossible to store each possible assignment in the table. Another problem is related to the number of parameters that need to be specified if there are many parents. This number grows exponentially with the number of parents. Nevertheless, there are certain domains where tabular representation is not only sufficient, but also brings very good performance.

Context-specific conditional probability distribution

Another popular representation was proposed by Boutilier et al. [Bou+96]. They noticed that sometimes CPD tables contain repetitions when certain assignments are dominated by subset of parent random variables. In order to create more conceptually appealing model they proposed *tree-CPD* representation where each node of Bayesian network contains a tree structure which can be traversed to find the right distribution for the given parents assignment.

They also pointed out that in some domains it might be easier to describe distribution in terms of rules that can be directly interpreted by human experts. These *rule-CPDs* are defied by a set of rules which specify entries in CPDs and the context where they are applicable.

Noisy-OR model

Another option to avoid repetitions in CPD tables is to exploit the causal mechanism underlying the relation between variables. One of the most popular solution from this category is *Noisy-Or* model, which can provide a logarithmic reduction of the number of parameters required to specify a CPD [ZD06]. It is applicable to problems where there are multiple independent causal mechanisms that can lead to a certain event and its likelihood does diminish if several of them act simultaneously.

This model works only with binary random variables. Let Y be a binary random variable with k binary parents $X_1, ..., X_k$ the CPD is *Noisy-Or* if there exist $k+1$ noise parameters $\lambda_0, ..., \lambda_k$ such that:

$$P(Y = 0|X_1, ..., X_k) = (1 - \lambda_0) \prod_{i:X_i=1} (1 - \lambda_i) \tag{3.4}$$

Noise parameters $\lambda_1, ..., \lambda_k$ are used to model situation where the causal mechanism is not fully deterministic meaning that even when a given cause it present ($X_i = 1$) it does not certainly imply presence of the consequence ($Y = 1$). The first noise parameter λ_0, on the other hand models different phenomenon. It is commonly referred to as a *leak probability* and it represents influence of causes that are not explicitly included in the model [Loc99].

Sometimes it might be challenging to choose good values for these parameters. They should be selected according to the domain and the phenomenon they model. However, in practice they are typically set manually to very low values so that the whole CPD resembles standard OR logical operator.

Continuous variables

All the models discussed so far were restricted to discrete variables with finitely many values. In many situations, however, some variables are best modelled with values from a continuous space (e.g. position, velocity, temperature). Fortunately the framework of Bayesian networks is flexible enough to encode joint distributions over continuous space as well. The only requirement is that the CPD $P(X|Parents(X))$ has to be defined for every assignment of $Parents(X)$, where parent variables might be discrete or continuous.

One local probability model which is a good approximation in many practical applications is *Linear Gaussian CPD* [KF09]. Not only does it capture many interesting dependencies, but also scales well with the number of parents making this a very general solution. Let Y be a continuous random variable with k continuous parents $X_1, ..., X_k$. We say that Y is described by a *Linear Gaussian* model if there exist parameters $\beta_0, ..., \beta_k$ and σ^2 such that the following Equation 3.5 holds for each assignment to parent variables $x_1, ..., x_2$.

$$p(Y|x_1, ..., x_k) = \mathcal{N}(\beta_0 + \beta_1 x_1 + ... + \beta_k x_k, \sigma^2) \tag{3.5}$$

As the name implies these formula can be interpreted as a statement that Y is a linear function of the variable $X_1, ..., X_2$, with addition of Gaussian noise with mean 0 and variance σ^2. This model, although very useful in practice, has some limitations. For example variance of the child

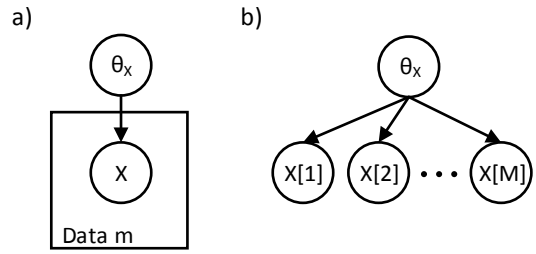


Figure 3.2 Example of a template-based representation
a) plate model; b) ground network

variable Y cannot depend on the actual values of its parents. In order to capture such relation one needs to use more complex models.

3.2.3 Template-based representation

Bayesian networks specify a joint distribution over a fixed set of random variables $\boldsymbol{X}$. However, in many domains probabilistic models have to relate to much more complex space. For example in a temporal setting we might want to track robot's location as it moved in the word and gathers observations. We might limit the space where the robot moves, but to correctly model trajectories we need to set positions in time. Depending on the duration of the observations the model might need to represent series of samples of different length or even infinite series.

In order to make Bayesian networks applicable to such problems they were extended with template-based representation, which can represent an entire class of distribution from the same type (e.g. distributions of trajectories of different length). One of the simplest and best-established solutions in this space are *plate models*. It is an object-relational framework where each instance of the same class share the same set of attributed and the same probabilistic model.

The simplest example of a plate model is presented in Figure 3.2 part *a*. It describes multiple random variables $\boldsymbol{X}$ sampled from the same distribution. In the schematic language of plate models this is indicated by enclosing all the variables in a box titled with the domain over which it

can be expanded. The box represents entire stack of identically distributed variables, thus it is called *plate* to make the analogy to a stack of identical plates.

Given a fixed data set of size M we can instantiate the model from Figure 3.2 part *a* and end up with a Bayesian network similar to the one from part *b*, which is commonly referred to as the *ground network*. This is very simple structure, but it is possible to construct more elaborate networks by nesting plates. It is even possible to share some variables between several plates in a similar way to how the intersection is represented in the Ven Diagrams. Then, such variable is instantiated for each possible combination of items from the surrounding domains. In practice some combinations might be of low interest and they can be trimmed after template is fully expanded to the ground network.

3.3 Inference

Once Bayesian network is constructed it can be used to answer queries of interest regarding the joint probability distribution it encodes. The most common query type one might use is the *probability query*. It consists of two parts:

- *evidence* - a subset $\boldsymbol{E}$ of random variables in the model together with their instantiation $\boldsymbol{e}$,
- *query variables* - a subset of $\boldsymbol{Y}$ random variables in the network.

Then, the goal of the inference task can be defined as computation of the following probability:

$$P((Y)|\boldsymbol{E} = \boldsymbol{e}) \tag{3.6}$$

that is the *posterior probability distribution* over values $\boldsymbol{y}$ of $\boldsymbol{Y}$, conditioned on the fact that $\boldsymbol{E}$ has assignment $\boldsymbol{e}$. This type of query is very common in practice where it is natural to have some evidence which can be used to reason about state which is not directly observable. The discussion about inference algorithms which are applicable to different models is beyond the scope of this section, but it can be found in a great book by Kollar and Friedman [KF09, p. 285-692].

3.4 Learning

Development process of Bayesian networks not only consists of modelling relations between random variables but also finding their distribution. There are three main approaches to this task. First is to specify distributions manually with the help of an expert. This solution, although feasible for some problems, can be nontrivial and expensive task even for modestly sized networks especially if there is disagreement between experts regarding specific probabilities. Additionally this approach is not applicable to domains where properties of distributions are changing over time because that would require experts to constantly update the network.

In order to overcome limitation of manually defined networks we can estimate model parameters from data provided that we have access to samples generated from the distribution we try to model. Currently, with all the advances in data storage and processing solution it becomes much easier to get large data sets than to obtain information from human experts. Ideally samples should be independent and identically distributed (IID). Although, this requirement cannot be always satisfied in practice, it should considered when designing data collection process.

Finally for problems where the data sets have insufficient size or do not describe the complete domain the two approaches discussed so far can be combined together. Such models make use of the prior expert knowledge as a starting point and enhance it with the learnings from the data. This approach was selected for the model discussed in this dissertation.

3.5 Bayesian troubleshooters

One distinctive application of Bayesian network which is worth mentioning in the context of this dissertation is the system fault diagnosis. It is a separate branch of research and the models used for this problem are referred to as Bayesian troubleshooters. They are regular Bayesian networks, but their random variables correspond directly to: causes or defects, symptoms, resolutions and other concepts from the specific domain they model.

Intensive development in this space started in the middle of 90's when *Microsoft* started hiring Bayesian mathematicians and researchers to work

on the systems for diagnosing problems with variety of software products including *Windows* [Loc99]. Apart from many articles and tutorials that came from this group they created a component-centric toolkit for modeling and inference with Bayesian networks *MSBNx* [KHH01]. Although this toolkit was publicly available it was mainly used internally at Microsoft.

Other companies also invested in Bayesian troubleshooters. In particular *Hewlett-Packard Company* created *SACSO* troubleshooter capable of diagnosing problems of printer systems [SJK00]. Apart from the probabilistic query results users could also specify that they are interested in getting suggestions about the optimal repair plans. More recent research and installations confirm that this is a very efficient solution to the problem of system fault diagnosis [PWN09; WKB10; Fra+12].

3.6 Summary

When designing automatic reasoning system for a real world application it necessary to deal with uncertainty. It can be caused by many factors such as impossibility to make observations about all the relevant state or accuracy inherent to the measurement procedures and selected devices. Additionally, uncertainty might be a result of the discrepancy between the system and not fully understood relationships and processes from the real world it tries to model. Probability theory provides mathematically consistent framework to quantify and operate with uncertainty.

Bayesian networks and graphical models in general embrace probability and provide powerful representation which can encode joint distribution over many related random variables. Other approaches have been proposed in the past. Although some of them are still in use, probabilistic graphical models gained almost universal acceptance and can be considered as a state of the art solution for building reasoning systems.

The flexibility of Bayesian networks makes them applicable to problems in discrete and continuous domains. The difference is only in the local probability models used for the random variables. A great effort was put in the research on these models and as a result there is an impressive selection of well understood option one can choose from when designing the network. Apart from the distributions one has also fine grained control over relationships between variables because there are hardly any

constraints on the edges that can be created in the graph.

Once the network is constructed it can answer important queries regarding the joint probability distribution. The most common is a probability query which aims to find the posterior probability distribution over variables of interest, conditioned on a set of observed evidence random variables. It is applicable to many problems encountered in practice where there is a clear separation between observable and not-observable state.

An intensive research on Bayesian networks was focused on the system fault diagnosis. It was driven by commercial companies, which wanted to use this technique to improve reliability of the services they provide, as well as the software and products they sell. This effort formed a branch of Bayesian troubleshooters which showed to be a suitable solution in many domains. They were also selected as the modelling technique for the diagnosis system described in this dissertation.

4 Data set

In order to teach a statistical model it is necessary to prepare a data set which represents the analyzed domain. Previously, chapter **??** discussed methods of estimating parameters for Bayesian networks. They all assumed that there is available a data set consisting of fully observed instances of the random variables the network is built from.

Although continuous integration systems produce a lot of data there is no standard data set which can be used in the research in this space. Therefore, it was necessary to first design a format for samples which will hold enough information to both train and evaluate models. Then, design and implement a data collection procedure which can be integrated with existing CI systems used in the industry and finally run it for a sufficiently long period of time to collect enough samples for the research. This chapter explores above mentioned topics in details.

4.1 Format

In order to estimate parameters for a Bayesian network from the data it is necessary to collect a data set where each data instance contains fully observed random variables used in the network. Therefore, in theory it is enough to preserve only the vector of values of the random variables and not the source data that was used to observe them.

Such approach has some significant drawbacks which are relevant for a long running research. In particular it assumes that the structure of the network and the types of random variables are known upfront. If the structure was modified by adding a new variable it would be necessary to either use algorithms which perform parameter estimation based on partially observed data or collect a new data set. In many cases the latter option is prohibitively expensive or even impossible when the source data is no longer available.

In real-life applications these problems are solved by preserving the relevant source data and making observations on demand [KF09]. With the current prices of the storage space this solution is much more cost effective and provides the flexibility needed in the long running research.

Of course, the decision about which data is relevant depends strongly on the analyzed domain. For the problem of automatic broken integration diagnosis the following four categories of data were selected:

- **Build configuration:** Information about the settings for tools used in the build.
- **Build logs:** Text files with reports generated by the tools used in the build.
- **Build trace:** Trace file contains the information about all processes that were spawned and the files they accessed.
- **Changes:** Collection of new changesets which were committed in the repository since the last successful integration build.
- **Causes:** Collection of changesets which were reverted with relation to the failure.

To make it easier to manage, a single data instance was represented by an archive with a list of relevant files. The following sections explain in detail their content and the process in which they were collected.

4.1.1 Build configuration

Build configuration contains the information necessary for the CI server to start and run the integration build which is preserved outside of the version control repository. It exists because it is necessary to pass to inform the server about the location of the repository before it can connect and update its working copy of the project. Apart from the information about the version control system it may also contain the definition of the scope of the integration, what components have to be built, and which tests should be executed. Finally, this category includes all the information that can be implicitly used in the build process such as: version of the

operating system, version of the compiler and other tools, values of the environmental variables.

Because this information is not typically preserved it is crucial to collect it during the build and save it as an artifact. Most, if not all, the data is textual, therefore a list of text files is a convenient way to preserve it. This is the format that was used this research.

4.1.2 Build logs

In build process there can be many different tools used. Typically, they produce logs or write the reports about the execution directory to the standard output stream, which can be redirected to a file. Apart from the execution trace, logs contain warnings and errors, which are very important to diagnose problems.

Log files can be organized in many different ways. Each tool can write to a dedicated file or the reports can be combine to decrease the total number of files. In the latter option there is a risk that the logs entries coming from concurrently running tasks will be interleaved. This problem can be solved by adding a prefix to each entry which will clearly identify the task it is coming from.

In the data set collected the log files were added to the archive directly without any transformation, but with structured names which identifies the task that created them.

4.1.3 Build trace

In order to save the information about the tasks that were executed in the build and the files they used it is necessary to instrument the build process to record all file access operations. Although some of the tools expose an option to log this information as they are running, this is exceptional case and will not be further considered. Most of the applications used in the build process do not keep track of the files they read and write. Therefore, for the solution to be truly generic and robust the instrumentation has to be done at the level of the operating system.

This is not a trivial task as such instrumentation is not a standard feature of the available operating systems. With access to the appropriate source code a custom version of the OS could be created. However, when

it comes to the commercial software, researches seldom have access the relevant sources.

In order to solve this problem Microsoft Research created and published *Detours* [HB99]. It is a library for intercepting arbitrary *Win32* binary functions on *x86* and *x64* machines. It has been used to extend applications and the Windows operating systems itself without modifying the binary files. There are other solutions to achieve system level interception but they did not reached the stage of being officially supported by the software vendors [LS00].

With *Detours* it was possible to inject instrumentation into the relevant functions in the operating system and record the information about the processes and file system operations they performed. There was no standard format of preserving such data, thus a custom format was designed together with a library for writing and reading it.

4.1.4 Changes

The goal of the diagnosis is to find a changeset which introduced a defect, thus it is crucial to keep track of all the potential candidates. As it was explained in the section 2.1 at page 6, the status of a changeset is considered unknown until it is verified in a full integration build. Consequently, the list of candidates consists of changes that were committed to the repository since the last successful integration. Their number depends on the frequency of builds and on the ability of the development team to fix issues. The more often the mainline branch represents the correct state the fewer candidates to consider.

After establishing which changesets should be included in the data set a decision had to be made regarding the level of details saved. Of course all the information is stored in the version control system so in theory a list of changeset identifiers is enough to make all the observations needed. However, in practice such solution would result in increasing the load on a repository used day to day by the development team and had a potential to decrease the productivity of the people who are working on the project.

In order to make the data set more independent of the production environment more information can be saved. There three basic levels of details that were considered:

- identifiers,
- identifiers with metadata,
- identifiers with metadata and diff.

To illustrate these options better there were three listings created with an output generated from a *Git* version control system [CH09]. They show the same changeset[1], but from different perspectives. The repository and the change itself were created for demonstration purpose, thus they are very minimalistic.

The command used in the first listing uses option `--format=%H` to change the default format and print just the identifier, which is a hexadecimal number with 40 digits.

```
1 >>git log -1 --format=%H
2 f6d945eb06f1444f62c600069b5bae5ec9e02863
```

Listing 4.1 Example of an identifier of a changeset

The command from the second listing uses option `--name-status` to include more metadata in the output. Apart from the identifier in the line 2 there is information about the author, time-stamp and comments. There is also a one-element list of files that were modified in the line 8. The letter 'A' indicates that the file was added.

```
1 >>git log -1 --name-status
2 commit f6d945eb06f1444f62c600069b5bae5ec9e02863
3 Author: Stanislaw Swierc <stanislaw.swierc@gmail.com>
4 Date:   Sat May 10 18:26:32 2014 +0100
5
6     Initial commit
7
8 A       sample.txt
```

Listing 4.2 Example of an identifier and metadata of a changeset

In the last listening option `--patch` causes the diff to be printed, which line by line presents the changes introduced. The format that was used

[1] In *Git* taxonomy changesets are called *commits*.

here is called *unified diff format* and is one of the most popular formats there are [Joh96]. It shows not only that the change added a file *sample.txt* but also content of the lines that were added.

```
>>git log -1 --patch
commit f6d945eb06f1444f62c600069b5bae5ec9e02863
Author: Stanislaw Swierc <stanislaw.swierc@gmail.com>
Date:   Sat May 10 18:26:32 2014 +0100

    Initial commit

diff --git a/sample.txt b/sample.txt
new file mode 100644
index 0000000..08fe272
--- /dev/null
+++ b/sample.txt
@@ -0,0 +2 @@
+first line
+second line
```

Listing 4.3 Example of an identifier, metadata and diff of a changeset

The presented listings were generated artificially. In practice the changes can be much bigger and more complicated. In order to predict which level of details is the most appropriate for the data set, a study of Alali et. al [AKM08] was analyzed. They examined changes in nine open source software repositories to find characteristics of a typical change. The measure they considered were number of files, number of lines and number of blocks. The summary of their research for *GNU gcc* project, which they selected as a good representation, is presented in the Table 4.1. The categories they selected were based on the statistics analysis under assumptions that all the changesets were sampled from the same distribution.

In the project they analyzed most of the changes were small or extra-small. This implies that if a diff was included in the data set the size of a single instance would not grow significantly on average. However, the data also shows that the large and extra-large changes have a non-negligible share. In the most extreme case there was a changeset which modified over 203K lines of code. It was probably a change in the project structure, where the files were moved from one directory to another. The diff of this change was very big and could even reach double size of all

	Number of files		Number of lines		Number of blocks	
Size	Range	Share	Range	Share	Range	Share
x-small	1 - 1	8.4%	0 - 5	19.0%	0 - 1	10.3%
small	2 - 4	68.0%	6 - 46	55.3%	2 - 8	65.2%
medium	5 - 7	12.8%	47 - 106	11.1%	9 - 17	10.7%
large	8 - 10	4%	107 - 166	4.3%	18 - 26	4.1%
x-large	11 - 5K	6.7%	167 - 203K	9.4%	27 - 8K	9.7%

Table 4.1 Changesets characteristics summary for the *gcc* project

the source files in the project. If it was included in the data set it would notably increase the total size.

Based on the observations of the changeset characteristic distributions a decision was made to include the identifiers and the metadata in the data set and to leave the diffs outside. If the training procedure requires some observations to be made on the content of the change that information can be fetch directly from the version control system.

4.1.5 Causes

Finally, the data set to be used in a supervised learning setup has to contain the information about the causes that were manually found by the build engineers or developers. As discussed in the chapter 2 there are two basic types of failures in the CI system, those that are related to the changes in the source code and those that are not. The former type is what this research is focused on.

In order to tell which changeset introduced a defect which broke the integration build it is enough to save its identifier. Additionally, it may be helpful to also preserve the information about the circumstances that led to the failure. Such data can be expressed in a free-form text and associated with the changeset.

Each instance in the data set contains a list of tuples describing the cause with the following elements:

- changeset identifier,
- text explaining the issue.

4.2 Collection scenarios

Continuous Integration system are designed primary for executing integration builds and collecting the information about the problems they encounter, but they do not correlate them with the changes that were committed to the project. This is a complicated task which is typically performed manually by either build engineers or the developers. Unfortunately because their main business objective is to put the project back into correct state the information about the causes is rarely preserved.

This section describes several scenarios of failed integration builds and shows how the data set collection can be enabled in each of them.

4.2.1 Forward fix scenario

One of the most popular strategies of resolving broken integration is a forward fix. It leaves all the changesets, including the one which introduced the repository and commits a change which a fix on top. This scenario was depicted in the sequence diagram in Figure 4.1.

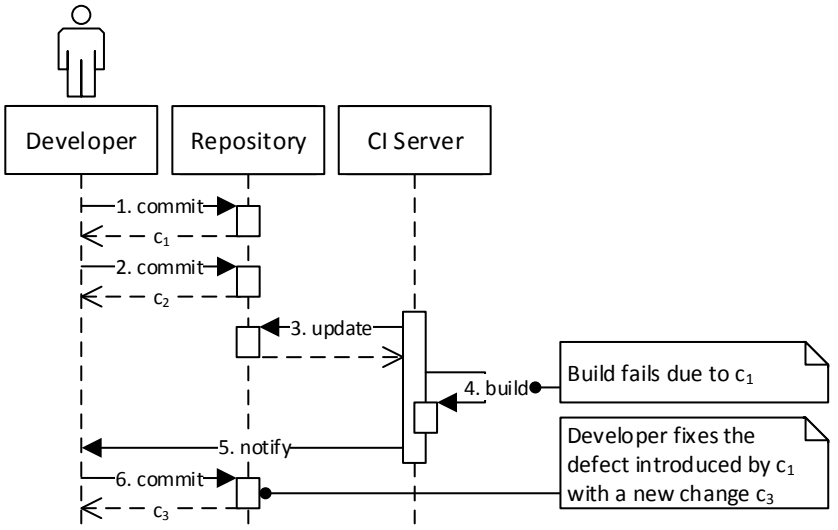


Figure 4.1 Forward fix sequence diagram

Developer commits two changes c_1 and c_2 to the repository of which the first one introduces a defect. They are detected by the CI Server which updates its working copy of the project and starts a new integration build. The build fails and the developer is informed about it. He diagnoses the problem and prepares a fix which is committed on top of other changes.

In order to collect information about the cause a developer has to enter to the system an identifier of a changeset that introduced the defect. In the analyzed scenario it can happen after the fix is committed in the last step.

4.2.2 Backward fix scenario

In some systems a different strategy is used, where all the changes which introduced defects are immediately reverted from the repository.

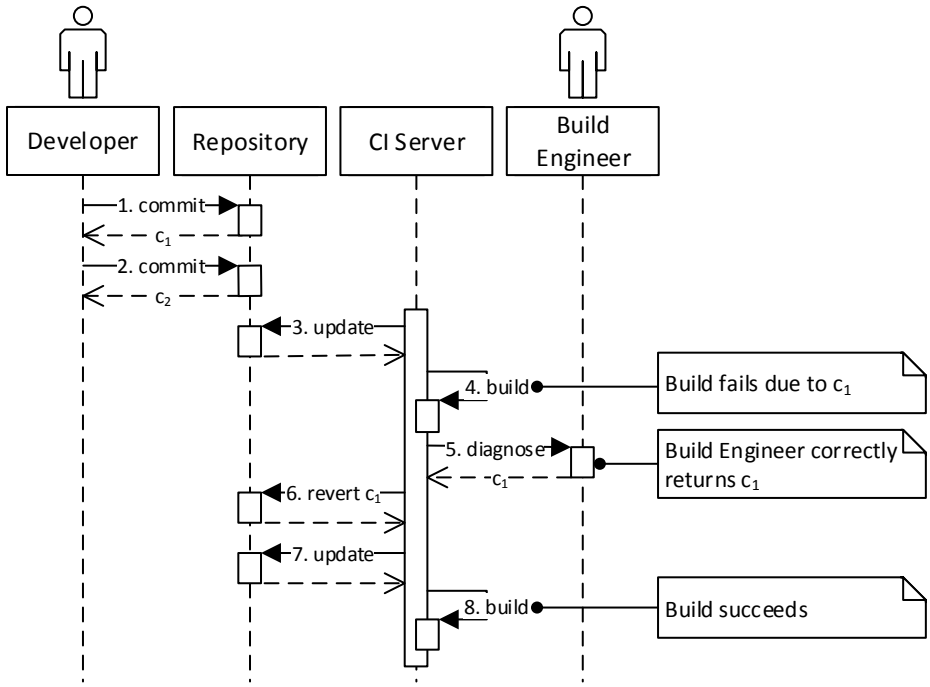


Figure 4.2 Backward fix sequence diagram

Similarly to the previous scenario developer commits two changesets c_1 and c_2. The first change introduces a defect which causes the integration build to fail. This time, however, the diagnosis is performed by the build engineer, who correctly finds the culprit. Because the fix requires detailed knowledge about the project he decides to revert the change with expectation that it will remove the defect from the source code. The subsequent integration build succeeds and confirms that this was the right decision.

This scenario is particularly interesting from the data set collection point of view because the users do not need to enter the information about the problem to in an external system. Every action is tracked in the repository and can be used later to find the mapping between failures and changes that introduced defects.

Although this is somewhat simple scenario because it assumes that the build engineer will correctly find and revert all the bad changesets, it can be frequently observed in practice. This scenario was the main source of data set instances collected in this research.

4.2.3 Ambiguous backward fix scenario

It is not always possible to collect a high quality data set instance in the backward fix scenario. Sometimes there are more defects and they appear one by one in the subsequent integration builds. This results in an ambiguity that cannot be resolved other than by manual intervention. This problem is depicted in the sequence diagram in Figure 4.3.

It extends the previous scenario by adding a second defect to the changeset c_2 on top of existing c_1. The first integration build fails and the logs show evidence of a first defect, but the second is hidden because the build was terminated before it reached the step where the error would have appeared. Build engineer correctly finds and reverts the culprit. This time the second build fails as well and reveals another defect, which is handled in the same way. Finally, the project is rebuilt once more to make sure its state is correct.

Apart from the obvious difference in the number of failed builds, there is one aspect that makes it ambiguous from a data set collection perspective. By looking just at the information that was preserved in the version control system it is impossible to tell if the build engineer made the right decision reverting the change c_1. If it was correct the sequence diagram would

look exactly the same. Therefore, there is no way to tell which changeset introduced a defect which caused the first build to fail.

This scenario is also one that is frequently observed in practice. In order to improve the quality of the data set and avoid the ambiguity a decision was made to ignore the first integration build (step 4) and collect a single data set instance for the second build (step 8) where one can be certain about what caused it to fail.

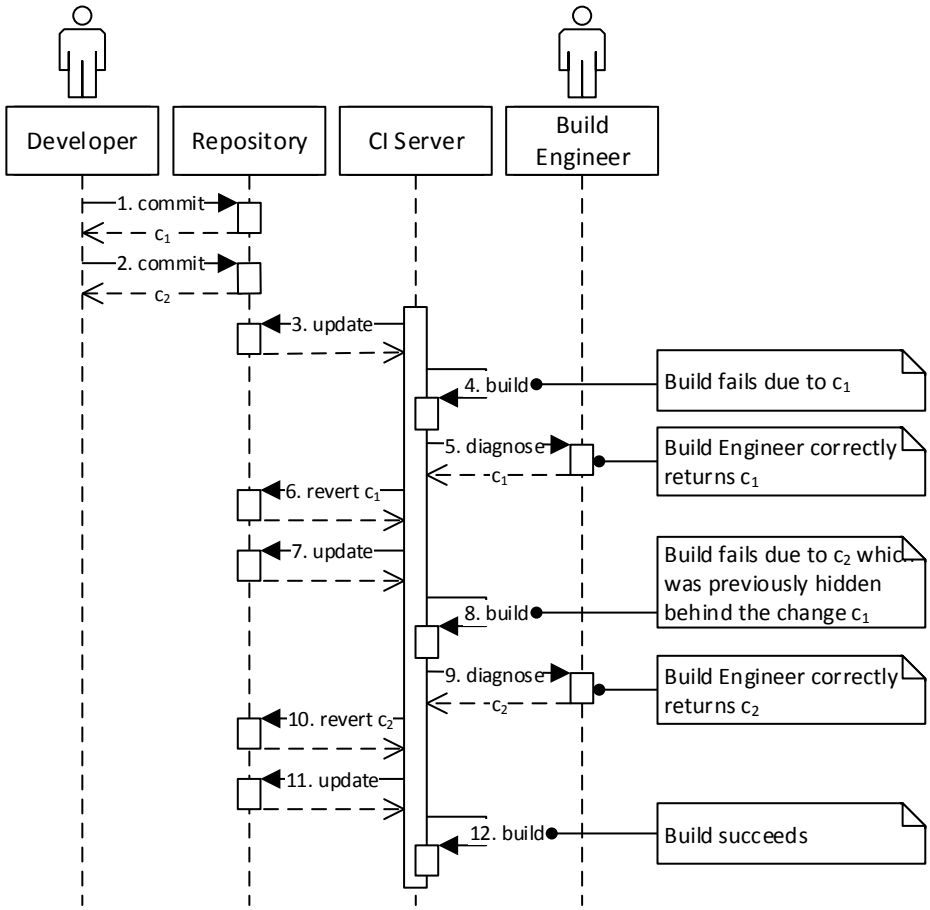


Figure 4.3 Sequence diagram with ambiguous data set collection scenario

4.3 Quality improvement

Shortly after the data set collection mechanism was enabled in the CI system the quality of samples was inspected to make sure that the type and the quality of information is sufficient to perform the diagnosis. At the beginning the performance of a naive diagnosis model was measured. Then, an outlier detection algorithm was run against the data set to find samples that lay unusually far from the rest of the samples.

4.3.1 Initial quality assessment

In the initial analysis a naive diagnosis model was used, which finds all the changesets that modified at least one file used in the build process and sets their probability of being culprits to one. This effectively creates two groups, one with changesets that are related to the build and another with all the other changesets. Because the probability within each group is the same the changeset have an arbitrary order. The performance of this model was presented in the ROC chart in Figure 4.4.

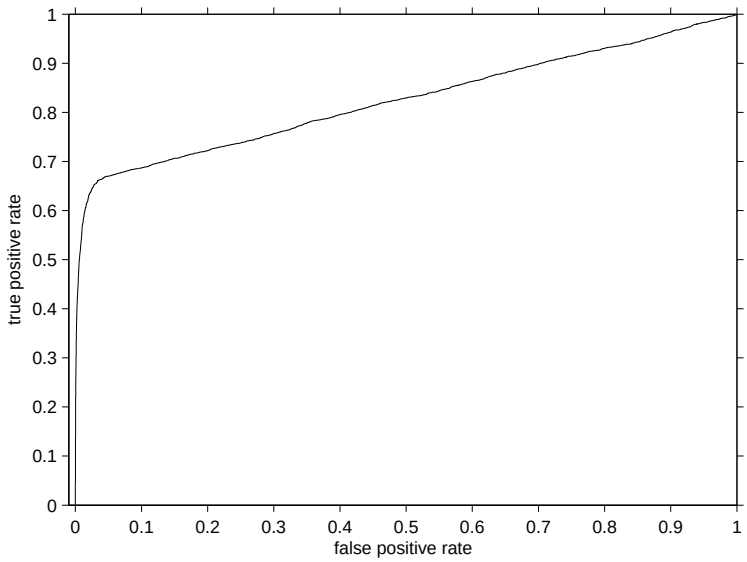


Figure 4.4 ROC chart for naive diagnosis model

It is clear that there are two ranges $(0, 0.05)$ and $(0.05, 1)$ where the curve behaves differently. At the beginning it increases rapidly to reach the level of about 0.66 true positive rate. Then, it flattens out and increases almost linearly up to the top right corner of the chart.

The linear shape of the curve in the second range resembles the curve of a *no-informative* classifier which assigns samples to classes at random [Jam+13]. After closer inspection of samples it was clear that this is precisely what was happening. All the changes which appeared in the second range, had the probability set to zero meaning that they did not modified any files that were used in the build.

In order to find out what is the cause of this behavior 50 integration builds were selected and inspected manually. It turned out that there were some deficiencies in the way the training set was captured which resulted in the higher number of false negatives. These problems are described in the following sections.

The problem of concurrent builds

The data set collection scenarios presented in the previous section were simplified because there was single integration machine involved. In practice capturing was performed in a distributed build system where many integration builds were running concurrently for different platforms and configurations. In this setup there is a risk of a problem depicted in the sequence diagram in Figure 4.5.

The scenario starts in the same way as in the previous examples. The developer commits two independent changes to the repository. The first change c_1 is verified locally by running build targeting platform *x64*. It succeeds but it introduces a defect to the *x86* version of the application. Then, the developer switches to a bug reported for the *x86* platform. Naturally he switches to this environment and uses it to verify the fix in a local build. Unfortunately the fix introduces a defect for the other platform. Both changes pass local verification but introduce two distinctive issues to the project.

Continuous Integration server detects changes in the repository and delegates tasks to the build machines targeting platforms *x86* and *x64* respectively. Both builds fail at around the same time and are diagnosed

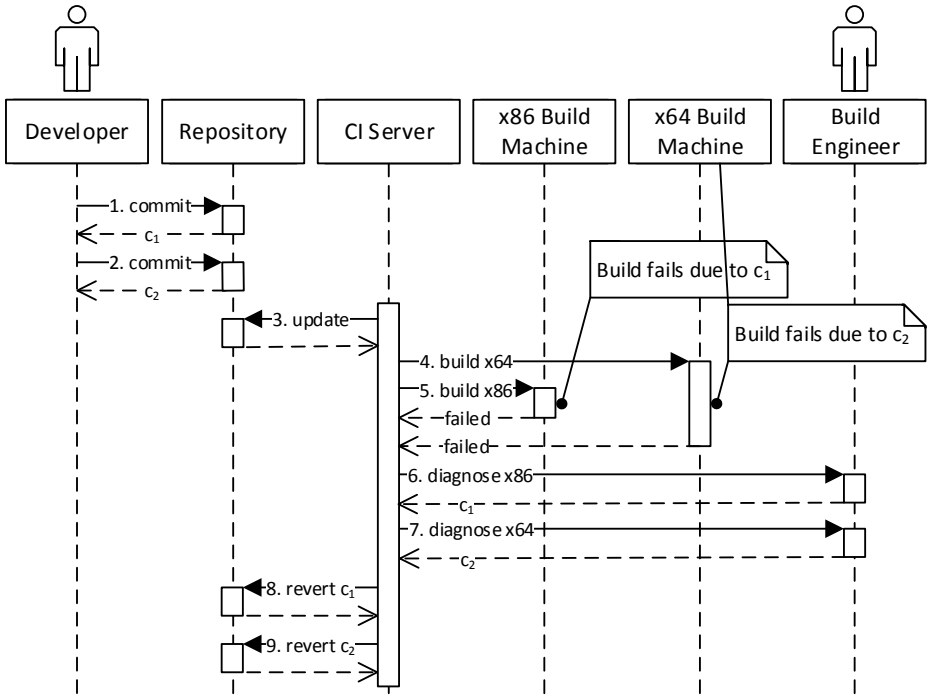


Figure 4.5 Concurrent failures of builds for two different platforms

by the build engineer. In the result both c_1 and c_2 changes get reverted. Finally, the server starts a new integration build which succeeds for all platforms and confirms that the project is back in the correct state.

In the absence of the diagnosis result from steps 6 and 7, the only way to tell if a change was good or bad is by inspecting the repository. The build for platform *x86* failed in the first run and succeeded in the second after changes c_1 and c_2 were reverted, thus both changes appear to be responsible for the failure. Similar reasoning is applicable to *x64* platform. As a result both changes are marked as culprits for both builds regardless if they modified files used in the process or not. Since only one change was responsible for the failure the second can be seen as a noise in the data set.

To quantify this phenomenon the ROC chart from Figure 4.5 was analyzed. The point at which the curve becomes linear is $(0.047, 0.67)$. It

means that the naive model was unable to find any relation between the failure and the changes in approximately 33% of cases.

The problem of voluntary revert operations

Another source of potential noise in the data set was identified through discussions of the practices used in the teams who were using the system. Developers admitted that in certain situations whey would voluntary revert their changes even if they do not cause a build break in the first integration build. This can happen if they are aware of a defect that would either appear only in the higher level build in a staged build environment or pass unseen through the current validation steps.

Although the presence of this phenomenon was confirmed by the users it was impossible to measure its impact on the data set because the voluntary revert operation is indistinguishable from a normal revert performed by a build engineer to fix the defect as depicted in Figure 4.6.

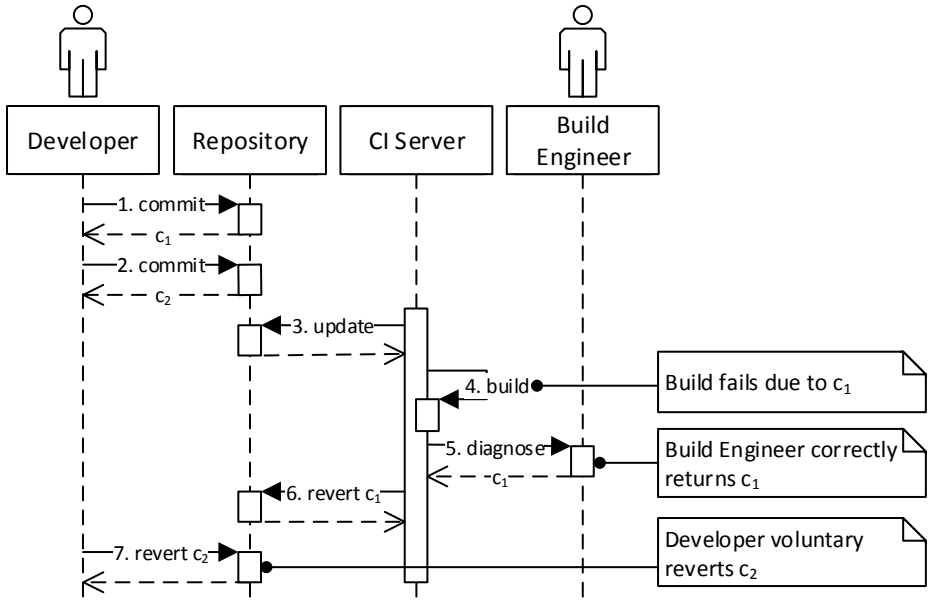


Figure 4.6 Sequence diagram with a voluntary revert scenario

Post processing step

In order to solve the problem discussed previously, a post processing step was introduced to prune all the changes marked as culprits if the probability returned by the naive model was precisely zero. There was a risk that such step would lead to overfitting. Therefore, it was necessary to confirm that it is correct by manual inspection of the effect it has on the representative subset of the training set.

The ROC chart created after pruning is presented in Figure 4.7. At the start it is similar to the previous chart and quickly increases, but instead of reaching the level of 0.66 it goes up to 1 of true positive rate.

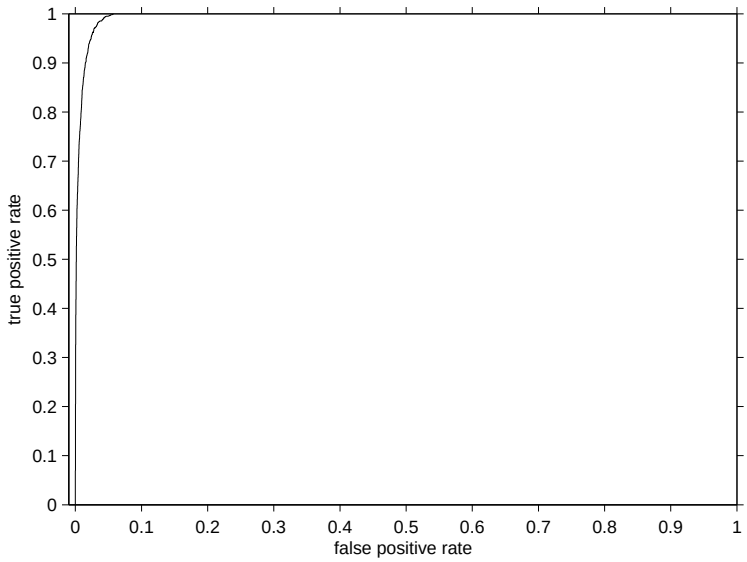


Figure 4.7 ROC chart created after training set was pruned

4.3.2 Outliers analysis

Since the data set was collected in a real setup there was a risk that it would contain some samples which represent extraordinary situations that are not relevant for the diagnosis problem. In order to detect them a statistical analysis was performed. The goal was to identify *outliers*, which

are defined as observations that lie unusually far from the main body of the distribution [DB07].

The environment where the data was collected run two main types of integration builds. There were long running builds which were validating multiple project and short builds focused on specific components. For the purpose of the outlier detection the data set was split by the type of the builds into two categories A and B which were then analyzed independently.

The following measures were taken into account:

- *committed* - total number of new changes in a build,
- *reverted* - total number of changes reverted in relation to a build.

The first question that had to be answered was whether the selected measures have to be analyzed together or in other words is there a relation that has to be taken into account. To answer that the linear correlation coefficient was calculated (denoted by Cor) [Jam+13]. It is a measure of the strength and the direction of a linear relationships between two variables X and Y. It is defined in the Equation 4.1, where n stands for the number of pairs of the data.

$$Cor(X,Y) = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}} \tag{4.1}$$

The correlation coefficients was calculated for each type of the build separately. They both had very low values, what indicates that there is random or nonlinear relationship between the two variables. Consequently the further analysis can be performed for each of them independently.

- $Cor(committed_A, reverted_A) = 0.135$
- $Cor(committed_B, reverted_B) = 0.148$

The distributions of the analyzed measures are not Gaussian, thus in order to find outliers a more flexible approach had to be applied which does not make such strict assumptions. One of the available methods is based on the $fourthspread$ (denoted as f_s) defined as the difference

between the third and the first quartile. An observation is said to be an outlier if it is farther than $1.5f_s$ from the closest quartile [DB07, p. 37].

In order to analyze outliers a pictorial summary was created using *boxplots*. This type of visualization is a popular choice during exploratory data analysis because it summarizes the distribution, but also shows individual samples which does not necessarily fit to the rest. This type of chart presents such features of the data like: center, spread, the extent and nature of any departure from symmetry and outliers. The last element is the most relevant in the context of this section. Each small circle above the whiskers represents a sample which was classified as an outlier

The boxplots for the two measures and two build types are presented in Figure 4.8. In order not to disclose sensitive information the number of committed changes have been standardized to have zero mean and a standard deviation of one. As a result the first boxplot may appear counterintuitive with negative numbers in the ordinate axis.

Both distributions are asymmetrical. The spread between the first quartile and median is much smaller than the spread between the third quartile and median. This tendency is visible for all four distributions. Each boxplot contains some outliers, but both their number and the distance from the main body is higher for the standardized number of committed changes.

Detected outliers were inspected manually. It turned out that they were all related to advanced operations in the version control system. Some development teams were working on features that were classified as potentially disruptive to the rest of the product and required an extra isolation. Therefore, they created and worked in a branch of the project for a longer period of time. When the feature was ready it was merged with the mainline. This appeared as a spike in the number of new changes that had to be validated in the integration build. With different scope of the features there were different levels of the spikes but they impacted all the distributions.

This scenario does not happen very often. There is an extra cost associated with doing development in the isolation and the teams choose it only when it is absolutely necessary. One of the factors that needs to be taken into consideration is the effort it takes to integrate changes back to the mainline. Since this problem is well understood and planned for, it

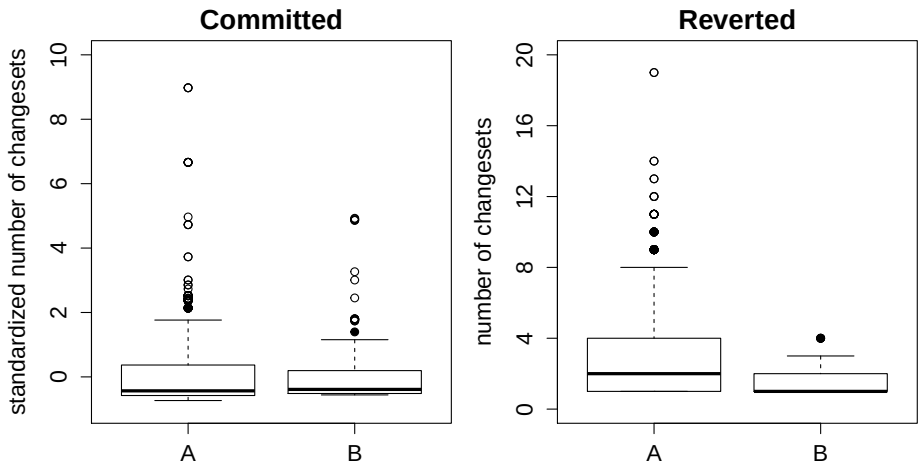


Figure 4.8 Boxplots for the number of changes committed and reverted for two types of builds

is not critical to automate it. Therefore, all outliers related to advanced branch manipulation were removed from the data set. The data instances that were preserve represent casual integration builds.

4.4 Summary

This chapter talks about the problem of collecting data set in an industrial Continuous Integration system. The format of data instances has to be planned carefully to make sure they contain enough information to train the diagnosis model and to evaluate its performance. The proposed solution satisfies these requirements.

There are different patterns of interactions with CI systems. Some of them are good for fully automated data set collection, in others people have to take an extra step and preserve the information which would be lost otherwise. Finally, there are some ambiguous scenarios where it is impossible to collect a high quality data samples.

For the problem of automated diagnosis the backward-fix scenario is preferred. Developers instead of fixing the defect remove completely the change that introduced it. The main advantage of this scenario is that the

information about defect is stored together with the source code inside the version control system, which is already in use and developers are already familiar with it. This information can be extracted automatically and added to the data instances to make them complete.

If the data set is collected in a real, industrial environment its quality may be suboptimal and should be carefully assess before it is used to train the model. Close inspection of the data instances showed that they contain some noise. Concurrent integration builds and platform sensitive defects were identified as the main sources of errors in the data set. When there are multiple changesets reverted at the same time it is hard to map them to the observed defects. This can result in the lower true positive rates.

This problem can be mitigated with a post processing step where all the changesets marked as culprits are inspected to see if they modified files used in the build. Changesets that did not modify any files can be safely ignored. This process can improve the overall quality of the data set.

Finally, the data set may contain some extraordinary cases which do not need to be included in the training process. Outliers detection combined with manual investigation showed that the advanced version control operations, such as branching and merging, can lead to integration builds with either unusually high number of new changesets to validate or the number of defects. Since they are always a result of a conscious decision of the development team such samples can be removed from the data set with an assumption that the automatic diagnosis is not critical in these scenarios.

From a perspective of data collection the backward-fix scenario is preferred because it enables to fully automate the process. Instead of asking users to enter the information about which changesets introduced defects, this data can be obtained directly from the version control system by looking at revert operations and the time when they were performed.

5 Diagnosis model

In the previous chapter we described the format of the data set instances. The main requirement was for it to capture enough information to train the diagnosis model. Now we will show how this data can be used to find defects in projects as well as the changes that introduced them. This model was first presented by Świerc, O'Flaherty, and Rodríguez [SOR14] in March 2014 shortly before it was deployed to a commercial Continuous Integration system.

5.1 Requirements

Before the diagnosis model is explain it is worth mentioning the requirements that such solution should satisfy to make it successful in a commercial environment. They were gathered by analysis of similar efforts in the field of advanced automation as well as through interviews with subject matter experts.

5.1.1 Machine Learning solution

In Continuous Integration systems as their name implies the process does not stop. As long as developers work on the project and commit new changesets there is a need to run integration builds. Each run generates new data with a potential value for a diagnosis agent. This environment creates a great opportunity for a Machine Learning system.

There are many definitions of what learning means in the context of a computer system [Rus+95]. For the purpose of requirements specification a convenient definition was proposed by T.M. Mitchell [Mit97]. It states that a computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks T, as measured by P, improves with experience E. This definition captures the fundamental aspects of learning systems.

In the context of automatic failure diagnosis in the CI systems this requirement implies that the model should be designed in a way that takes advantage of the examples recorded in the data set. Not only should it improve the accuracy of the diagnosis but also adapt to the changes in the environment, where it is being used.

5.1.2 Incorporation of the existing expert knowledge

As it was discussed in chapter 2, currently the most popular solution for handling issues in the CI systems is to delegate the task of diagnosing and fixing problems to developers or to set up a dedicated team of build engineers, who can own this process. This implies that there are already human experts in this field, who understand this problem well.

People who have been working in this domain for many years have knowledge which is scarce and very expensive resource. The model should be designed in a way to make it possible to capture the knowledge from many experts and combine it with what can be learned from the data. Such solution can lead to a system that has a better performance or even outperform any single expert [Sch11].

5.1.3 Interpretable results

Social problems may be an impediment for an adoption of a Machine Learning system. People may feel fear of replacement, fear of change or simply be skeptical to technology [Sch11]. One option to mitigate this risk is to design the system to return easy interpretable results.

Previous research in this field showed that managers are less likely to make decisions based on results from systems they do not fully understand [HZ06]. In order to maximize the chance of the system being adopted in practice the diagnosis model and the results it produces should be clear for the domain experts to examine and verify its correctness.

5.1.4 Scalability with the number of distinctive defects

One of the critical aspects of the diagnosis model is its capability to scale well with the number of distinctive defects. The tools used in the CI systems can generate many different errors and warnings, which in certain cases can be also considered as issues and prevent the integration

Tool	Warnings	Errors
Microsoft Visual C# Compiler (version 12.0.21005.1)	176	953
Microsoft C/C++ Optimizing Compiler (version 18.00.21005.1)	410	981
Microsoft Incremental Linker (version 12.00.21005.1)	46	71

Table 5.1 Counts of errors and warnings returned by selected tools

build from reporting success. Table 5.1 gathers the numbers of distinctive warnings and errors for some popular tools.

It is clear that an integration build which executes C# Compiler can fail with over a thousand different error messages. The number of distinctive defects, however, is harder to estimate because there is no simple mapping. On the one hand, all the syntax errors can be modelled with a single generic "syntax error" defect. On the other hand, a single error can correspond to multiple defects. For example "type not found" error can be caused by adding a reference to a missing type, deleting the original type definition or even modifying project configuration to omit a relevant file.

Based on the interviews with subject matter experts the initial number of distinctive defects that has to be supported by the diagnosis agent was established at around one hundred. This value may change in time if new tools or verification steps are included in the integration builds. For this reason, it is important for the model to scale well to match this demand.

5.2 Diagnosis procedure

With the requirements in mind several possible solutions were considered including rule-based systems and regression trees. The former were rejected because of the challenges in creating a rule-base machine learning systems, while the latter does not incorporate existing expert knowledge. One solution that satisfies all the requirements are the Bayesian trou-

bleshooters which extend Bayesian networks described in the chapter 3.

With explicit probabilistic modelling it is possible to design a model to leverage both data set and available expert knowledge. Additionally, the structure of the network defines clear paths of reasoning that can be followed to get insights about how the results were formulated. Finally, if designed correctly the network can have clear extensibility points which set the direction of future development.

Although the proposed design uses a Bayesian troubleshooter at its core there are other important steps that have to be performed before the network can be constructed. These steps as well as the steps directly related to the Bayesian network are described in the remaining part of this section.

5.2.1 Create a build graph from logs and build trace

In the first step the build graph is built from the information captured in the build trace file. As it was descried in the section 4.1.3 at page 47 the trace file contains the data about all the Operating System processes that were created in the context of the integration build and the file access operations they performed. Although this information is independent from the solution used to coordinate build tasks, it is convenient to describe it in terms of *GNU Make* or simply *Make*, which is one of the most popular utility used in this space [Mec04]. It is important to note, however, that the diagnosis model is not necessarily tied to this particular piece of software, and has applications to other systems used in the industry [HB10; Bai+07].

With *Make* the build specification is described in terms of rules in a dedicated language and generally saved in a file named *Makefile*. A rule consists of three parts: the *target*, *prerequisites*, and the *command* to perform. A *target* is a file or a collection of related files which will be created after successful execution of the command. It is also possible to define targets which do not produce any files and represent more abstract operations like deploying build artifacts to the staging environment. The *commands* correspond to normal shell commands or scripts. They are executed only when all the *prerequisites* expressed in terms of dependent targets are satisfied.

With this terminology it is possible to define a *build graph* as a di-

rected acyclic graph where vertices represent the build target and edges connecting targets with their prerequisites. The edges are pointing from the prerequisites to targets along with the actual flow of the build process. Each vertex holds additional information about the commands that were executed to build the target.

The most important piece of information is whether the target succeeded or not. Normally, it can be inferred from the status code returned by the command used to build it. In *Make* a status of zero indicates that the command succeeded, whereas a status of nonzero indicates some kind of failure. Some programs use the return status code to indicate something more meaningful, but when the build graph is built this information is converted to a binary flag representing success or failure.

By looking at the commands that were executed for a given target it is possible to correlate this data with the information saved in the build trace file to get a more detailed picture of the execution. In particular, each target can be mapped to the operating system process hierarchy created for its commands. Then, by looking at the file system access operation a set of files used to build the target and a set of files produced by the target can be found. These sets are crucial for the diagnosis and are attached to the vertices.

With the data described previously the build graph is complete in a sense that it can be used in the subsequent diagnosis steps. However, there are some optional refinements that can improve the performance of the system.

Refinement 1: Splitting targets

In large scale software systems the costs of maintaining single *Makefile* file can be prohibitively high. One way to solve this problem is to divide the system into a set of components, store them in distinctive directories and create separate *Makefile* file for each of them. Then, the entire system is built using a top-level *Makefile* that invokes the *Makefile* for each component in the proper order. This approach is called *recursive make* because the top-level *Makefile* invokes *Make* recursively [Mec04]. It is a common technique for handling componentized systems.

With *recursive make* targets in the top-level *Makefile* represent not a single but a whole hierarchy of targets. This complicates the process of

constructing a build graph. One simple option is to focus just on the top-level targets and not traverse the hierarchy. A more advanced option is to do a pre-order depth-first traversal of the hierarchy and extract child-targets to the top level with preserving all their dependencies and the dependencies of their ancestors.

Refinement 2: Latent errors

In order for the *Make* to know if a command was successful or not it must return a meaningful status code. Although, most of the tools used in this space satisfy this requirement, there is a risk that some commands will return status zero regardless of their actual outcome. This can happen in a situation when the script is custom made by a developer who does not fully understand how the build process works or simply when there is a bug in the script. This can lead to non-deterministic failures which are not surfaced in the build summary.

One optional extension of the build graph creation step is to set the status of each target not only based on the status code but also based on the content of the log files. A target can be considered as failed if the status code is nonzero or if there are some error messages in the logs. This extension increases the total cost because it requires a linear scan of all the log files, but can increase the accuracy of the information in the build graph.

Refinement 3: Latent dependencies

After build graph is filled with data from the build trace file it is possible to decorate it with latent dependencies that have not been listed explicitly in the build specification. For example if a target A creates a file F which is later read by target B there should be an edge between targets A and B regardless it appeared in the build specification or not. This rule is applicable to all the files including intermediate files which are easier to miss than build artifacts.

In certain situations this step may lead to the violation of acyclic property of the graph. Although it is a sign of a problem in the build specification the graph algorithms used during diagnosis can handle this problem well, thus there is no need to artificially restore this property.

5.2.2 Find the set of leading-failed build targets

When the build graph is complete, it is used to find the set of *leading-failed targets*, which are defined as failed targets whose all upstream dependencies succeeded. The name comes from the fact that this procedure divides the build graph into two subgraphs. First, contains all the targets which succeeded, whereas the second contains failed targets and the targets which succeeded despite some of their dependencies have failed. The sources of the second subgraphs are special in a way that they are "leading" all the failed targets.

The leading-failed targets have an interesting property that failure cannot be explained from the structure of the build graph by a failure of any other target. This property if frequently exploited by human domain experts, who start diagnosis by exploring errors of the failed targets with no failed upstream dependencies, and move down the build graph only when they seek for extra information to support their hypothesis regarding the defect. This step was added to the diagnosis model to emulate this behavior of a human expert.

When the diagnosis is focused on the leading-failed targets there is a risk that a genuine defect will stay unseen just because all its symptoms appeared in a downstream target. This is a valid concern, but typically human experts ignore it because once all the clearly visible defects are removed from the system the next integration build will progress pass the previous point of the failure and expose previously skipped defects. This is an iterative process where each iteration decreases the size of a set of failed targets by removing defects visible in the leading-failed targets until this size goes down to zero, at which point in time the whole integration build is successful.

An example of a build graph with some failed targets was presented in the Figure 5.1. The graph consists of nine nodes, where five of them have failed. The set of leading-failed targets includes only 3 and 4 because their only upstream dependency is target 1 which succeeded. Targets 6 and 8 do not belong there because their immediate dependencies have failed. Whereas, the target 9 was skipped because it is transitively dependent on the target 4.

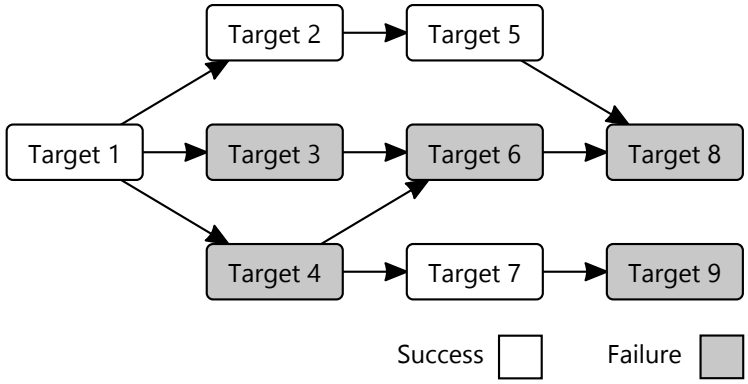


Figure 5.1 Example of a build graph with failed targets

5.2.3 Extract the information regarding errors from log files

For every leading-failed target there are most likely some errors in the execution log thread. They carry a lot of information about the nature of the problem like the identifier of the tool, its error code and the name of the source file where the error was detected. This information has to be extracted and made available in the diagnosis procedure.

Similarly to the leading-failed targets the first error in the thread is typically sufficient to identify the defect, while the following errors can be used to increase the confidence of a given hypothesis. Based on this observation the diagnosis model was designed to focus primary on the first error.

The concept of errors in Continuous Integration system is slightly different from how this term is commonly used. The error is any condition that makes the integration build fail. In particular, a warning returned by a tool used in the process can also be considered as an error if that is the policy of the project and its builds. Therefore, warnings should also be taken into consideration when scanning the log thread if a given project requires it.

Some of the tools can make this task easier by exposing special configuration or command-line options. For example *Microsoft Visual C# Compiler* supports an option `warnaserror`, which allows the user to adjust how the compiler reports warnings with the granularity down to a

level of a specific warning code. Two variants of this option together with short description are presented in the Listing 5.1. Such options should be preferred in the integration build specifications over any other solutions.

```
>>csc.exe /help
[...]
                      - ERRORS AND WARNINGS -
/warnaserror[+|-]             Report all warnings as errors
/warnaserror[+|-]:<warn list> Report specific warnings as errors
[...]
```

Listing 5.1 Selected Microsoft Visual C# Compiler warning options

5.2.4 Reduce the set of leading-failed build targets

Each item in the set of leading-failed build targets can potentially be caused by a distinctive defect in the code base and is inspected thoroughly. Therefore, whenever it is possible, the set should be reduced to decrease the problem size which consequently decreases the diagnosis time and the computational cost required to perform it.

There are certain scenarios where a single defect can lead to a large number of leading-failed targets in the build graph. One of the most common is when a target represents a core library with reusable functionality that is referenced by many projects and consequently many build targets depend directly on it. A change to the library itself which alters its public interface can compile correctly in isolation but break the downstream targets when included in an integration build. Such problem is easy to detect because all the leading-failed targets will be located in the same part of the build graph.

Another more complicated scenario is when a single configuration file is used as an input by many targets which can be scattered across the guild graph. In this case the locality cannot be used as an indicator and it is necessary to observe file access patterns or inspect the log messages to detect similarity in the data. The following sections describe different strategies that may be used to solve this problem.

Duplicate-based reduction

One of the most effective way to reduce the set of leading-failed targets is to search the corresponding error messages for duplicates. Two targets which failed with the exactly same error messages are expected to be related to the same defect regardless of their place in the build graph.

A good example of when this strategy works is when a rarely used function is deleted from a core library and some downstream project fail with a compiler error complaining that a specific symbol was not recognized. For every target both the symbol name and the surrounding error message will match what can be easily detected as a duplicate.

There are certain error types where the strict comparison does not work well. The tool that reports the error may include date and time or other contextual information specific to the target that was built. This problem is hard to solve in a unified way because it is not clear which parts of the error message can be ignored. Fortunately in real world these errors appear rarely and can be handled in the case by case basis.

Topological reduction

Another strategy does not need to inspect error messages but relies on the information present in the build graph. It is based on an assumption that leading-failed targets which have the same upstream and downstream dependencies are similar to each other and presumably are related to the same defect. The higher number of shared dependencies the stronger the similarity relation, but in practice it is sufficient to require at least one dependency for every direction.

Example of build graph where topological reduction is applicable is presented in the Figure 5.2. There are eight targets in total and five of them are marked as failures. The initial set of leading-failed targets consists of targets 2, 3, 4, and 9. The first three depend on target 1 and act as dependencies for target 6. Because they all failed there is strong indication that there is a defect which was introduced to the source code of target 1 and any target 2, 3, or 4 can be used as a starting point for the diagnosis to find it.

Such patterns are very common in projects which are localized into many different languages. Each localization task corresponds to a single

target and due to the strong similarity they all appear in the same place of the build graph with the same dependencies.

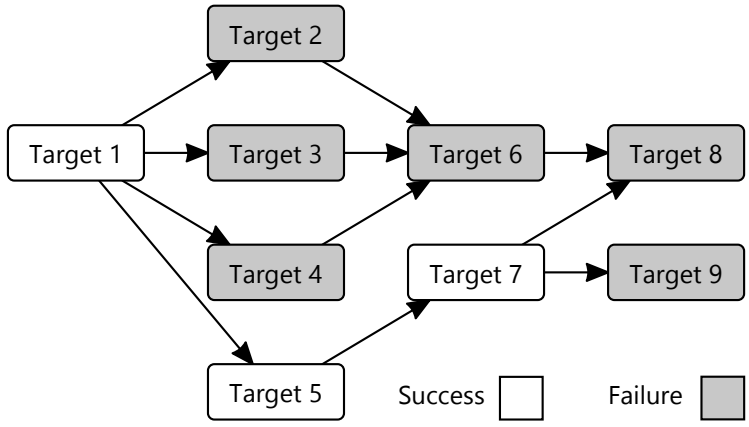


Figure 5.2 Build graph where topological reduction is applicable

Policy-based reduction

Previous strategies describe how the set of leading-failed targets can be reduced in an automatic way to ignore certain failures which are expected to bring little value to the overall diagnosis result. However, sometimes it may be necessary to deliberately ignore certain known failures to avoid alerting the development team about them over and over again. This can be solved with a simple policy-based reduction. A policy may be set to always remove certain target from the set.

It is worth noticing that a policy-based reduction should be used carefully because ignoring failures can reduce the benefits of practicing Continuous Integration. Ideally it should be enabled only for a short period of time during the integration of disruptive changes in the project when the failures are expected and diagnosed by a person who is responsible for the integration process.

5.2.5 Build Bayesian network describing the problem

Once the diagnosis task is scoped to the reduced set of leading-failed targets a Bayesian network describing the problem can be constructed. An instance of the network is built from a template represented as a plate model (see section 3.2.3). The structure of the model is presented in the Figure 5.3. The random variables are divided into five layers by their type and functionality, and placed on four plates, three of which are organized in a hierarchical structure while one is cutting through.

Target

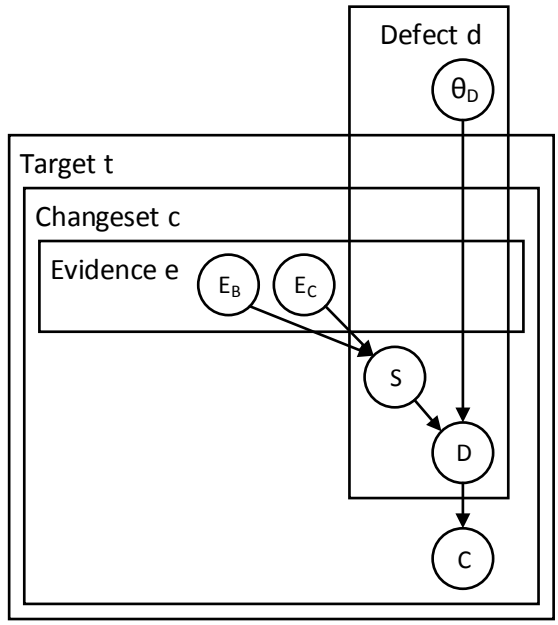
The outermost plate represents the leading-failed target and defines boundary of a subgraph with nodes which are related to a particular target. The only edge that is crossing its border connects defect with its hyperparameter. A probabilistic interpretation is that all targets are conditionally independent given the values of the hyperparameters.

This of course is a simplification because in reality all the targets are related, but it makes the network more manageable in terms of the design and interpretation of the results. When a knowledge engineer defines which symptoms are relevant for a given defect he can focus on a single failure without trying to find how it might be related to the other coincidental failures. It also simplifies the work of a build engineer who might be looking for an explanation why a particular target failed because it limits the set of random variables he might need to inspect.

This plate does not have any nodes on its own because everything is considered in the context of the target and the set of candidate changesets which could have introduced the defect. In an alternative design one could introduce target level evidence random variables for observations that are not related to any changesets. However, nested plates were selected because they can encode the same probability distributions while preserving unified framework for working with evidence.

Changeset

Inside the target there is a *Changeset* plate which represents a set of candidate changesets. The most important at this level is the culprit C binary



- $\theta_D(d)$ - prior parameter which controls the probability distribution of the d defect type.
- $E_B(t, c, e)$, $E_B(t, c, e)$ - true if the evidence of type e appeared for the changeset c and the build target t.
- $S(t, c, d)$ - true if symptoms suggest that a changeset c could have introduced a defect of type d to the component built by target t.
- $D(t, c, d)$ - true if the changeset c introduced a defect of type d to the build target t.
- $C(t, c)$ - true if the changeset c introduced a defect to the build target t.

Figure 5.3 Plate model of the Bayesian network used for diagnosis

random variable which indicates if the changeset introduced a defect to the build target. It cannot be observed directly, but we can use probabilistic inference to find its value based on the observations of its predecessors.

For its local probability model we selected the *Noisy-Or* model described in section 3.2.2 at page 37. It assumes that all the defects use different causal mechanism, and each of them should flag a changeset to be considered as a culprit. If we take into account that the defects are modelled as N_D binary random variables we can define the formula for the conditional probability distribution function as:

$$P(C(t,c) = 0 | D(t,c,1), ..., D(t,c,N_D)) = (1 - \lambda_C(c)) \prod_{i:D(t,c,i)=1}^{N_D} (1 - \lambda_D(i)) \tag{5.1}$$

The noise parameters λ_D can be used to model a situation in which the defect is present but it does not surface during the build. Since the process of making observations is fully automated the chance of skipping a defect is very low, thus it is practical to set all of these parameters to value close or equal to one.

The leak parameter λ_C, on the other hand has a more important role in the diagnosis. It models the influence of defects that are not explicitly included in the network. Its value corresponds to the probability of a changeset being a culprit despite the fact that there are no visible symptoms. During the development phase it is acceptable to set it to a higher value, but as more defect types are supported by the system it should go down to e.g. 0.001.

Sometimes builds fails when there were no defects introduced to the project. It might happen when some external systems are down, like the artifact store server where the build artifacts are uploaded towards the end of the build. If the integrator cannot reach the server it will mark the build as failed to indicate that no artifacts are available. This class of problems are modelled by adding a *system pseudo-changeset* with a special null identifier.

Evidence

The innermost plate contains evidence random variables which represent observable phenomena. Their values can always be set in the network, however, it might not always be practical because there is a cost associated with each observation. The cost is mainly expressed in the time it takes to execute all the actions required to make the observation.

The system supports two classes of evidence with a corresponding nodes in the template: E_B and E_C. Fist, represents *basic evidence* which is inexpensive to observe, thus, it can be fully set in each network used for the diagnosis task. As a consequence these variables do not need to have their prior probabilities specified.

Good examples of such evidence are assertions which can be checked by inspecting local resources such as logs, build trace, or the basic information about the changesets. The following points present interpretation of some evidence random variables instantiated to diagnose a failure in a *"app"* target which builds a *MSBuild* project *"app.csproj"* using *Microsoft Visual C# Compiler*. The project fails because the changeset *"4312"* adds a reference to a missing source file.

- File *"app.csproj"* is an input of a target *"app"*.
- Changeset *"4312"* modified file *"app.csproj"*.
- Changeset *"4312"* modified project file *"app.csproj"* of the leading-failed target *"app"*.
- Target *"app"* failed with the *Microsoft Visual C# Compiler* error *"CS1504 (NoSourceFile)"* for the file *"foo.cs"*.

Second class contains *complex evidence* which is expensive to observe. In contrast to the basic evidence these random variables do not need to be set and in some situations they can be completely ignored if the result inferred from the basic evidence is sufficient to find the defect.

Consequently it is necessary to specify the prior probability for these variables. They all make use of Bernoulli distribution with parameters set manually by the knowledge engineer who is defining the diagnosis procedure for a particular defect type. If some complex evidence is reused in

multiple symptoms it might be necessary to synthesize opinions of different authors to reach an agreement regarding the values of the parameters.

Typically this evidence is expensive because in order make the observation it is necessary to connect to external systems, fetch more information and process it. To illustrate that we will continue example of the *,,app"* target failure caused by adding a reference to a missing source file to the project.

- Changeset *"4312"* modified project file of the target *"app.csproj"* by adding a reference to the source file *"foo.cs"*.

In order to observe such evidence it is necessary to:

1. Connect to the Version Control Repository and download two revisions of the *"app.csproj"* file, one for the changeset *"4312"* and one for its immediate predecessor in the revision graph.
2. Parse both project files.
3. Compare the sets of referenced source file and check if the *"foo.cs"* exists in one revision but not in the other.

The first step is the most expensive one as it makes a network call. This cost may be justified in a situation when there are many changesets that touched the project file and the basic evidence is insufficient to identify the culprit. By inspecting multiple revisions we can find the point in time when the reference to the file *"foo.cs"* was added and it will be strong indicator of the defect.

Defect

Evidence alone does not indicate the type of defects that exist in the project. They need to be grouped together to form higher level constructs modelled with symptom random variables $S(t, c, d)$ in the *Defect* plate. These variables depend on both types of evidence.

The plate modelling framework has some limitations in the expressiveness when it comes to the irregular dependencies. According to the canonical interpretation of the template presented in the Figure 5.3 each symptom variable depends on all available evidence variables. However, it does

not completely match the design. Symptom variables are defined in the context of specific defect types and they depend on a selected subset of evidence variables. This could be modeled by including an extra deterministic variables to the model, which would mask the evidence, but it is debatable if such change improves the interpretability of the template.

For each defect type there exists precisely one symptom in the model. This implies that if the knowledge engineer finds that there are multiple symptoms which can all indicate the same general defect it is necessary to split it into several specialized defects to satisfy the constraints of the model.

The symptom is assumed to be appear only when all of relevant evidence is present. Therefore, it is modelled with a *Noisy-And* construct similar to the *Noisy-Or* described previously in the context of the culprit variables. The formula for the conditional probability distribution can be defined in the following way:

$$\begin{aligned} &P(S(t,c,d)=1|E_B(t,c,1),...,E_B(t,c,N_{EB}),\\ &E_C(t,c,1),...,E_C(t,c,N_{EC})) =\\ &(1-\lambda_S(d)) \prod_{i:E_B(t,c,i)=1}^{N_{EB}} \lambda_{EB}(i) \prod_{j:E_C(t,c,j)=1}^{N_{EC}} \lambda_{EC}(j) \end{aligned} \tag{5.2}$$

The noise parameters λ_{EB} and λ_{EC} can be used to model evidence whose presence does not certainly imply the presence of the symptom. Whereas the leak parameter λ_S limits the maximum probability for the symptom variable. All the parameters should be set manually by the knowledge engineer who is working on the specific defect type.

Presence of a symptom indicates only that the circumstances are right for the defect to exist, but it does not imply it. The uncertainty regarding the defect is modelled with a $D(t,c,d)$ random variable, which uses a rule-based representation for its local probability model [Bou+96]. Its distribution is conditioned on the value of the symptom random variable and it might either be zero or a Bernoulli's distribution with prior parameter $\theta_D(d)$. The formula for the defect distribution can be defined as:

$$P(D(t,c,d)=1|S(t,c,d),\theta_D(d)) = S(t,c,d)\theta_D(d) \tag{5.3}$$

The parameter $\theta_D(d)$ is itself a random variable known as a prior parameter because if it was removed from the network it would be necessary to define a regular parameter of the prior distribution. This variable is special in a way that it is the only variable that appears outside of the hierarchy surrounded by the *Target* plate. This model structure implies that all the failed targets are conditionally independent given $\theta_D(d)$ and its value is shared not only between failed targets within a single build, but also across many builds. This is a key aspect for the Bayesian parameter estimation which will be discussed in section 5.3.2. The distribution is defined as follows:

$$P(\theta_D(d)) = \gamma \theta_D^{\alpha_{1d}-1}(1-\theta_D)^{\alpha_{0d}-1} \tag{5.4}$$

The constant γ is a normalizing constant for the Beta distribution introduced in section 3.4, whereas α_{1d} and α_{0d} are two hyperparameters, which are estimated from the samples for each defect type separately.

5.2.6 Observe basic evidence

After the Bayesian network is built from the template we can start adding information about the failure by observing the basic evidence. It has priority over its complex counterpart because it is more cost effective.

For every evidence type there is a predicate function that is responsible for performing all the actions required to determine whether the variable should be set to true or false. They are expressed with an imperative code and executed by the diagnosis agent.

5.2.7 Execute inference procedure

The inference boils down to execution of a set of probabilistic queries, one per each target and changeset combination. The query has a form of a posterior probability and it finds likelihood that a changeset introduced a defect given the observed basic evidence $\boldsymbol{E}_B$ and the vector of prior parameters $\boldsymbol{\theta}_D$. This task can be formally defined as:

$$P(C(t,c) = 1 | \boldsymbol{E}_B, \boldsymbol{\theta}_D) \tag{5.5}$$

5.2.8 Observe complex evidence

In some situations the basic evidence can be insufficient to find a strong changeset candidate. Then, it is necessary to start observing complex evidence. Due to its cost the processing is not batched, but is happens sequentially one observation at a time.

There can be many strategies for selecting the next complex evidence to observe. One practical heuristic method is to rank the random variables by the number of dependent symptoms. The higher this number is the greater impact the variable might have on the changeset posterior probability. If two complex evidence variables are used in the same number of symptoms then they can be further ranked by the number of basic evidence variables that participate in these symptoms. This strategy follows an assumption that more specialized defect types should have priority over general ones.

With the second class of evidence the probabilistic query introduced in Equation 5.5 can be updated by including more conditioning random variables:

$$P(C(t,c) = 1 | \boldsymbol{E}_B, \boldsymbol{E}_C, \boldsymbol{\theta}_D) \tag{5.6}$$

The process of observing more evidence and updating the probabilities should continue until there is a changeset candidate with a high posterior probability that is clearly standing out from the rest, or the execution time limit is reached, or there is no more evidence to observed.

5.2.9 Collect results

When the termination condition is reached and the network is complete in terms of observed evidence the posterior probabilities of the *Culprit* random variables are calculated once more for each combination of leading-failed target and changeset, and returned as the diagnosis result. This format is perfect for a downstream automation that might act upon the output. It the result needs also to be presented to users, in order to make it more appealing it can be grouped by the target and ordered descending by the probability. That way the most relevant or actionable information floats to the top.

Additionally, the whole network or its parts can be persisted to give a better justification in case somebody wants to understand why a particular

changeset was blamed to have introduced a defect. This is important for the knowledge engineers when they are adding support for new defect types.

5.3 Training procedure

The Bayesian network presented in the previous section supports two basic types of training. Subject matter experts can set the values of hyperparameters which are combined with the statistics calculated from the data to form the prior parameters for defect distributions. One advantage of this procedure is that the manual steps appear very early and once they are done the model can be automatically retrained when more samples are added to the training.

We recall that defects are modelled with Bernoulli's distributions with prior parameters coming from a Beta distributions. This implies that the posterior probability of observing new defect $D[M+1]$ conditioned on the presence of the right symptoms $S[M+1]$ and the previous M observations $D[1], ..., D[M]$ can be described with Equation 5.7.

To make the formula more succinct the index variables for target, changeset, and defect type were omitted. Therefore, it should be interpreted with an assumption that all the random variables mentioned in Equation 5.7 are for the same defect type observed in the context of M independent failures.

$$P(D[M+1]|S[M+1], D[1], ..., D[M]) = \frac{\alpha_1 + M[D=1, S=1]}{\alpha_0 + \alpha_1 + M[S=1]} \quad (5.7)$$

This equation combines the hyperparameters of the Beta distribution α_0 and α_1 with the counts of certain events recorded in the training set. Count $M[D=1, S=1]$ represents the number of situations when the defect was observed in the presence of related symptoms, whereas $M[S=1]$ is the total number of situations when the symptoms were observed.

It is interesting to examine the effect of the hyperparameters over the size of the training set. Initially, when there are very few samples their values dominate the probability. However, as more samples are observed and the respective counts grow this effect diminishes. By selecting the right initial values one can control its strength with relation to the data.

5.3.1 Expert elicitation

The existing expert knowledge is captured partially in the network structure and partially in the values of hyperparameters which describe distributions of different defect types. Their values should be established as a consensus from opinions of build engineers in the process known as expert elicitation.

The task of quantifying uncertainty can be very daunting especially without solid understanding probability theory. However, with Beta distribution it is possible to formulate the task differently so that it is approachable by practically everyone. Hyperparameters α_0 and α_1 appear in Equation 5.7 next to the count statistics. For this reason they are often referred to as *pseudo-count* [KF09, p. 740]. We can think of them as the number of times we have seen particular events in our prior experience before data set collection process started.

With this observation it is possible to prepare a survey for the people involved in the process of build break management and ask them two simple questions for each defect type:

- How often do you see builds failing with S symptoms?
  (Never, Rarely, Occasionally, Regularly, Always)

- When you observe S symptoms, how often do you find that the failure is caused by D defect type?
  (Never, Rarely, Occasionally, Regularly, Always, Not Applicable)

Their answers can be translated directly to the values of the hyperparameters. The first answer corresponds to sum of α_0 and α_1, whereas the second defines the ratio between them. For example if we observe answers (*Occasionally, Always*) and select mapping (20, 0.95) we can calculate $\alpha_1 = 20 \cdot 0.95 = 19$ and $\alpha_0 = 20 \cdot (1 - 0.95) = 1$, and conclude that the prior parameter of the defect type has distribution $Beta(19, 1)$.

There is some freedom in selecting the mapping used to translate answers to numbers, but there are several constraints it needs to satisfy. Answer *Always* to the first question must not have too high value in relation to the number of samples in the training set so that it does not dominate the probability even if there is enough information in the data

to learn the distribution. Answers *Never* and *Always* to the second question must not translate to 0 and 1 values because that could give users false confidence when in fact there are hardly any examples of defects that are truly certain.

Finally, answer *Never* to the first question should be taken into special consideration. If the defect has never been observed, then why the diagnosis agent should even support it? To answer this question it is necessary to think about the skill-set of build engineers. They are very knowledgeable about the technologies used in the project they are directly involved in, but they might not know the tools used in other projects. In a complex system there are typically certain people who own specific areas. By selecting *Never* as the answer, what respondents are really doing is indicating that this defect type does not belong to their area of expertise, thus, such answers should be ignored altogether. Only if this pattern repeats for all build engineers should the defect type be reevaluated and possibly removed from the system.

5.3.2 Offline training

In order for the diagnosis agent to have good performance the moment it is deployed it is necessary to collect a training set use it to learn from it in an offline strategy. The framework of Bayesian network supports several types of learning tasks, but in the context of the diagnosis agent we will focus on learning as a parameter estimation problem.

Equation 5.7 provides a good description of how the estimation should be performed. It involves calculating count statistics for the events $M[D = 1, S = 1]$ and $M[S = 1]$. While the latter is straightforward because we can directly observe if a given symptom is present in the sample, the former brings significant challenges because it requires us to know when a certain defect type was added to the project. As explained in the chapter 4 this information is very hard to obtain, especially in an automated fashion.

In order to solve this problem we may train the model under additional assumption that the defect existed if the observed symptoms supported it and the related changeset was in fact labeled as the real culprit. Although not always true, this is a very practical simplification because it lets us work with less expensive, automatically labelled samples and with sufficiently large training set its impact becomes negligible. With this as-

sumption the right side of Equation 5.7 can be replace with the following fraction:

$$\frac{\alpha_1 + M[C = 1, S = 1]}{\alpha_0 + \alpha_1 + M[S = 1]} \tag{5.8}$$

Count $M[C = 1, S = 1]$ represents the number of events when the symptoms for a given defect type were present and the changeset was identified as the real culprit. The main advantage of this version is that all the events which appear in the formula can be observed in the training set. The modified learning procedure rewards all the defect types which in the light of the available evidence could have existed.

We recall from the section 5.2.5 that the proposed model support two classes of evidence: simple and complex. The latter brings another challenge to the training task because it might not always be practical to fully observe it. This problem is less significant than during the real diagnosis because the timing constrains are much weaker, meaning that we can tolerate expensive observations in the training phase, but it still exists.

Consequently, the statistics which appear in Equation 5.8 cannot be simply set by counting events because there is some uncertainty they need to account for. We can solve this problem with probabilistic modelling by replacing the exact counts with their expected values:

$$\bar{M}[C = 1, S = 1] = \mathbb{E}[M[C = 1, S = 1]] \tag{5.9}$$

$$\bar{M}[S = 1] = \mathbb{E}[M[S = 1]] \tag{5.10}$$

Distributed training

The training task can be executed on a single machine, but it can also be distributed across many machines. It is possible because the addition operation, which is used to calculate the statistics appearing in Equation 5.8, is associative meaning that the samples can be processed in an arbitrary order.

This task fits perfectly to the *MapReduce* programming paradigm supported by several distributed computation systems including a very popular option - *Apache Hadoop* [Whi09]. For the problem to be executed on

a *Hadoop* cluster it has to be decomposed into two tasks. The first is the *Map* task, which takes a set of data and converts it into a collection of key-value pairs. Second, is the *Reduce* task, which takes output from the *Map* task and for each key it combines the pairs into smaller set of pairs.

In order to use *MapReduce* paradigm in the training phase the *Map* map tasks should, for each sample, build the Bayesian network, observe the random variables including the *Culprit* variables and emit the count values keyed by the index of the defect type. The subsequent *Reduce* task can preserve the key and sum up all the partial results to form the total counts returned as a result of the whole operation.

5.3.3 Online training

The agent is designed to diagnose the failed integration builds as they happen. This stream of tasks can also be turned into a stream of samples and used to update the model parameters with the new information. That way there is no need to persist all the samples in the training set and the model gets extended to adapt to the changing environment.

In order to enable online training it is necessary to close the feedback loop. The most significant challenge is related to the *Culprit* random variable, which cannot be observed at the diagnosis time and becomes available much later, after the defect gets removed from the project. To work around this gap in time the Bayesian network used in the diagnosis can be saved to the temporary store and loaded once there is sufficient information to update the parameters. This process can be implemented with the following steps:

1. perform diagnosis,
2. save the Bayesian network to a file,
3. wait for the first successful build,
4. check the backward fix scenario for ambiguity,
5. load the network from a file,
6. observe the *Culprit* variable,
7. update the event counts for all the defect types.

This scenario is possible only if the project uses the backward-fix strategy for managing broken builds (see section 4.2). In all the other cases, the steps three and four can be replaced with one manual step where build engineer points out the culprit.

5.4 Practical considerations

As described in previous sections diagnosis model returns a list of possible culprits ordered by the probability rank. How this output is used next depends on the environment in which the model operates. There are three main scenarios which can be used in practice:

Decision support system - diagnosis results help build engineers find culprits faster but no change gets reverted unless human approves it.

Full automation - diagnosis results are used by the automation to revert identified culprits.

Mixed - diagnosis results are used by the automation to revert culprits with high probability, but the system falls-back to build engineers if none of the culprits have sufficiently high probability.

In the current Continuous Integration systems build engineers are responsible for managing broken builds. The most natural way of enabling diagnosis model in such environments is to use it as a decision support system [WDN07]. This scenarios does not change existing processes. Just as before human experts are responsible for finding defects, but by inspecting potential candidates in the descending order of the probability, they should be able to accomplish this task faster. Only after they confirm that the change introduced a defect it gets reverted form the repository. This guarantees that the false positive rate will stay at the same level at the cost of having human involved in the process.

Full automation mode on the other hand lowers the costs of maintaining the CI system by delegating all the tasks to autonomous agents. In this scenario agent responds to each broken integration build by either reverting the top candidate or restarting the build if symptoms indicate that this is an intermittent system issue. A disadvantage of this solution is

that it can have higher false positive rate and does not handle new defects well.

When build engineers come across errors they have not seen before they can always reach out to the development team for help. Similar approach is adapted in the mixed mode where the diagnosis agent is allowed to act upon a failed build provided that the culprit was identified with high probability. If symptoms do not clearly identify what the problem is, or there are many candidates with equally high ranks, the agent can escalate to build engineers and aid them with diagnosis results.

In this scenario system switches from automation to decision support mode based on the value of the probability. The rate at which it happens can be controlled by defining minimal threshold at which the agent can still revert culprits. This threshold can take values in the range $(0, 1)$ and can be selected according to the unique project characteristics. It can be lower for projects where developer can tolerate higher false positive rates.

Mixed mode combines advantages of previous solutions because trivial defects get fixed automatically while hard problems are brought to the attention of human experts. Project owners can control this process by setting value of the probability threshold for automatic revert. As a result the false positive rate can be kept low while the costs of maintaining the system go down because there are fewer interventions form build engineers.

5.5 Summary

The diagnosis model presented in this chapter satisfies all the requirements gathered in the planning phase. It is a Machine Learning system because the more samples are added to the training set the more accurate estimates of model parameters are becoming and consequently expected performance improves as well.

Not only does the model learn from the training set, but it also incorporates the existing expert knowledge by capturing it in the Bayesian network structure and values of hyperparameters, which control the prior probability distributions for the supported defect types. It makes the model more flexible in a sense that it can diagnose rare defects provided that they are well known and understood by the experts. Additionally, by participating in the shaping of the model people involved in the process

of build break management can understand how it works and trust the results it gives.

To make the results interpretable the random variables used in the Bayesian network represent concepts commonly used by the build engineers such as evidence, symptoms and defects. They were also organized into five layers by their function. That design makes it approachable to users who can examine and verify the correctness of the results before they act upon them.

Finally, the structure of the Bayesian network was designed to scale well with the number of distinctive defect types. If an integration build fails with a new, unknown defect the network can be extended to support it by defining new evidence, combining it to form symptoms and specifying prior probability for the defect type. These steps are the same regardless the complexity of the problem, thus the network is always modified in the additive fashion.

In order to achieve this property the network structure encodes some implicit conditional independences assumptions. In particular, all the defects are conditionally independent given their prior parameters. Consequently, the results of the probabilistic queries for the *Culprit* random variables can be skewed towards positive value. Nevertheless, as it will be demonstrated in the following chapter, this affects all the changesets in the similar way and does not change their relations significantly when they are ordered by the probability rank.

The diagnosis model was design to solve a very specific problem of finding defects in the Continuous Integration systems. Although it has potential to work in other domains, there were no attempts to test this hypothesis.

6 Study of the effectiveness

Previous chapter described the diagnosis procedure. In this chapter we will use this design to train the model in different configurations and look at the quality of the results it produces. We will follow grey-box testing approach where most of the time we will analyze output, but in order to explain certain phenomena we will refer to the implementation details of the underlying model.

6.1 Data set used in the research

The data set used in this research was collected in an industrial Continuous Integration system at *Microsoft Corporation* in the period from Nov 2012 to Feb 2014. During this time the system was used by many teams working on thousands of different projects and using different technologies, but sharing the same build definition and execution technology. Both the collection process and the format of samples were described in details in chapter 4 at page 45.

There were two main types of integration builds running in the system. First type consisted of short builds focused on specific self-contained software components or projects. Typically they were verifying up dozen new changeset and their outputs were mainly consumed by the team members working directly with the source code. Those people were also responsible for managing the process and making sure that all defects get fixed in a timely fashion.

Builds of the second type, on the other hand, took longer to execute because they were integrating changes across many projects which are grouped together to form complete products. Because the impact of defects in this phase was much higher there was a dedicated team of build engineers who were looking after this process.

Although the rate of failed integration builds was very low, the high number of executions happening every day compensated for it and made it

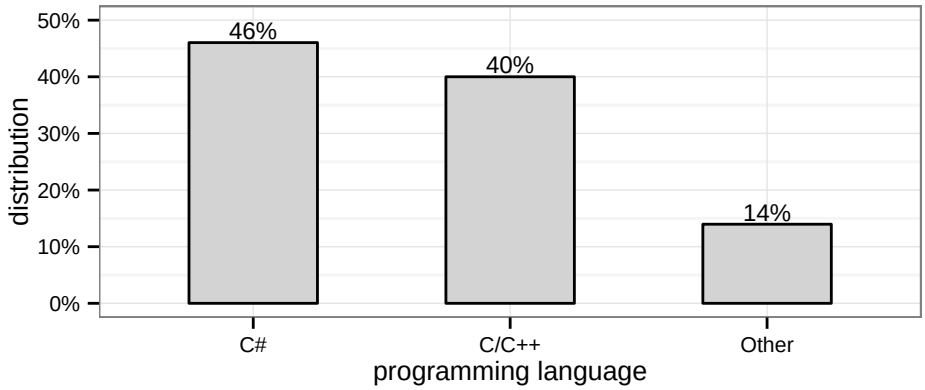


Figure 6.1 Programming languages distribution in the data set

possible to collect a data set of a size sufficient for the research. Moreover, thanks to the diversity in projects and types of integration builds, the samples represent a broad range of problems to diagnose.

6.1.1 Programming languages

Projects which used the common CI system were being written in different programming languages with the distribution presented in Figure 6.1. It is clear that the primary languages of choice were C# in the managed space (46%) and C/C++ for the native code (40%). The remaining part (14%) contained more specialized languages for writing automation scripts, and domain specific languages for creating installers and deployment packages.

Without formal analysis, if we assume that there is weak correlation between programming languages and the number of defects detected in the integration builds we can expect that a model capable of diagnosing failures for the top two languages would cover approximately 80% of all the issues.

6.1.2 Expert elicitation

Apart from the data set it is necessary to capture prior expert knowledge to build the model and sets all of its hyperparameters. This task was

performed according to the procedure described in section 5.3.1 at page 85. After the main defect types were identified a survey was prepared for people who are involved in the build break management.

In order to capture insight for defects appearing in both types of builds described previously the survey was handed to both a team of build engineers, who deal with issues in the CI system on a daily basis, and a team of developers who occasionally have to fix problems in the build they own. Since the survey was carried out in an industrial environment the response rate was high. Results were combined to form a single *Beta* probability distribution for each defect type.

6.2 Research approach

The data set used in the research and the diagnosis model have some unique characteristics which require special handling to reduce the estimation bias for the quality measures. This section presents main factors that were taken into consideration.

6.2.1 Prior expert knowledge

The model is designed to incorporate the prior expert knowledge. In order to measure its impact on the effectiveness the analysis will be performed in two phases. First, the model will be evaluated with *uninformative priors*, which are defined as assignments to the hyperparameters which maximize the information brought by the data [KF09].

For *Beta* distributions selected for the defect types this corresponds to setting both α_0 and α_1 to 1. With these values we can focus on the influence of the size of the training set without worrying that the results are shifted because some parameters were set manually. In the second phase we will look into what happens when we start to incorporate prior expert knowledge.

6.2.2 Complex evidence

Model supports two classes of evidence: basic and complex. The difference is that the latter can be expensive in terms of time it takes to observe the evidence and in practice the values of corresponding random variables

will not always be set. This design is important because it allows to set upper limits on the execution time, however, it bring some challenges in the analysis because it adds an extra parameter which can influence the quality of results.

During this research complex evidences were used several times for very specific defect types. For example they checked if a changeset removed definition of a constructor still in use which resulted in *C# Compiler* error *CS1729 (BadCtorArgCount)*. It is complex task because it requires at least one network call to Version Control System to download the previous version of the file and check if the constructor existed before.

Taking into account that complex evidences are used for very specific defect types, which have low frequency in the data set, and the challenges it brings to the analysis, a decision was made to treat them the same as basic evidences in a sense that they is always observed. This corresponds to a situation when there are no limits on the execution time. With this simplification the estimates of quality measures will be optimistically biased but this effect is expected to have low impact on the overall performance.

6.2.3 Noise parameters

The proposed network contains many *Noisy-OR* and *Noisy-And* local probability models whose noise and leaks parameters have to be set. They express the uncertainty about procedures used to observe random variables and relations between them. Description together with interpretation of each parameter was included in the section 5.2.5 at page 76. Here we present assignment which was used throughout the research to emphasize how the model behaves in different setups. In practice these parameters can take different values to fine-tune the system.

Because the diagnosis procedure is fully automated there is little uncertainty added in the execution steps. For example, if a compiler finds a problem, it always emits an error to the log where it will certainly be discovered and translated to evidence. Therefore, all the noise parameters were set to its neutral value which is at 1.

$$\lambda_D = \lambda_{EB} = \lambda_{EC} = 1 \tag{6.1}$$

Similar reasoning is applicable to the leak parameter for the symptom random variable because it is used mainly to combine evidences and not

add to the uncertainty. In contrast to the noise parameters its neutral value is at 0.

$$\lambda_S = 0 \tag{6.2}$$

The only parameter which is set to a significant value is the leak parameter for culprit random variables. It plays a key role in the network size analysis because it models the influence of defect that are not explicitly included in the network structure. Its value corresponds to the probability of a changeset being a culprit despite the fact that there are no visible symptoms. This is particularly important for very small networks which do not support many defect types.

The value of this parameter can be set once for each changeset, but in practice it is good to define different values for normal changes and special system pseudo changeset added to model system defects. The reason is that the absence of symptoms can indicate either that the issue is not supported or that the data sources used to populate evidence random variables were unavailable due to the system failure. It was observed that the latter scenario is more popular so its leak parameter was set accordingly to a value ten times higher than for the normal case.

$$\lambda_C(c) = \begin{cases} 0.01 & \text{if } c \text{ is } \textit{system pseudo changeset} \\ 0.001 & \text{otherwise} \end{cases} \tag{6.3}$$

6.2.4 Cross-Validation

The way data set was collected creates a challenge for estimating the test error associated with a particular model trained with it. A very popular approach is *K-Fold Cross-Validation*, where the available samples are randomly divided into k groups, or *folds*, of approximately equal size. The first fold is treated as a validation set and the model is trained on the remaining $k-1$ folds. The mean squared error is then computed and saved. This procedure is repeated k times, with each group treated once as a validation set. The final result is computed by averaging the partial results.

This approach in its original form would lead to estimates biased towards lower errors. It is because the data does not satisfy the indepen-

dence assumption made implicitly in the random division step. At a given point in time there can be several integration builds running for the same project, each using slightly different configurations. A single defect in the project can cause all of them to fail and make the samples highly dependent on each other.

If the builds were randomly divided into training and test sets then an easy way to design a diagnosis agent would be to mark changesets from the test set as culprits when they are known to have introduced a defect to some projects in the training set. Such model would appear to perform surprisingly well in cross-validation while in practice it would not be able to detect new defects at all.

In order to eliminate this problem in the experiments, the cross-validation was performed on a groups created not by random sampling, but by dividing them accordingly to their start time with a caveat that the cutoff point cannot split consecutive or concurrent failed builds of the same project.

The result of this step is presented in Figure 6.2. It is clear that the folds are uneven with an average around 10%. Fold number 8 is much bigger than its neighbors which indicates that there were many concurrent builds failing close to the cutoff point of this fold. The extra samples it received were taken from the fold number 9, which is much smaller.

This approach does not guarantee the estimates will be unbiased, but it significantly reduces the bias by making folds independent with regards to defect instances, what is required by the cross-validation procedure. They are not fully independent because they contain samples from the same projects, but these are the patterns we want the model to discover and exploit in the diagnosis.

6.3 Network size sensitivity analysis

One of the most interesting aspects of the model is how the diagnosis quality changes with the number of distinctive defect types random variables included in the Bayesian network. It corresponds directly to the requirement of scalability defined in section 5.1.4. The expectation is that the quality will improve as more defect types are supported by the system.

This section is divided into three parts. First we will look into a baseline

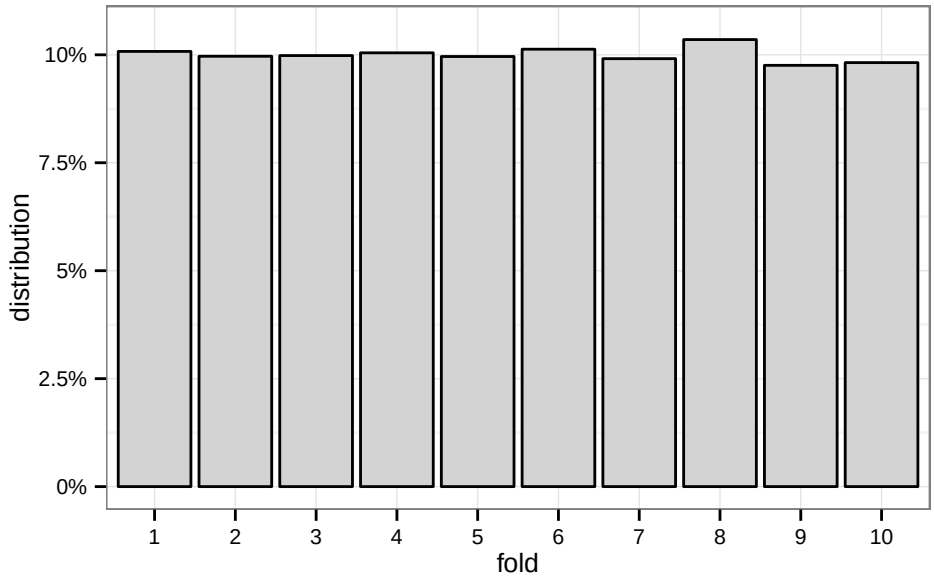


Figure 6.2 Example of data set division into uneven folds

model with very limited capabilities. Its simplicity will help us define the measures and introduce charts that will be used in subsequent sections to reason about behavior and efficiency of the model. With this solid background we will observe how the results change when we add support for general and then specific defect types.

6.3.1 Baseline analysis

We will start the analysis with the simplest issue, yet the one which can be seen in practice. It is a failed compilation due to a syntax error, which indicates that the source code does not adhere to the grammar rules of the programming language. Each compiler can have its own set of syntax errors but they are all similar in the sense of the causal mechanisms and scenarios in which they are created.

One example of such problem is the *CS1514, (LBraceExpected)* error reported by the *C# Compiler* with a message similar to the one presented in the following listing. The lack of "`{`" character in the source code violates the grammar rules and prevents the compiler from processing the

source file.

```
1 A.cs(7,30,7,30): error CS1514: { expected
```

Listing 6.1 Example of a *C# Compiler* error *CS1514*

A single error code can correspond to multiple defect types. Even as simple problem as syntax error can be introduced in several scenarios listed below.

- Developer made a syntax error when writing the code.
- Developer missed a syntax error after his or her change was automatically merged with other concurrent changes.
- Developer modified a macro construct in a header file included in the sources which, when expanded, created a syntax error.
- Project owner updated build configuration to include old erroneous file in the process.
- Project owner changed the version of the code-generation tool used during the build.

The first scenario is by far the most popular case among all the other examples. It is very similar to the second scenario which additionally explains the circumstances of the error. For diagnosis it is sufficient to find the changeset which introduced the defect, thus the first scenario was selected for the baseline.

A plate scheme for this model is presented in Figure 6.3. Dashed lines around evidence and defect indicate that instead of representing a whole set of random variables they are fully expanded plates. In this particular case they expand to one node each.

Evidence random variable $E_b(1)$ takes value true when the changeset c edited a file which was mentioned in the error message related to target t. In order to observe this variable it is necessary to prepare a procedure that will parse logs and load the information from the version control system about recent modifications. Hyperparameter $\theta_D(1)$, on the other hand, is

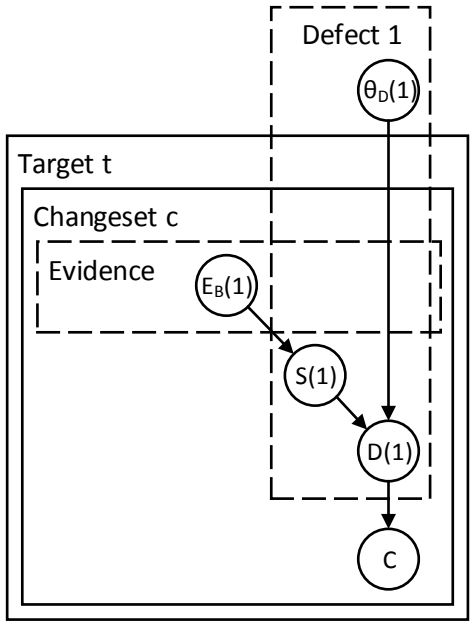


Figure 6.3 Plate schema for the baseline model

not related to any target and encodes the probability for the defect type alone.

Syntax error is an example of a problem within the project'a source code. However, integration builds might fail also due to system failures. For example *C# Compiler* might run out of available memory and fail with an error *CS0003 (NoMemory)* as presented in the listing below.

```
1 error CS0003: Out of memory
```

Listing 6.2 Example of a *C# Compiler* error *CS0003*

This can happen when there are multiple builds scheduled to run at the same physical machine at the same time. Such error has nothing to do with the changes in the project, thus the diagnosis agent should indicate it by selecting system pseudo changeset as the culprit.

System defects represent a unique class of problems and they deserve to be analyzed separately. Therefore, all charts presented in this chapter

are doubled to account for that. Although it is interesting to see if the model can handle both classes, system defects are rarely observed and their frequency goes down when the Continuous Integration system becomes more reliable. Taking that into account the primary focus will be on the project defects.

We will assume that the diagnosis agent works in the mixed mode, which was first introduced in section 5.4 at page 89. It assumes that there exists a certain probability threshold above which the agent automatically takes an action on behalf of build engineers or escalates the problem to human experts if none of the failure explanations have sufficiently high probability. Naturally selecting the right value of threshold is critical to achieve good performance in practice. To help us understand how the quality measures change with this value it will be used as the horizontal axis for almost all visualization we will see.

Outcome rates analysis

The first chart that we will introduce focuses on the impact of the diagnosis agent on the health of the project and the efficiency of the system. There exist five main scenarios which we will take into consideration:

Fixed (clean): Real culprit was identified and correctly reverted from the repository. After this action project state was valid again and subsequent build succeeded.

Fixed (collateral): Real culprit was identified and correctly reverted, but there were some innocent changesets above the threshold which were incorrectly reverted as well. After this action project state was valid again and subsequent build succeeded.

Bad revert: All the changesets that were reverted were actually innocent and the real culprit was left in the code base. After this action project state stayed invalid and subsequent build failed with the same error.

Bad retry: Diagnosis result incorrectly indicated that the failure was caused by the system defect and the build should be retried. However, subsequent build failed with the same error.

Fallback: None of the changesets had sufficiently high probability rank to be considered as a culprits and the problem was escalated to a human expert for a manual intervention.

Each of these scenarios has some unique implications. The most desired outcome is *fixed (clean)* because it restores valid state of the project without generating extra work for the team. Only the developer who committed changeset with a defect needs to revisit it and improve the code before committing it again, but his task would need to be completed regardless.

Fixed (collateral) outcome, on the other hand, as its name implies has some collateral damage because apart from the real culprit some innocent changesets are removed as well. Their authors need to spend some time to confirm their changes are correct and commit them again. This is an extra work for them. Depending on the scale of the damage this scenario can also be considered as a positive one when the costs of the extra work it generates are lower than the costs related to productivity drop which results from the project staying in an invalid state This is particularly important for teams with dozens of people who might be impacted by a failed integration build.

The worst case is *bad revert*. Similarly to *fixed (collateral)* innocent changesets get removed, but this time the real culprit is left in the repository undetected. This means that developers need to face the cost of committing their changesets again and they get nothing back because the project is still in invalid state, however, they learn about it only after subsequent build fails with the same error. If it happens users might get impression that the system works against them so it is important to keep this rate at the minimum level.

Another negative outcome is *bad rollback* where system incorrectly retries the build with expectation that the second run will not hit the same intermittent issues. Just as for *bad revert* the subsequent build fails with the same error. The main difference, however, is that the bad decision did not generate extra work for developers. Instead the only consequence is that the system is unavailable while the second build is being executed so the overall efficiency is decreased. This rate can be tolerated at a much higher level than the *bad revert* rate.

Finally, in certain situations the safest thing diagnosis agent can do is to escalate the problem to build engineers for manual intervention, which we will refer to as *fallback* scenario. It is worth mentioning that even very high fallback rate (approaching one) can be acceptable in practice because in existing systems which do not automatically diagnose failures this rate is equal to 1, yet they are operational. By the way the diagnosis model was designed setting probability threshold to one drives the fallback rate one as well because due to non-zero leak parameters the model will never point at a real changeset with absolute certainty[1]. This relationship can help us understand charts in this section because we can always start the interpretation from threshold equal to 1, which is almost identical to a situation where the model is completely disabled.

With good understanding of all the possible outcome rates we can present them all in a single area plot in Figure 6.4. Different outcomes are represented with grey-scale color scheme where *bad revert* intentionally stands out with its black color to represent most severe mistake. Immediately visible is that the *fallback* rate increases almost linearly with the threshold. This increase becomes stepwise for higher values because there were fewer samples in a data set which got such high probability ranks. As explained in the previous paragraph it reaches value one together with the threshold.

When the probability threshold is set to zero *fixed (collateral)* rate dominates the distribution because for such low value all changesets are reverted including the real culprits. This of course is related to high collateral damage which is unacceptable in practice. From there the rate changes in two phases. Initially it decreases rapidly along with threshold to reach 8% at the point 0.09. Then it changes gently up to the point 0.5 where it drops down to 0.

In the first phase most of the changesets had arbitrary order because in the presence of no applicable defect types their probability was set entirely based on the leak parameters. The point at which the rate transitions into second phase corresponds to builds with one changeset and one pseudo system changeset, each with their leak parameters defined in the

[1]This rule does not hold for degenerated case of builds with no changesets where the failure is certainly caused by a system issue.

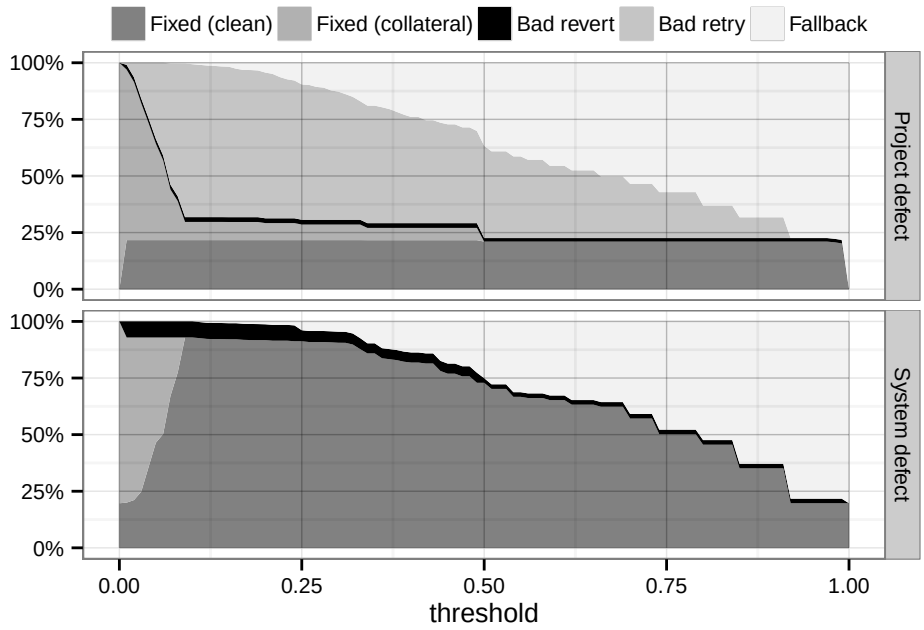


Figure 6.4 Outcome rates for baseline model

Equation 6.3. We can easily calculate it as $(0.001)/(0.01 + 0.001) \approx 0.9$.

The second phase shows that in the data set there were builds where more changesets were assigned higher probabilities than what would result from the leak parameter alone. Taking into account the only defect type supported by the baseline model we may infer that there were multiple changes to the same file which was mentioned in the error message. Because there was no additional evidence to break the tie builds where this happened could only be automatically fixed by removing all the related changesets with collateral damage.

The point at which the second phase ends and the rate drops to zero is very interesting. It is at $^1/_2$ and setting threshold to this value guarantees that the agent will revert at most one changesets because non-zero leak parameter for system defects it is impossible for two changesets to have a probability of $^1/_2$. Therefore, it is impossible for *fixed (collateral)* rate to be greater than zero. Other similar points are at $^1/_3$, $^1/_4$ for which similar guarantees hold for at most two and at most three reverts respectively.

This predictable behavior is very important in practice.

The most stable outcome is *fixed (clean)*. It barely changes in the full range of probability threshold except the extreme points where it goes down to zero. This indicates that the most intuitive procedure which is to blame the change to a file mentioned in the error message is in fact very reliable and can handle approximately 20% of all the project related issues in the data set. However, it does not always work. For threshold equal to 0.95 both *fixed (clean)* and *bad revert* rates are greater than zero indicating that there must have been some examples where the changeset was correct even though the error mentioned the file it modified.

Because the model supported only a single defect type it was unable to find relationship between the failure and changesets and for low values of the threshold the only reasonable action the agent could take was to repeat the build. This resulted in a high level of *bad retry* rate which decreases with threshold.

Up to this point we focused only on the project defects, however, most of the statements were true for system defects as well. The main difference is that because the leak parameters for system defects is much higher than for project defects in the case of baseline model the *fixed (clean)* rate is much higher as if it took the share of the *bad retry* rate.

Precision and recall analysis

Outcome rates are great to get an overview of how the model performs in a mixed mode switching between full automation and decision support system. In order to understand better how it behaves in each of these roles we will need some different measures. The reason why it is so important becomes obvious if we think about the *fixed (collateral)* rate. It tells us only that the real culprit was reverted, but it does not tell us anything about the scale of collateral damage. To get better insights we can use precision and recall plots such as the ones presented in Figure 6.5.

If we compare these plots to what we discussed in the previous section we can observe that recall curve has a very similar shape to sum of *fixed (clean)* and *fixed (collateral)* rates, whereas precision curve includes additionally mistake rates *bad revert* and *bad retry*.

Although the shapes are very much alike the actual values are different

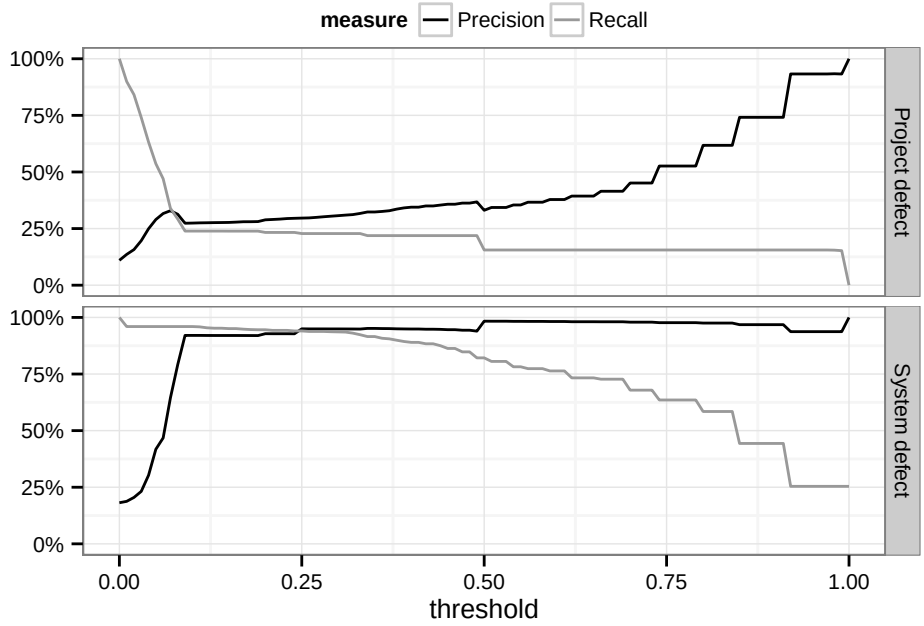


Figure 6.5 Precision and recall plots for baseline model

because rates operate at the level of integration builds while precision and recall tell us more about what is happening at the level of changesets. For example previous 20% of *fixed (clean)* rate coincide with only 15% recall. This particular discrepancy exists because some builds in the data set had multiple defects.

The presence of multiple defects also explains the rather strange shape of precision around the characteristic point 0.5. In general precision increases along with threshold, but at this point we can observe small drop. This happens because above this point model ignores previously correctly detected defects to satisfy the at most one revert guarantee discussed together with *fixed (collateral)* rate.

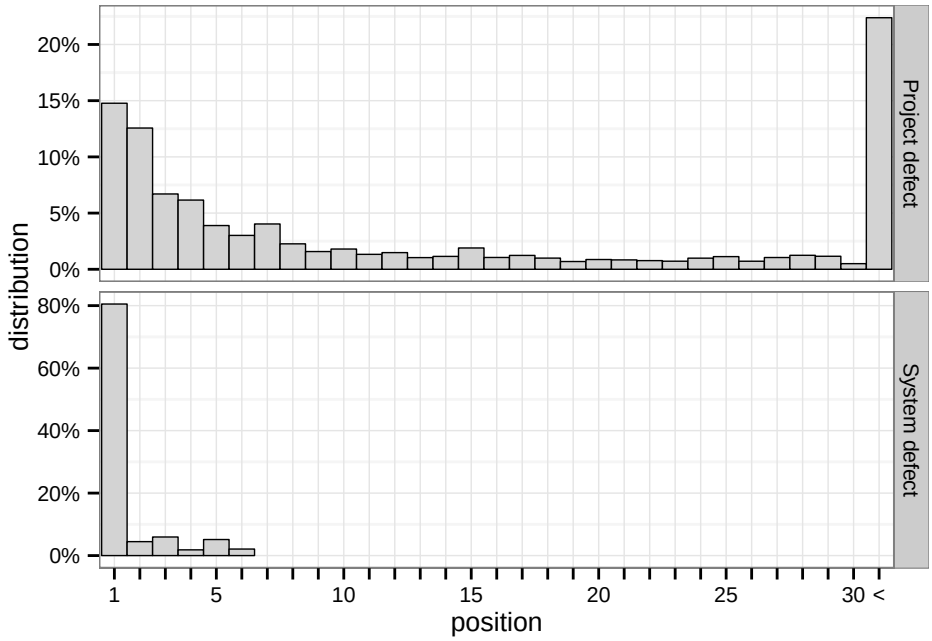


Figure 6.6 Histogram of culprit positions for baseline model

Culprit position analysis

When the model is used as a decision support system then the agent does not automatically revert bad changesets, but instead it prepares a detailed report with a list of candidates ordered by the probability of having introduced a defect. In this setup one of the most important quality measure is the position of the real culprit in the report. Of course it is best when it appears first because then user can find it immediately. This is not always possible and sometimes users will need to go through a list and inspect other, correct changesets.

It is worth mentioning that culprit position is insensitive to the probability threshold used previously. In fact in decision support mode there is no threshold because the user is responsible for making a call about where the defect is. Consequently, even changeset with a very low probability rank can appear at the top of the list, provided that all the other candidates have lower ranks. Because probability has to sum up to one this is

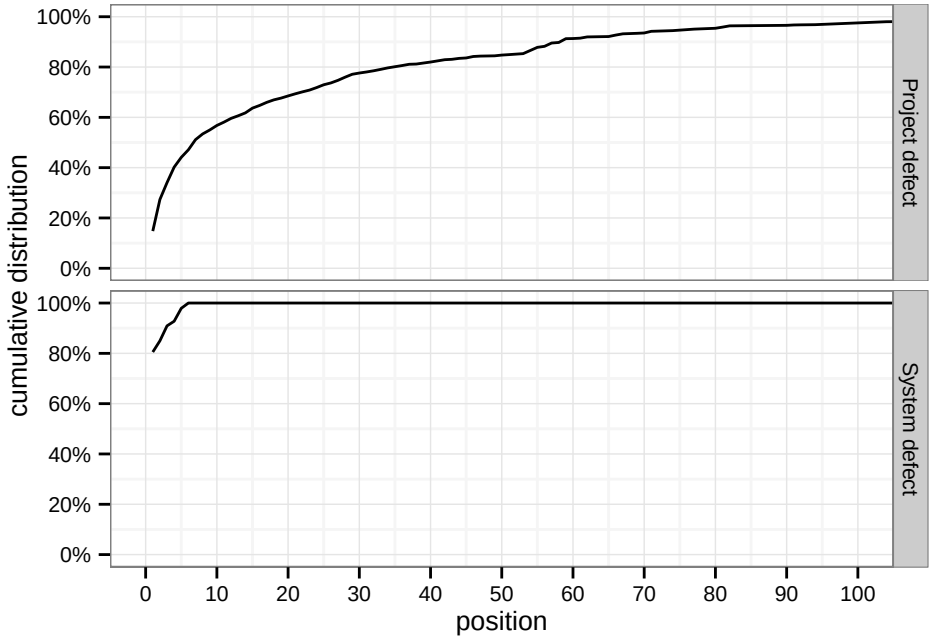


Figure 6.7 Cumulative distribution of culprit positions for baseline model

only possible for builds with a high number of changesets to consider.

In order to present the distribution of culprit positions for the baseline model we created a histogram presented in Figure 6.6. The first bar for project defects reaches only 15%, which not surprisingly matches the level of recall for high values of threshold from Figure 6.5. Second bar is high as well with the level of 12%. At the third position on the other hand we can see a drop after which the distribution slowly decreases to zero at the point beyond the range presented in the histogram.

This pattern can be easily explained by looking at the set of defect types supported by the baseline model. It consists of only single specific defect type which can be observed only in 29% of builds in the data set. When supporting evidences are there, the model assigns high probability ranks to associated changesets bringing them to the top of the list (first two positions). In builds which failed due to other defects the real culprit will appear at random position. The reason why we do not see this uniform

distribution in the histogram is that there are more builds with small number of changesets, thus the density is gradually decreasing.

For system defects 80% of the mass is located in the first position while the rest is distributed in the range up to sixth position. This shows what 20% of system defects manifested with error messages which mentioned specific files, but the baseline model was unable to distinguish them. In order to fix this problem the model would need to be extended by adding support for these very specific defect types.

Histograms are great for visualizing distributions, but it might be challenging to use them to compare several distributions, which is something we will need to do in the following sections. For this tasks cumulative distribution functions seem like a better fit. Figure 6.7 present such functions created for the baseline model. Apart from much wider range of on the horizontal axis this solution has advantage of having the same scales for vertical axis because it is standardized to percentage.

Now it is clear that the distribution for project defects does not finish around 30^{th} position. The tail of the distribution is much longer. Even at position 100 cumulative distribution is still lower than one. In next sections we will observe how adding more defect types can help shorten this tail and improve the overall quality of results.

6.3.2 General defect types

When diagnosis agent fails to correctly identify the culprit then most likely it is because it does not support defect type which caused the failure. At any point in time the model can be extended by adding support for new types. Each such effort consists of analyzing the causal mechanism of the defect, defining evidence which is a good indication of the failure and implementing them in code, add corresponding random variables for evidence, symptom and defect to the Bayesian network template.

With additional defect types the Bayesian network grows as it is depicted in Figures 6.8. Original defect plate is repeated once for each type supported and contains several nodes. Evidences are outside because they can be reused in multiple defects. All of the nodes meet at a single culprit variable at the bottom of the changeset plate. This procedure can be repeated to add support for an arbitrary number of defect types.

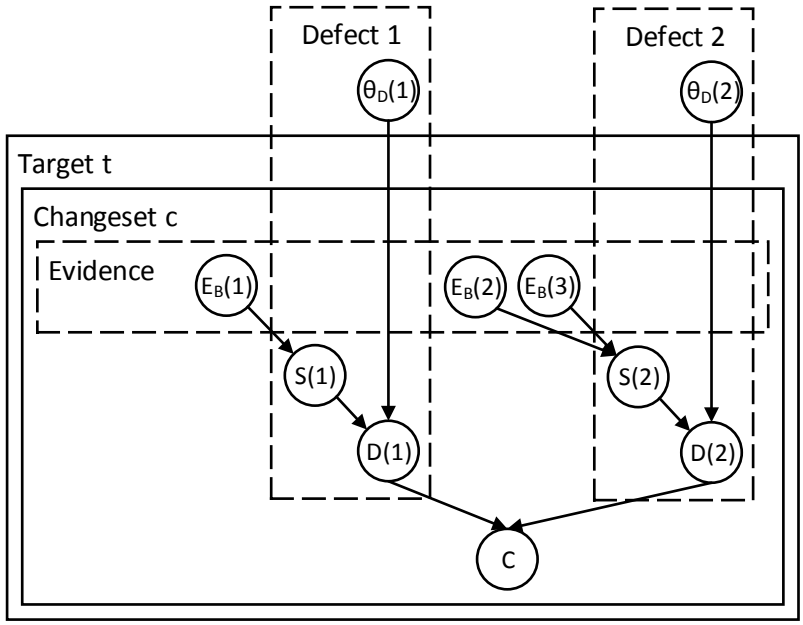


Figure 6.8 Plate model of network supporting two defect types

When extending the model one should select the specificity level for the defect type. If a diagnosis procedure is very specific it will identify culprits with high confidence, however, the number of builds for which it will be applicable will be low. General defect types on the other hand can have supporting evidence in many builds, but they do not necessarily point at the right changeset or they might identify multiple equally plausible candidates. We saw this problem previously for the baseline model when the *fixed (collateral)* rate was greater than zero.

This section focuses on general defect types which exploit such properties of errors as:

- file system locality,
- build graph locality,
- file extension associativity.

	Defect types count		
Case	General	Specific	Total
A	1	0	1
B	3	0	3
C	7	0	7
D	11	0	11
E	1	10	11
F	1	20	21
G	1	30	31
H	11	30	41

Table 6.1 Supported defect types counts in analyzed cases

First two assumes that the change which introduced the defect is close to the place where the error was observed in terms of a distance measured in the file system location or the build graph location. Both distances are valuable because they focus on different aspects. Former can detect big and risky changes even if the associated files have not been processed by tasks in the build graph whereas latter captures dependencies between projects which are located in completely different file structures.

Last property focuses on the file extensions because they can tell a lot about the tools used to process the files at build time. Even if the error message does not mention any file, but has an error code of *C# Compiler* we can expect that the defect is somewhere in a **.cs* file which is typically used for *C#* source code.

We will analyze how effectiveness of the model changes as more defect types are added by looking at results from several runs described in Table 6.1. Each case has its unique identifier and set of defect types. In this section we will focus only on the runs from *A* to *D*, others will be covered in the following section.

Because we need to compare many run at the same time the area plot presented in Figure 6.4 as to be replaced with a generalized version. Figure 6.12 shows outcome rates in stacked bar charts grouped by the value of threshold.

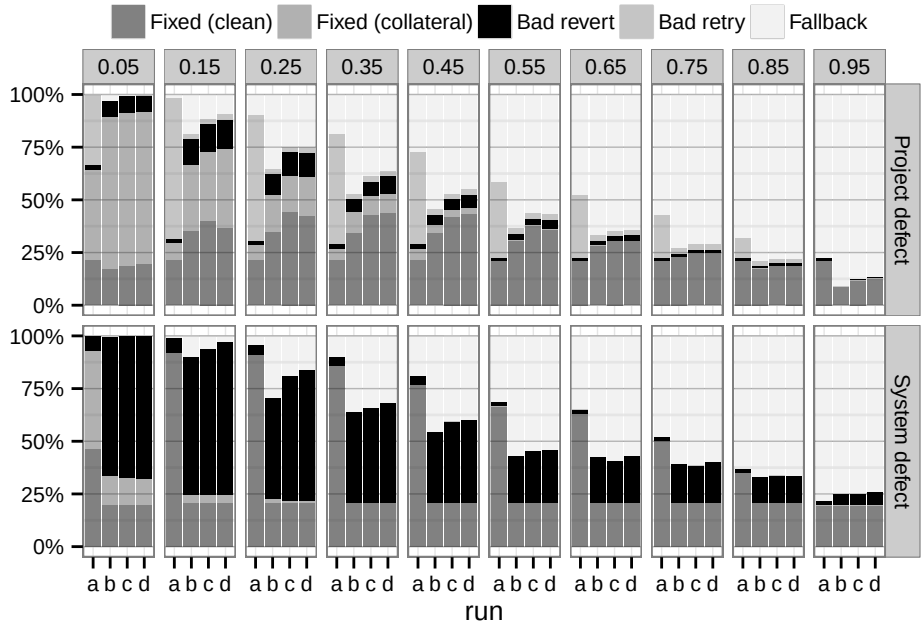


Figure 6.9 Outcome rates for model with general defect types

The most clearly visible difference from the previous chart is that there are high *bad revert* rates for system defects. That is because all the new types were project defects and other issues were neglected. Diagnosis agent was able to find more reasons to revert some changesets than to retry a build. This is not a completely bad situation because system defects happen so rarely, but there is definitely room for improvements.

Outcome rates for project defects were slightly impacted for the extreme values of probability threshold, where it was close to zero or one, and significantly different in the range from 0.15 to 0.65. We can see that the more defect types were added to the model, *fixed (clean)* rate was going up at the cost of *bad revert* rate. Outcome *bad retry* was rarely observed because the model would always find some reasons to revert certain changesets.

Precision and recall charts for the analyzed runs were presented in Figure 6.10. As soon as the first two defect types were added there was con-

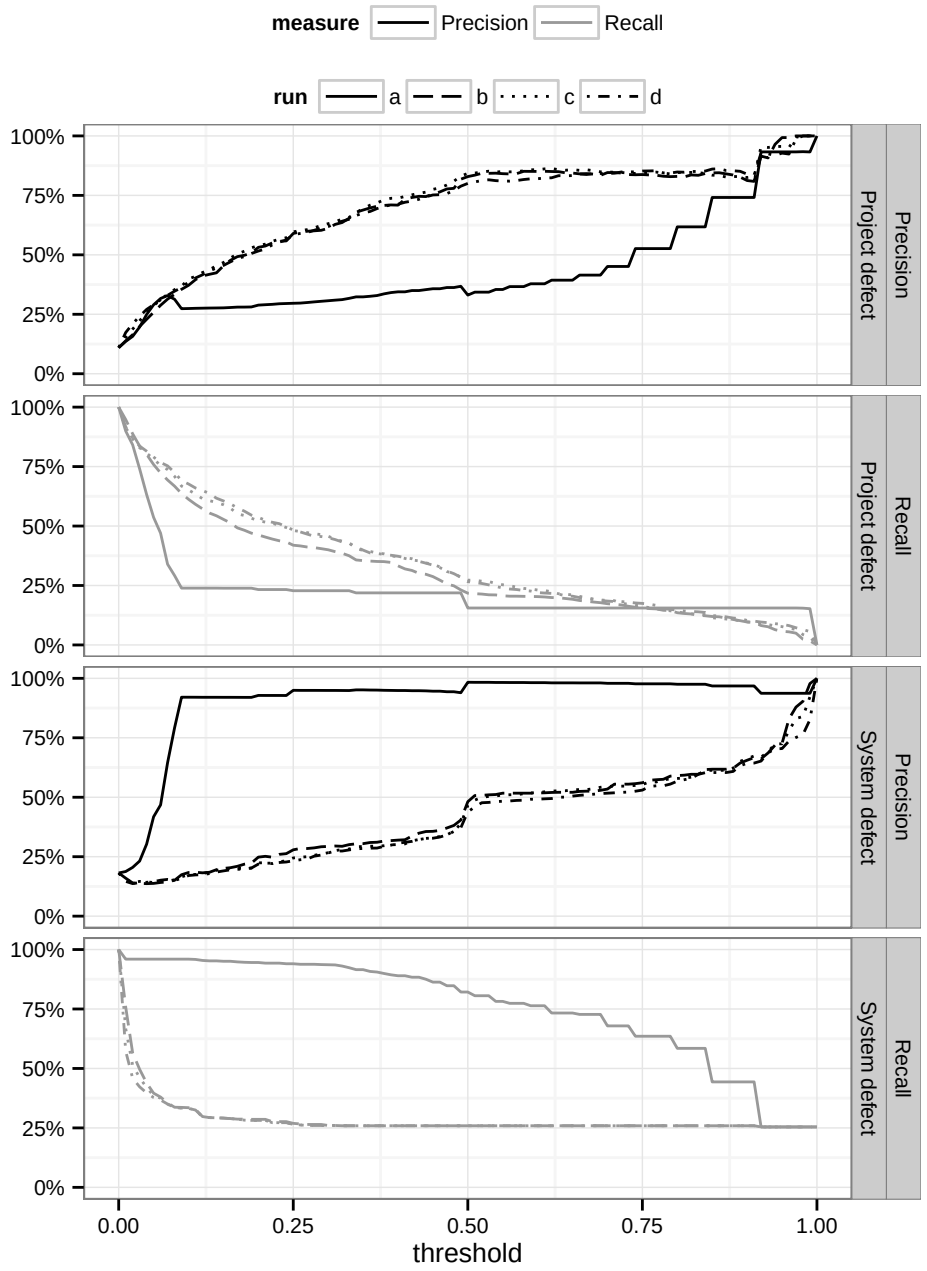


Figure 6.10 Precision and recall plots for model with general defect types

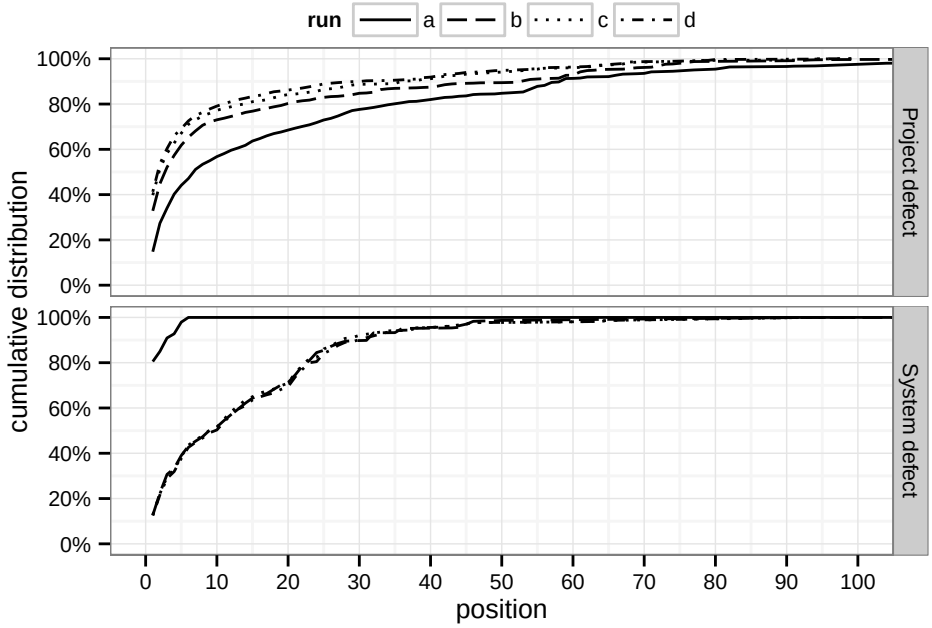


Figure 6.11 Cumulative distribution plot of culprit positions for model with general defect types

siderable improvement in both precision and recall. These defects types were looking for issues based on the file system and build graph locality. As it turns out these are very good predictors. Other defect types only slightly altered the measures.

The change becomes more visible when we look at the cumulative distribution of culprit positions in Figure 6.11. Model which supported more defect types would produce reports with more culprits located at leading positions. In particular in the best run D 40% culprits appeared on the first position in comparison to the 15% of the baseline.

Additionally the upper limit for the position was improved as well. Previously the baseline model could not guarantee that all culprits would appear in the first 100 results. With more defect types for all the runs the cumulative distribution converged to zero before the 80^{th} position.

6.3.3 Specific defect types

Model can also be extended by adding support for specific defect types. In comparison to defects discussed in the previous section they do not rely on common properties applicable to many errors such as build graph locality, but focus on explicit modelling scenarios for specific error codes. That makes them more focused and once all the necessary evidences are there the conclusions can be made with higher confidence.

A good example or a scenario for a specific defect type is when developer delete a C# class in a library without making sure it is not used by the projects which depend on the library. In the integration build this can manifest with a *C# Compiler* error *CS0234 (DottedTypeNameNotFoundInNS)* as similar to the one presented in Listing 6.3. The message indicates that a compiler found an identifier in the source code in the context where it was expecting either a namespace or a type, but it was unable to resolve it.

```
1 A.cs(15,14): error CS0234: The type or namespace name 'C' does
2   not exist in the namespace 'B' (are you missing an assembly
3   reference?)
```

Listing 6.3 Example of an error message used for a specific defect

In order to translate this scenario to a diagnosis procedure it is necessary to create basic evidence random variables to indicate if the error code was observed in logs, if a changeset modified a file which by convention should contain the definition of the missing type. Additionally one can define a complex evidence that will be set when the definition was in fact deleted. This task can requires downloading and parsing two versions of the same source file to inspect types it contains.

After adding more specific defect types the outcome rated changed as depicted in Figure 6.12. Because identified culprits had much higher probability ranks than what we saw in the previous section the increase in *fixed (clean)* rate can be observed even for extreme values of the threshold. In particular at the point of 0.95 there is still a clear increase. This improvement appears also for other points and around 0.45 has its maximum. That is where the model was capable of fixing problem in 50% of all integration

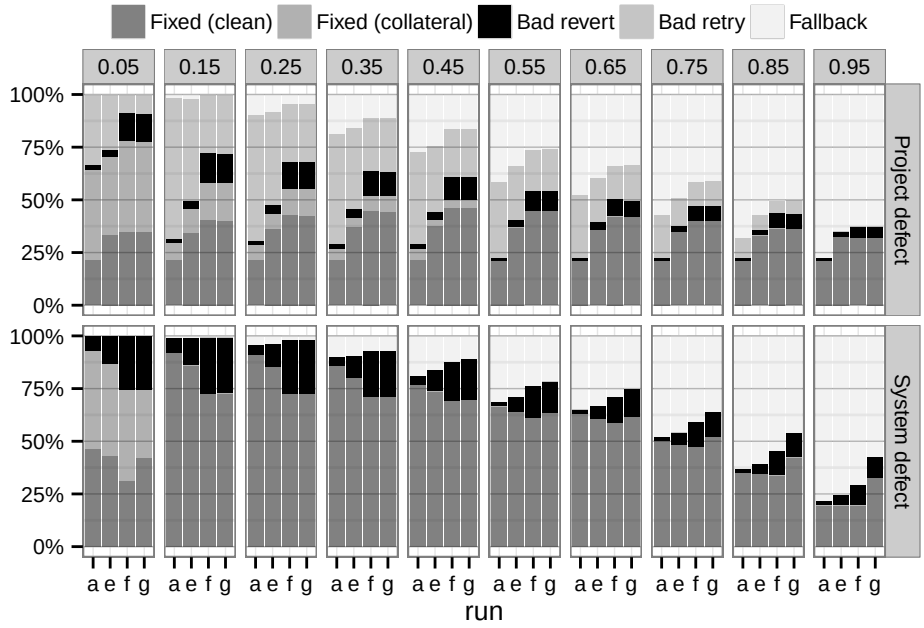


Figure 6.12 Outcome rates for model with specific defect types

builds in the data set with only small collateral damage. Unfortunately, at this point *bad revert* rate is also higher than for the baseline model.

Because specific defect types explain only a subset of all failures the *bad retry* is greater than zero for all values of threshold meaning that meaning that external defect despite being incorrect was the only plausible explanation for many builds. In the situations where it was a correct conclusions, in the lower plot, *fixed (clean)* stays at a high level, comparable to the baseline model.

High values of *bad revert* rate are also reflected in the precision and recall plots in Figure 6.13. This does not match the intuition because specific defect types should come into play only if many specific conditions are met. However it also means that the diagnosis agent can make bad decisions with high confidence for a small fraction of all builds. This is a negative effect which ideally should be eliminated from the system.

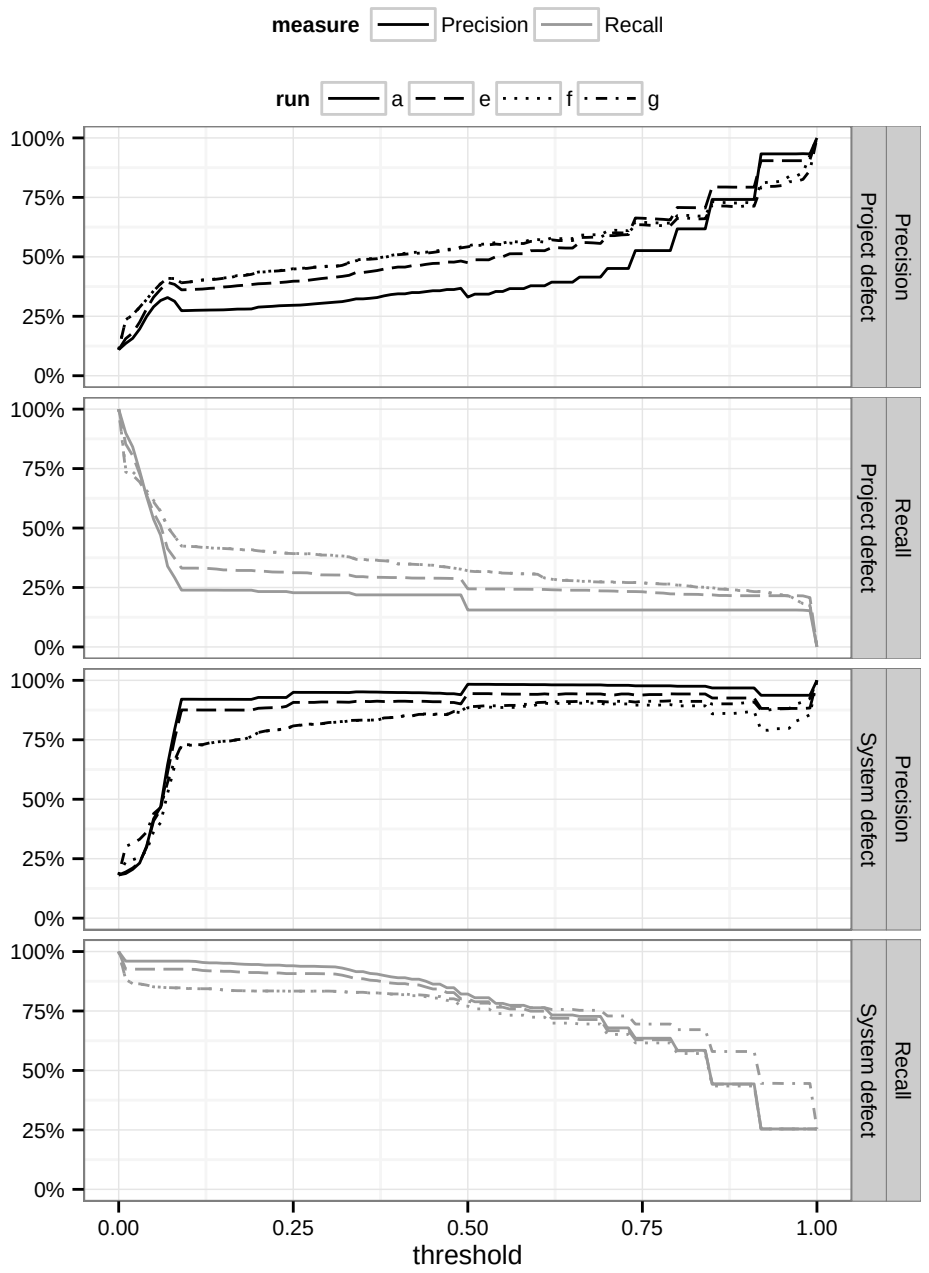


Figure 6.13 Precision and recall plots for model with specific defect types

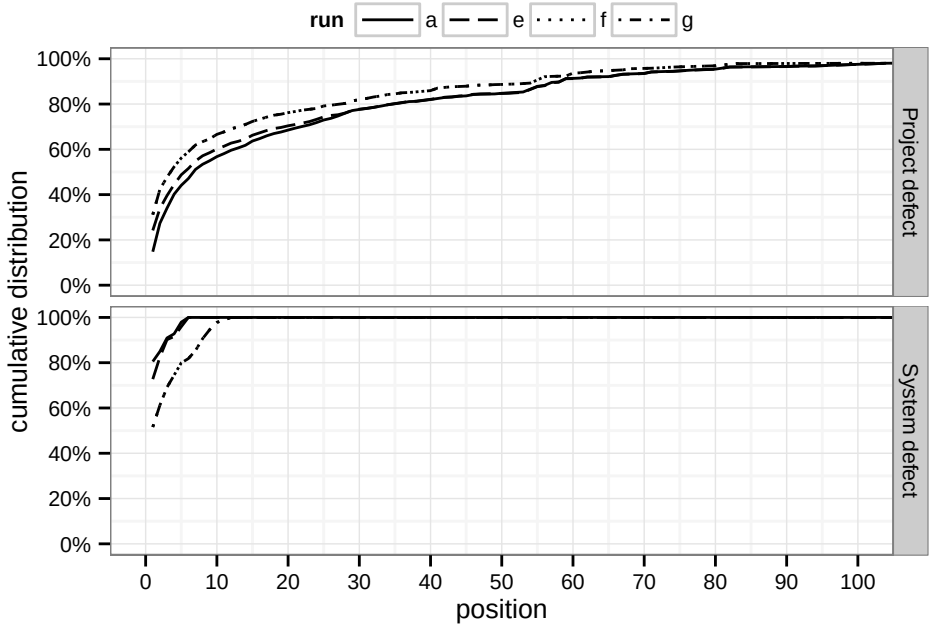


Figure 6.14 Cumulative distribution plot of culprit positions for model with specific defect types

Cumulative distribution plots for specific defect types presented in Figure 6.14 show that as the model was being expanded the quality was improving as well, however, the rate of improvement was decreasing. There is hardly any difference between cases F and G despite the fact that the latter supports 10 additional defect types (see Table 6.1).

This highlights the challenge in quality evaluation for specific defect types. Some of them are observed rarely and are underrepresented in the data set, thus their impact on the global quality measures may appear to be insignificant. However, for build engineers who come across such defect in production environment it can make a significant difference whether model supports it or not.

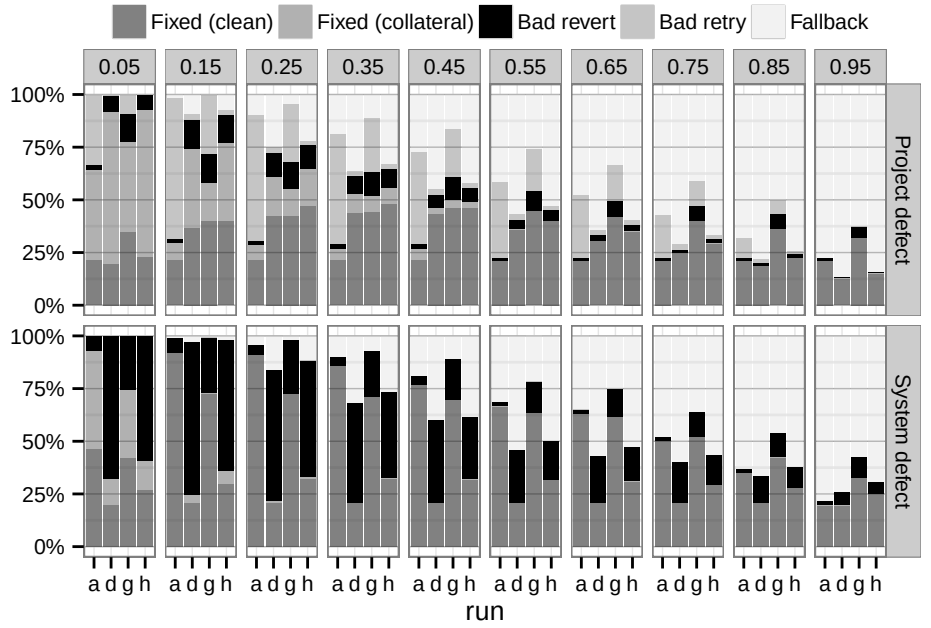


Figure 6.15 Outcome rates for combined model with both types of defects

6.3.4 Combined model

Previous two sections showed that general and specific defect types have very different characteristics when it comes to quality measures. Each of them have their strengths and weaknesses. In this section we will look into what happens if we create one model capable of diagnosing all the defects discussed so far. This case appeared in Table 6.1 with H identifier. It supports a total number of 41 defect types.

The outcome rates for combined model as well as the major models from the previous section is presented in Figure 6.15. The most interesting pattern that emerges for project defects for *fixed (clean)* rate is that it increases for the first three cases to finally drop in H to the level between cases D and G. It is very clearly visible for probability threshold 0.65 and its immediate neighborhood. If this rate was the only measure we care about this would be degradation of quality, however there are other rates

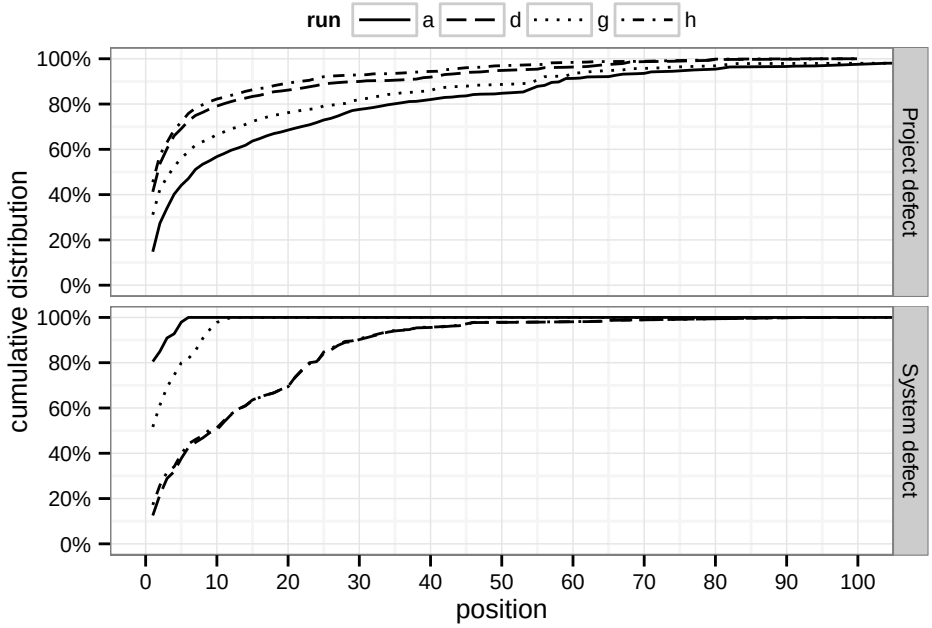


Figure 6.16 Cumulative distribution plot of culprit positions for combined model

that also need to be taken into consideration. For example *bad revert* rate is associated with extra costs because it requires developers to do extra work and commit their changesets again.

If we focus on the negative rates we can observe that they follow a similar pattern and drop for the last case. For threshold 0.65 *bad revert* rate decreased four times in comparison to G case. Even better improvement was observed for *bad revert* which decreased almost to zero for the combined model. This change, however, was at the cost of decrease in *fixed (clean)* rate for system defects.

Synergy between general and specific defect types is also visible in cumulative distribution of culprit position in Figure 6.16. Combined model outperformed all the other cases. However, the change from D to H is much smaller than from A to G because defects overlap in certain builds.

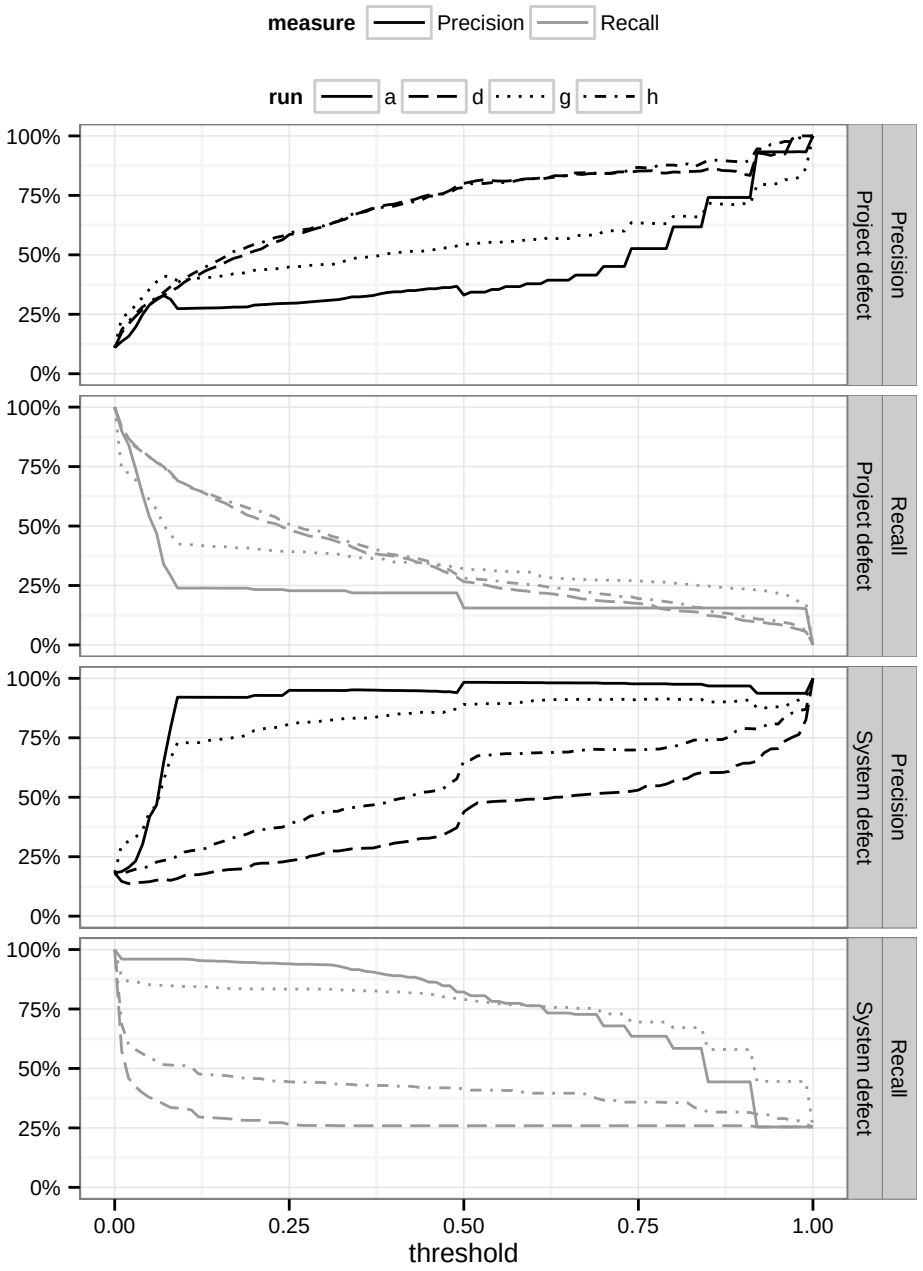


Figure 6.17 Precision and recall plots for combined model

The reason why combined model works better than other options is visible in precision and recall plots in Figure 6.17. In all facets curves for case *H* lies in between cases *D* and *G*. As it turns out combining defect types leads to a balanced model.

6.4 The impact of prior expert knowledge

The analysis so far was focused on the model trained with uninformative priors to highlight the ability of the model to train from the data set. However, in practice there is available prior expert knowledge which in theory can be leveraged to further improve the accuracy of the agent. Current section explores this process by looking into what happens when the feedback from human experts is included in the model.

As discussed in chapter 5, expert knowledge is encoded in the model in the form of hyperparameters of probability distributions used to describe defect types. Consequently in order to analyze how quality measures change when model makes use of it we need to select models which already support some defect types for the baseline.

Table 6.2 gathers information about cases which were used in the analysis. Because models will be compared pairwise there are two pairs or models, each with different properties. First are *D* and *I* models which support only general defect types. They are followed by *G* and *J* which, on the other hand, support primary specific defects. For each pair one model uses non-informative while the other has priors set by experts.

	Includes prior	Defect types count		
Case	expert knowledge	General	Specific	Total
D	*false*	11	0	11
I	***true***	**11**	**0**	**11**
G	*false*	1	30	31
J	***true***	**1**	**30**	**31**

Table 6.2 Selected cases for prior expert knowledge analysis

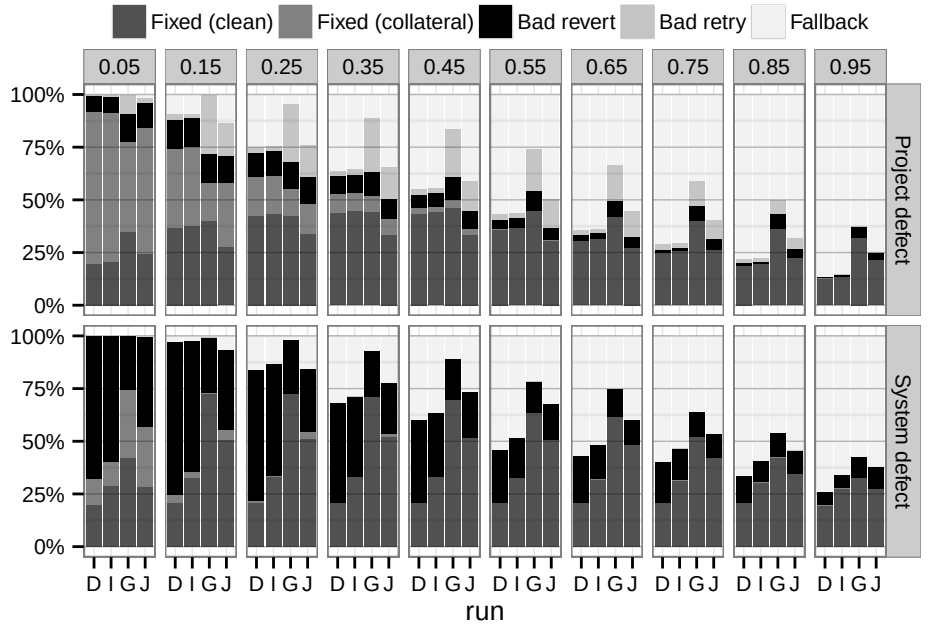


Figure 6.18 Outcome rates for model with prior expert knowledge

Outcome rates for the analyzed models were summarized in Figure 6.18 with bar plots introduced in the previous section. It is clear that the pairwise difference strongly depends on the defect type. For models D and I there is hardly any difference in rates calculated for project defects for all the values of probability threshold, while the model was able to better diagnose external issues.

In the second pair the difference is clearly visible everywhere, however, in contrast to what one could expect adding expert knowledge made the model worse in the sense of proposed quality measures. There is a significant drop in *fixed (clean)* and other rates, where agent takes an action while *fallback* rate increases.

Different patterns seen for two pairs can be explained by going back to Equation 5.7. General defect types have typically good coverage in the data set. They have a lot of samples which can be used to calculate parameters of probability distributions and their priors become unimportant. Specific defect types, on the contrary, can have only few samples to

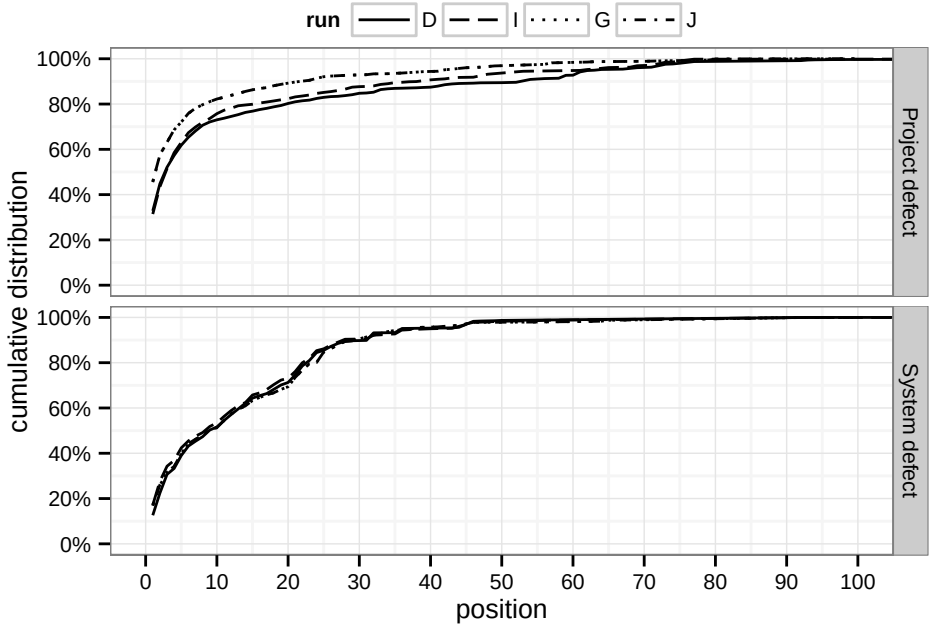


Figure 6.19 Cumulative distribution plot of culprit positions for model with prior expert knowledge

support them and that is when the end distribution will be dominated by prior probabilities provided by experts.

This also explains why there appears to be drop in quality for case *J*. Because prior expert knowledge, by its very nature, is supposed to come from observations made before the data set collection process started, when it is included in the model it can lead to decrease in performance measured against the same data set or its subsets (cross-validation). However, what we are really interested in is the good performance on the new samples which have not been included in the data set so the initial warning signs for outcome rates does not undermine the whole principle.

In fact, if we look at the cumulative distribution of culprit positions in Figure 6.19 we can see that the curves not only did not get worse but even improved for case *I*. This indicates that including prior expert knowledge

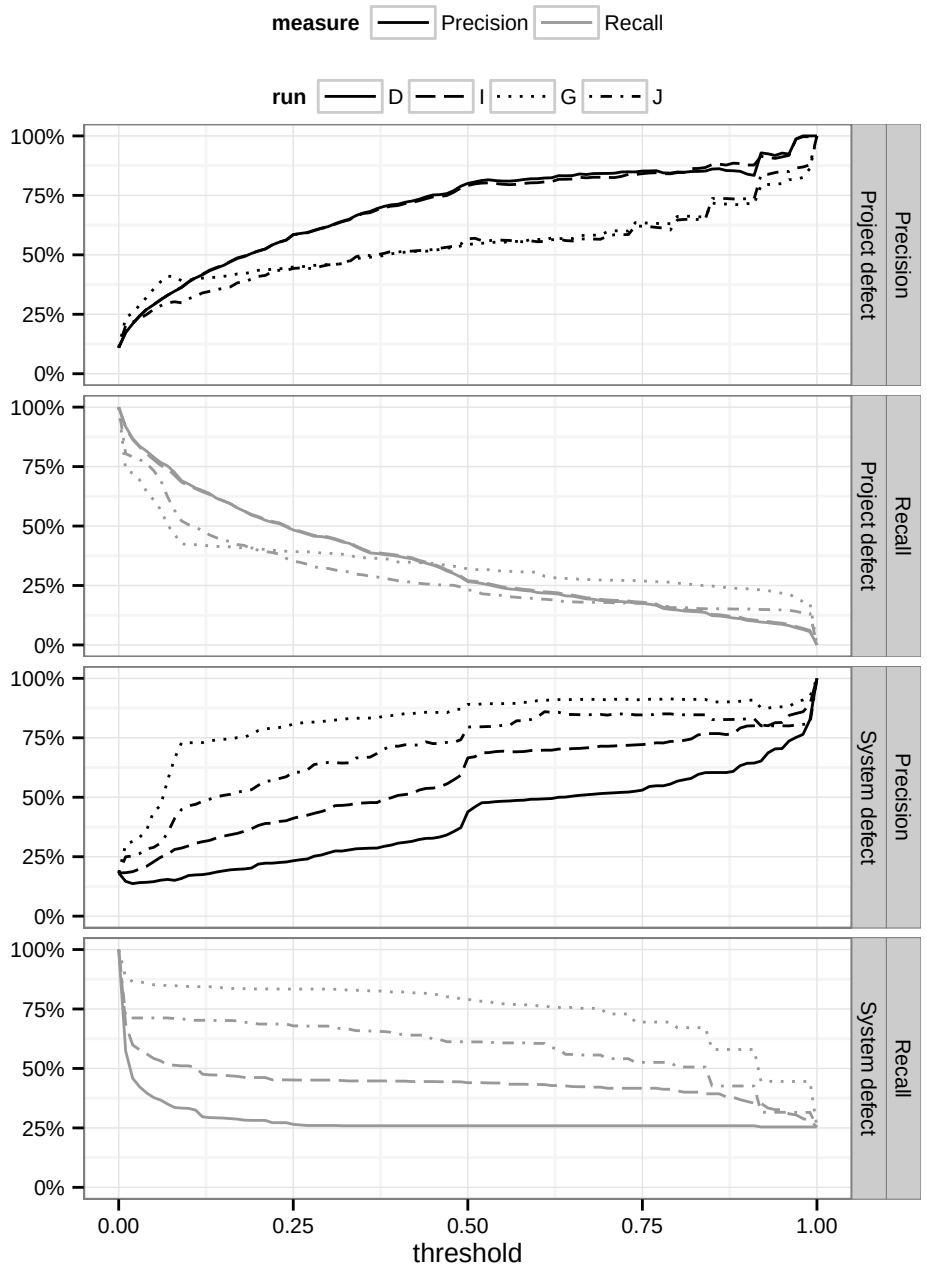


Figure 6.20 Precision and recall plots for models which make use of the prior expert knowledge

in the model might skew returned probability values, but the overall order of changesets should be preserved with real culprits floating to the starting positions.

Precision and recall curves presented in Figure 6.20 are plotted against the probability threshold, thus there is a drop in quality similar to what we observed previously in Figure 6.18. This time, however, it is visible that the rates were mainly affected by lowering recall while the precision stayed mainly at the same level.

In the absence of strong evidence showing that prior expert knowledge is valuable to the model we cannot reject a null hypothesis sating that it has no positive effects. However, during the research we came across examples of defect types which gained a lot from manually set priors. The first system defect type from section 6.3.1, which was created for *C# Compiler* error *CS0003 (NoMemory)*, is one of them. With only two supporting samples and non-informative priors its probability would have been set to approximately 0.75. Of course such error almost certainly indicate the problem in the system and the probability should be approaching 1. The only way to enforce that in the model is by setting its prior based on opinions of human experts.

6.5 Summary

Data set used in the research was collected over a period of more than a year in an industrial Continuous Integration system used across many projects. Although the rate of failed integration builds was very low the high number of executions happening every day made it possible to collect sufficient number of samples. Additionally, to make the diagnosis model operational it was necessary to capture prior expert knowledge and translate it to a supported form.

The designed model had some unique challenges which were addressed in the research. Prior expert knowledge was eliminated completely in the network size analysis by adoption of uninformative priors which maximize the information brought by the data itself. This highlighted patterns and behaviors which emerged when more random variables related to defect types were added to the model. Only in the final phase of the research was the prior expert knowledge included to measure its impact in isolation.

Network size analysis was performed in three steps. First we looked into how different quality measures change when model is extended with general defect types. Then, we switched to specific defect types just to finish with a combined model.

With general defect types model practically stopped making a mistake attributing failure to a system failure and the *bad retry* rate dropped almost to zero. At the same time *fixed (clean)* rate, which is one of the most important quality measures, improved slightly. Specific defect types, on the other hand led to a model which has much higher *fixed (clean)* rates at the cost of also increasing *bad revert* rate. Only the combined model, which supported both classes of defect types, was able to produce high positive rates and keep the negative rates at reasonably low levels.

The impact of prior expert knowledge was different depending on the types of supported defect types. If there was enough samples in the data set to estimate the probability parameter from then manually set prior probability was unimportant and had no effect on the quality measures. However, for models witch specific defect types both data and expert knowledge were combined together to produce one result. This part of the research was inconclusive because some quality measures improved while others declined, but one thing was clear - there are certain defect types which can gain a lot of manually set priors and the model should support them.

7 Conclusions

In this dissertation we proposed a novel improvement to existing Continuous Integration systems which removes the burden of selected manual tasks by introducing an autonomous software agent capable of diagnosing faults in integration builds and automatically fixing them by reverting changesets which introduced defects in the project source code. We confirmed its utility by training and evaluating it on the data set collected over the period of 16 months in a commercial CI system used at *Microsoft Corporation* by many different teams.

After analysis of several modern CI systems used in practice we discovered that they do not preserve enough information to build a robust data set for statistical learning applications in fault diagnosis. Whenever an integration build fails users care primary about quickly resolving the issue and less about documenting the circumstances and the root cause. We argued that in order to improve in this space it is necessary to preserve information about the build configuration, execution, failure, new changes in the project's source code and resolution steps. We designed a new format for data samples which can compactly store all this information.

We looked into several real-world scenarios of handling failed builds and proposed modifications which make it possible to not only fix the issue but also collect data samples. We argued that backward-fix strategy, where all the failures are resolved by reverting culprit changes from the version control system, is superior over forward-fix strategy because it makes it possible to fully automate the collection process so that there are no additional steps for the people who look after this process. However, we pointed out that in certain cases when multiple defects surface at the same time or when there are multiple integration builds running concurrently it is impossible to automatically collect high quality samples without extra input from human experts.

The main contribution of this dissertation is defining clear analogy between the problem of finding changesets which introduced defects in the

source code and well understood task of fault diagnosis. This opened an opportunity to use a state of the art modelling techniques of building systems for reasoning under uncertainty. The proposed expert system is based on a Bayesian troubleshooter and can answer probability queries about the posterior probability that a changeset had introduced a defect, conditioned on the observed evidence from the completed integration build. By framing the problem in this way we were able to design the agent to take actions based on the probability ranks and made this process easy to control by introducing a probability threshold under which the agent will refrain from reverting changes automatically. When the agent does not act upon results it can pass them to human experts and effectively work as a decision support system.

The diagnosis procedure we proposed was created to be applicable to real-world problems, thus it contains many practical assumptions. For example, the model expects there it precisely one defect to find either in the project's source code or in the CI system itself. It makes the agent infeasible for environments where multiple-faults are standard, but we confirmed that it did not impact the overall efficiency of the model in a significant way when evaluated against the available data set. The benefit, however, was cognitive simplicity because users can expect all the probability ranks to sum to one.

The structure of the proposed Bayesian network scales well with the number of possible defect types. This requirement was critical because there are hundreds of reasons why a compilation phase might fail and there are typically other phases as well which have to be taken into consideration. The model satisfied also all the other requirements gathered for the agent. It is a Machine Learning system because the more samples are added to the training set the more accurate estimates are calculated for model parameters. Not only does the model learn from the training set, but it also incorporates existing expert knowledge. Finally, it produces results which can be understood by the users.

We studied the effectiveness of the proposed model on the data gathered in a real-world, commercial Continuous Integration system. We focused on answering questions regarding what happens when the Bayesian network grows in terms of the number of distinctive defect types it supports, and how inclusion of prior expert knowledge changes the quality measures.

We showed that for the best results the model should support both

general and specific defect types. Only this combination led to high rate of correctly fixed problem and kept the rate of mistakes made by the model at reasonably low levels. In the best observed case the model was capable of successfully handle 50% of all failed integration builds, made mistakes for 7% and left the rest for manual intervention.

On the occasion when agent refrains from taking action it can act as a decision support system and provide users with reports in the form of ranked list of changesets paged by 10 items. We estimated that users can expect the real culprit to appear at the first page in 80% of cases and within first two pages in 90%. The rest are the more challenging failures, where the agent showed to be no better than a random ranking algorithm. In order to improve performance for these rare cases it would be necessary to extend the set of defect types supported by the system, but we decided against doing it due to increasing costs. This led to a conclusion that with plethora of possible problems some of them are still better handled manually.

In the study we divided all defects into two categories: project and system. We noticed that performance for system defects was considerably worse. It was partially because the diagnosis model had only limited support from this category. However, it drew our attention to several cases where evidence clearly indicated problems with the project source code, but it was in fact a system defect related to updating the local workspace on the integration machine. The proposed model does not handle such cases well, as it has certain requirements regarding the reliability of the system where it operates correctly.

The solution presented in this dissertation is not applicable to every software project. When either development velocity or the concurrency level is low it is typically obvious which changeset introduced a defect. Adding extra diagnosis step to the process would only increase the total time it takes to execute an integration build without a real benefit to the people working on the project.

Another case where probabilistic fault diagnosis is not the right solution is when the project is small and it takes just several minutes to run a full integration build. Then, it is possible to find the changeset which introduced a defect by doing a binary search on the new changesets that have been checked-in to the repository. Alternatively, if the compiler can run in an incremental mode it might be better to check all the changesets

sequentially in chronological order. Both solutions are deterministic and guarantee correct results provided that builds are reproducible. The only problem is that this approach is limited to short builds.

The proposed solution fits perfectly to large-scale Continuous Integration systems and large projects where it takes up to several hours to execute the compilation phase, the rate at which new changesets are checked in is high and there are many people working on the same product possibly from locations distributed geographically in different time zones. In such environments build break management strategy is critical to make the system successful and shortening the time it takes to diagnose issues can improve productivity of everyone working on the project.

In June 2014 the implementation of the software agent described in this dissertation was enabled on a voluntary basis for selected projects at *Microsoft Corporation*. At the time of writing it has been constantly running for almost a year and bringing value each time there is a failure in an integration build that needs to be diagnosed. It has been running both in full automation and decision support mode depending on the individual preferences of teams and project owners.

Bibliography

Artificial Intelligence

[Ada84] J. B. Adams. "Probabilistic reasoning and certainty factors". In: *Rule-Based Expert Systems* (1984), pp. 263–271.

[Bou+96] C. Boutilier et al. "Context-specific independence in Bayesian networks". In: *Proceedings of the Twelfth international conference on Uncertainty in artificial intelligence*. Morgan Kaufmann Publishers Inc. 1996, pp. 115–123.

[BS+84] B. G. Buchanan, E. H. Shortliffe, et al. *Rule-based expert systems*. Vol. 3. Addison-Wesley Reading, MA, 1984.

[Cyr08] K. A. Cyran. "Modified indiscernibility relation in the theory of rough sets with real-valued attributes: Application to recognition of fraunhofer diffraction patterns". In: *Transactions on Rough Sets IX*. Springer, 2008, pp. 14–34.

[DB07] J. L. Devore and K. N. Berk. *Modern mathematical statistics with applications*. Cengage Learning, 2007.

[Fra+12] U. Franke et al. "Availability of enterprise IT systems: an expert-based Bayesian framework". In: *Software quality journal* 20.2 (2012), pp. 369–394.

[Hec90] D. Heckerman. "Probabilistic interpretations for MYCIN's certainty factors". In: *Readings in uncertain reasoning*. Morgan Kaufmann Publishers Inc. 1990, pp. 298–312.

[HH87] D. E. Heckerman and E. J. Horvitz. "On the Expressiveness of Rule-based Systems for Reasoning with Uncertainty." In: *AAAI*. 1987, pp. 121–126.

[HHN91] D. E. Heckerman, E. J. Horvitz, and B. N. Nathwani. "Toward normative expert systems: The Pathfinder project". In: *Methods of information in medicine* 31 (1991), pp. 90–105.

[HN92] D. Heckerman and B. N. Nathwani. “An evaluation of the diagnostic accuracy of Pathfinder”. In: *Computers and Biomedical Research* 25.1 (1992), pp. 56–74.

[IWD05] G. Ilczuk and A. Wakulicz-Deja. “Rough sets approach to medical diagnosis system”. In: *Advances in Web Intelligence*. Springer, 2005, pp. 204–210.

[IWD07] G. Ilczuk and A. Wakulicz-Deja. “Visualization of rough set decision rules for medical diagnosis systems”. In: *Rough sets, fuzzy sets, data mining and granular computing*. Springer, 2007, pp. 371–378.

[Jam+13] G. James et al. *An Introduction to Statistical Learning*. Springer, 2013.

[KF09] D. Kollar and N. Friedman. *Probabilistic graphical models: principles and techniques*. The MIT Press, 2009.

[KHH01] C. Kadie, D. Hovel, and E. Horvitz. “MSBNx: A component-centric toolkit for modeling and inference with Bayesian networks”. In: *Microsoft Research, Richmond, WA, Technical Report MSR-TR-2001-67* 28 (2001).

[KW12] M. Kurzynski and M. Wozniak. “Combining classifiers under probabilistic models: experimental comparative analysis of methods”. In: *Expert Systems* 29.4 (2012), pp. 374–393.

[Loc99] J. Locked. “Microsoft bayesian networks: Basics of knowledge engineering”. In: *Kindred Communications Troubleshooter Team Microsoft Support Technology* 12 (1999).

[Min+14] T. Minka et al. *Infer.NET 2.6*. Microsoft Research Cambridge. http://research.microsoft.com/infernet. 2014.

[Mit97] T. M. Mitchell. *Machine Learning*. McGraw-Hill Science Engineering, 1997.

[Ole+89] K. G. Olesen et al. “A munin network for the median nerve-a case study on loops”. In: *Applied Artificial Intelligence an International Journal* 3.2-3 (1989), pp. 385–403.

[Pea86] J. Pearl. “Fusion, propagation, and structuring in belief networks”. In: *Artificial intelligence* 29.3 (1986), pp. 241–288.

[Pea88] J. Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, 1988.

[PHF10] A. Patil, D. Huard, and C. J. Fonnesbeck. "PyMC: Bayesian stochastic modelling in Python". In: *Journal of statistical software* 35.4 (2010), pp. 1–5.

[PWD07] P. Paszek and A. Wakulicz-Deja. "Applying Rough Set Theory to Medical Diagnosing". In: *Rough Sets and Intelligent Systems Paradigms*. Springer, 2007, pp. 427–435.

[PWN09] A. Pernestål, H. Warnquist, and M. Nyberg. "Modeling and troubleshooting with interventions applied to an auxiliary truck braking system". In: *Proceedings of 2nd IFAC workshop on Dependable Control of Discrete Systems*. 2009.

[Rus+95] S. J. Russell et al. *Artificial intelligence: a modern approach*. Prentice hall Englewood Cliffs, 1995.

[SB75] E. H. Shortliffe and B. G. Buchanan. "A model of inexact reasoning in medicine". In: *Mathematical biosciences* 23.3 (1975), pp. 351–379.

[Sch11] R. J. Schalkoff. *Intelligent Systems: Principles, Paradigms and Pragmatics*. Jones & Bartlett Publishers, 2011.

[SJK00] C. Skaanning, F. V. Jensen, and U. Kjærulff. "Printer troubleshooting using Bayesian networks". In: *Intelligent Problem Solving. Methodologies and Approaches*. Springer, 2000, pp. 367–380.

[WB05] J. M. Winn and C. M. Bishop. "Variational message passing". In: *Journal of Machine Learning Research*. 2005, pp. 661–694.

[WDN07] A. Wakulicz-Deja and A. Nowak. "From an information system to a decision support system". In: *Rough Sets and Intelligent Systems Paradigms*. Springer, 2007, pp. 454–464.

[WH86] B. P. Wise and M. Henrion. "A framework for comparing uncertain inference systems to probability". In: *Proceedings of the First Conference on Uncertainty in Artificial Intelligence*. Morgan Kaufmann Publishers Inc. 1986, pp. 99–108.

[WKB10] W. Wiegerinck, B. Kappen, and W. Burgers. "Bayesian networks for expert systems: Theory and practical applications". In: *Interactive collaborative information systems*. Springer, 2010, pp. 547–578.

[Woz04] M. Wozniak. "Proposition of boosting algorithm for probabilistic decision support system". In: *Computational Science-ICCS 2004*. Springer, 2004, pp. 675–678.

[Woz11] M. Wozniak. “Knowledge source confidence measure applied to a rule-based recognition system”. In: *Intelligent Information and Database Systems*. Springer, 2011, pp. 425–434.

[ZD06] A. Zagorecki and M. J. Druzdzel. “Knowledge Engineering for Bayesian Networks: How Common Are Noisy-MAX Distributions in Practice?” In: *ECAI*. 2006, p. 482.

Software Engineering

[Abl+08] R. Ablett et al. “Build notifications in agile environments”. In: *Agile Processes in Software Engineering and Extreme Programming*. Springer, 2008, pp. 230–231.

[AKM08] A. Alali, H. Kagdi, and J. Maletic. “What’s a typical commit? A characterization of open source software repositories”. In: *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*. IEEE. 2008, pp. 182–191.

[BA99] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 1999.

[Bai+07] S. Bailliez et al. *Apache Ant 1.9.4 Manual*. Apache Ant. 2007.

[Bro08] G. Brooks. “Team pace keeping build times down”. In: *Agile, 2008. AGILE’08. Conference*. IEEE. 2008, pp. 294–297.

[Bug09] Y. Bugayenko. “Quality of code can be planned and automatically controlled”. In: *Advances in System Testing and Validation Lifecycle, 2009. VALID’09. First International Conference on*. IEEE. 2009, pp. 92–97.

[BZ10] R. Buse and T. Zimmermann. “Analytics for software development”. In: *Proceedings of the FSE/SDP workshop on Future of software engineering research*. ACM. 2010, pp. 77–80.

[BZ14] A. Begel and T. Zimmermann. “Analyze This! 145 Questions for Data Scientists in Software Engineering”. In: *Proceedings of the 36th International Conference on Software Engineering*. Hyderabad, India, 2014.

[CH09] S. Chacon and J. C. Hamano. *Pro git*. Vol. 288. Springer, 2009.

[CS98] M. A. Cusumano and R. W. Selby. *Microsoft secrets: how the world’s most powerful software company creates technol-*

ogy, shapes markets, and manages people. Simon and Schuster, 1998.

[Cze+11] J. Czerwonka et al. "Crane: Failure prediction, change analysis and test prioritization in practice–experiences from windows". In: *Software Testing, Verification and Validation ICST, 2011 IEEE Fourth International Conference on*. IEEE. 2011, pp. 357–366.

[DMG07] P. M. Duvall, S. Matyas, and A. Glover. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[DPH12] J. Downs, B. Plimmer, and J. G. Hosking. "Ambient awareness of build status in collocated software teams". In: *Proceedings of the 2012 International Conference on Software Engineering*. IEEE Press. 2012, pp. 507–517.

[FF06] M. Fowler and M. Foemmel. "Continuous integration". In: *Thought-Works, 2006* (2006).

[Fow99] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[FP98] N. E. Fenton and S. L. Pfleeger. *Software metrics: a rigorous and practical approach*. PWS Publishing Co., 1998.

[GS12] M. L. Guimarães and A. R. Silva. "Making software integration really continuous". In: *Fundamental Approaches to Software Engineering*. Springer, 2012, pp. 332–346.

[HB10] S. I. Hashimi and W. Bartholomew. *Inside the Microsoft Build Engine: Using MSBuild and Team Foundation Build*. O'Reilly Media, Inc., 2010.

[HB99] G. Hunt and D. Brubacher. "Detours: binary interception of Win32 functions". In: *3rd Usenix Windows NT Symposium*. 1999.

[HF10] J. Humble and D. Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[HJ07] J. Holck and N. Jørgensen. "Continuous integration and quality assurance: A case study of two open source projects". In: *Australasian Journal of Information Systems* 11.1 (2007).

[HX10] A. E. Hassan and T. Xie. "Software intelligence: the future of mining software engineering data". In: *Proceedings of the*

FSE/SDP workshop on Future of software engineering research. ACM. 2010, pp. 161–166.

[HZ06] A. E. Hassan and K. Zhang. "Using decision trees to predict the certification result of a build". In: *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*. IEEE. 2006, pp. 189–198.

[Joh96] M. K. Johnson. "Diff, patch, and friends". In: *Linux Journal* 1996.28es (1996), pp. 2–4.

[Kim+08] S. Kim et al. "Automated continuous integration of component-based software: An industrial experience". In: *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society. 2008, pp. 423–426.

[Lac09] F. J. Lacoste. "Killing the Gatekeeper: Introducing a Continuous Integration System". In: *Agile Conference, 2009. AGILE'09*. IEEE. 2009, pp. 387–392.

[LS00] J. R. Lorch and A. J. Smith. "The VTrace tool: building a system tracer for Windows NT and Windows 2000". In: *MSDN Magazine* 15.10 (2000), pp. 86–102.

[LSL12] F. Lier, S. Schulz, and I. Lütkebohle. "Continuous integration for iterative validation of simulated robot models". In: *Simulation, Modeling, and Programming for Autonomous Robots*. Springer, 2012, pp. 101–112.

[Mar11] R. C. Martin. *The Clean Coder: A Code of Conduct for Professional Programmers (Robert C. Martin Series)*. Prentice Hall, 2011.

[Mec04] R. Mecklenburg. *Managing projects with GNU make*. O'Reilly Media, Inc., 2004.

[Mil08] A. Miller. "A hundred days of continuous integration". In: *Agile, 2008. AGILE'08. Conference*. IEEE. 2008, pp. 289–293.

[Mis05] S. C. Misra. "Modeling design/coding factors that drive maintainability of software systems". In: *Software Quality Journal* 13.3 (2005), pp. 297–320.

[Mor+10] G. Moreira et al. "Software product measurement and analysis in a continuous integration environment". In: *Information Technology: New Generations (ITNG), 2010 Seventh International Conference on*. IEEE. 2010, pp. 1177–1182.

[Ras04] J. Rasmusson. “Long build trouble shooting guide”. In: *Extreme Programming and Agile Methods-XP/Agile Universe*. Springer, 2004, pp. 13–21.

[Rob04] M. Roberts. “Enterprise continuous Integration using binary dependencies”. In: *Extreme Programming and Agile Processes in Software Engineering*. Springer, 2004, pp. 194–201.

[Rog03] R. O. Rogers. “CruiseControl. NET: Continuous integration for. NET”. In: *Extreme Programming and Agile Processes in Software Engineering*. Springer, 2003, pp. 114–122.

[Rog04] R. O. Rogers. “Scaling continuous integration”. In: *Extreme Programming and Agile Processes in Software Engineering*. Springer, 2004, pp. 68–76.

[Sch04] K. Schwaber. *Agile project management with Scrum*. O’Reilly Media, Inc., 2004.

[SE04] D. Saff and M. Ernst. “An experimental evaluation of continuous testing during development”. In: *ACM SIGSOFT Software Engineering Notes*. Vol. 29. 4. ACM. 2004, pp. 76–85.

[SOR14] S. Świerc, M. O’Flaherty, and M. Rodríguez. “Automated failure diagnosis in large-scale build system”. In: *The Practive of Machine Learning*. Vol. 1. Microsoft. Redmond, USA, 2014.

[Sto07] T. van der Storm. “The Sisyphus Continuous Integration System.” In: *11th European Conference on Software Maintenance and Reengineering*. IEEE. 2007, pp. 335–336.

[Sto08] T. van der Storm. “Backtracking incremental continuous integration”. In: *12th European Conference on Software Maintenance and Reengineering*. IEEE. 2008, pp. 233–242.

[Sto09] S. Stolberg. “Enabling agile testing through continuous integration”. In: *Agile Conference, 2009. AGILE’09.* IEEE. 2009, pp. 369–374.

[Whi09] T. White. *Hadoop: the definitive guide: the definitive guide*. O’Reilly Media, Inc., 2009.

Notation Index

α_0, α_1	Parameters of *Beta* distribution
λ	Noise parameter of *Noisy-Or* and *Noisy-And* models
θ	Prior parameter which controls dependent distribution
$\boldsymbol{\theta}$	Vector of prior parameters which control dependent distributions
c	Changeset domain index
C	Random variable which indicates whether changeset introduced a defect
d	Defect domain index
D	Random variable which indicates whether specific defect was observed
e	Evidence domain index
E	Random variable which indicates whether specific evidence was observed
$\boldsymbol{E}$	Vector of random variable which indicate whether specific evidence was observed
$\mathbb{E}[X]$	Expectation (mean) of X
$M[x]$	Counts of event x in the data
$\bar{M}[x]$	Expected counts of event x
$\mathcal{N}$	Gaussian distribution
N_X	Counts of X random variable
S	Random variable which indicates whether specific symptom was observed
t	Build target domain index
$Val(X)$	Possible values of X random variable

Index