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.

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 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 selected steps in the integration process, which would have to otherwise be performed by the developers. 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 and fix defects. 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 depicted in the sequence diagram in Figure 1.

In this scenario Developer commits two changesets c_1 and c_2. The first change introduces a defect which causes the integration build to fail. In response Build Engineer performs diagnosis 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.

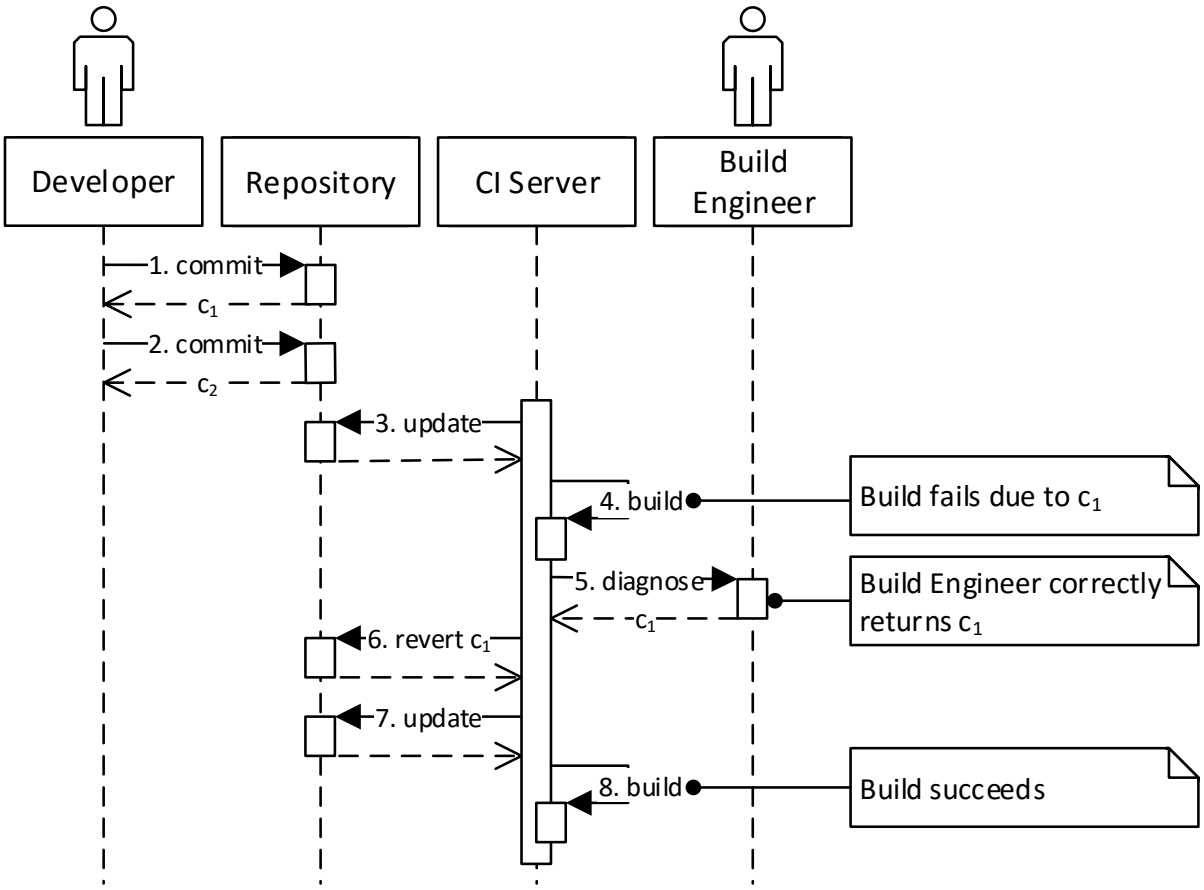

Figure 1. Backward fix sequence diagram

Depending on the project size the role of Build Engineer can be assigned to a developer or can be a permanent position.

2 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 described in the previous section can be automated. Additionally, with backward-fix policy it is possible not only to find the problem but also make automatic corrections. 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.

The dissertation described an interdisciplinary research on Software Engineering and Bayesian modelling of reasoning systems. It starts with detailed description of Continuous Integration and explains the associated concepts. Then, it switches to the problem of reasoning under uncertainty and Bayesian networks as one of the solutions. It contains several significant contributions: novel design of sample format for storing information about failures in CI systems, data set collection procedure and the diagnosis model itself. The effectiveness of the proposed model is studied on the data set gathered in a real-world, commercial CI system used daily by hundreds of developers.

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.

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 [KF09]. One of their realization are Bayesian networks [Pea88].

One distinctive application of Bayesian network which is very important 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.

Initially the research in this area 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 [Loc99, SJK00]. Bayesian troubleshooter showed their effectiveness in many applications. For this reason, they have been selected as a modelling technique to build the diagnosis agent used in this research.

4 Data set

In order to teach a probabilistic model, it is necessary to prepare a data set which represents the analyzed domain. In particular, parameters of Bayesian networks can be estimated based on the available a data set consisting of fully or partially observed instances of the random variables.

Although Continuous Integration systems produce a lot of data there was no standard data set which could be used in this research. 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. Finally run it for a sufficiently long period of time to collect enough samples for the research.

4.1 Format

The structure of the network and consequently the set of random variables that need to be observed was not known upfront to the research. Therefore, it was vital to design a data set format which preserves the relevant source data and makes it possible to observe variables on demand [KF09]. In the research 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.

4.2 Collection scenarios

Data samples can be collected in various scenarios depending on the broken build management policies. Analysis performed as part of the research revealed that the backward-fix scenario presented in Figure 1 is the most convenient for data collection. Its main advantage is that it can be fully automated. Users do not need to enter the information about the problem into any external systems because every action they take is tracked in the Version Control System and can be programmatically accessed from there.

Data set used in this research was collected in this way 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.

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

5 Diagnosis procedure

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.

Proposed diagnosis procedure uses a Bayesian troubleshooter at its core, however, there are many other important steps that have to be performed before the network can be constructed. These steps are described in the remaining part of this section.

5.1 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 which contains information 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*, which is one of the most popular utility used in this space [Mec04].

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. 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 directed 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 such as the file system access operation performed and the set of files used to build it.

5.2 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 their 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.

5.3 Extract information about 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 on the first error.

5.4 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.

5.5 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 [KF09]. The structure of the model is presented in Figure 2. 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.

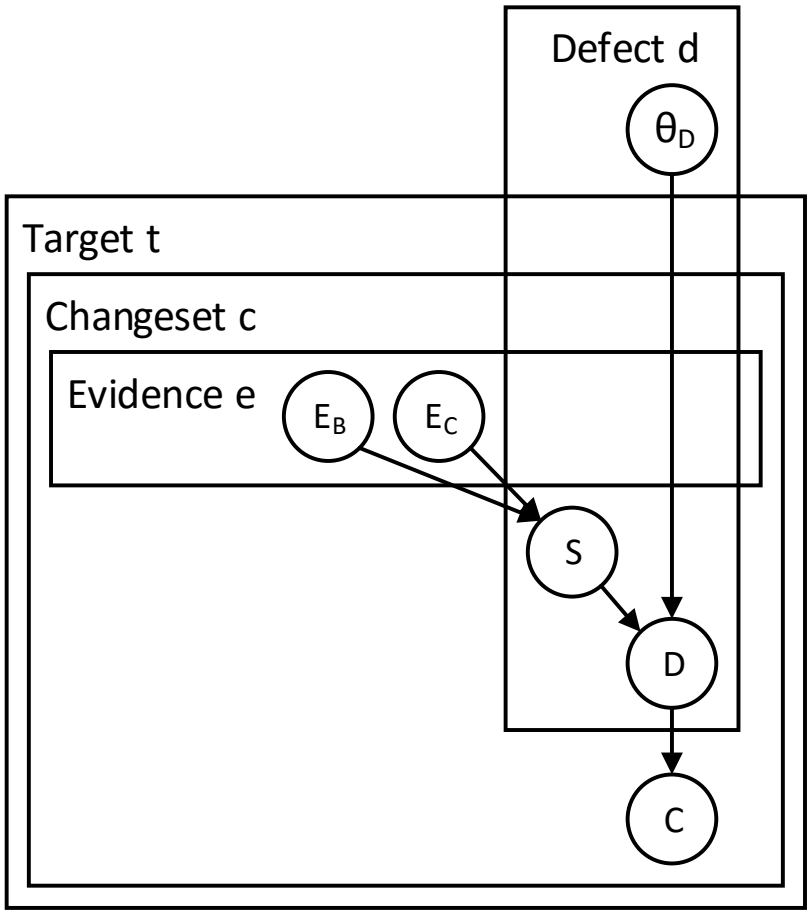

Figure 2. Plate model of the Bayesian network used for diagnosis

- $\theta_D(d)$ - 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*.

5.6 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 the value the variable should be set to. They are expressed with an imperative code and executed by the diagnosis agent.

5.7 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 E_B and the vector of prior parameters ϑ_D. This task can be formally defined as:

$$P(C(t,c)|E_B,\vartheta_D)$$

5.8 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. With this additional class of information the probabilistic query introduced in the previous section can be updated by including more conditioning random variables:

$$P(C(t,c)|E_B,E_C,\vartheta_D)$$

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.

5.9 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 appears at the top.

6 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 approach is that prior knowledge can be incorporated at the beginning and the model can be automatically retrained when more samples are collected.

Defects are modelled with Bernoulli's distributions with prior parameters coming from 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 $D[1],\ldots,D[M]$ can be described with equation:

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

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.

7 Study of the effectiveness

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. The expectation was that the quality will improve as more defect types are supported by the system. Moreover, model is designed to incorporate the prior expert knowledge, thus, this aspect is also included in the study.

By looking at how the output of the model can be interpreted one can identify five main outcomes that have to be taken into consideration. They depend on the *threshold* defining minimal value of probability at which the agent automatically performs an action on behalf of the user.

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.

When the model is used purely as a decision support system one might not be interested in the possible outcomes, but instead focus on the detailed report with a list of candidates ordered by the probability of having introduced a defect. In this problem 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.

In order to cover all the relevant aspects of the model the study was focused on both outcome analysis and culprit position analysis. It was also divided into two main stages. First, the model was studied without prior expert knowledge with so called uninformative priors, which are defined as assignments to the hyperparameters which maximize the information brought by the data. Second incorporated expert knowledge gathered from a survey filled out by people involved in the process of broken build management. Configuration of all the cases included in the research was summarized in Table 1.

Case	Prior expert knowledge	Defect types count		
		General	Specific	Total
A	**false**	**1**	**0**	**1**
B	false	3	0	3
C	false	7	0	7
D	**false**	**11**	**0**	**11**
E	false	1	10	11
F	false	1	20	21
G	**false**	**1**	**30**	**31**
H	**false**	**11**	**30**	**41**
I	**true**	**11**	**0**	**11**
J	**true**	**1**	**30**	**31**

Table 1. Supported defect types counts in analyzed cases

7.1 Baseline analysis

Single general defect type included in case A is example of a 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.

With good understanding of all the possible outcome rates we can present them all in a single area plot in Figure 3. 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.

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

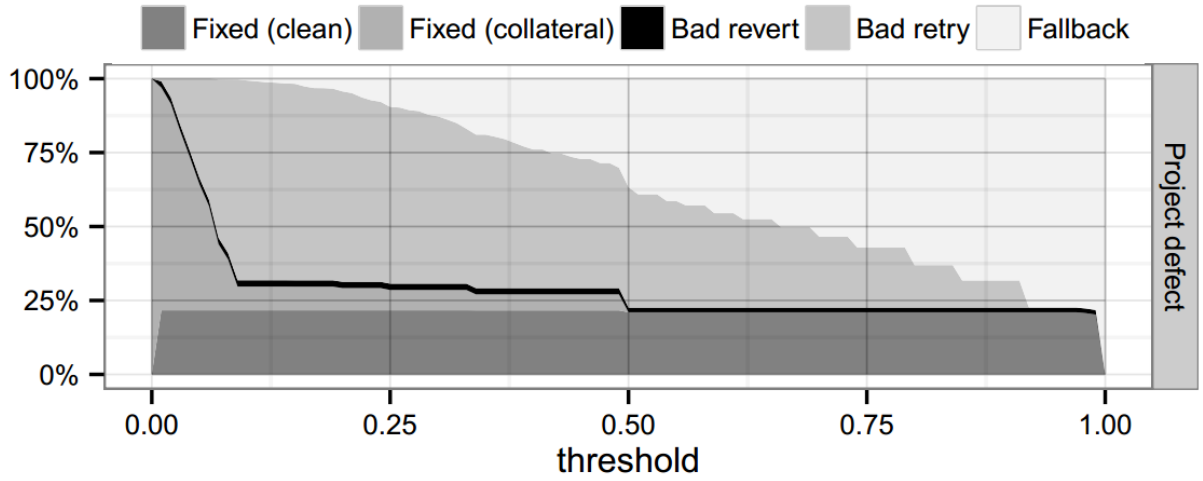

Figure 3. Outcome rates for baseline model

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. Similar pattern can be observed in the Precision and Recall plot in Figure 4.

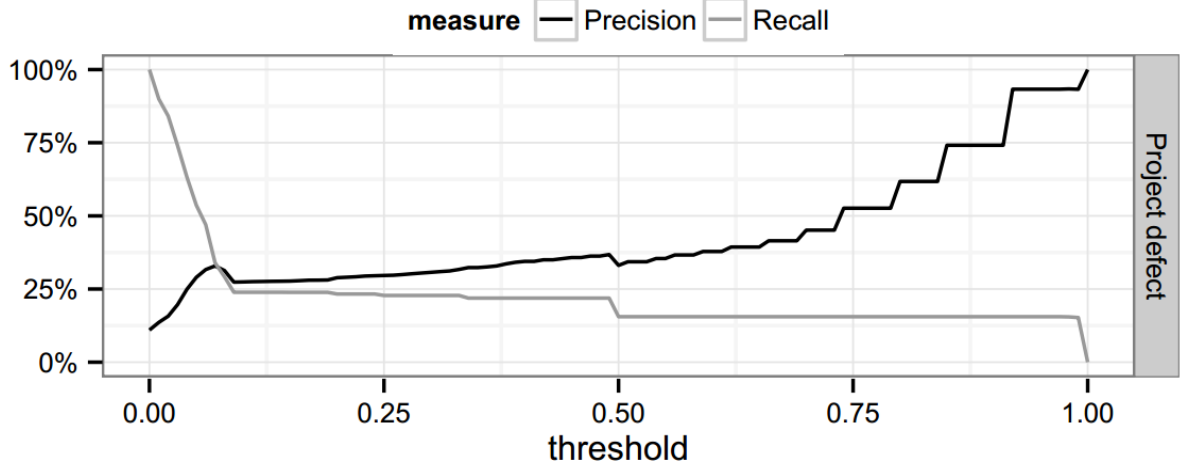

Figure 4. Precision and recall plots for baseline model

When the model is used as a decision support system position of real culprits is critical. Its distribution the baseline model is presented in the histogram in Figure 5. The first bar for project defects reaches only 15%, which not surprisingly matches the level of recall for high values of threshold from Figure 4. 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.

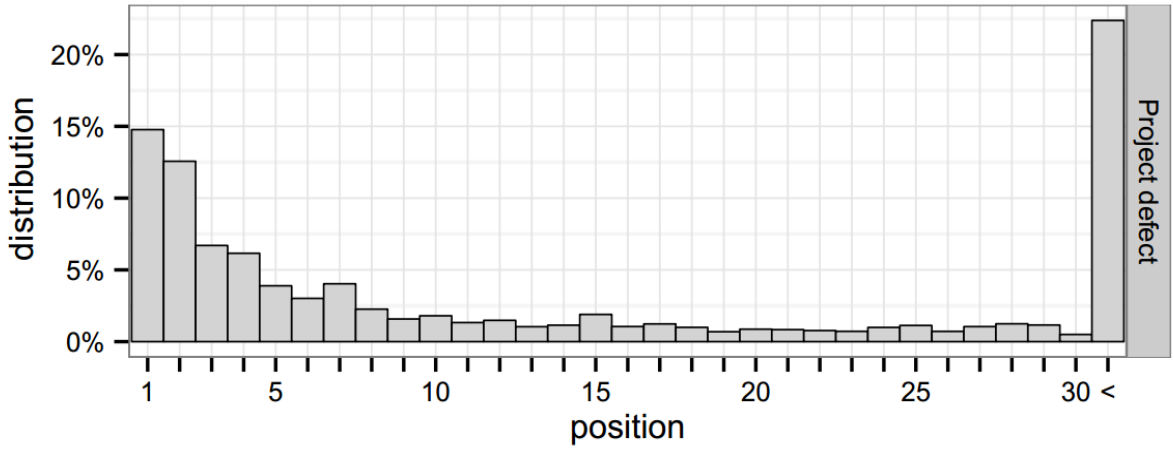

Figure 5. Histogram of culprit positions for baseline model

7.2 Mixed model

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.

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.

Model can also be extended by adding support for specific defect types. In comparison to general defects 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.

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 1. Because we need to compare many run at the same time the area plot presented in Figure 3 has to be replaced with a generalized version. Figure 6 shows outcome rates in stacked bar charts grouped by the value of threshold.

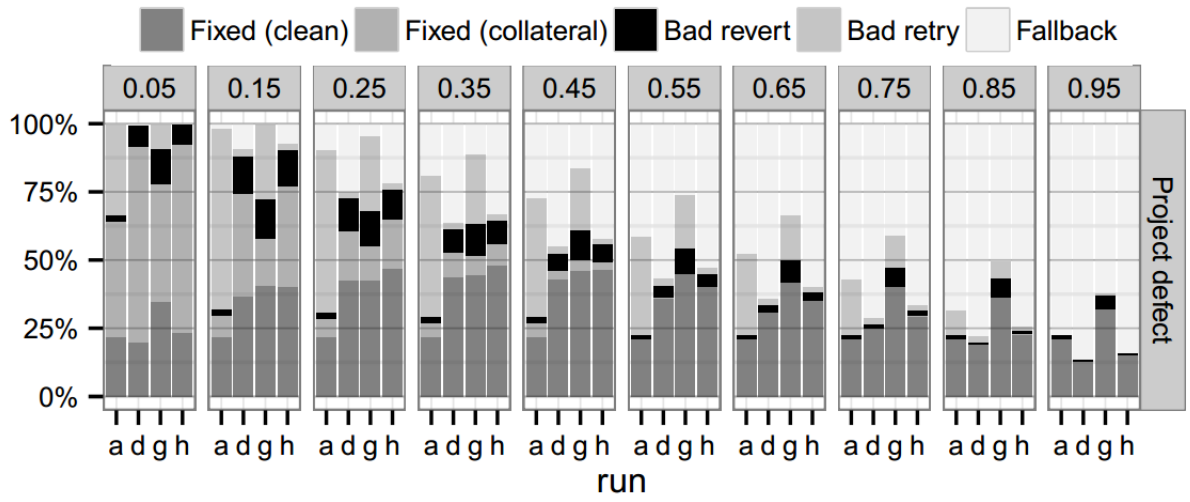

Figure 6. Outcome rates for mixed model with both types of defects

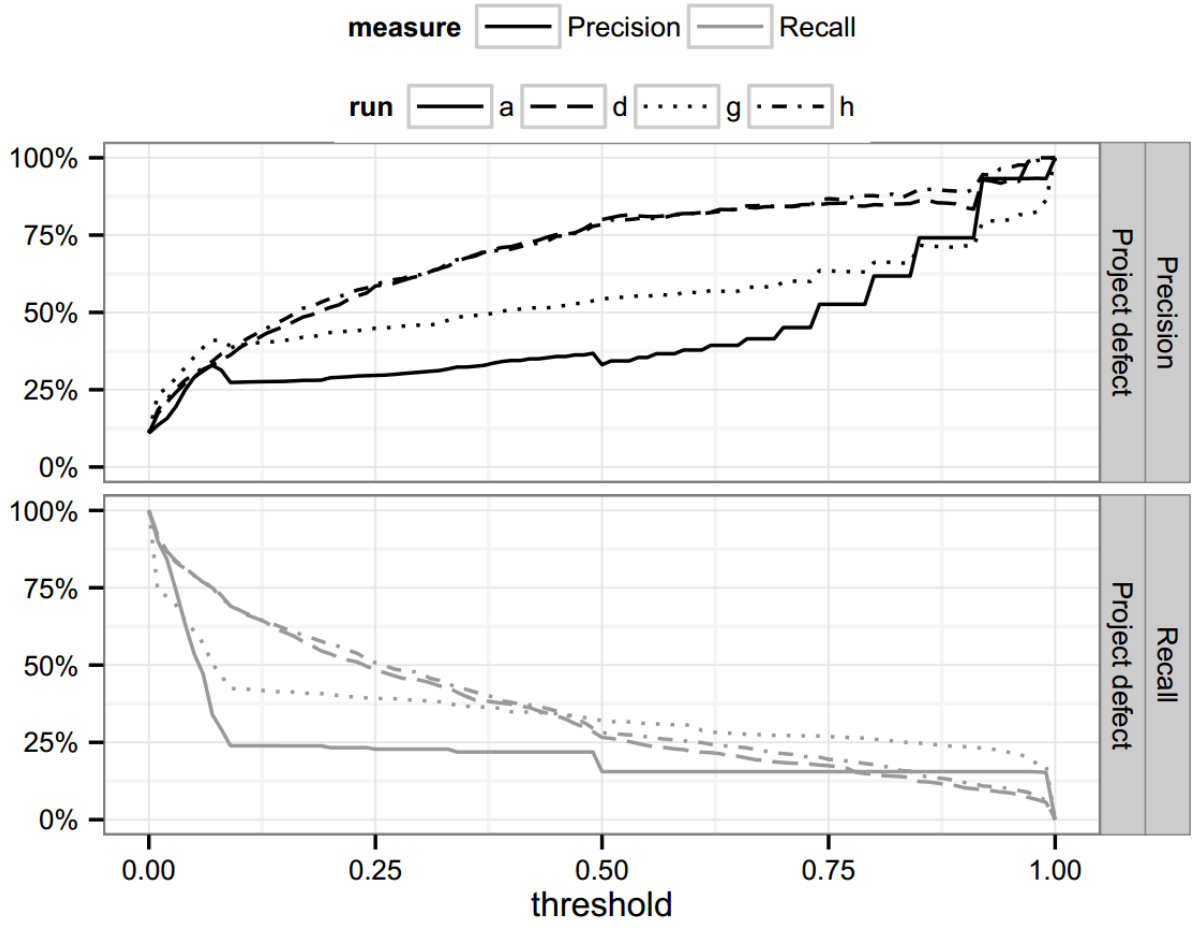

Figure 7. Precision and recall plots for mixed model

For mixed model H as well as the major models with general defect types D and specific defect types G 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.

Synergy between general and specific defect types is also visible in Precision and Recall charts in Figure 7. In both facets the curves for case *H* lies in between cases *D* and *G*. As it turns out combining defect types leads to a balanced model.

Different pattern emerged in the cumulative distribution of culprit position in Figure 8. Instead of laying in between the curve for mixed model outperformed all the other cases. The change from *D* to *H* is much smaller than from *A* to *G* because defects overlap in certain builds. Nevertheless, it is a clear indication that adding diversity to the set of supported defect types can improve the overall efficiency.

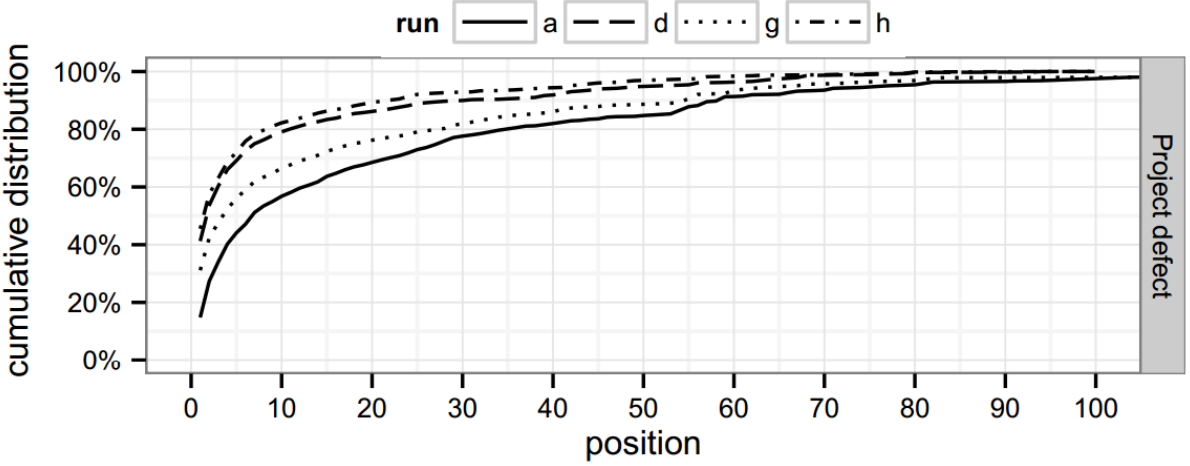

Figure 8. Cumulative distribution plot of culprit positions for mixed model

7.3 Model with 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.

In this study we will do a comparison of two pairs of selected models., each with different properties. First are *D* and *I* models which support only general defect types. Then, *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 as introduced in Table 1.

Outcome rates for the analyzed models were summarized in Figure 9 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.

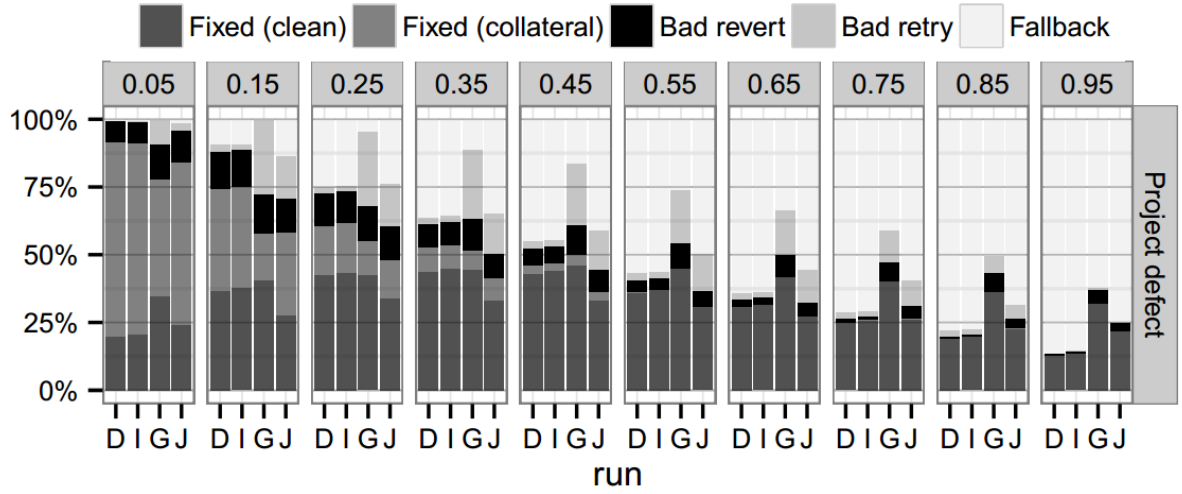

Figure 9. Outcome rates for model with prior expert knowledge

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 Section 6. 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, they can be supported only by few samples and their distribution gets 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 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. Defect 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 indicates a problem in the system, thus its 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.

8 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.

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.

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.

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.

Bibliography

[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.

[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.

[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.

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

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

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

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

[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.

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