CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Continuous Integration in Component-Based Embedded Software Development: Problems and Causes

Master's Thesis in Software Engineering and Technology

Ívar Gautsson
Þórhildur Hafsteinsdóttir

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2017

Master's thesis 2017

Continuous Integration in Component-Based Embedded Software Development: Problems and Causes

ÍVAR GAUTSSON
ÞÓRHILDUR HAFSTEINSDÓTTIR

Department of Computer Science and Engineering
Division of Software Engineering and Technology
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2017

Continuous Integration in Component-Based Embedded Software Development: Problems and Causes
ÍVAR GAUTSSON
ÞÓRHILDUR HAFSTEINSDÓTTIR

Supervisor: Miroslaw Staron, Department of Computer Science and Engineering
Advisor: Jesper Kullgren, Delphi Automotive Systems Sweden AB
Examiner: Eric Knauss, Department of Computer Science and Engineering

Master's Thesis 2017
Department of Computer Science and Engineering
Division of Software Engineering and Technology
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2017

Continuous Integration in Component-Based Embedded Software Development: Problems and Causes
ÍVAR GAUTSSON
ÞÓRHILDUR HAFSTEINSDÓTTIR
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Background: Continuous integration is a widely used practice in which software developers are expected to do frequent changes to a common code base. Potential integration issues can therefore be identified early on and fixed quickly. However, adopting continuous integration may sometimes be difficult, e.g. when developing embedded software, since specialized hardware is needed to run and test the software.

Aim: The goal of this study is to assess what problems come with the use of continuous integration in the context of component-based embedded software development together with identifying the causes for those problems. The continuous integration system being used for the development of an automotive infotainment head unit software system is analyzed, and the main problems and their causes are then identified.

Method: The aforementioned continuous integration system is modeled using the Cinders framework to gain an understanding of the system. Interviews with seven employees working on the project reveal what problems they are facing with regards to continuous integration. Ishikawa diagrams are used to show the main problems and some of their causes.

Results: Four main problems associated with using continuous integration in the development of component-based embedded software are identified: late discovery of defects, the overall integration process takes too much time, the system build breaks too often and interrupted development flow. The causes for these problems are also identified.

Conclusion: This study reveals some of the continuous integration related problems in the development of a component-based embedded software project. The results might not be relevant to all projects using continuous integration as some of the results are specific to component-based and/or embedded software development. Only one software project was under study and the results might therefore not be generalizable.

Keywords: Agile software development, continuous integration, component-based software, embedded software, Cinders

Acknowledgements

Our time at Chalmers University of Technology has been both enjoyable and rewarding. We would like to extend our gratitude to all our teachers in the Software Engineering program.

Miroslaw Staron, who was our teacher and supervisor, deserves our utmost thanks for his invaluable help and guidance. We are also grateful to our teacher and examiner, Eric Knauss. His course on the fundamentals of Agile software development was a useful preparation for this thesis.

We offer our thanks to Delphi Automotive for providing us with the opportunity to do this thesis work. The Gothenburg office welcomed us and served us delicious breakfast every morning. We are deeply grateful to our supervisor at Delphi, Jesper Kullgren. Special thanks go to Lars-Christian Aadland and all other employees at Delphi, especially the participants of this study.

Big thanks also to our fellow students at Chalmers, especially Martin Calleberg, Linus Hagvall and Ómar Þór Ómarsson.

Finally, we would like to thank Gunnar Hinrik Hafsteinsson and Þorvaldur Gautsson, who read the thesis and proposed improvements.

Ívar Gautsson & Þórhildur Hafsteinsdóttir, Gothenburg, June 2017

Contents

List of Figures

List of Tables

Abbreviations

API	Application Programming Interface
ASIF	Automated Software Integration Flow
CI	Continuous Integration
CIViT	Continuous Integration Visualization Technique
IHU	Infotainment Head Unit
KPI	Key Performance Indicator
SDK	Software Development Kit

1 Introduction

Large software projects generally require multiple developers to collaborate with each other. In most cases, this means that more than one developer is working on the same code base in parallel. At some point in time, the developers need to integrate their individual code changes, which can be an incredibly demanding and risky process [12]. Developers often have no way of knowing how long this integration phase will take, so uncertainty levels can therefore be high [12]. One practice that has proven useful in combating this problem of integrating code from multiple developers is called *continuous integration* (CI) [12].

Continuous integration means that the members of a team integrate their work frequently, e.g. once a day or more often [12]. Each integration is then verified by an automated build, which usually includes compiling and testing the software, allowing integration errors to be detected early on [12]. Risk reduction is often claimed to be the main benefit of continuous integration, since integration problems are addressed immediately and in smaller increments [11, 12]. The risk of introducing defects also becomes reduced [11, 12]. By running tests continuously, it becomes possible to find and fix bugs *before* they are introduced to the software [11]. It also becomes easier to assess the health of the software project as a whole, since the practice enables projects to generate working software many times a day [11].

Adopting continuous integration can have its difficulties. Duvall [11] claims that continuous integration should preferably be implemented early in a project. It is also possible to adopt it late in a project, but he finds that doing so can lead to people being under more pressure and more likely to resist change. Some research has been done in the field of continuous integration with regards to the challenges and problems that can appear when companies adopt the practice [9, 18, 21, 25]. However, more research is needed to assess the problems that can emerge and how they can be mitigated, especially when working with software systems combined with electrical and mechanical systems [21].

A systematic literature review on continuous integration (and other continuous practices) found that a considerable amount of the studies in this area lack contextual information [33]. Many papers do not provide any information on the domain or type of application that continuous integration is being used to develop, and organizational factors such as size and domain are also frequently left out. They recommend that future research should include more contextual information, as it will likely improve the quality and credibility of research in this area [33]. This thesis therefore includes a section in which contextual factors are stated, i.e. Section 1.4.

1.1 Problem Statement

The company Delphi Automotive Systems Sweden AB has somewhat recently started using continuous integration for one of their Infotainment Head Unit (IHU) projects. Their goal is to adopt *continuous delivery* in the near future, but in order to achieve that they need to fix the problems they are currently facing with regards to their continuous integration system [25]. One of those problems is that system builds are frequently broken, which has a negative impact on the development flow. Another problem is that the lack of automation for system tests requires developers to do manual regression testing, which can be quite time-consuming.

1.2 Purpose

The purpose of this study is to investigate the main problems associated with using continuous integration in the development of component-based embedded software together with finding the causes for those problems. The architecture of a continuous integration system being used for an IHU project is modeled with *Cinders* [37], which is an architecture framework for modeling continuous integration and delivery systems. That model, together with unstructured and semi-structured interviews with members of the IHU project, is used to identify continuous integration related problems and their causes. This results in a model which consists of a number of *Ishikawa* diagrams (also known as cause-and-effect diagrams). This model can help the case company identify how the continuous integration system can be improved. By identifying the main problems and their causes, it becomes easier for the company to fix or mitigate those problems.

Nilsson et al. [24] claim that visualizing testing activities in a model can help with identifying key improvement areas. Bosch and Ståhl [37] similarly state that modeling continuous integration systems can help with the identification and planning of improvements. This means that modeling the continuous integration system, which has never been done before in the case company, can help the company identify what needs to be improved. Identifying and understanding continuous integration related problems that a component-based embedded software project is experiencing will make a contribution to the field of continuous integration. Other organizations that wish to adopt continuous integration or are facing similar problems might also benefit from the results of the study. Additionally, since Cinders has never been used before in published research, this will be the first study using that framework to model a continuous integration system.

1.3 Research Questions

The research questions that are addressed in this thesis are:

RQ1: What are the main problems associated with using continuous integration in the development of component-based embedded software?

RQ2: What are the main causes for the problems associated with using continuous integration in the development of component-based embedded software?

In RQ1, the main problems associated with using continuous integration in the development of component-based embedded software will be listed. In order to understand why these problems occur, the causes of the problems need to be identified, which in turn addresses RQ2.

1.4 Infotainment Head Unit Project

This study is done in collaboration with the company Delphi Automotive, which provides technological solutions to the automotive and transportation sectors. The company is quite large, with over 173.000 employees and a presence in 44 countries. In this study, the continuous integration system being used for one of the projects at Delphi is studied. The project in question is an infotainment head unit system designed for automotive vehicles. The software being developed is a component-based embedded operating system and has been under development for approximately one and a half years. Many different teams located in several countries are working on the project and there are over 250 employees working on it, most of whom are engineers.

1.5 Outline of the Paper

The remainder of this thesis is organized as follows. In Chapter 2, the theoretical framework is provided, and in Chapter 3, related work is presented. The research design applied to this study is presented in Chapter 4 and the results are then shown in Chapter 5. A discussion is provided in Chapter 6 and Chapter 7 includes the concluding remarks, where a summary of the main findings is presented.

2 Theoretical Framework

Definitions of important concepts are provided in this chapter. First, continuous integration is discussed in general in Section 2.1. Then, the concepts of component-based software and embedded software are defined in Section 2.2 and Section 2.3, respectively. Next, the architectural framework Cinders is presented in Section 2.4. Finally, the Ishikawa diagram is described in Section 2.5.

2.1 Continuous Integration

The term *continuous integration* comes from the Extreme Programming development process and is one of its original practices [12]. As mentioned before, continuous integration requires team members to integrate their work frequently, e.g. once a day or more often. This means that it is possible to detect integration errors quickly, since each integration is verified by an automated build which usually consist of compiling the code and executing tests as well. Many developers think that this practice results in more rapid software development and fewer integration problems [12]. Continuous integration has received a lot of attention as of late from both industry and research. There is plenty of information available on this topic, for instance Martin Fowler's 2006 article *"Continuous Integration"* [12], which has been influential in this field and is still a useful resource. Before continuous integration is explained in more detail, it is worth noting that many implementation variations exist and disagreements are common on many aspects of the practice [36].

According to Fowler [12], the adoption of continuous integration usually entails having some specific tools in place, as shown in Fig. 2.1. He states that one of the tools needed is a version control system, which makes it possible to store source code in a common repository. The version control system manages versions of the software being developed and stores each version as a commit in the repository [27]. In other words, commits are used by version control systems to keep track of the changes that have been made to a software project [22]. Another tool that Fowler finds to be helpful is a continuous integration server. The continuous integration server monitors the repository for changes and takes care of verifying that those changes did not break anything. This verification is done every time a developer pushes code to the repository and usually involves an automated build [12]. Quite often, there is more than one type of build being done, so it can be confusing to just say "the build". Usually, there is a *commit build* which is run by the continuous integration system for every single new commit [12]. This build should provide fast feedback to developers, and ten minutes is often stated as a guideline for how long it should take [12], although a literature review by Laukkanen and Mäntylä suggests that a build time of two minutes is optimal [19]. To keep this build fast, usually only small or medium tests are included in it. Developers are therefore not blocked for long and can be notified when the build is done [47]. Another common build is the *second stage build*, which includes more end-to-end tests and other time-consuming tests [12]. This build is not run for every single commit, since it can take quite a lot of time, so it is usually run periodically, e.g. every 10 minutes or every N commits [47]. Setting up a good continuous integration infrastructure is, however, not enough. Some changes also need to be made to the development workflow. Everyone working on the team needs to ensure that new changes both pass the build and also have a minimal impact on the production environment [23].

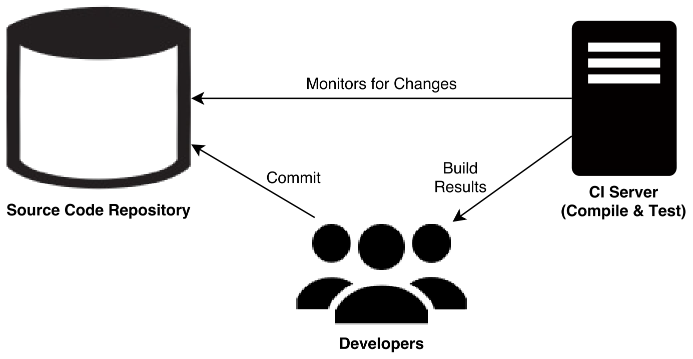


Figure 2.1: Basic continuous integration infrastructure.

Testing in continuous integration should be automated and executed continuously, which helps to find defects quicker. Both low-level and high-level testing can be conducted in this context. Low-level tests are usually white box tests, e.g. unit tests, while high-level tests are often gray or black box tests that test the components of the system as a whole and how they work together, e.g. GUI-based testing [4]. Duvall [11] claims that in continuous integration, unit tests, component tests, system test and functional tests should be automated. They state that unit tests verify small elements in a software system whilst component tests verify bigger portions. This means that component tests have more code coverage and take a longer time to run than unit tests. These tests may also require some external dependencies or even the fully installed system. System tests are tests that verify the complete software system, which means that such tests require a fully installed system [11]. Functional tests, which are sometimes called acceptance tests, are defined as tests that verify the functionality of the system from the client's viewpoint [11].

2.2 Component-Based Software

Fully adopting continuous integration seems to be a struggle in many projects, especially large ones, and Bosch and Ståhl [38] argue that breaking down large software systems is a key enabler for continuous integration at scale. One way to break down a software system is to use a *component-based software architecture*. In component-based software development, the software is built from multiple components which can be developed independently from each other [17].

There are many different definitions of what a software component is, and some even claim that the term "component" cannot be defined [7]. Szyperski [44] claims that a component has three characteristic properties. The first property is that *a component is a unit of independent deployment* [44]. To achieve independence, a component therefore needs to be separated well from both other components and its environment as a whole. The fact that a component is a unit of deployment means that a component is never deployed partially — only as a single unit. The second property of a component is that *a component is a unit of third-party composition* [44]. This means that a component has to come with specifications of how it can be used, since it might have been developed by a third party. Interfaces must therefore be well-defined, so that it is clear what services a component requires to function properly, and also what services it provides to its environment. A simple component diagram can be seen in Fig. 2.2, which shows how two components communicate through their interfaces. The third and final property of a component is that *a component has no (externally) observable state* [44]. This stateless nature of components means that it would make little sense to load more than one copy of the same component into a system, since the copies would be indistinguishable from each other [44].

When component-based software is being developed, there is a risk that software components and applications have different kinds of requirements, which can result in components not satisfying the application requirements and vice versa [6]. Crnkovic [6] states that when changes are made on the application level (e.g. components are updated), then there is a risk that the changes cause system failure. Therefore, component-based software tends to be quite sensitive to changes [6]. This could be considered a challenge when using continuous integration, since frequent changes to the code base is one of the key factors in continuous integration.

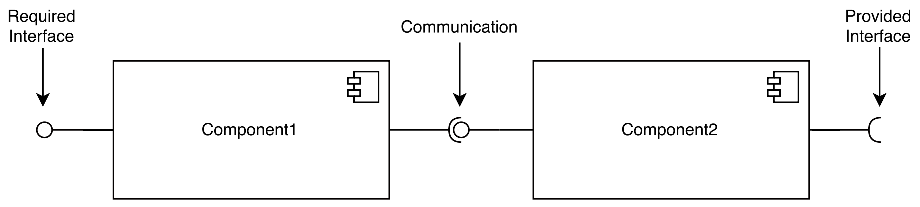


Figure 2.2: In this example component diagram, Component2 provides an interface (i.e. some service) to Component1.

As mentioned in Section 2.1, there are commonly two builds in continuous integration: a commit build that is fast and a second stage build which executes more thorough tests. This can, however, be done differently when the software architecture is modular. For example, if the software is component-based, then each component can have its own independent build and continuous integration processes, and another build can then be triggered when a new component dependency is made available [29, 40].

2.3 Embedded Software

Embedded software is software written for embedded systems [43] and Mårtensson et al. [21] define software-intensive embedded systems as software systems combined with electrical and mechanical systems. In the past, embedded software was not considered to be as important as it is today, as the software was just something that was quickly developed when the hardware was ready [43]. The software was then typically developed by someone who knew the hardware well. Nowadays, systems have become more complex which has resulted in using software specialists instead to write the code for embedded systems. Walls lists several things that one has to bear in mind when working in embedded software development and they are listed below.

- **Memory size**. Embedded systems have limited memory and because of power and cost consumption demands, excess memory is not an option. This is different for developers who are working with desktop systems as they often do not have to think about memory, mainly because memory is cheap and operating systems can expand it using hard drive space.
- **CPU power**. The amount of CPU power for embedded systems is often only just enough for the function required, which has an effect on code and operating system efficiency. This is due to strict cost and power consumption requirements. Desktop computers, on the other hand, have CPUs that are cheap and developers do not have to worry about heat dissipation and power consumption.
- **Code optimization**. Developers working with embedded software need to develop optimized code, but it depends on their needs what they want to optimize. Controllability is, for instance, a key feature of an embedded compiler. Code optimization is also performed by developers who are working with desktop systems. However, embedded systems and desktop applications often have different priorities, where the latter would probably focus more on e.g. the speed of the software.
- **Operating system**. An embedded system may have an operating system which could be a special variant of Windows or Linux, an in-house developed operating system or one of many real-time operating systems. A desktop computer, on the other hand, would probably only have Windows or Mac OS X as its operating system.
- **Real time behavior**. Most embedded systems are real time, which means that the system is predictable-deterministic and not necessarily fast. Desktop systems, on the other hand, are seldom real time.
- **Development paradigm**. For embedded systems, code is developed on the "host" (desktop computer) and then later executed on the "target" (the embedded system itself). This is different for desktop applications since the software is usually developed and executed on the same machine. The execution phase is also more complex for embedded systems than desktop computers because the software must either be run under some kind of simulation environment or be transferred to a target.
- **Execution paradigm**. The execution of software for most embedded systems

goes on continuously from start-up until the device is powered down. This is different for desktop applications since the execution of software depends on what the user wants.

- **Every embedded system is different**. Embedded systems can be different in so many ways, e.g. different operating systems, memory architecture etc. This means that embedded software developers not only have to learn how to work close to the hardware and real-time systems, but also that they might need to learn about different operating systems, a selection of development tools, multiple CPU architectures and so forth. This may be the biggest difference between embedded and desktop systems; desktop systems have only a few variations, while every embedded system is different.

2.4 Cinders

Cinders is an architectural framework designed specifically for the purpose of describing continuous integration and delivery systems created by Ståhl and Bosch [37]. It contains four architectural viewpoints, each of which represents a unique aspect of the continuous integration and delivery system. These viewpoints are:

- **Causality Viewpoint**: Causality implies that one action or task causes some other action or task to be triggered. This viewpoint is therefore used to represent triggering relationships in continuous integration and delivery systems. The Causality Viewpoint's meta-model can be seen in Fig. 2.3.

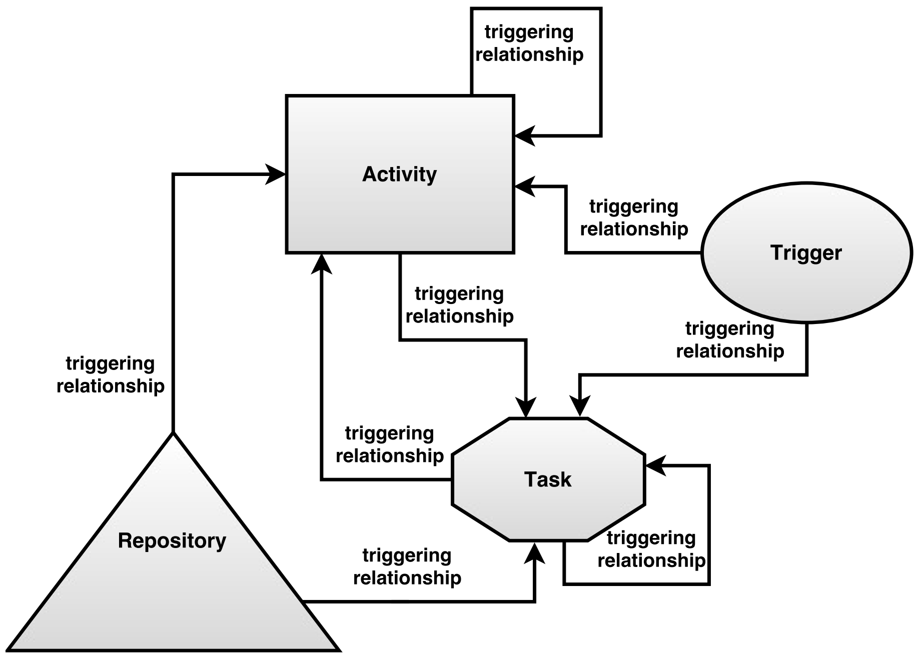


Figure 2.3: A meta-model of the Causality Viewpoint in Cinders.

- **Production Line Viewpoint**: This viewpoint is similar to the Causality Viewpoint, but instead of showing the causal relationships between activities and tasks, it shows consuming relationships. The purpose of this viewpoint is to show how artifacts, i.e. source code revisions and their derivatives, pass through the system. Some other differences from the Causality Viewpoint are that repositories, tasks and activities are described in more detail and external triggers are omitted. The meta-model for the Product Line Viewpoint is shown in Fig. 2.4, and Table 2.1 describes what each base set in the viewpoint is.

Table 2.1: The base sets used in the Product Line Viewpoint.

Base set	**Class**	**Definition**	**Values**
pre-integration-procedure	Repository	The steps that are needed before new code is integrated with the integration target	**one or more of** {none, review, queue, tests, automated}
integration-target	Repository	The kind of storage or branch that the repository node represents	**one of** {private, team, development, release}
automation	Activity and Task	The degree of automation	**one of** {none, supporting, full}
functional-confidence	Activity	How much confidence the activity provides in regards to functional aspects	**one of** {none, some, extensive}
non-functional-confidence	Activity	How much confidence the activity provides in regards to non-functional aspects	**one of** {none, some, extensive}
queue-time	Activity	The minimum and maximum time a triggered activity spends in a queue, before it can execute some activity	$\{t_{min}, t_{max}\}$
duration	Activity	The minimum and maximum time spent executing some activity	$\{t_{min}, t_{max}\}$
system-completeness	Activity	The degree of system completeness at which the activity operates	**one or more of** {unit, partial, full, customer on-site}
lead-time	Activity and Task	In a fully automated system, this would be the sum of duration values and upstream queue-time. If manual triggering is involved, then this value might not be as meaningful	$\{t_{min}, t_{max}\}$

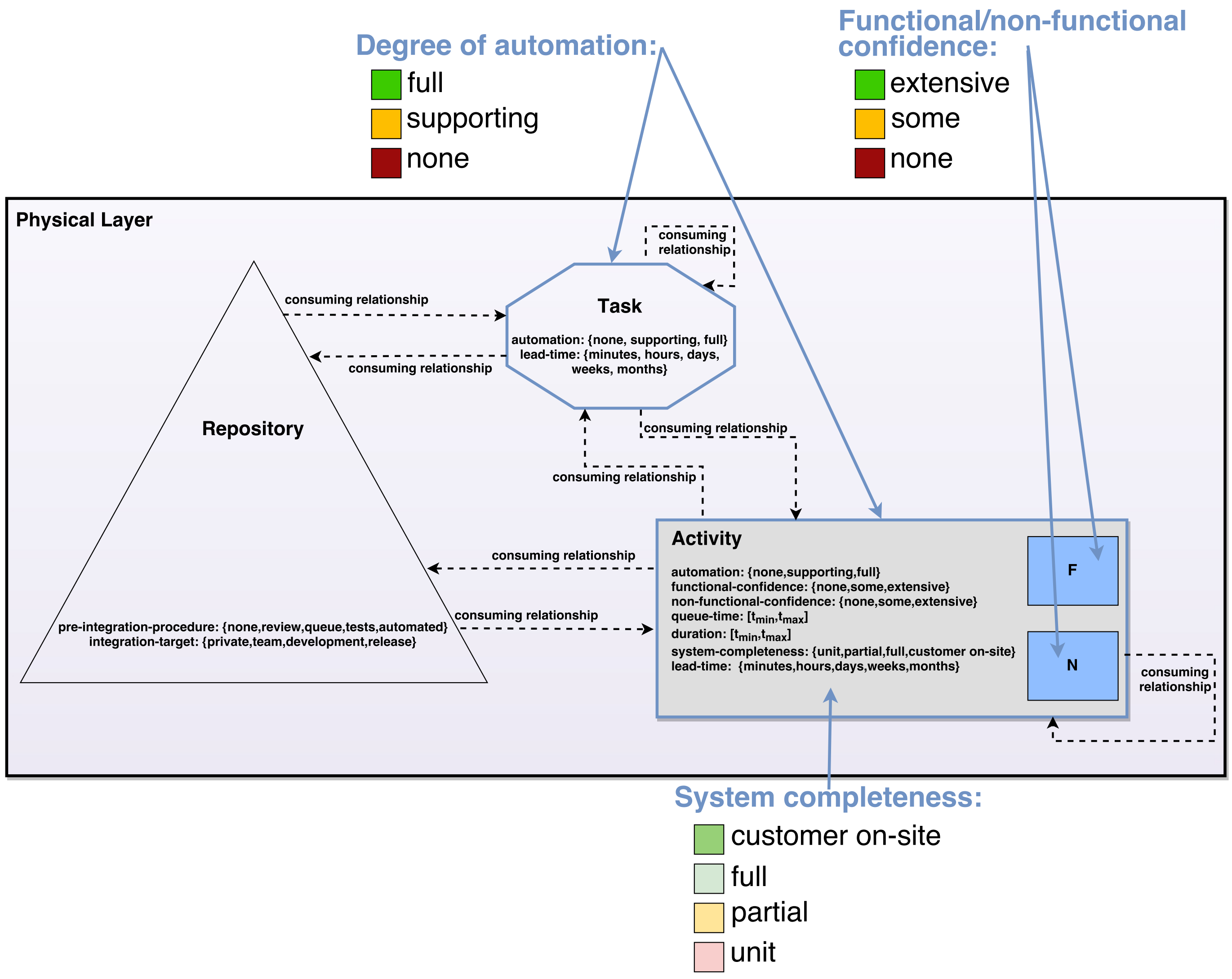

Degree of automation:
🟩 full
🟨 supporting
🟥 none
Functional/non-functional confidence:
🟩 extensive
🟨 some
🟥 none
Physical Layer
consuming relationship
Task
automation: {none, supporting, full}
lead-time: {minutes, hours, days, weeks, months}
Repository
pre-integration-procedure: {none,review,queue,tests,automated}
integration-target: {private,team,development,release}
Activity
automation: {none,supporting,full}
functional-confidence: {none,some,extensive}
non-functional-confidence: {none,some,extensive}
queue-time: [tmin,tmax]
duration: [tmin,tmax]
system-completeness: {unit,partial,full,customer on-site}
lead-time: {minutes,hours,days,weeks,months}
F
N
System completeness:
🟩 customer on-site
🟩 full
🟨 partial
🟥 unit

Figure 2.4: A meta-model of the Product Line Viewpoint in Cinders.

- **Test Capabilities Viewpoint**: Provides an overview of which test activities are performed. This viewpoint should indicate the level of system completeness and the feedback lead time for each test activity. The meta-model for this viewpoint can be seen in Fig. 2.5.
- **Instances Viewpoint**: This is the most detailed view and is supposed to illustrate implementation details. It is considered to be an optional part of the framework and in the paper where Cinders is presented, no example illustration is provided for this viewpoint.

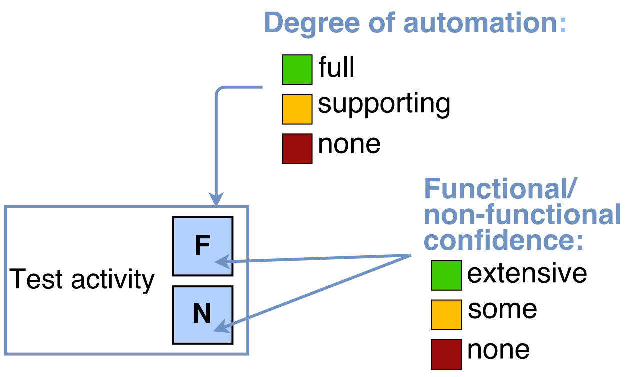


Figure 2.5: A meta-model of the Test Capabilities Viewpoint in Cinders.

2.5 Ishikawa Diagram

The Ishikawa diagram, which is also known as the cause-and-effect diagram, shows the relationship between the cause and the effect. In other words, it is a causal diagram which lists possible causes for a particular effect. The effect could, for instance, be a quality characteristic or a problem [13]. Fig. 2.6 shows what this type of diagram looks like, where each cause of a specific problem is categorized.

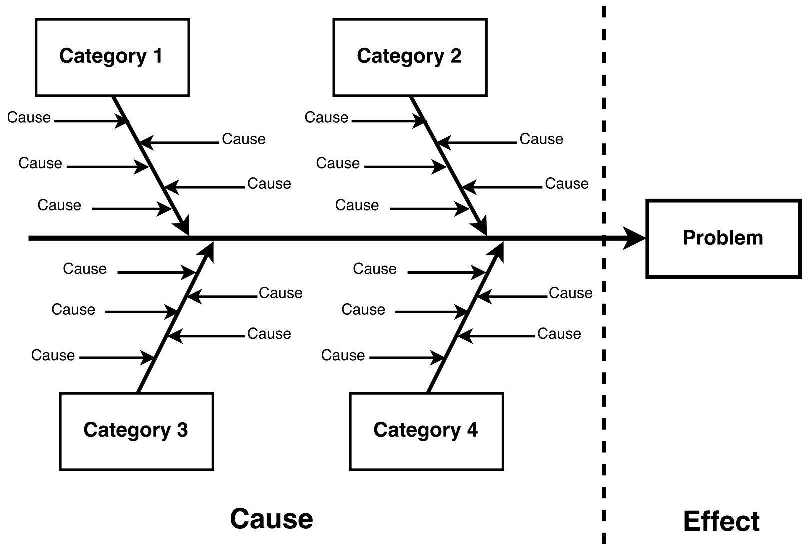


Figure 2.6: An Ishikawa diagram (also known as a cause-and-effect diagram).

3
Related Work

Several literature reviews have been published on continuous integration or related topics. Some of them were useful for finding research that is relevant to this thesis and will therefore be briefly discussed. One of these literature reviews, which was done by Bosch and Ståhl [36], discusses the continuous integration implementation variations that are to be found in published research. They claim that the practice is interpreted and implemented differently from case to case and that there is currently no consensus on continuous integration as a single, uniform practice. They then state that it would be worthy to investigate if any contextual factors of software projects influence which variation gets chosen. Another literature review by Shahin et al. [33] discusses different approaches, tools, challenges and practices in the areas of continuous integration, delivery and deployment. They found that a considerable amount of the studies in these areas lack contextual information such as the domain or type of application being developed. Organizational factors such as size and domain are also frequently left out. Future research should in their opinion include more contextual information, as it will likely improve the quality and credibility of research in these areas. Yet another systematic literature review by Laukkanen, et al. [18] focuses on the problems that arise when adopting continuous delivery. They identify the causes of some of the problems and solutions are even provided in some cases. The problems they found are divided into seven themes: build design, system design, integration, testing, release, human and organizational, and resource.

As mentioned previously, the aspects of continuous integration are often defined differently. One source of variation is how build failures and successes are defined. Commonly, a build is considered to have failed if any test fails during the build [1, 28]. However, that is not always the case. For example, Rogers [30] proclaims that for most development teams, it is fine to allow acceptance tests to break over the course of an iteration, as long as all tests pass before the end of the iteration. Other requirements sometimes need to be fulfilled before a build is considered a success, e.g. there is a certain level of test coverage [46] or a specified threshold for some metric is not exceeded [15]. Another requirement for a successful build could be that there is an absence of severe code analysis warnings [15]. Some sources also mention that for a build to succeed, the compilation must succeed as well [10,15,46], although that usually seems to be implicitly assumed. There is not a lot of research that explores why builds fail and how they can be prevented from breaking when using continuous integration. Kerzazi et al. [16] investigated the main factors impacting build failure. A quantitative analysis that they did found that build failures correlate with the number of simultaneous contributors on the branch, the type of work performed on a branch (feature development, bug fix, etc.), the build type (integration build vs. continuous build), and the roles played by the stakeholders of the builds. They also did a qualitative investigation, which showed that the typical circumstances under which a build breaks are missing referenced files, mistakenly checking in work-in-progress, and transitive dependencies. Storm [41] proposes a solution to mitigate the impact of build failures in the context of component-based software development. His solution uses backtracking, which means that if the build of a component fails, then other components that depend on it will be built using the previous successful build of the component that had the build failure. His reasoning for why this is a good solution is that any build is better than no build at all.

An industrial experience paper by Kim et al. [17] introduces an integration procedure and automated integration system for a software project with hundreds of components. In that paper, there were three main lessons that they learned from implementing their system. The first lesson was that users need to be trained. They state that developers and component maintainers need to be pressed to closely monitor the integration status and also need to respond to defects found during the integration. One idea that they like is to punish component maintainers when their components have trivial defects. They also say that component maintainers should be encouraged to do frequent small releases, and avoid adding lots of features altogether, since that makes it harder to detect the source of a problem. The second lesson was that the component maintainer's task should be automated. Component maintainers can often do simple mistakes or skip basic tests because of overconfidence. The third lesson relates to the testing environment. If developers set up their own testing environment, then that could lead to problems for others running different environments. Therefore, the same testing environment should be provided to everyone with the actual integration.

The dependency between components and component interfaces makes modularization difficult and is a challenge that Bosch mentions in his book [4]. This also makes development teams highly dependent on each other. He also mentions that it is difficult to have automated testing and daily builds when working with embedded systems involving hardware with slow development cycles. Applying continuous integration to software systems combined with electrical and mechanical systems seems to be challenging, but research in this area is scarce. A paper by Mårtensson et al. [21] lists which factors must be taken into account when applying continuous integration for the development of these kinds of systems. They define issues within seven topics, i.e. long build times, complex user scenarios, compliance to standards, many technology fields, security aspects, test environments and architectural runway.

According to a case study by Debbiche et al. [9], coordinating integration dependencies became more difficult after the adoption of continuous integration. In that study, four different issues related to the adoption of continuous integration were reported by developers. The first one was that component interfaces needed to be more clearly defined. The second one was that it was harder to locate the source of errors during integration, because code is delivered from different teams. The third one was that more failures were experienced during integration. The fourth and final one was that there was a need to wait until other components or parts were done before integrating work.

Another case study by Olsson et al. [25] analyzed a company that was transitioning to continuous integration. That company was dependant on code from many different suppliers, which one developer thought had a negative effect on the company's development speed. Several interviewees also mentioned that fitting different components from different suppliers was time consuming, so not only was the development lead time long, but also the integration of components. The paper mentions that one common barrier is the dependency between components and the dependency between component interfaces. This means that separation is difficult, and development teams are therefore very dependant on each other. The paper also mentions that when transitioning towards agile development, there should be a shift to small development teams and a focus on features instead of components.

Modeling continuous integration systems has many benefits according to Bosch and Ståhl [37]. They found that modeling such systems can improve understanding and also help with identifying improvement areas. The authors of this paper found two techniques for modeling continuous integration systems and one architectural framework. Automated Software Integration Flow (ASIF) is a descriptive model created by Ståhl and Bosch [34], which shows automated activities, together with nodes that represent external triggering factors and inputs, that are connected with input and triggering relationships. Continuous Integration Visualization Technique (CIViT) is a technique created by Nilsson et al. [24] that visualizes the testing activities performed around a product or a product platform. This technique aims to give an overview of end-to-end testing activities, which can prevent or mitigate problems such as duplicate testing efforts and slow feedback loops. In another paper by Ståhl and Bosch [35], they applied both the CIViT and ASIF continuous integration modeling techniques to four separate industry cases in three companies and investigated what impact it would have. The Cinders framework, which was introduced in another paper by Ståhl and Bosch [37], is based on both CIViT and ASIF, and is described in Section 2.4,

4 Research Design

The main goal of this study is to find out what continuous integration related problems the case company is having. The research method used in this thesis is *design science research.* In design science research, an artifact is created [42], which in this case is a model. This model consists of a number of *Ishikawa* diagrams which show the main continuous integration related problems and their causes. According to Ishikawa [13], identifying the relationship between cause and effect of a problem makes it possible to take an action to solve it. The case company can therefore utilize the model to improve its continuous integration system. In addition, researchers can utilize the model to see what problems a real world software project has faced with regards to continuous integration and what the causes of these problems are.

Another improvement method, *action research*, is often a viable alternative to design science research. In action research, changes (or actions) are made in a problematic situation and data collection techniques need to be used before, during and after the action-taking stages [8]. This method was not chosen, because no direct actions were performed to change the continuous integration system at the case company. In addition, data collection was only performed once, i.e. to identify the main problems and their causes. Furthermore, according to Baskerville [3], design science research emphasizes on solving problems by creating an artifact, which in this study is a model, whilst action research emphasizes on solving problems through social and organizational change.

Design science research will be explained in more detail in Section 4.1. Then, the plan of the study will be described in Section 4.2. Finally, the data collection methods and data analysis methods used in this thesis work will be described in Section 4.3 and Section 4.4, respectively.

4.1 Design Science Research

According to a paper by Vaishnavi and Kuechler [39], design-science research focuses on creating new knowledge by designing an artifact and analyzing that artifact, in order to improve and understand some specific parts of information systems. The same paper also mentions that design science research is sometimes called "Improvement Research", which puts emphasis on its problem-solving or performance-improving nature. The design science research process model is visualized in Fig. 4.1, where an overview is presented of how such research is conducted.

The following five process steps are followed in this thesis work: awareness of problem, suggestion, development, evaluation and conclusion. Awareness of problem

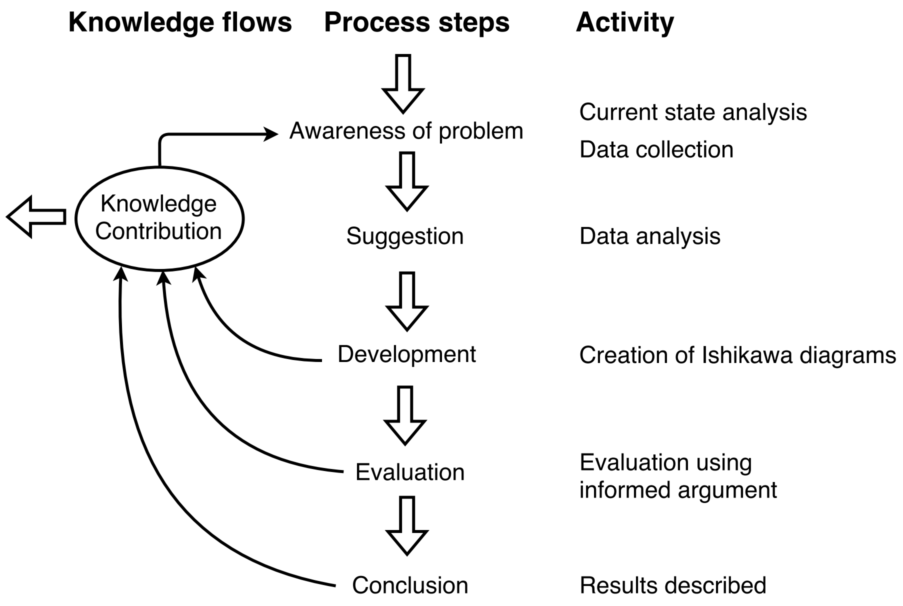


Figure 4.1: An overview of the design science research methodology. Activities that were performed for this thesis are mapped to specific steps in the process.

is the first process step and it involves two activities: a current state analysis and data collection. The current state analysis results in a model of the IHU project's continuous integration system created using Cinders. The aim of the data collection activity, which includes documentation analysis, unstructured and semi-structured interviews, is to identify any problems that employees in the IHU project have when they use the continuous integration system. In the next step, i.e. the suggestion step, the data analysis is performed which results in identifying the main problems of the continuous integration system and also identifying the causes for those problems. In the development step, the problems and their causes from the previous step are further investigated and a number of Ishikawa diagrams are created, which is the model. This model is then evaluated in the evaluation step with a descriptive method called *informed argument*, where information from relevant research is used to build a convincing argument for the model's utility [42]. The literature used in the informed argument method is collected using the *backward snowballing* approach. In that approach, highly cited studies, e.g. systematic literature reviews, are identified and used to find more literature from the reference list [45]. The evaluation of the model is presented in Section 6.1. Then, the final step is the conclusion step, where the results are described. The knowledge gained from the process, i.e. design science knowledge, may well serve as the subject of further research [39].

During this thesis work, the seven guidelines for design science by Hevner et al. [42] are followed. The first guideline is designing an artifact and according to the second guideline, the artifact must be created for a specified problem domain. Guideline 3 demonstrates that an evaluation of the design must be made. Guideline 4 states that a design science research must provide clear and verifiable contributions in the areas of the design artifact, design foundation, and/or design methodologies. According to Guideline 5, design science research requires the application of rigorous methods in both the construction and evaluation of the designed artifact. This entails that appropriate data collection and analysis methods will be used. Guideline 6 demonstrates that a design is a search process where many different design alternatives are generated and the goal is to find the most effective one that meets the requirements and constraints. According to guideline 7, the result of a design science research method should be communicated effectively both to technical and managerial audience.

4.2 Plan of Study

In this section, the research plan is presented. This study is split into two phases: a current state analysis phase and a problem identification phase. In the first phase, the continuous integration system used in the IHU project is analyzed. In the second phase, continuous integration related problems that the project is facing are presented and some of their causes identified.

4.2.1 Current State Analysis

This thesis is not just an analysis of the current state of the continuous integration system architecture being used in the IHU project. The focus is rather on finding out what the project's main problems are with regards to continuous integration, together with identifying their causes. However, the authors of this paper feel that the current state of a continuous integration system's architecture has to be understood well before the main problems and their causes can be identified. This subsection therefore explains how the current state of the system was analyzed.

In order to understand the workflow and the continuous integration system being used in the IHU project, unstructured interviews were conducted with two members of the integration team and documents created by the same team were also analyzed. In case some of the documents had outdated information, more discussions were held with the integration team to acquire correct information. The creation of a model of the current state of the continuous integration system helped gain more understanding of it and the intended workflow. This model was created using *Cinders*, which is described in Section 2.4. A reason for choosing Cinders, but not the other continuous integration modeling techniques, i.e. ASIF or CIViT, is that Cinders has been proven to be an improvement over those method, as it combines the best from the other two techniques. Cinders represents the integration flows in two different viewpoints (the Causality Viewpoint and the Product Line Viewpoint) and the overview of the test activities performed in another one (the Test Capabilities Viewpoint). These viewpoints are rendered from the same underlying data model and therefore give a more complete overview of the continuous integration system than ASIF and CIViT alone [37].

4.2.2 Problems and Their Causes

Since RQ1 is about finding out what the main continuous integration related problems are in the IHU project, unstructured interviews were held with two members in the integration team and two developers working in that project. This led to identifying some areas of the continuous integration system that were of special concern. These areas were then addressed in the interview question written in order to find out which problems are the most serious ones. These questions were then used in semi-structured interviews conducted with various employees. The aim of these interviews was to shed light on the problems that the employees working in the project have faced or are currently facing with the continuous integration system, together with finding out what the main causes for these problems are, which in turn addresses RQ2. The data collected from the semi-structured interviews was analyzed with a theory generation method called the *constant comparison method*, which is a qualitative data analysis method [32]. This method is described in more detail in Section 4.4. This analysis resulted in several Ishikawa diagrams, which is the model. This type of diagram is explained in Section 2.5. A reason for choosing this diagram is that according to Ishikawa [13], it can be used for any problem. By knowing the relationship between cause and effect of a problem, then it becomes possible to take an action to solve it.

4.3 Data Collection

This chapter provides a general description of the data collection methods that are used in this thesis work.

4.3.1 Documentation Analysis

Documentation analysis involves analyzing documents generated by developers. These documents can be comments in the code or separate documents that describe a software system. The advantage of using this technique is that it can serve as an introduction to the software and the team. The documents written about the system can present a glimpse of at least one person's understanding of the software system. The main disadvantage of using this technique is that this can be time consuming and written material may be inaccurate [20]. The documents that were analyzed in this study are two documents written by the integration team:

- *Develop & Test Software* document, which explains the developer's work-flow for committing code and how to baseline a new version of a repository.
- *Continuous Integration* document, which summarizes the Continuous Integration and Testing work-flow in the SUT.

4.3.2 Interviews

An interview is a conversation where at least one researcher talks to at least one respondent at a time. The advantage of interviews is that they are highly interactive. The disadvantages are that interviews are time consuming and cost-inefficient. A researcher must schedule a meeting with the respondent which could take some time, since the respondent might be busy [20]. According to Runeson and Höst [31], interviews can be unstructured, semi-structured and fully-structured. In this thesis, unstructured and semi-structured interviews were conducted. Unstructured interviews are interviews where the interview conversation will develop based on the researcher's interest of the subject. The interview questions are therefore put together as interests and general concerns from the researcher. Semi-structured interviews involve predefined questions that are not necessarily asked in the same order as they are listed.

As was mentioned in Section 4.2.1, unstructured interviews were held with two members in the integration team in order to understand the continuous integration system being used in the IHU project and the workflow. After the current state had been analyzed, unstructured interviews were held again with the same members of the integration team and two developers working in the IHU project. Those interviews were conducted to identify which areas of the continuous integration system were of special concern and should thus to be addressed in the semi-structured interviews. Semi-structured interviews were then held to identify the main problems the employees have had with regards to continuous integration. Runeson and Höst recommend selecting interview subjects based on differences. This is preferable to selecting similar subjects in an attempt to replicate similarities. Therefore, employees working in the IHU project that hold various positions and are members of different teams were interviewed, as can be seen in Table 4.1. The interview questions can be seen in Appendix 1, together with Fig. A.1 and Fig. A.2 which were in some cases shown to the interviewees in order to make it easier for them to understand the interview questions. The interviews, which took 20-60 minutes, were recorded by both researchers in order to make sure that all the data was stored safely.

4.4 Data Analysis

The data analysis in this study is done using the *constant comparison method*. The constant comparison method is a theory generation method, which is used in qualitative data analysis. This method involves attaching labels or codes to text pieces that are relevant to an idea or a theme that is of interest in the study. The codes and subcodes that were assigned to the text pieces can then be used to group the text pieces into patterns. Next, a field memo is written in order to express a proposition from the coded data [32].

Table 4.1: Background information about the interviewees.

ID	**Position**	**Team**	**Years at the company**	**Software Development Experience [years]**	**CI Experience [years]**
A	Configuration Manager	Integration Team	5	5	5
B	Integrator	Integration Team	0.5	1	0.5
C	Scrum Master	T1	1.5	15+	8-9
D	Developer	T2	1.5	12	5
E	Developer	T3	0.5	16	15-16
F	Developer	T4	5	5-6	5
G	Developer	T5	9	9	3-4

5
Results

The findings of the study are presented in this chapter. First, the continuous integration system being used in the IHU project is explained and modeled with Cinders in Section 5.1. Then, the main problems that the project is facing with regards to continuous integration are listed, together with their causes in Section 5.2.

5.1 Current State Analysis

One of the traditional practices of continuous integration is that a single source repository should be maintained [12]. In the IHU project, each component has its own source repository, which includes all of the latest changes for that specific component. There is also a project-level repository, as shown in Fig. 5.1, which stores so-called recipes — one for each component in the project. A recipe is a file which provides a set of instructions on how some unit (e.g. a component or some other piece of software) should be built. These instructions state e.g. where the source repository is located, which commit should be used for building the unit and also what dependencies the unit has. A build tool can use such a recipe file to automatically build a single component or even the software project as a whole. The project-level repository described above is required when doing the full system build, as the recipes provide instructions on how to build every single component.

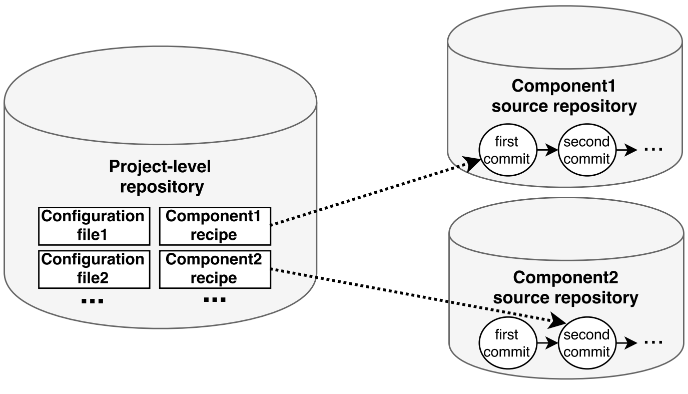


Figure 5.1: The project-level repository stores recipes, one for each component. In this example, Component1's recipe states that the first commit should be used for the system build, while Component2's recipe states that the second commit should be used.

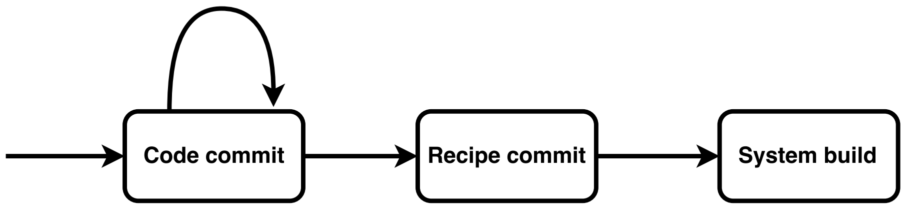


Figure 5.2: The three integration flows of the continuous integration system.

Cinders, which is described in the theoretical framework chapter, is an architecture framework used to model the current state of the continuous integration system architecture being used in the IHU project. In this framework, three architectural viewpoints are used to describe the system, i.e. the Causality Viewpoint, the Product Line Viewpoint and the Test Capabilities Viewpoint. The Causality Viewpoint and Product Line Viewpoint show three different integration flows. The first flow is for the code commit, i.e. when a developer commits code to a component's repository. The second flow is for a recipe commit, i.e. when a component maintainer commits a recipe to the project-level repository. The recipes are then used for the system build, which is the third flow. An overview of these three integration flows can be seen in Fig. 5.2. The aforementioned diagram shows that developers can do multiple code commits, and when the component maintainer decides to include these commits in the system build, then he or she does a recipe commit.

5.1.1 Causality Viewpoint

The Causality Viewpoint represents triggering relationships in the continuous integration system. The Causality Viewpoint for the *code commit*, which is on a component level, is shown in Fig. 5.3. This viewpoint shows what happens when a developer decides to check-in or "push" code changes to a component source repository. *Git* is the version control system being used, and the first thing that happens after a developer pushes his or her changes with Git is that a new change record is created in *Gerrit*. Gerrit is the system that is used for code reviews, since every code change needs to be accepted by other team members. The new change record in Gerrit then triggers two activities that work in parallel, i.e. the code review and the component build. Both of these activities need to succeed in order to trigger the merge activity. The component build is done under the target's Software Development Kit (SDK) using the build automation tool *Jenkins*. This build entails compiling and running unit tests for that component. When the code review is completed, i.e. either accepted or rejected, a status mail will be sent automatically to the committing developer. If the code review is rejected, then the committing developer needs to change the code and push a new commit. When the component build activity is finished, a mail is sent to the committing developer that informs him or her how that activity went. If there is enough time, then static code analysis is run using the tool *Coverity*. However, the static code analysis is not required and running it can take quite some time. Hence, developers sometimes stop it in order to speed things up. If Coverity gets the chance to finish the analysis, then a defect list

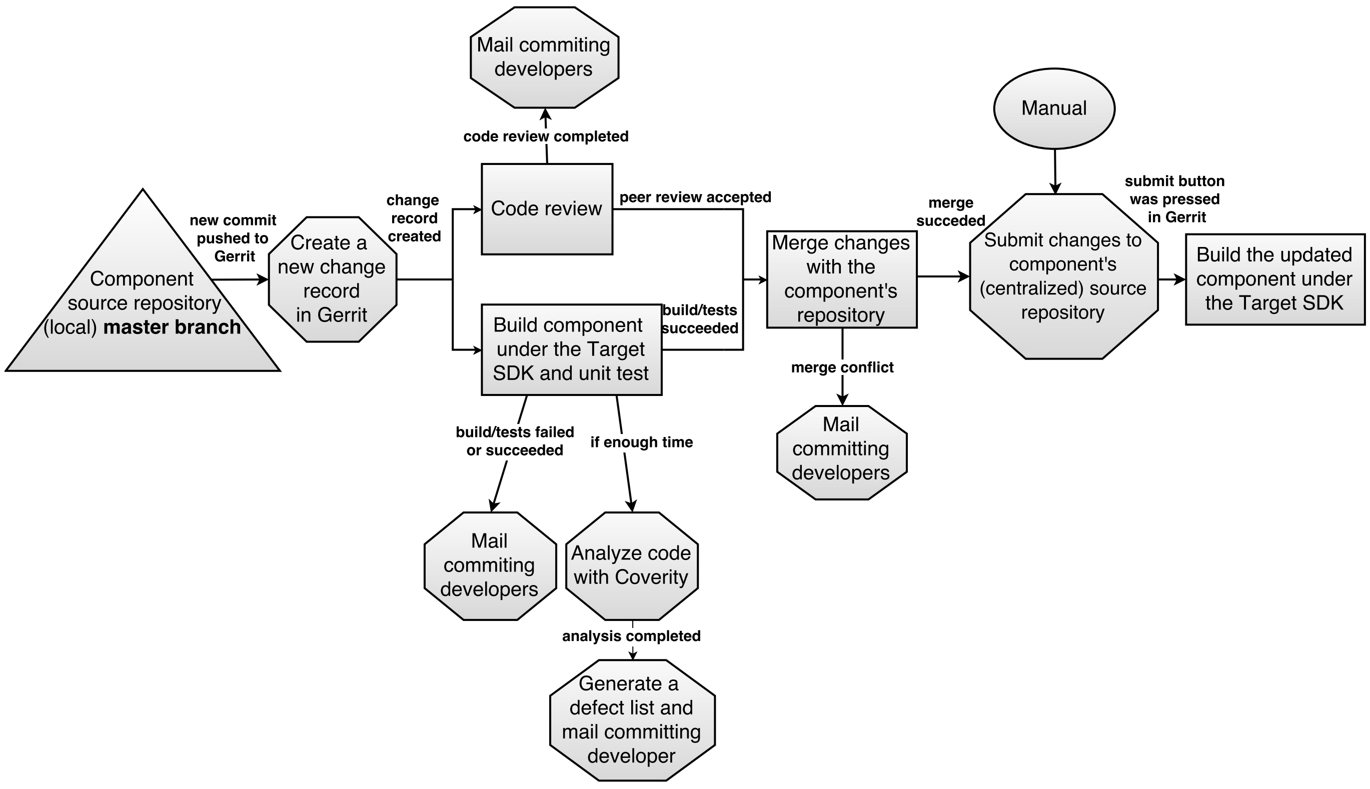

Mail commiting developers
code review completed
Manual
submit button was pressed in Gerrit
change record created
Code review
peer review accepted
merge succeded
new commit pushed to Gerrit
Create a new change record in Gerrit
Merge changes with the component's repository
Submit changes to component's (centralized) source repository
Build the updated component under the Target SDK
Component source repository (local) master branch
Build component under the Target SDK and unit test
build/tests succeeded
merge conflict
build/tests failed or succeeded
if enough time
Mail commiting developers
Mail commiting developers
Analyze code with Coverity
analysis completed
Generate a defect list and mail committing developer

Figure 5.3: Causality Viewpoint for a code commit.

is generated and the committing developer receives a mail. This defect list is only informative and does not have any effect on whether the change will be accepted or not. When the build, tests and code review have succeeded, the merge activity is triggered automatically. The merge activity attempts to merge the code changes with the component's repository. If there is a merge conflict, then it triggers a task where a mail is sent to the committing developer, telling him or her that there was a merge conflict. However, if the merge succeeds, then the committing developer must manually press the submit button, which submits the changes to the component's source repository on the centralized git server. This task then triggers a build, where the updated component is built under the target SDK.

In Fig. 5.4, the Causality Viewpoint for the *recipe commit* is presented. This viewpoint shows what happens when a component maintainer decides to "bump" a recipe, i.e. when he or she wants a new version of that component to be included in the system image. Component maintainers can have various roles; they can e.g. be developers, scrum masters or integrators. When the component maintainer commits an updated recipe for some component to the project-level repository, then a new change record is created in Gerrit. This new change record then triggers two activities that work in parallel, i.e. a code review and an incremental build on target software. The code review works in the same way as in the Causality Viewpoint for the code commit in Fig. 5.3, but it often takes less time, mainly because the recipe files are usually tiny and only have a few lines. It would take too long to do a clean system build for every single recipe bump, so an incremental build is done instead. This incremental build only builds the parts of the system

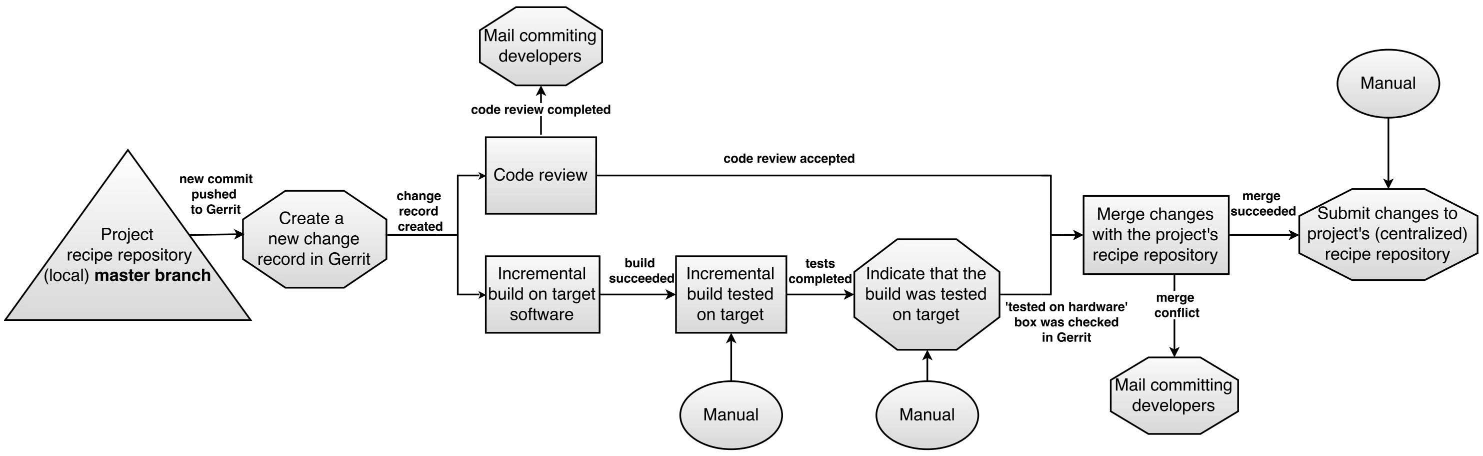

Mail commiting developers
code review completed
Manual
Code review
code review accepted
new commit pushed to Gerrit
change record created
Project recipe repository (local) **master branch**
Create a new change record in Gerrit
Incremental build on target software
build succeeded
Incremental build tested on target
tests completed
Indicate that the build was tested on target
'tested on hardware' box was checked in Gerrit
Merge changes with the project's recipe repository
merge succeeded
Submit changes to project's (centralized) recipe repository
merge conflict
Mail committing developers
Manual
Manual

Figure 5.4: Causality Viewpoint for the recipe commit.

that have changed, and uses cached builds when possible. Only components that were affected by the changes introduced in the recipe bump will therefore need to be rebuilt. The software being built could be one component, a couple of components or even the whole system. This task could take a long time if there is a long build queue, i.e. a queue of commits that need to be built. When the incremental build on the target software has succeeded, it triggers an activity where the incremental build is tested on target. As was mentioned in Section 2.3, a target is defined as an embedded system, which in this case is an IHU. This test is done manually and after that, the developer must indicate that the build was tested on target by manually ticking the "tested on hardware" checkbox. This checkbox is problematic, since developers can tick the "tested on hardware" checkbox without having tested the build on hardware. When both the code review has been accepted and the "tested on hardware" checkbox has been ticked, the merge activity can start. The merge activity merges the changes to the project's recipe repository on a centralized git. As in the previous viewpoint, in Fig. 5.3, if there is a merge conflict, a mail is sent to the committing developers, which is the component maintainer in this case. Otherwise, if the merge succeeds, then the component maintainer must manually press the submit button, which submits the changes to the project-level repository on the centralized git server.

The Causality Viewpoint for the *system build* can be seen in Fig. 5.5. This build is done once a day during the night, and is considered to be a *clean build*, which means that every single component is built from scratch, unlike the incremental build. It is important that this build succeeds, since a designated smoke test team needs it in order to perform daily smoke tests. If this build fails, then the smoke team has nothing to test. A broken system build also has an effect on the component teams, as will be discussed further in Section 5.2.3 and Section 5.2.4.2.

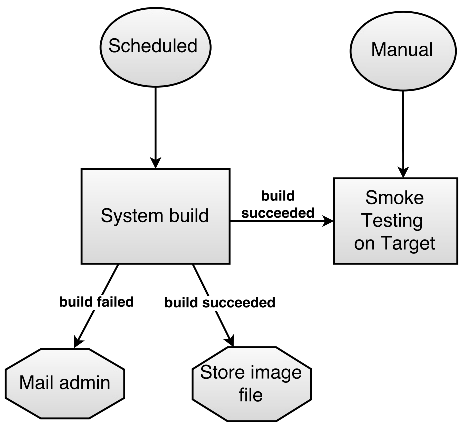


Figure 5.5: Causality Viewpoint for the system build.

5.1.2 Product Line Viewpoint

The Product Line Viewpoint, which is described in the theoretical framework chapter, shows how artifacts travel through the system. There are three Product Line Viewpoints for the case company's continuous integration system. The first Product Line Viewpoint is for a *code commit* and it can be seen in Fig. 5.6. This viewpoint describes the same integration flow as Fig. 5.3. The second one is for the *recipe commit* and it can be seen in Fig. 5.7. This viewpoint describes the same flow as Fig. 5.4. Lastly, Fig. 5.8 is a Product Line Viewpoint for the *system build* and it describes the same flow as Fig. 5.5.

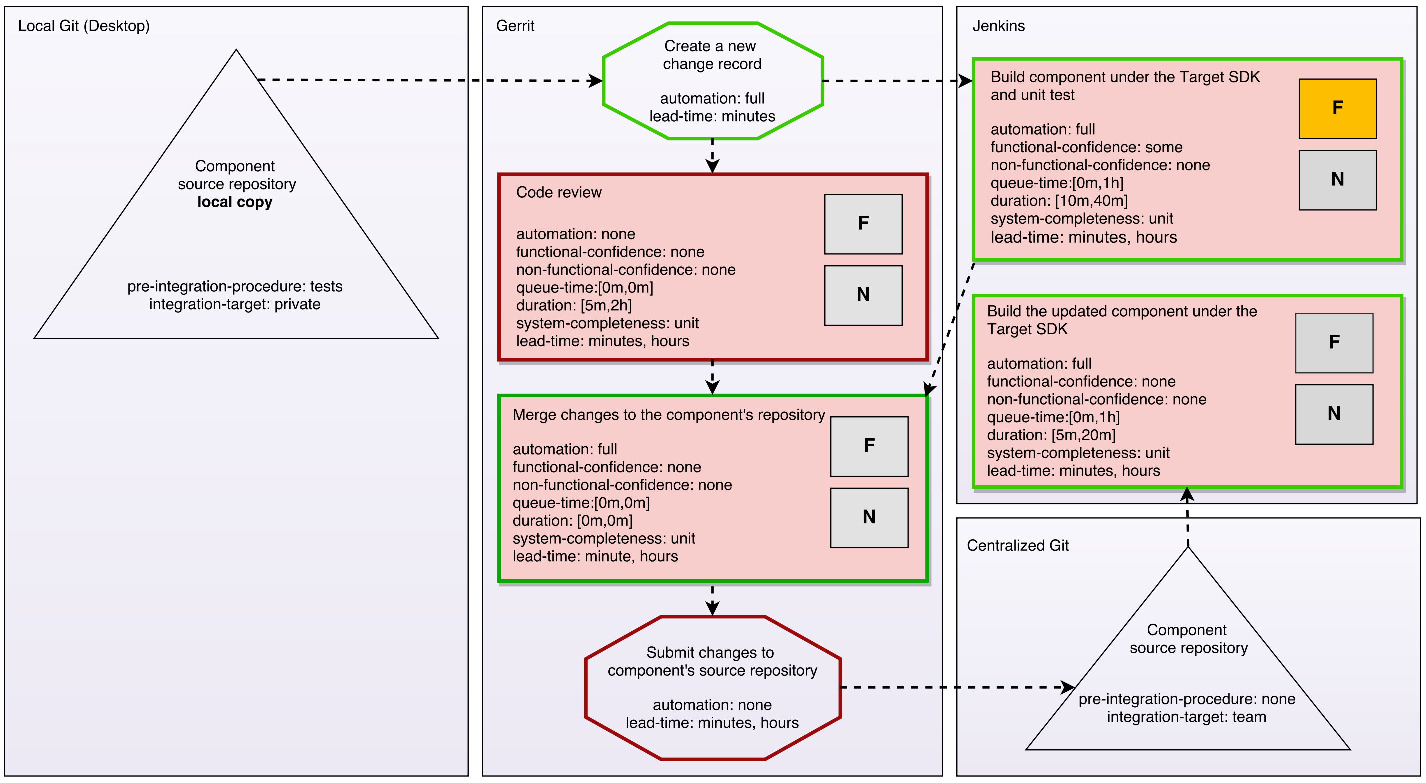

Local Git (Desktop)
Component source repository local copy
pre-integration-procedure: tests
integration-target: private
Gerrit
Create a new change record
automation: full
lead-time: minutes
Code review
automation: none
functional-confidence: none
non-functional-confidence: none
queue-time:[0m,0m]
duration: [5m,2h]
system-completeness: unit
lead-time: minutes, hours
F
N
Merge changes to the component's repository
automation: full
functional-confidence: none
non-functional-confidence: none
queue-time:[0m,0m]
duration: [0m,0m]
system-completeness: unit
lead-time: minute, hours
F
N
Submit changes to component's source repository
automation: none
lead-time: minutes, hours
Jenkins
Build component under the Target SDK and unit test
automation: full
functional-confidence: some
non-functional-confidence: none
queue-time:[0m,1h]
duration: [10m,40m]
system-completeness: unit
lead-time: minutes, hours
F
N
Build the updated component under the Target SDK
automation: full
functional-confidence: none
non-functional-confidence: none
queue-time:[0m,1h]
duration: [5m,20m]
system-completeness: unit
lead-time: minutes, hours
F
N
Centralized Git
Component source repository
pre-integration-procedure: none
integration-target: team

Figure 5.6: Product Line Viewpoint for the code commit.

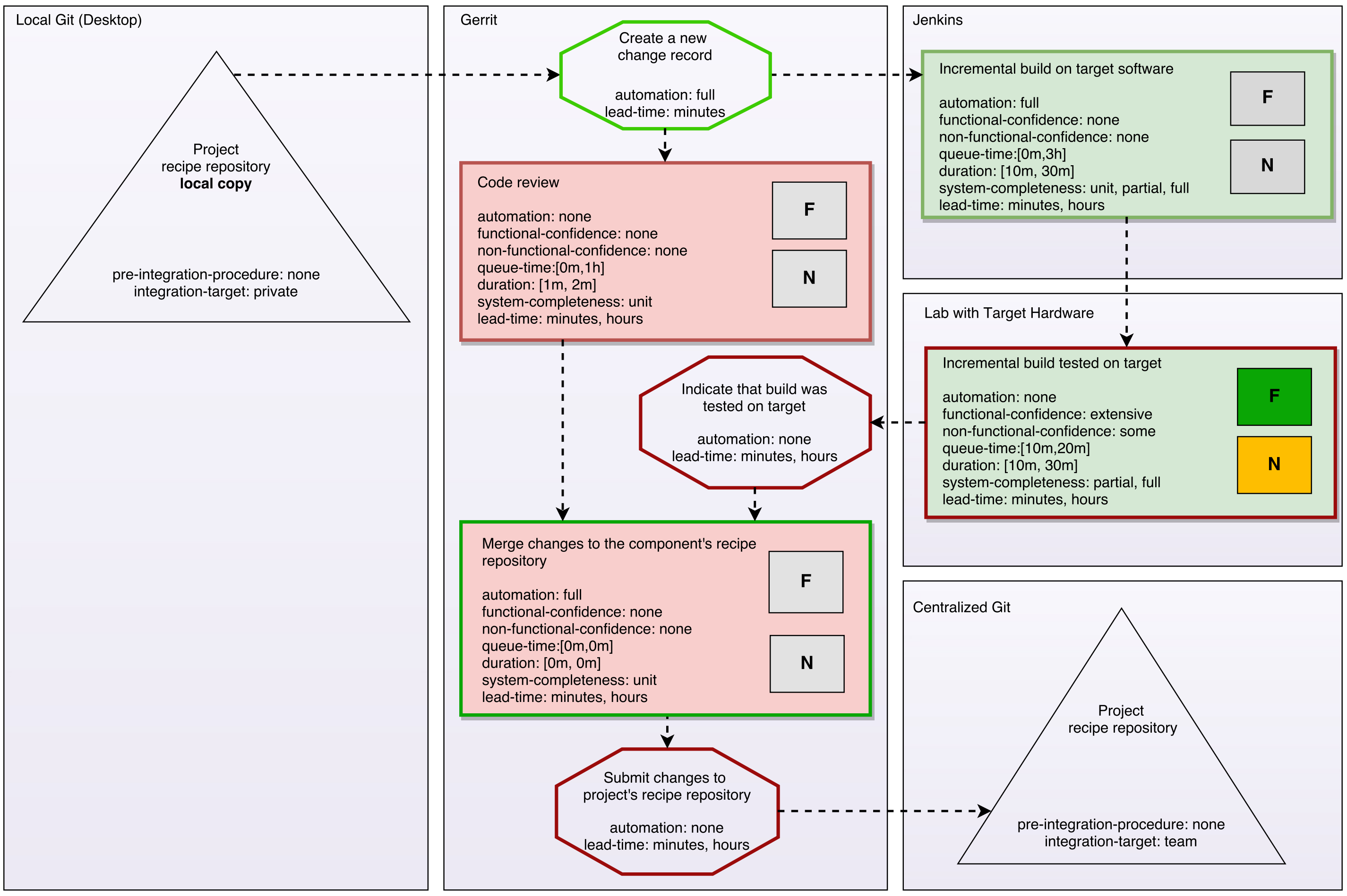

Local Git (Desktop)
Project recipe repository local copy
pre-integration-procedure: none
integration-target: private
Gerrit
Create a new change record
automation: full
lead-time: minutes
Code review
automation: none
functional-confidence: none
non-functional-confidence: none
queue-time:[0m,1h]
duration: [1m, 2m]
system-completeness: unit
lead-time: minutes, hours
F
N
Indicate that build was tested on target
automation: none
lead-time: minutes, hours
Merge changes to the component's recipe repository
automation: full
functional-confidence: none
non-functional-confidence: none
queue-time:[0m,0m]
duration: [0m, 0m]
system-completeness: unit
lead-time: minutes, hours
F
N
Submit changes to project's recipe repository
automation: none
lead-time: minutes, hours
Jenkins
Incremental build on target software
automation: full
functional-confidence: none
non-functional-confidence: none
queue-time:[0m,3h]
duration: [10m, 30m]
system-completeness: unit, partial, full
lead-time: minutes, hours
F
N
Lab with Target Hardware
Incremental build tested on target
automation: none
functional-confidence: extensive
non-functional-confidence: some
queue-time:[10m,20m]
duration: [10m, 30m]
system-completeness: partial, full
lead-time: minutes, hours
F
N
Centralized Git
Project recipe repository
pre-integration-procedure: none
integration-target: team

Figure 5.7: Product Line Viewpoint for the recipe commit.

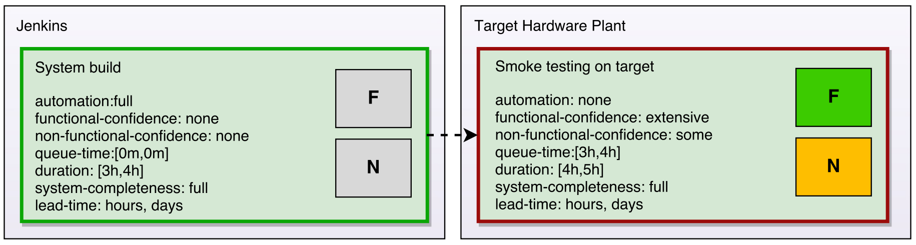


Figure 5.8: Product Line Viewpoint for the nightly build.

5.1.3 Test Capabilities Viewpoint

The Test Capabilities Viewpoint is shown in Fig. 5.9. This viewpoint provides an overview of the test activities performed in the IHU project. It should be noted that only the test activities that are relevant to the continuous integration system are discussed in this thesis. Any late-cycle tests or tests done by independent testing teams are therefore ignored. For each test activity in the Test Capabilities Viewpoint, it can be seen what their feedback lead time is and at which level of system completeness they are executed [37]. The tests that have red borders are not automated and the ones that have green borders are automated. *Smoke testing* is done in India during the night and they focus more on functional related testing, to see if everything is working. The smoke testers follow a list of instructions on what to test, and then have a report ready in the morning when the developers show up to work. The smoke testing is included in the system build viewpoints and can be seen in Fig. 5.5 and Fig. 5.8. *Testing on target* is done by the component teams to check if their code works by testing their code manually on target. Test on target is therefore quite similar to the smoke tests, but the difference is that the latter tests the whole system in the intended production environment, but the former sometimes only tests a part of it manually on target. The target tests are included in the recipe commit viewpoints and can be seen in Fig. 5.4 and Fig. 5.7. *Static code analysis* is done in Coverity, which is sometimes done in the component build, as can be seen in Fig. 5.3 and Fig. 5.6.

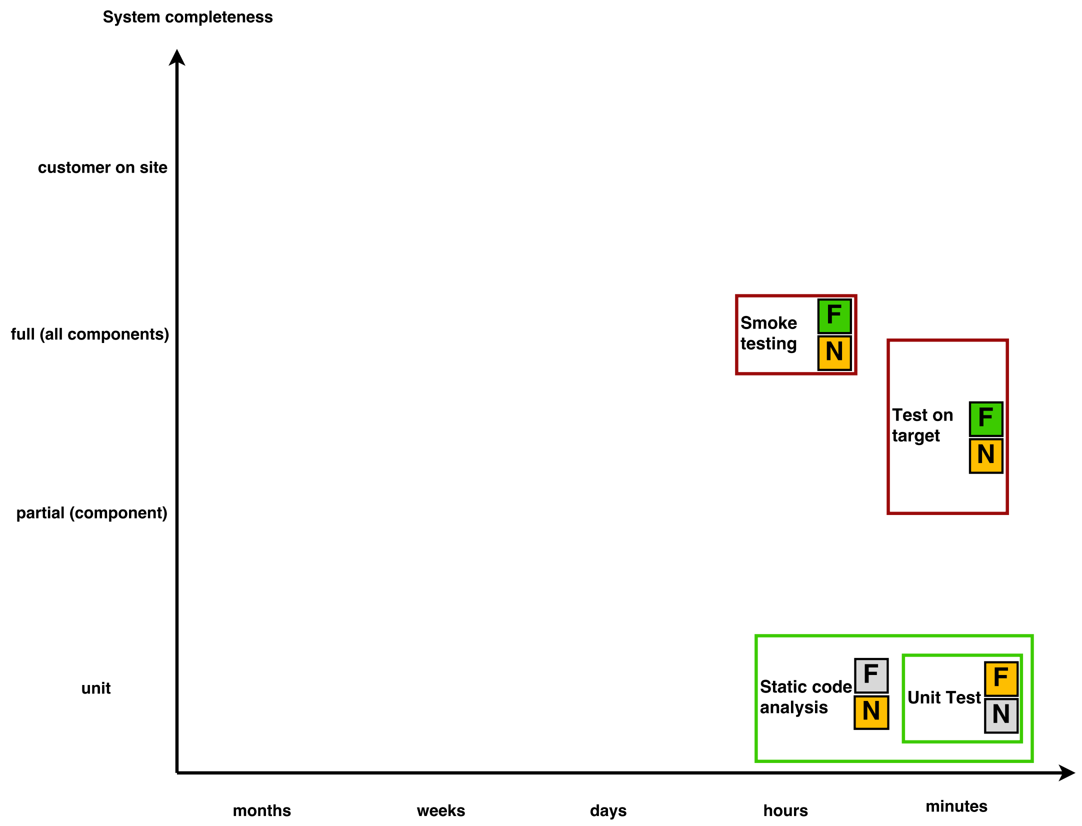


Figure 5.9: The Test Capabilities Viewpoint. This diagram only shows the tests that are most relevant to the continuous integration system.

5.2 Problems and Their Causes

The qualitative interviews, which are described in Section 4.3.2, helped shed light on what problems the case company is having with continuous integration (CI). The main problems can be found in Table 5.1. The interviews also revealed causes for these problems, and an overview of these causes can be found in Table 5.2. The problems and causes can also be seen in Fig. 5.15, Fig. 5.16, Fig. 5.17 and Fig. 5.18, where they have been tagged to specific activities and tasks in the continuous integration system.

Table 5.1: Problems with continuous integration.

Problem	**Description**
Late discovery of defects	*Defects are found during late-stage testing instead of being found by the continuous integration system.*
The overall integration process takes too much time	*It can take a long time for a commit to get accepted into the main build.*
System build breaks too often	*The main system build, which is run during the night, breaks too often.*
Interrupted development flow	*The normal development flow of developers is sometimes interrupted, e.g. because of excessive waiting times.*

Table 5.2: Problems with continuous integration and their causes.

Problem	**Causes**
Late discovery of defects	*Testing on target is not extensive enough, testing on target skipped, tests are lacking in the CI system, test bench is different from production environment, static code analysis ignored or skipped.*
The overall integration process takes too much time	*Testing on target is time-consuming, system build is only run once a day, static code analysis is time-consuming, component build takes too much time, code reviews take too much time, too long build queues.*
System build breaks too often	*Missing dependencies not discovered in the incremental build.*
Interrupted development flow	*Lack of inter-team communication and synchronization, system build breaks too often, the overall integration process takes too much time.*

5.2.1 Late discovery of defects

Code changes sometimes introduce new defects into software. At the case company, the test suite is not optimized to find defects before they are introduced; there is more of a reliance on smoke tests and other late-cycle tests to find defects in the product. This problem, i.e. late discovery of defects, is tagged to the *Submit changes to project's (centralized) recipe repository* task in Fig. 5.16. It is tagged to that task because defects, which could have been found by tests, are sometimes submitted to the project's mainline. The causes for this problem can be seen in Fig. 5.10 and are explained in more detail in the following subsections.

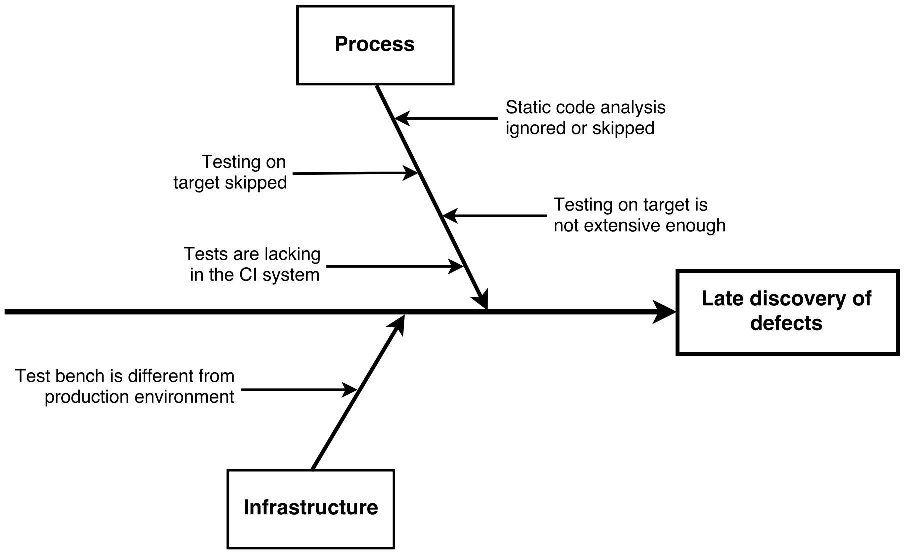

Process
Static code analysis ignored or skipped
Testing on target skipped
Testing on target is not extensive enough
Tests are lacking in the CI system
Late discovery of defects
Test bench is different from production environment
Infrastructure

Figure 5.10: Late discovery of defects (cause-and-effect diagram).

5.2.1.1 Testing on target not extensive enough

One of the causes for late discoveries of defects is that the manual regression tests performed on target by developers are often not extensive enough:

> "*People think they have tested on hardware — they have the illusion that they tested it — but they didn't test it long enough.*"
> - Configuration manager A

The same interviewee expands on that point:

> "*People are usually only testing their own functionality and not checking for example if someone else's is breaking.*"
> - Configuration manager A

That is something that developers D and G also mention in their interviews, i.e. that when developers do testing on target, they tend to test only the feature they are working on and do not check whether they are breaking something else.

This cause, i.e. *testing on target not extensive enough*, is both tagged to the *incremental build tested on target* activity in the Causality Viewpoint for a recipe commit in Fig. 5.17 and also tagged to *test on target* in the Test Capabilities Viewpoint in Fig. 5.18.

5.2.1.2 Testing on target skipped

Developers are required to do manual regression testing on target before submitting their changes to the mainline. In the case company's continuous integration system, there is a "tested on hardware" checkbox that needs to be checked before code can be submitted to the project's recipe repository. However, it is possible for developers to tick that checkbox, without having actually tested the code on target. One developer says the following:

> "*The problems is that we have a real complex project. It's really, really complex, so there is no way that a single software engineer can oversee all of the side-effects that your code actually causes. You can introduce time-delays or latency in some ways that affects all the system, and you don't see that, if you don't test on target.*" - Developer G

Scrum master C mentions that timestamps in logs clearly show that developers sometimes skip the manual regression testing on target. If the build is ready and then two minutes later the code has been tested on target, then the developer has definitely not tested the code on target. The interviewees reasoning is that it is not possible to flash the software on target and do the manual tests in two minutes. Since there are many reasons for why testing on target is skipped, a separate cause-and-effect diagram is presented in Fig. 5.11, where those reasons are highlighted.

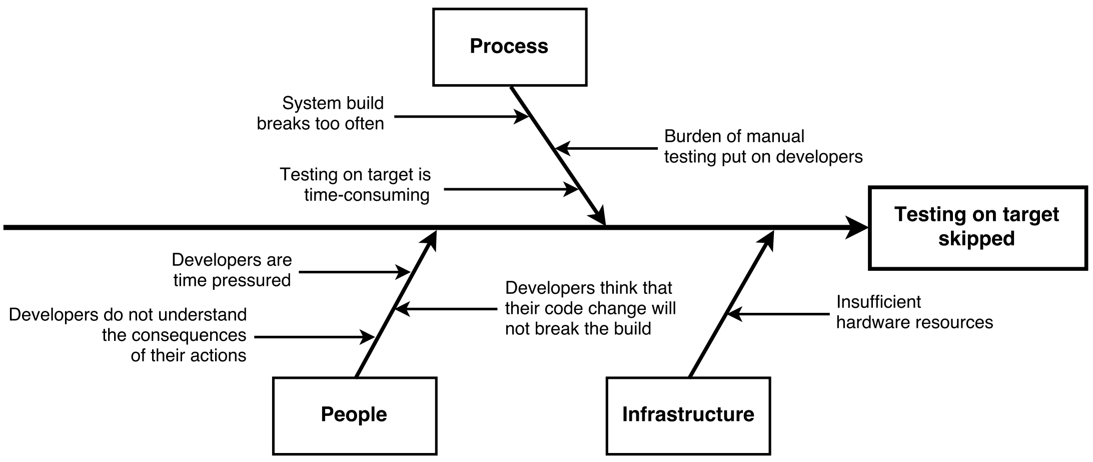

Process
System build breaks too often
Burden of manual testing put on developers
Testing on target is time-consuming
Testing on target skipped
Developers are time pressured
Developers think that their code change will not break the build
Insufficient hardware resources
Developers do not understand the consequences of their actions
People
Infrastructure

Figure 5.11: Testing on target skipped (cause-and-effect diagram).

One of the causes for skipping testing on target is because of a *broken system build.* This is because a broken system build breaks the incremental build, which is needed to do the on-target tests:

> "*I think the main reason [why testing on target is not performed] is that someone else broke [the system build]. I can't test my functionality but I know 99% that it works, so I check it in, because someone has broken the master already and I can't test until they fix the master.*"
> - Configuration manager A

System build breaks too often is defined as one of the four main problems and is explained in more detail in Section 5.2.3.

Developers D, F and G; scrum master C and integrator B all think that one of the reasons for skipping the on-target tests is because *developers think that their code change will not break the build*:

> "*[A reason for ticking the 'tested on hardware' checkbox without testing the code on target is that] you think that: 'yeah, my feature doesn't mess up something else', because you have quite good confidence in your own code.*" - Developer G

Scrum master C agrees, but also thinks that *developers sometimes do not understand the consequences of their actions*:

> "*Some people just don't know [that their code change can break the build] and they don't really understand the whole picture. And then there are some senior developers who think that: oh, this is such an easy change, that can't have any impact to any other place! I can skip [testing] this one [on target].*" - Scrum master C

Developer D agrees that *developers sometimes do not understand the consequences of their actions*:

> "*We have so many different teams committing code and sometimes it's hard to see how one commit can affect the entire tree, basically. Sometimes you just commit code and hope to hell it doesn't crash. [...] A build is 40, 50, 60gb, so it's quite huge with all the source code and everything. So, it's a bit hard to get a grasp on, maybe.*" - Developer D

Developer D also thinks that a reason for skipping testing on target is that *testing on target is time-consuming* and *developers are time pressured*:

> "*It's quite laborious [to test on target]. [...] Flashing it takes 5-10 minutes, and then you need to reboot. That's 15-20 minutes. [...] It might take too long for some. [...] It's also because people are stressed — I think that's also [one reason for skipping testing on target] — they have a lot of commits coming before a release.*" - Developer D

Testing on target is time-consuming is also a cause for the problem *the overall integration process takes too much time* and this cause is explained in more detail in Section 5.2.2.2.

Developer F and G mention that *insufficient hardware resources* is a reason for skipping the on-target tests:

> "*I think you do that [tick the 'tested on hardware' checkbox, without testing the code on target] because you don't have access to hardware, because the hardware is blocked in some way.*" - Developer G

Developer E, who is used to working with fully-automated continuous integration systems, thinks that testing on target should be automated. His view is that it is too much to ask of developers to do full regression testing manually before submitting code changes:

> "*I think the responsibility to put that on a software developer is too much. [...] What does it mean that a developer should do full regression test before submitting? [...] I don't blame people for not doing it. [...] It should be automated.*" - Developer E

This cause is listed under process in Fig. 5.11 and is called *Burden of manual testing put on developers.*

Testing on target skipped, is tagged to *test on target* in the Test Capabilities Viewpoint in Fig. 5.18 and is also tagged to the *incremental build tested on target* activity in the Causality Viewpoint for the recipe commit, which can be seen in Fig. 5.16.

5.2.1.3 Tests are lacking in the CI system

Testing-related problems and causes can be seen in the Test Capabilities Viewpoint in Fig. 5.18. According to this viewpoint, unit tests and static code analysis are automated in the continuous integration system. On the other hand, smoke tests and on-target tests have to be performed manually. Some teams have started to do component tests, but most developers just focus on having a high code coverage of unit tests:

> "*People concentrate too much on unit tests I think. Unit tests are quite good but we have a very high demand on coverage.*" - Developer D

Developer E thinks that the focus should perhaps be shifted from unit testing to component testing or software integration testing:

> "*Maybe we should shift focus from unit testing to component testing, or software integration testing. That's my understanding, that yeah, people*

> *are just trying to reach a set KPI and maybe don't really understand why they should just reach this KPI."* - Developer E

Developer G states the same thing, i.e. that developers are driven by Key Performance Indicators (KPIs) and sometimes just write unit tests to get the code coverage percentage up:

> *"Sometimes KPIs — like unit test coverage, something like that — drive, because sometimes you just write tests to get the coverage up. But it looks on the KPI as if it's good"* - Developer G

Unit tests are important, but Developer E still feels that integration tests are sometimes more suitable:

> *"I think unit testing is important too, where you have units with very complicated logic and of course you should have unit testing because it's much easier to find all the corner cases and test the corner cases. But if you don't have that complicated logic, then it's much more value, better price-performance, benefit ratio, bang for the buck to do integration tests."* - Developer E

Developer E also mentions that this focus on unit testing has a high maintenance cost, in particular when an Application Programming Interface (API) is changed:

> *"It's also a maintenance cost. If you're just writing unit tests, to get to a certain level of code coverage, then you will pay a lot for maintaining those. As soon as you start changing APIs on your unit, then you will need to update all the test cases accessing those APIs and it probably would have been cheaper if you're just out for code coverage to do integrative testing"* - Developer E

Three other interviewees (A, B, D), also think that component and integration tests are lacking in the continuous integration system and say that they should be automated as well. One of them says the following:

> *" What we need is more component testing and automating the component testing I would say, because that's usually where the issues are. When we have a component talking to a component and the API changes, and if only one makes the change then it will break for the other team. [...] So I would say for component teams what we are lacking is component tests and integration between components."* - Configuration manager A

This cause is also tagged to the *build component under the target SDK and unit test* activity in the Causality Viewpoint for the code commit, which can be seen in Fig. 5.15.

5.2.1.4 Test bench is different from production environment

Some regressions that are not found when testing in bench might show up in the production environment:

> "*Testing in bench doesn't give you the same results as testing in a real car.*" - Configuration manager A

This cause is both tagged to the *incremental build tested on target* activity in the Causality Viewpoint for the recipe commit, which can be seen in Fig. 5.16 and also tagged to *test on target* in the Test Capabilities Viewpoint in Fig. 5.18.

5.2.1.5 Static code analysis ignored or skipped

As mentioned in Section 5.1, the static code analysis is only done if there is enough time. Developers tend to stop it when they do the component build, in order to speed things up. This means that one of the reasons for skipping code analysis is because *running the static code analysis is time-consuming*, which is discussed further in Section 5.2.2.4. Integrator B says that static code analysis is needed, but also that it is of low priority and developers often do not care about the result from it. This cause refers to the *analyze code with Coverity* activity in the Causality Viewpoint for the code commit, which can be seen in Fig. 5.15. This cause is also tagged to *static code analysis* in the Test Capabilities Viewpoint in Fig. 5.18

5.2.2 The overall integration process takes too much time

It can take a long time for a commit to get accepted into the main build. All of the developers (D, E, F, G) think that the overall integration process is too long. Developer G states that the overall process sometimes takes almost half a day. Developer E wants it to be possible to deploy new software in the car in 24 hours, but thinks that they need to be much faster to be able to achieve that:

> "*I think it [the overall integration process] takes too long. [...] It won't be possible with the current pace to put new software in the car in 24 hours. We need to be much faster.*" - Developer E

Another developer says the following:

> "*I think the CI environment is just slow. [...] It's basically just too long of a feedback loop.*" - Developer D

The causes for this problem can be seen in Fig. 5.12 and are explained in more detail in the following subsections.

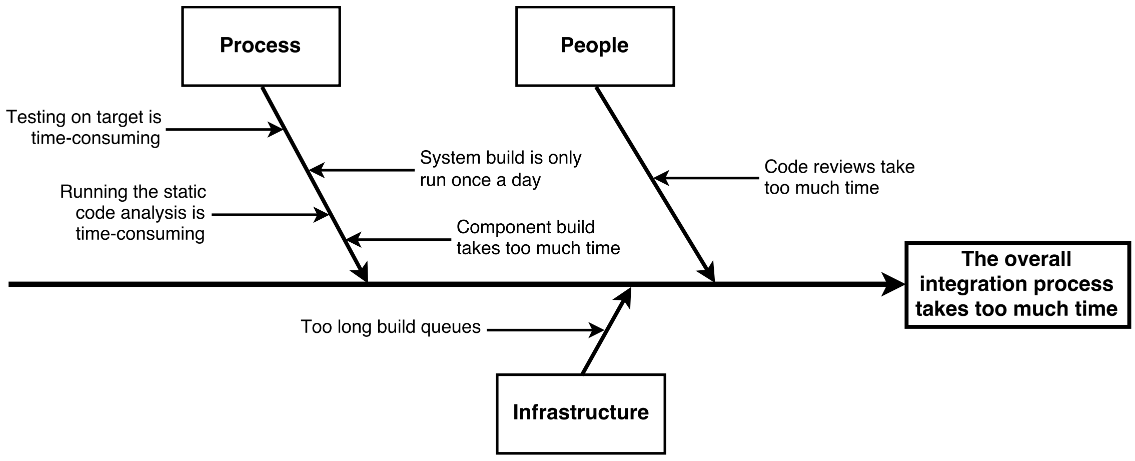

Process
People
Infrastructure
Testing on target is time-consuming
Running the static code analysis is time-consuming
System build is only run once a day
Component build takes too much time
Code reviews take too much time
Too long build queues
The overall integration process takes too much time

Figure 5.12: The overall integration process takes too much time (cause-and-effect diagram).

5.2.2.1 Component build takes too much time

Developers often have to wait a long time for the component build to finish. Most respondents state that they would want the component build to take between 5 and 15 minutes maximum. However, the build times, which are of course different for each component, are much higher in many cases. As can be seen in the Product Line Viewpoint for a code commit in Fig. 5.6, component builds usually take between 10 and 40 minutes. There is sometimes an additional queue-time which can take up to 1 hour. Configuration manager A states that the component builds probably take around half an hour on average:

> "*It depends on which component, I would say. Some are down to ten minutes, maybe even lower. Some may take up to forty minutes. [...] [The component builds take on] average thirty minutes I would say, if you do them in parallel.*" - Configuration manager A

Developer D also states that the tests are not causing the high build times, but the compilations instead. This cause, i.e. *Component build takes too much time*, is tagged to the *Build component under the Target SDK and unit test* activity in the Causality Viewpoint for the code commit, which can be seen in Fig. 5.15

5.2.2.2 Testing on target is time-consuming

The fact that *testing on target is time-consuming* does not only lead on-target testing being skipped, as discussed in Section 5.2.1.2 — it also causes the overall integration process to take too much time. Developer D thinks that testing on target takes too much time and adds the following:

> "*It's quite laborious [to test on target]. You get test build, you flash it on hardware and then you need to reboot, do a change.*" - Developer D

This cause is tagged to the cause *testing on target skipped* which is both tagged to the *incremental build tested on target* activity in the Causality Viewpoint for a recipe commit in Fig. 5.17 and also tagged to *test on target* in the Test Capabilities Viewpoint in Fig. 5.18.

5.2.2.3 System build is only run once a day

The system build is only done once a day, i.e. on a nightly basis, and is therefore not done in a continuous manner. This cause is tagged to the *system build* activity in the Causality Viewpoint for the system build, which can be seen in Fig. 5.17.

5.2.2.4 Running the static code analysis is time-consuming

Configuration manager A and developer G think that the execution of the static code analysis tool being used, Coverity, takes too much time:

> "*Coverity is good, but it's too high build time for it to actually be efficient enough in continuous integration. [...] It can add one to four times the build. So, if your build is ten minutes without Coverity, Coverity can increase it up to twenty or even forty, fifty minutes.*"
>
> \- Configuration manager A

Since it is so time-consuming, the *static code analysis is sometimes ignored or skipped*, as described in Section 5.2.1.5. This cause refers to the *analyze code with Coverity* activity in the Causality Viewpoint for the code commit, which can be seen in Fig. 5.15. This cause is also tagged to *static code analysis* in the Test Capabilities Viewpoint in Fig. 5.18

5.2.2.5 Code reviews take too much time

Developers need to do code reviews before their code is merged to a component source repository (in the code commit integration flow). Code reviews are also required before a recipe is merged to the project recipe repository (in the recipe commit integration flow). This problem refers to the activity which takes place in the code commit integration flow, see Fig. 5.15. According to the Product Line Viewpoint for the code commit in Fig. 5.6, the code review can take between 5 minutes and 2 hours. Developer G says that the code reviews slow them down, since they need two team members to approve the code before it is sent through. Sometimes it takes a long time for the two team members to review the code or start the code review activity. The developer committing the code can therefore get stuck in this review phase. The same developer also explains the main reasons for this cause:

> "*One [reason for why peer reviews slow developers down] is that we have a team where everyone is detail oriented and [...] they are quite picky*

on how the code looks like basically. [...] [Another reason] is that there is a lot to do, so we don't actually have time to review as well, so then we get stuck on that." - Developer G

5.2.2.6 Too long build queues

The continuous integration system can only run a certain amount of builds at a time, so a queue is created when many builds need to be run at the same time. Configuration manager A states that long build queues can cause long integration times. Developer F says the same thing, but adds that this is only a problem at the end of a sprint:

"*Sometimes when it's at the end of the sprints I would say there is a lot of traffic and then lots of jobs in the queue, so it takes of course a longer time. [...] Other than that, then there are not that many jobs in queue."* - Developer F

This cause is tagged to the *build component under Target SDK and unit test* activity in the Causality Viewpoint for the code commit, in Fig. 5.15 and also in the *Incremental build on target software* activity in the Causality Viewpoint for the recipe commit, in Fig. 5.16.

5.2.3 System build breaks too often

The main system build, which is done once a day, breaks often. Configuration manager A, scrum master C, and developers F and G all think that a broken system build is quite common. The scrum master's opinion is that there are too many broken builds and that it is a big issue in the project. Developer F mentions that a broken build is sometimes a bottleneck, since everyone is dependant on the system build's image. Some teams are, however, less effected by a broken build than others:

"*When there is a broken build then we are not affected that much. [...] It depends on the issue as well, actually, and what feature we are developing as well, because some features demand us to use the latest nightly build and then it could block us."* - Developer G

This problem is tagged to the *system build* activity in the Causality Viewpoint for the system build, which can be seen in Fig. 5.17. The cause for this problem can be seen in Fig. 5.13 and is explained in more detail in the following subsection.

5.2.3.1 Missing dependencies not discovered in the incremental build

Some code change might break the system build, even though it does not break the incremental build:

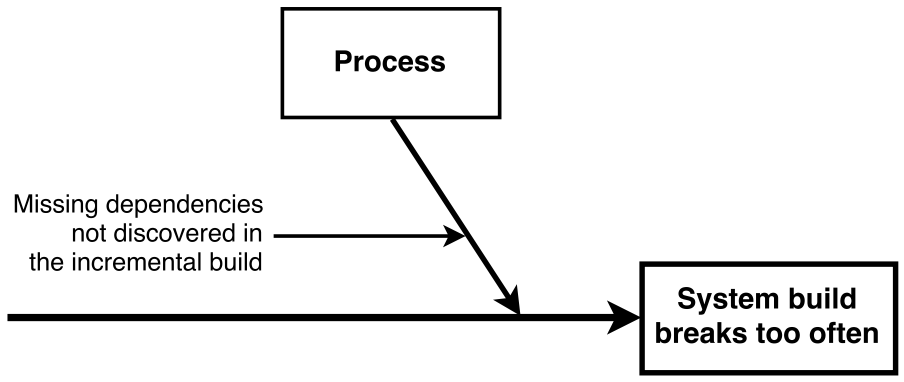

Process
Missing dependencies not discovered in the incremental build
System build breaks too often

Figure 5.13: System build breaks too often (cause-and-effect diagram).

> "*All the incremental builds can be fine, and you don't see any issues and you can even test that thing locally and everything works [but the clean system build could still break.]*" - Scrum master C

In the incremental build, a component might depend on an older version of a component which then updates in the daily system build. Configuration manager A mentions that in the system build, everything is built from scratch, and that the build order of components matters in that case. The same interviewee also notes that in the incremental build, only the components that are affected by your changes need to be built, which makes dependency problems less likely. This cause is tagged to the *incremental build tested on target software* activity in the Causality Viewpoint for the recipe commit, which can be seen in Fig. 5.16.

5.2.4 Interrupted development flow

Developers experience interrupted development flow when they are blocked from finishing some task that they are working on. Integrator B states that if the main build is broken, then the teams cannot continue working and developers are delayed, since they cannot submit new changes to the mainline. Developer D mentions that every team gets blocked if the main build is broken, and also says that interrupted development flow is extremely common for his team, since they depend a lot on the other teams. This problem is tagged to the *new commit pushed to Gerrit* triggering relationship in the Causality Viewpoint for the code commit, which can be seen in Fig. 5.15. The causes for this problem can be seen in Fig. 5.14 and are explained in more detail in the following subsections.

5.2.4.1 Lack of inter-team communication and synchronization

Teams need to be more synchronized, because changes that one teams does can affect other teams. Scrum master C and developers D and E think that there is a lack of inter-team communication and synchronization. One of the interviewees states the following:

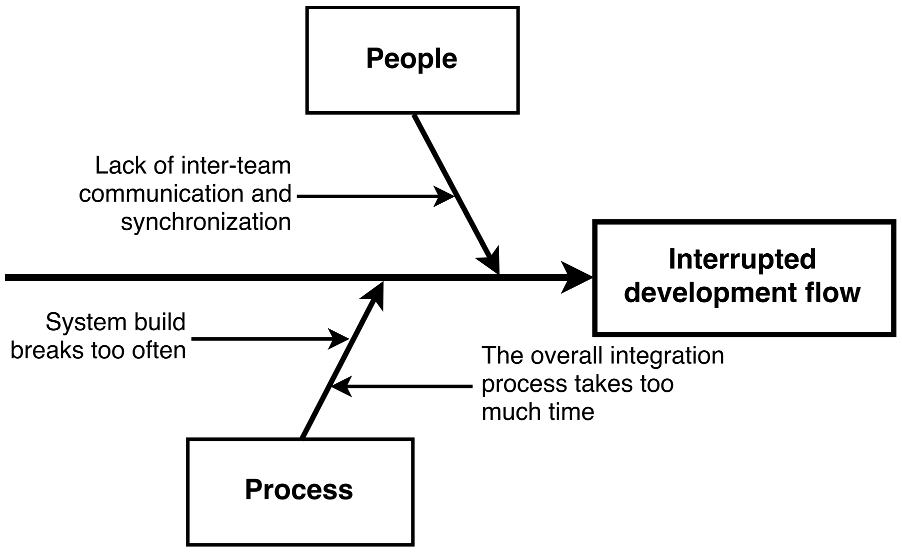

People
Lack of inter-team communication and synchronization
Interrupted development flow
System build breaks too often
The overall integration process takes too much time
Process

Figure 5.14: Interrupted development flow (cause-and-effect diagram).

> "*I think a lot of it [interrupted development flow] has been because the teams which we rely on have not been synchronized enough with our needs. So it's a synchronization issue in the project. I think that's the main reason [for interrupted development flow].*" - Developer E

5.2.4.2 System build breaks too often

Four of the interviewees (B, C, D, F) mention that a broken system build causes an interrupted development flow, since developers often need to wait until the build gets fixed before they can continue their work. One of the interviewees states the following:

> "*Well, if the build is broken, in the master branch, [...] teams cannot continue. We can't continue working if we cannot submit changes to the master, so it delays, I would say, the developers.*" - Integrator B

This cause, which is also one of the main problems, is explained in more detail in Section 5.2.3 and is tagged to the *system build* activity in the Causality Viewpoint for the system build, which can be seen in Fig. 5.17.

5.2.4.3 The overall integration process takes too much time

Scrum master C says that the fact that the system build is only done once every 24 hours can interrupt developers:

> "*If they [developers] have fixed a bug, they need to have a nightly official build to verify that bug. In this case, it means that it will have 24 hours delay immediately and those people have already jumped to some other*

areas. They need to switch between different areas and tasks and that is always taking some time and it's kind of unnecessary jumping back and forth. Yeah, that's probably one of these things which people don't really realize has quite a big impact." - Scrum master C

This cause, which is also one of the main problems, is explained in more detail in Section 5.2.2 and is tagged to the *new commit pushed to Gerrit* triggering relationship in the Causality Viewpoint for the code commit in Fig. 5.15.

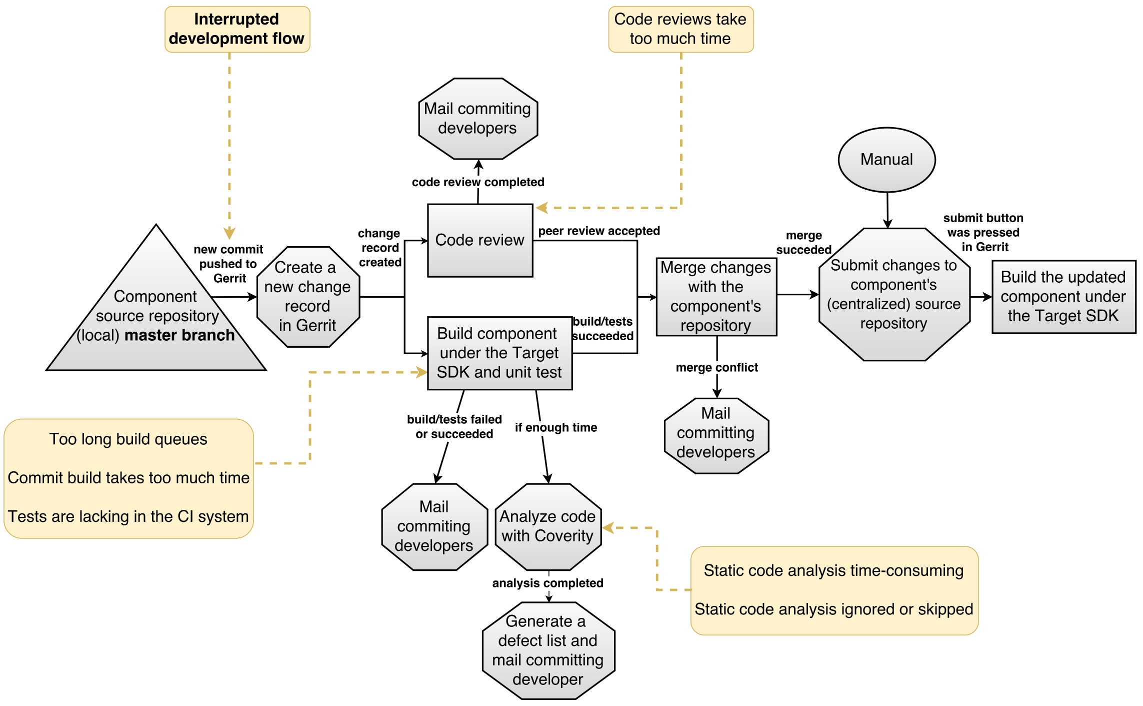

Interrupted development flow
Code reviews take too much time
Mail commiting developers
code review completed
Manual
Component source repository (local) master branch
new commit pushed to Gerrit
Create a new change record in Gerrit
change record created
Code review
peer review accepted
Merge changes with the component's repository
merge succeded
Submit changes to component's (centralized) source repository
submit button was pressed in Gerrit
Build the updated component under the Target SDK
Build component under the Target SDK and unit test
build/tests succeeded
merge conflict
Mail committing developers
Too long build queues
Commit build takes too much time
Tests are lacking in the CI system
build/tests failed or succeeded
if enough time
Mail commiting developers
Analyze code with Coverity
Static code analysis time-consuming
Static code analysis ignored or skipped
analysis completed
Generate a defect list and mail committing developer

Figure 5.15: Causality Viewpoint for a code commit. Relevant problems (bold) and causes are shown.

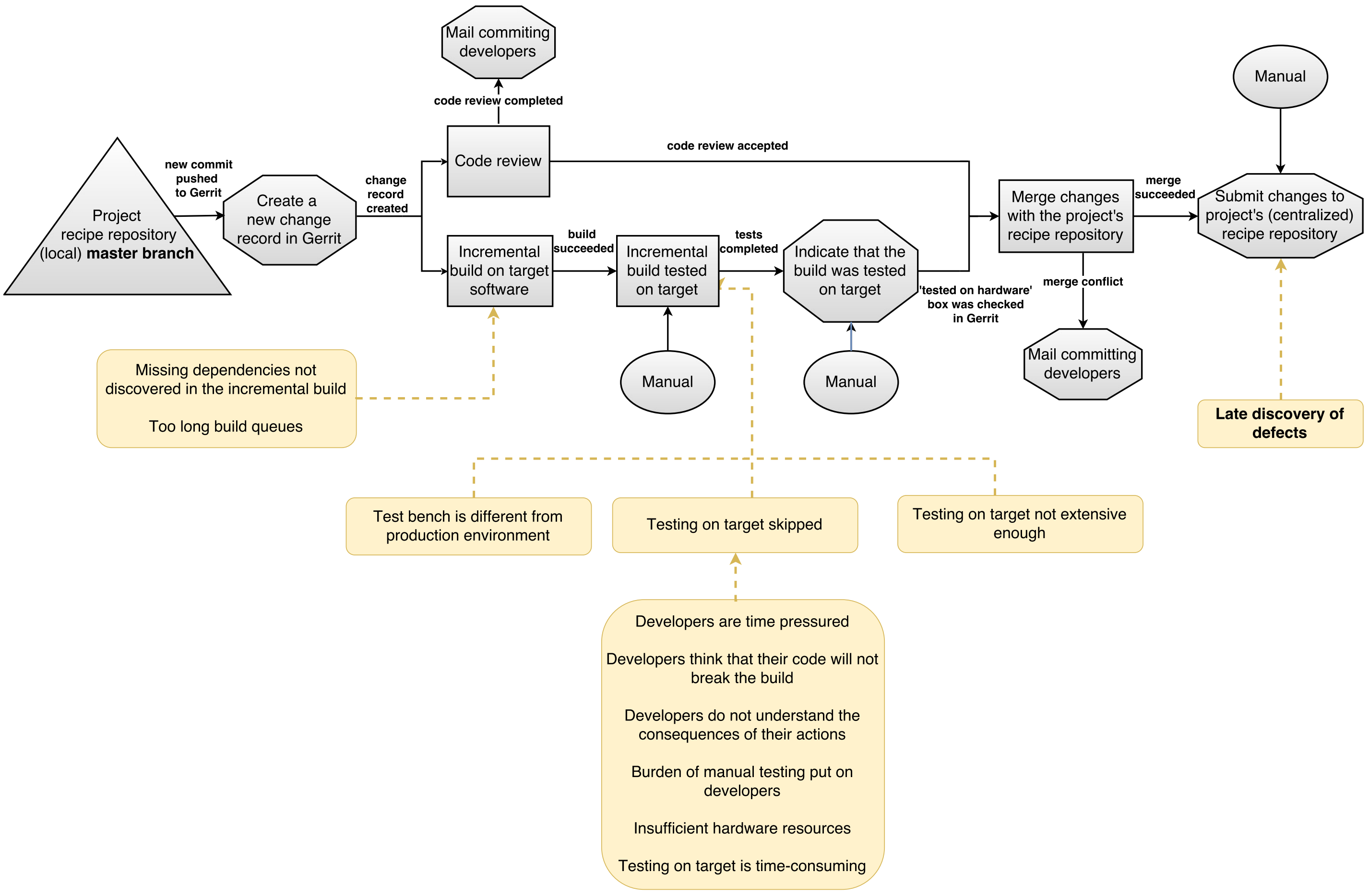

Mail commiting developers
code review completed
Code review
code review accepted
new commit pushed to Gerrit
Project recipe repository (local) **master branch**
Create a new change record in Gerrit
change record created
Incremental build on target software
build succeeded
Incremental build tested on target
tests completed
Indicate that the build was tested on target
'tested on hardware' box was checked in Gerrit
Merge changes with the project's recipe repository
merge succeeded
Submit changes to project's (centralized) recipe repository
merge conflict
Mail committing developers
Manual
Manual
Manual
Manual
Missing dependencies not discovered in the incremental build
Too long build queues
Late discovery of defects
Test bench is different from production environment
Testing on target skipped
Testing on target not extensive enough
Developers are time pressured
Developers think that their code will not break the build
Developers do not understand the consequences of their actions
Burden of manual testing put on developers
Insufficient hardware resources
Testing on target is time-consuming

Figure 5.16: Causality Viewpoint for a recipe commit. Relevant problems (bold) and causes are shown.

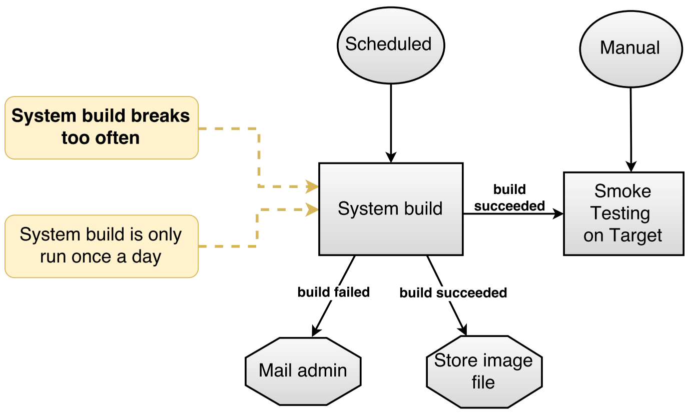


Figure 5.17: Causality Viewpoint for a system build. Relevant problems (bold) and causes are shown.

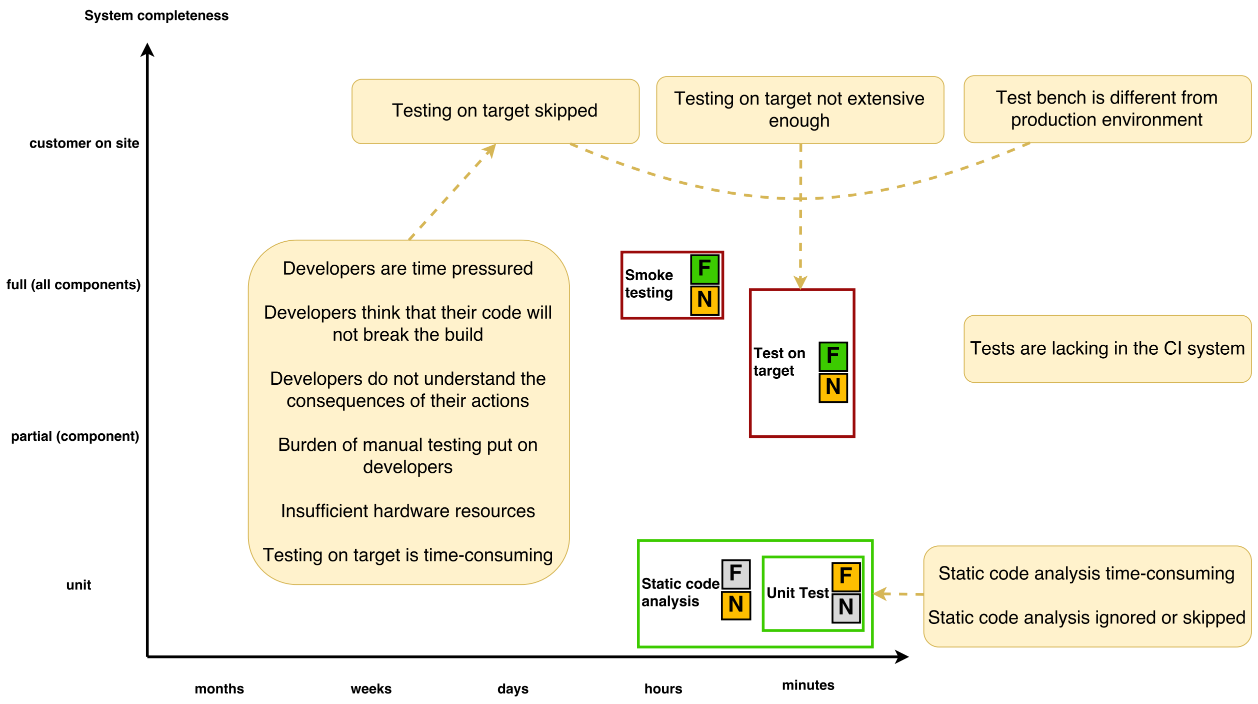

System completeness
customer on site
full (all components)
partial (component)
unit
months
weeks
days
hours
minutes
Testing on target skipped
Testing on target not extensive enough
Test bench is different from production environment
Developers are time pressured
Developers think that their code will not break the build
Developers do not understand the consequences of their actions
Burden of manual testing put on developers
Insufficient hardware resources
Testing on target is time-consuming
Smoke testing
F
N
Test on target
F
N
Tests are lacking in the CI system
Static code analysis
F
N
Unit Test
F
N
Static code analysis time-consuming
Static code analysis ignored or skipped

Figure 5.18: Test Capabilities Viewpoint. Relevant problems and causes are shown.

6 Discussion

The results of this study are discussed in this chapter. In Section 6.1, the research questions are addressed and the results evaluated using informed argument. Then, potential threats to validity are discussed in Section 6.2.

6.1 Research Questions

This study has two research questions:

RQ1: What are the main problems associated with using continuous integration in the development of component-based embedded software?

RQ2: What are the main causes for the problems associated with using continuous integration in the development of component-based embedded software?

6.1.1 The Problems (RQ1)

The four main problems that are listed in Table 5.1 will be discussed here.

6.1.1.1 Late discovery of defects

The first problem is the *late discovery of defects*. One of the benefits of continuous integration, according to Duvall [11], is that defects are detected and fixed sooner when it is used. His reasoning is that running tests and inspections several times a day leads to a higher probability of discovering defects *when they are introduced*, instead of during late-cycle testing. Fowler [12] similarly states that one of the benefits of continuous integration is that it leads to dramatically less bugs. However, he also states that the degree of this benefit is directly tied to the quality of the test suite. At the case company, the test suite is not optimized to find defects before they are introduced. They rely more on smoke tests and other late-cycle tests to find defects in the product. It can be challenging to automate some tests and run them continuously when specialized hardware is needed [21], so this problem is partly because the software is embedded.

6.1.1.2 The overall integration process takes too much time

The second problem is that *the overall integration process takes too much time.* Fowler [12] states that rapid feedback is *"the whole point of CI"* and Duvall [11] similarly states that rapid feedback is *"at the heart of CI"*. In this context, rapid feedback means that developers are notified quickly whether their code change was accepted or not. Rapid feedback also implies that everyone involved in the project should be able to know the status of the project several times a day [11]. In the IHU project, the time from when a developer commits code to when that commit is merged to the mainline is quite long. A commit can reach the mainline the next day at the earliest — if everything goes smoothly (code reviews are finished quickly and so on). The next day is therefore the best case scenario, but usually it takes more than one day for a commit to reach the mainline.

6.1.1.3 The system build breaks too often

The third problem is that *the system build breaks too often.* Fowler [12] and Duvall [11] both say that builds need to be fixed immediately if they break. Fowler states that it is essential when using continuous integration to have a known stable base for developers to work on. Broken builds are therefore obviously not good for the project. If the build breaks too often, then it is vital to find out why it is breaking, so that future breaks can be prevented from happening.

6.1.1.4 Interrupted development flow

The fourth problem is *interrupted development flow.* This problem entails anything that decreases the productivity of developers and distracts them from writing code for some reason. Anything that blocks the developers from their normal development flow causes this problem. Many issues can arise when developers resume their work after having been interrupted. Examples of such issues are more errors, loss of knowledge, increased time to perform tasks and increased failures to perform critical tasks [26].

6.1.2 The Causes (RQ2)

The causes for each of the main problems are listed in Table 5.2 and will be discussed here.

6.1.2.1 Late discovery of defects

The first problem, *late discovery of defects*, has five causes that are all testing-related and can be seen in Fig. 5.10. One of those causes is *static code analysis ignored or skipped.* According to a paper by Ayewah et al. [2], static code analysis tools can be used to find important defects in code. Not running the static code analysis can therefore contribute to the late discovery of defects.

Another one of the causes for late discovery of defects is that *tests are lacking in the CI system.* A lot of effort is put into reaching a high code coverage for unit tests in the IHU project. This focus on unit tests means that component and integration tests get little attention. Interestingly, Jalote et al. [14] found that most defects of a component were found by testing other components. By running automated component testing for instance, one could detect more defects since those types of testing use more dependencies than unit testing [11].

Test bench is different from production environment is a cause for late discovery of defects and relates to embedded software development. The test environment becomes a limited resource if the hardware needed for the system under test is expensive or in short supply [21]. In the automotive industry for instance, it might not always be possible to test the software in an actual car (i.e. the production environment). If code changes are not regression tested with a car, then defects can go undetected.

Testing on target is not extensive enough is another cause for the late discovery of defects. It addresses the fact that manual hardware testing is sometimes not performed extensively enough. According to Duvall [11], 100% of all tests must pass for every single build when continuous integration is used. He claims that not running the tests can lead to broken code and lower-quality software. In the IHU project, the testing on target is not automated and has to be performed manually. Broken code can reach the mainline if these tests are not extensive enough. It could be argued that these manual regression tests do not "pass" if they are not done extensively enough. This would not be as big of a problem if this testing was automated, as Mårtensson et al. [21] and Duvall [11] state it should be.

The cause *testing on target skipped* has seven causes of its own, as seen in Fig. 5.11. Some of these causes are put in a category called *people* in the cause-and-effect diagram. Laukkanen et al. [18] list some similar human and organizational problems that make the adoption of continuous integration and delivery difficult. They define *lack of discipline* as a problem where developers lack discipline, e.g. in doing tests before committing their code. *More pressure* is also a human and organizational problem that they mention, which is sometimes caused by *lack of experience* which also causes *lack of understanding.* Debbiche et al. also define *increased pressure* as a challenge when adopting continuous integration [9]. *Hardware testing* is also defined as a problem according to Laukkanen et al. [18], which happens when the hardware needed in hardware testing is not always available and can be defined as the same cause listed under infrastructure in Fig. 5.11 and is called *insufficient hardware resources.* Another cause for skipping the on-target tests is that the *burden of manual testing put on developers.* Mårtensson et al. [21] mention in their paper that automated tests should be a prerequisite for continuous integration and add that manual tests are less predictable and take a longer time to execute than automated tests. Furthermore, Debbiche et al. [9] have also identified *too many manual tests* as a challenge associated with testing at a telecommunication services and infrastructure provider. They found in their case study that teams working closer to the hardware need to do more manual tests than the teams working on the application level.

6.1.2.2 The overall integration process takes too much time

The second problem, *the overall integration process takes too much time*, has six causes as seen in Fig. 5.12. Four of those causes relate to time-consuming activities, and together with *too long build queues*, they all slow down the overall integration process.

System build is only run once a day is also a cause for the overall integration process taking too much time. Duvall [11] claims that one cannot claim to be practicing continuous integration if the build is not run continuously. One of the cornerstones of continuous integration, in his view, is that several integration builds occur each day. He considers regular builds to be important so that everyone can learn the state of the project several times a day. It is therefore not enough to just run the build once a day on a hard-schedule, like it is done in the IHU project.

6.1.2.3 System build breaks too often

System build breaks too often is the third of the main problems listed and has only one cause, i.e. *missing dependencies not discovered in the incremental build*, as can be seen in Fig. 5.13. This cause is mainly due to the fact that the software is component-based. One interviewee mentioned that if the components are not built in the right order in the system build, then that build might break, even though the incremental build done previously succeeded. Debbiche et al. [9] mention in their paper that it is important to consider how code dependencies affect the integration process. They identify a couple of issues related to integration dependencies and one of them is that the amount of failures experienced during integration increased after the adoption of continuous integration.

6.1.2.4 Interrupted development flow

The last problem, *Interrupted development flow* has three causes as can be seen in Fig. 5.14. Two of them are main problems that were discussed earlier, i.e. *system build breaks too often* and *the overall integration process takes too much time.* Laukkanen et al. [9] also identified similarly that there is a causal relationship between a broken build and broken development flow.

The causes for the overall integration process taking too much time, which are listed in Fig. 5.12, can also cause an interrupted development flow. As was mentioned in Section 6.1.2.2, three of the causes for this problem are related to time-consuming activities. These activities are similar to the *time-consuming testing* problem that Laukkanen et al. identify [18]. According to them, time-consuming testing causes *interrupted development flow* (which they call *broken development flow*). Debbiche et al. [9] define *too long build queues*, which they call *integration queue* in their paper, as a challenge, since due to a constant stream of integrations the risk of blocking the integration queue increases, which can lead to the mainline being blocked for some time. Like a broken system build, a broken mainline can cause an interrupted development flow. The development flow can also be interrupted when builds take too much time according to Brooks [5].

System build is only run once a day can also be considered as a cause for interrupted development flow. Duvall [11] states that if developers have to stop their development activities because they need to wait for feedback, then that affects the development flow.

Lack of inter-team communication and synchronization is the final cause for interrupted development flow. This challenge relates to component-based software development, since development teams are highly dependent on each other [4, 25]. This means that development teams sometimes need to wait for other parts to be finished before integrating their work, which interrupts the development flow [9].

6.2 Threats to Validity

In this section, the threats to validity are discussed. Assessing the validity of a study is important, as it indicates how trustworthy the results are [31]. Four different categories of validity threats are addressed in this study and they are discussed in the sections below.

6.2.1 Construct Validity

Construct validity refers to what extent the operational measures that are studied represent what is investigated according to the research questions and what the researchers have in mind [31]. Runeson and Höst [31] say that triangulation can be used to improve research validity. In essence, triangulation means that different angles of the studied object are analyzed to get a broader perspective. This becomes especially important when qualitative data is being used, since it is less precise than quantitative data. Three types of triangulation were applied in this study:

- *Data triangulation* was used by collecting data from different sources. Seven subjects were interviewed from 5 different development teams and one integration team. The subjects also held various roles in the company.
- *Observer triangulation* was achieved by having two researchers conduct the interviews. In addition, both supervisors of this thesis, i.e. the one from university and the one from the case company, reviewed the interview questions. Their comments then led to some refinements being done before the interviews were conducted.
- *Methodological triangulation* was achieved by using different methods for collecting data, i.e. unstructured interviews, semi-structured interviews and documentation analysis.

The data from the semi-structured interviews was the principal source used to identify the main continuous integration related problems and their causes. It is worth noting that some interviewees seemed to have thought that *breaking the build* and *introducing broken code into the build* meant the same thing. Also, since no visualization of the continuous integration system is in place at the case company, developers were sometimes unsure about what build was being asked about. Some of the construct validity threats were addressed by showing the interviewees Fig. A.1 and Fig. A.2 in the semi-structured interviews, in order to show them what parts of the continuous integration systems they were being asked about.

6.2.2 Internal Validity

Internal validity needs to be considered when examining causal relations. This means that when the researcher is investigating whether factor A affects an investigated factor B, then there is risk that factor B is also affected by a third factor C. There is a threat to internal validity if the researcher is not aware of factor C or does not realize to what extent it affects factor B [31].

In this study there is a risk of internal validity threats since causal relations are being investigated. Interviewees helped identify the causes for certain problems and it is of course possible that the interviewees are sometimes wrong in their assessments. Interviewees might think that one factor causes another factor, but the reality might be that some third hidden factor was the actual cause.

6.2.3 External Validity

External validity refers to what extent the findings are of interest to people that are outside the investigated case and to what extent it is possible to generalize the findings [31]. The fact that only one project in a single company was under study in this thesis can be considered to be an external validity threat.

6.2.4 Reliability

Reliability is one aspect that indicates whether the results would be the same if another researcher later conducted the same study. This validity therefore refers to what extent the data and the analysis of the data are dependent on the researchers that conducted this study in the first place [31].

In this study, the main continuous integration related problems and their causes were identified with unstructured and semi-structured interviews with 7 employees at the case company. In order to get more reliable data, it would be essential to have interviews with more employees at the case company. As was mentioned in Section 6.2.1, observer triangulation was achieved by having two researchers conduct interviews, which would reduce bias by individual researchers. Thus, the reliability of the study is improved [31].

7 Conclusion

In this study, the problems associated with using continuous integration in the development of component-based embedded software are assessed. Additionally, potential causes for those problems are identified. The continuous integration system being used for a component-based embedded software project was modeled using Cinders, which is an architectural framework for describing continuous integration and delivery systems. The Cinders model and interviews conducted with project members helped identify four main problems related to the continuous integration system: late discovery of defects, the overall integration process takes too much time, the system build breaks too often and interrupted development flow. The causes for those problems were also identified and a model was created in that regard. This model includes five Ishikawa diagrams: four diagrams for the main problems and then one additional diagram for one of the potential causes. These diagrams list all the causes for a particular problem or a cause, which can help the case company identify how the continuous integration system can be improved. A lot of the research in the field of continuous integration is missing important contextual information, e.g. the type of application that continuous integration is being used to develop and what the domain is. This study therefore includes contextual information on the product and the organization developing it.

This study and others suggest that continuous integration works well with component-based software development. Breaking a large software project into components makes using continuous integration on a large scale more manageable. However, good inter-team communication and synchronization seem especially important when the software architecture is component-based. This seems to be the case, at least partly, because components are quite sensitive to changes, and developers are expected to do frequent changes to the common code base when continuous integration is used. One of the main problems at the case company is that the nightly system build, which builds the whole system from scratch, breaks too often. This problem is caused by missing dependencies and thus directly linked to the fact that the software is component-based. Before submitting changes to the mainline, an incremental build is run and the software is then tested on target. This incremental build is much faster than a full system build and therefore reduces the overall integration time. However, this reduction in time comes at a cost, as it leads to more broken builds. The nightly system build can break even though the incremental build was successful, as missing dependencies are sometimes not discovered in the incremental build.

Adopting continuous integration in embedded software development seems quite challenging. Some of the problems that the project discussed in this study is facing could be eliminated by automating tests. Developers in the project are required to perform manual regression tests on target, which often leads to those tests not being extensive enough or in some cases not performed at all. Automated testing is, however, hard to achieve in embedded software development since specialized hardware is needed to both run the software and perform tests.

To our knowledge, this is the first study which uses Cinders to model a continuous integration system — aside from the example presented in the original paper. We liked the framework and found using it to be straightforward for the most part. The most challenging part was to elicit the functional- and non-functional-confidence values of activities, as people were somewhat confused by what these terms meant. Also, we had to create all of the Cinders viewpoints manually, which was quite tedious.

Further work is needed on the challenges and problems that can emerge when adopting continuous integration, especially in the area of embedded software. The findings of this study can be extended by both proposing solutions to the identified problems and by including a larger sample of companies using continuous integration for embedded and/or component-based software development. Additionally, developing tool support for Cinders would make it easier to use and could thus be important for increasing its usage by both researchers and practitioners.

Bibliography

[1] R. Ablett, E. Sharlin, F. Maurer, J. Denzinger, and C. Schock. Buildbot: Robotic monitoring of agile software development teams. In *RO-MAN 2007 - The 16th IEEE International Symposium on Robot and Human Interactive Communication*, pages 931–936, Aug 2007.

[2] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, Sept 2008.

[3] R. Baskerville. What design science is not. *European Journal of Information Systems*, 17(5):441–443, 10 2008. Copyright - © Palgrave Macmillan Ltd 2008; Last updated - 2013-10-04.

[4] J. Bosch. *Continuous Software Engineering.* Springer, 2014.

[5] G. Brooks. Team pace keeping build times down. In *Agile 2008 Conference*, pages 294–297, Aug 2008.

[6] I. Crnkovic. Component-based software engineering — new challenges in software development. *Software Focus*, 2(4):127–133, 2001.

[7] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications.* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[8] R. Davison, M. G. Martinsons, and N. Kock. Principles of canonical action research. *Information Systems Journal*, 14(1):65–86, 2004.

[9] A. Debbiche, M. Dienér, and R. Berntsson Svensson. *Challenges When Adopting Continuous Integration: A Case Study*, pages 17–32. Springer International Publishing, Cham, 2014.

[10] J. Downs, J. Hosking, and B. Plimmer. Status communication in agile software teams: A case study. In *2010 Fifth International Conference on Software Engineering Advances*, pages 82–87, Aug 2010.

[11] P. Duvall, S. M. Matyas, and A. Glover. *Continuous Integration: Improving Software Quality and Reducing Risk (The Addison-Wesley Signature Series).* Addison-Wesley Professional, 2007.

[12] M. Fowler. Continuous integration, 2006. (accessed 15 May 2017).

[13] K. Ishikawa. *Guide to quality control.* Industrial engineering and technology. Asian Productivity Organization, 1976.

[14] P. Jalote, R. Munshi, and T. Probsting. *Components Have Test Buddies*, pages 310–319. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[15] A. Janus, R. Dumke, A. Schmietendorf, and J. Jäger. The 3c approach for agile quality assurance. In *2012 3rd International Workshop on Emerging Trends in Software Metrics (WETSoM)*, pages 9–13, June 2012.

[16] N. Kerzazi, F. Khomh, and B. Adams. Why do automated builds break? an empirical study. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 41–50, Sept 2014.

[17] S. Kim, S. Park, J. Yun, and Y. Lee. Automated continuous integration of component-based software: An industrial experience. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 423–426, Sept 2008.

[18] E. Laukkanen, J. Itkonen, and C. Lassenius. Problems, causes and solutions when adopting continuous delivery—a systematic literature review. *Information and Software Technology*, 82:55 – 79, 2017.

[19] E. Laukkanen and M. V. Mäntylä. Build waiting time in continuous integration: An initial interdisciplinary literature review. In *Proceedings of the Second International Workshop on Rapid Continuous Software Engineering*, RCoSE '15, pages 1–4, Piscataway, NJ, USA, 2015. IEEE Press.

[20] T. C. Lethbridge, S. E. Sim, and J. Singer. Studying software engineers: Data collection techniques for software field studies. *Empirical Software Engineering*, 10(3):311–341, 2005.

[21] T. Mårtensson, D. Ståhl, and J. Bosch. *Continuous Integration Applied to Software-Intensive Embedded Systems – Problems and Experiences*, pages 448–457. Springer International Publishing, Cham, 2016.

[22] M. Marzban, Z. Khoshmanesh, and A. Sami. *Cohesion Between Size of Commit and Type of Commit*, pages 231–239. Springer Netherlands, Dordrecht, 2012.

[23] M. Meyer. Continuous integration and its tools. *IEEE Software*, 31(3):14–16, May 2014.

[24] A. Nilsson, J. Bosch, and C. Berger. *Visualizing Testing Activities to Support Continuous Integration: A Multiple Case Study*, pages 171–186. Springer International Publishing, Cham, 2014.

[25] H. H. Olsson, H. Alahyari, and J. Bosch. Climbing the "stairway to heaven" – a mulitiple-case study exploring barriers in the transition from agile development towards continuous deployment of software. In *2012 38th Euromicro Conference on Software Engineering and Advanced Applications*, pages 392–399, Sept 2012.

[26] C. Parnin and R. DeLine. Evaluating cues for resuming interrupted programming tasks. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 93–102, New York, NY, USA, 2010. ACM.

[27] R. Preiel and B. Stachmann. *Git: Distributed Version ControlFundamentals and Workflows.* BrainySoftware.com, 2014.

[28] J. Rasmusson. *Long Build Trouble Shooting Guide*, pages 13–21. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[29] M. Roberts. *Enterprise Continuous Integration Using Binary Dependencies*, pages 194–201. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[30] R. O. Rogers. *Scaling Continuous Integration*, pages 68–76. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[31] P. Runeson and M. Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131, 2009.

[32] C. B. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on software engineering*, 25(4):557–572, 1999.

[33] M. Shahin, M. A. Babar, and L. Zhu. Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *CoRR*, abs/1703.07019, 2017.

[34] D. Ståhl and J. Bosch. Automated software integration flows in industry: A multiple-case study. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 54–63, New York, NY, USA, 2014. ACM.

[35] D. Ståhl and J. Bosch. Industry application of continuous integration modeling: A multiple-case study. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pages 270–279, New York, NY, USA, 2016. ACM.

[36] D. Ståhl and J. Bosch. Modeling continuous integration practice differences in industry software development. *Journal of Systems and Software*, 87:48 – 59, 2014.

[37] D. Ståhl and J. Bosch. Cinders: The continuous integration and delivery architecture framework. *Information and Software Technology*, 83:76–93, 2017.

[38] D. Ståhl, T. Mårtensson, and J. Bosch. The continuity of continuous integration: Correlations and consequences. *Journal of Systems and Software*, 127:150 – 167, 2017.

[39] V. K. Vaishnavi and W. Kuechler, Jr. Design science research in information systems, 2004; last updated: November 15, 2015. [Online; accessed 13-February-2017].

[40] T. van der Storm. The sisyphus continuous integration system. In *11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, pages 335–336, March 2007.

[41] T. van der Storm. Backtracking incremental continuous integration. In *2008 12th European Conference on Software Maintenance and Reengineering*, pages 233–242, April 2008.

[42] R. H. Von Alan, S. T. March, J. Park, and S. Ram. Design science in information systems research. *MIS quarterly*, 28(1):75–105, 2004.

[43] C. Walls. *Embedded Software: The Works.* Newnes, 2012.

[44] C. S. with Dominik Gruntz and S. Murer. *Component Software: Beyond Object-Oriented Programming.* ACM Press/Addison Wesley Publishing Co., New York, 2 edition, 2002.

[45] C. Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th international conference on evaluation and assessment in software engineering*, page 38. ACM, 2014.

[46] H. M. Yuksel, E. Tuzun, E. Gelirli, E. Biyikli, and B. Baykal. Using continuous integration and automated test techniques for a robust c4isr system. In *2009 24th International Symposium on Computer and Information Sciences*, pages 743–748, Sept 2009.

[47] C. Ziftci and J. Reardon. Who broke the build? automatically identifying changes that induce test failures in continuous integration at google scale. In

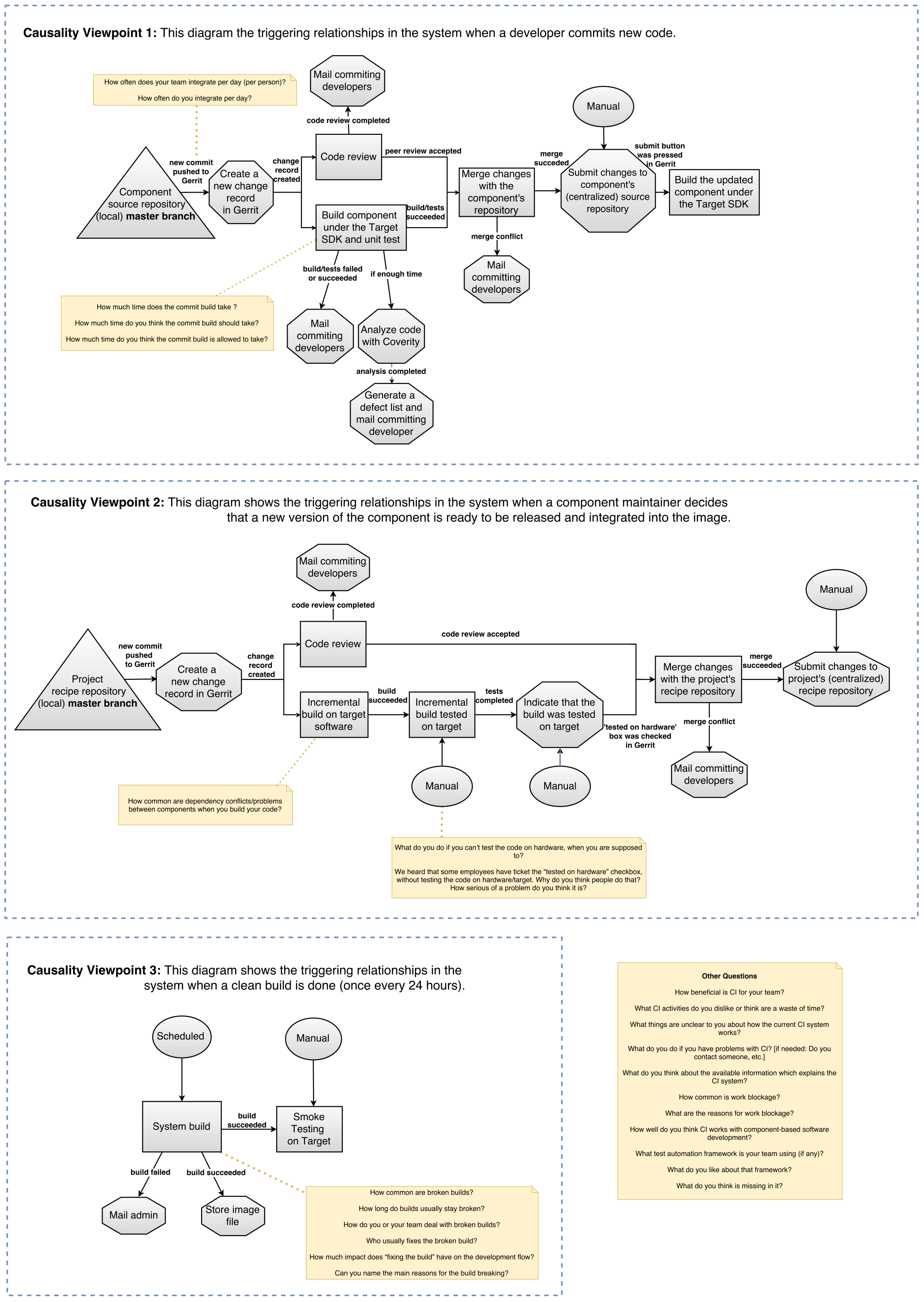
International Conference on Software Engineering (ICSE), Buenos Aires, Argentina, 2017.

A

Appendix 1

Interview Question	**Theme**
1. What is your name, job title and role at [the case company]?	
2. Which team are you in?	
3. How long have you worked at [the case company]?	*Background*
4. How long have you been working in the field of Software development?	
5. How long have you used Continuous Integration?	
6. How often does your team integrate per day (per person)?	
7. How often do you integrate per day?	
8. How beneficial is CI for your team?	
9. What CI activities in the current system do you dislike or think are a waste of time?	
10. What things are unclear to you about how the current CI system works?	*Human & Organizational*
11. What do you do if you have problems with CI? [if needed: do you contact someone, etc.]	
12. What do you think about the available information which explains the CI system?	
13. How much time does the commit build take (i.e. the component build and unit tests)?	
14. How much time do you think the commit build should take?	
15. How common are broken builds (i.e. the nightly ones)?	
16. How long do builds usually stay broken?	
17. How do you or your team deal with broken builds?	
18. Who usually fixes the broken build?	
19. How much impact does "fixing the build" have on the development flow?	*Integration*
20. Can you name the main reasons for the build breaking?	
21. How common is work blockage?	
22. What are the reasons for work blockage?	
23. What are your thoughts on the time the overall integration process takes, i.e. when changes are approved to the mainline?	

Interview Question	**Theme**
24. What kind of testing does your team do?	
25. What do you think about the quality of the current testing process?	
26. What types of testing do you think are missing in the CI system?	
(a) What about automated testing?	
27. How do you think the current testing process can be improved?	
28. What are your thoughts on the current execution time of the tests?	*Testing*
29. What do you do if you can't test the code on hardware, when you are supposed to?	
30. We heard that some employees have ticked the "tested on hardware" checkbox, without testing the code on hardware/target. Why do you think people do that? How serious of a problem do you think it is?	
35. How common are dependency conflicts/problems between components when you build your code?	
36. How well do you think CI works with component-based software development?	*System Design*
35. What test automation framework is your team using (if any)?	
(a) What do you like about that framework?	
(b) What do you think is missing in it?	*Resource*
36. What do you think is important to have in an automation framework?	


Causality Viewpoint 1: This diagram the triggering relationships in the system when a developer commits new code.

How often does your team integrate per day (per person)?

How often do you integrate per day?

How much time does the commit build take ?

How much time do you think the commit build should take?

How much time do you think the commit build is allowed to take?

Causality Viewpoint 2: This diagram shows the triggering relationships in the system when a component maintainer decides that a new version of the component is ready to be released and integrated into the image.

How common are dependency conflicts/problems between components when you build your code?

What do you do if you can't test the code on hardware, when you are supposed to?

We heard that some employees have ticket the "tested on hardware" checkbox, without testing the code on hardware/target. Why do you think people do that? How serious of a problem do you think it is?

Causality Viewpoint 3: This diagram shows the triggering relationships in the system when a clean build is done (once every 24 hours).

How common are broken builds?

How long do builds usually stay broken?

How do you or your team deal with broken builds?

Who usually fixes the broken build?

How much impact does "fixing the build" have on the development flow?

Can you name the main reasons for the build breaking?

Other Questions

How beneficial is CI for your team?

What CI activities do you dislike or think are a waste of time?

What things are unclear to you about how the current CI system works?

What do you do if you have problems with CI? [if needed: Do you contact someone, etc.]

What do you think about the available information which explains the CI system?

How common is work blockage?

What are the reasons for work blockage?

How well do you think CI works with component-based software development?

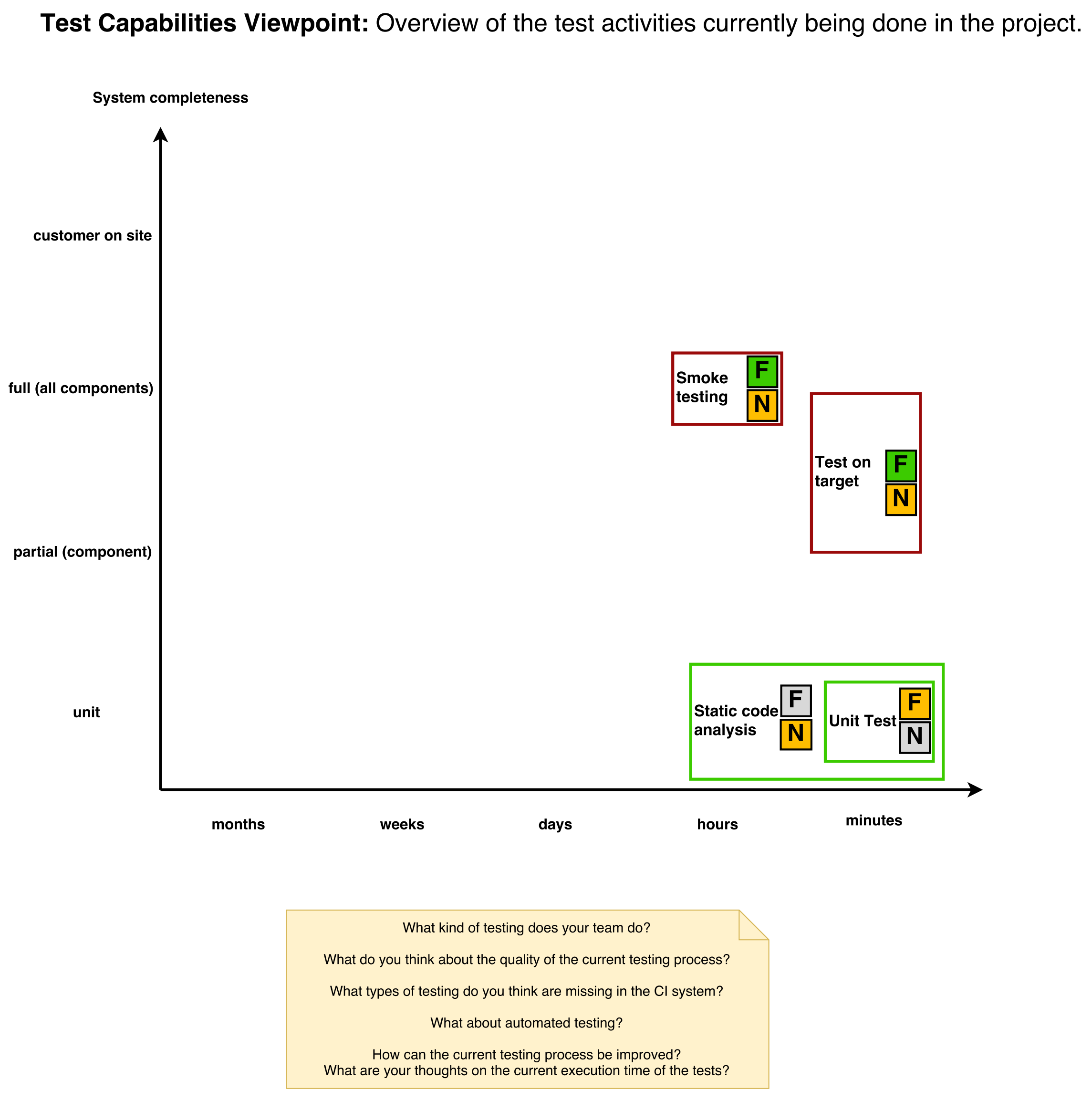
What test automation framework is your team using (if any)?

What do you like about that framework?

What do you think is missing in it?

Component source repository (local) **master branch**
new commit pushed to Gerrit
Create a new change record in Gerrit
change record created
Code review
code review completed
Mail commiting developers
peer review accepted
Build component under the Target SDK and unit test
build/tests succeeded
build/tests failed or succeeded
if enough time
Mail commiting developers
Analyze code with Coverity
analysis completed
Generate a defect list and mail committing developer
Merge changes with the component's repository
merge conflict
Mail committing developers
merge succeded
Manual
submit button was pressed in Gerrit
Submit changes to component's (centralized) source repository
Build the updated component under the Target SDK
Project recipe repository (local) **master branch**
new commit pushed to Gerrit
Create a new change record in Gerrit
change record created
Code review
code review completed
Mail commiting developers
code review accepted
Incremental build on target software
build succeeded
Incremental build tested on target
Manual
tests completed
Indicate that the build was tested on target
Manual
'tested on hardware' box was checked in Gerrit
Merge changes with the project's recipe repository
merge conflict
Mail committing developers
merge succeeded
Manual
Submit changes to project's (centralized) recipe repository
Scheduled
Manual
System build
build succeeded
Smoke Testing on Target
build failed
build succeeded
Mail admin
Store image file

Figure A.1: The Causality Viewpoints with their definitions and relevant interview questions.



What kind of testing does your team do?

What do you think about the quality of the current testing process?

What types of testing do you think are missing in the CI system?

What about automated testing?

How can the current testing process be improved?
What are your thoughts on the current execution time of the tests?

Figure A.2: The Test Capabilities Viewpoint with its definition and relevant interview questions.