Continuous Integration Impediments in Large-Scale Industry Projects

Torvald Mårtensson
Saab AB
Linköping, Sweden
torvald.martensson@saabgroup.com

Daniel Ståhl
Ericsson AB
Linköping, Sweden
daniel.stahl@ericsson.com

Jan Bosch
Chalmers University of Technology
Gothenburg, Sweden
jan@janbosch.com

***Abstract*—Based on interviews with 20 developers from two case study companies that develop large-scale software-intensive embedded systems, this paper presents the main factors that affect how often developers deliver software to the mainline. Further on, the paper describes the continuous integration behaviors in projects where up to 1,000 developers commit to the same mainline. The main factors that could enable more frequent integration of software are: "Activity planning and execution", "System thinking", "Speed" and "Confidence through test activities". Behind these main themes we also present a wide range of sub-categories ("Modular and loosely coupled architecture", "Test selection" etc) which summarizes what the developers themselves see as the continuous integration impediments in large-scale industry projects.**

Keywords—software integration; continuous integration; embedded systems; size; large-scale; modular architecture; loosely coupled architecture

I. INTRODUCTION

Continuous integration is often described as an efficient way of conducting software development. The practice and its benefits are described in both books and other research papers [2,5,12,14]. Still, many projects struggle to realize their implementations of continuous integration [10,13,15,16]. In previous work [17], we have analyzed the continuity of different continuous integration implementations in organizations with hundreds or thousands of developers, and investigated the correlation between the continuity of continuous integration and size of the organization. Related work [4] also describe that large-scale continuous integration is challenging. But the question still remains why the implementations of continuous integration in industry projects differ so much from the practice as it is described in literature, for example in Martin Fowlers popular paper [3]. We have, as one small step to answer this question, in previous work reported our own experiences of factors that could constrain a full implementation of continuous integration [8]. We believe that filling the gap between large-scale industry projects and continuous integration as described in literature is a very topical issue, especially as continuous integration is now implemented by many large-scale organizations (often together with continuous delivery).

The topic of this paper is to answer the following research question: *Which are the impediments that have to be overcome in order to make software developers deliver more frequently?* We will focus on continuous integration implementations for software-intensive embedded systems (software systems combined with electronic and mechanical systems). The contribution of this paper is three-fold. First, it provides interview results from two large-scale industry projects regarding the developers' continuous integration behaviors. Second, it sheds light on how developers describe pros and cons regarding committing to branch or directly to mainline (also called trunk). Third, based on interview results it provides researchers and practitioners with an improved understanding of the main factors that affect how often developers deliver software to the mainline – the main factors that could enable more frequent integration of software. The remainder of this paper is organized as follows. In the next section we present the research method, including a description of the case study companies. This is followed by an analysis of the interview results in Section III. Threats to validity are discussed in Section IV. The paper is then concluded in Section V.

II. RESEARCH METHOD

In order to answer the research question stated in Section I, a series of interviews were conducted with interviewees from two companies. The interviews were conducted according to the selected interview design, and were followed by an analysis of the interview results.

A. Case Study Companies

The case study companies are developing large-scale and complex software for products which also include a significant amount of mechanical and electronic parts. Both companies are multi-national organizations with more than 15,000 employees. In this paper, the companies are referred to as Company A and Company B. Continuous integration practices such as automated testing, private builds and integration build servers are implemented in various ways in different parts of the companies. The software teams commit to a common mainline for each software product. Build, test and analysis of varying system scope and coverage run both on event basis and on fixed schedules, depending on needs and circumstances. A wide range of physical target systems as well as a multitude of both in-house and commercial simulators are used to execute these tests. The interviewees from the case study companies were purposively sampled in line with the guidelines for qualitative data appropriateness that is given by Robson [11] (pp.166-168): "interviewing 'good informants' who have experienced the phenomenon and know the necessary information". The individuals were sampled for scope and variation to represent a wide range of sub-systems (with different characteristics) and project types (size and development context).

B. *Interview Design*

Twenty individual interviews were held with participants from two case study companies (ten at each company). The interviews were conducted as semi-structured interviews, held face-to-face or by phone using an interview guide with pre-defined specific questions. The interviewer was transcribing the interviewee's responses during the interview, and each response was read back to the interviewee to ensure accuracy. The interview questions were sent to the interviewee at least one day in advance to give the interviewee time to reflect before the interview.

The first part of the interview queried the software developer for contextual information, and how the interviewee currently commit their software. This information was gathered as the interviewee's answers may depend on their background and current work situations. In the second part of the interview the interviewee was asked questions about how the interviewee would prefer to commit software, and about the interviewee's opinions about continuous integration practices, which also might affect the answers for the question in the third part of the interview. In the third part, the interviewee was asked to describe impediments that must be removed in order to enable daily commits to the mainline. After the first answer from the interviewee, a list of aspects to consider (found in literature [6, 9]) was presented to the interviewee. This process was designed to allow spontaneous reflections from the interviewee, and then eliciting further in-depth responses.

The word "commit" is used in the research study as the term for integration of a software developer's code into the mainline. As there is no unified terminology, the use of "commit" (or any other word) could be seen as problematic. The word "integrate" could be seen as an alternative, but might create confusion as it could also be interpreted as rebase. The alternative "deliver" has connotations of formality and process, which is not a limitation we like to include. A third alternative is "push", but is really a term connected to a specific family of tools (distributed version control systems). That being said, we consider "commit" to be the best alternative.

C. *Analysis and Validity*

The data on how often the interviewed developers currently commit software and would wish to commit software (collected from the interviews) were collated into a table. Then min, average and max values etc were simply calculated from the information in the tables. The interviewees were quite generous with comments regarding both their opinions about benefits of continuous integration practices and about committing to branch vs directly to mainline. The comments were collated and processed with exploratory analysis. Representative quotes were selected from the transcripts to convey the interviewees' opinions and discussions.

The responses for the last interview question ("what needs to be changed") included a large amount of statements and comments. The interview results were analyzed based on thematic coding analysis as described by Robson [11] (pp.467-481), outlined in the following bullets:

- *Familiarizing with the data*: Reading and re-reading the transcripts, noting down initial ideas.
- *Generating initial codes*: Extracts from the transcripts are marked and coded in a systematic fashion across the entire data set.
- *Identifying themes*: Collating codes into potential themes, gathering all data relevant to each potential theme. Checking if the themes work in relation to the coded extracts and the entire data set. Revising the initial codes and/or themes if necessary.
- *Constructing thematic networks*: Developing a thematic 'map' of the analysis.
- *Integration and interpretation*: Making comparisons between different aspects of the data displayed in networks (clustering and counting statements and comments, attempting to discover the factors underlying the process under investigation, exploring for contrasts and comparisons). Revising the thematic map if necessary. Assessing the quality of the analysis.

The process was conducted iteratively to increase the quality of the analysis. The remaining themes were then described, with representative quotes selected from the transcripts included in the descriptions. Special attention was paid to outliers (interviewee comments that do not fit into the overall pattern) according to the guidelines from Robson [11], in order to strengthen the explanations and isolate the mechanisms involved.

Construct validity, internal validly and external validity have been considered during research design, during the interviews and as part of the analysis (as described in Section IV). The 12 threats to internal validity presented by Cook, Campbell and Day [1] are used to discuss internal validity.

III. Analysis of Interview Results

A. *Background Information*

The interviewed developers were generally very experienced, with an average of 12.15 years of experience of industry software development. Company A interviewees had experience spanning from 4 to 20 years, with an average of 11.6 years. Company B interviewees experience spanned from 4 to 32 years, with an average of 12.7 years.

The developers from Company A were all working in the same project and committed software to the same mainline, although all interviewees were working in different teams and with different subsystems. The project included around 500 software developers, of which about 150 worked with the product code on daily basis. The scope of the Company A project was to develop a new product (with a new platform) during a time-span of several years. The interviewees from Company B worked in several different projects, with the number of developers (committing to the same mainline) spanning from 50 to 1,000. The average number of developers for the projects in the Company B interviews was 626 developers. The scope of the Company B projects was in six of ten cases described by the interviewees as adding new features or systems to an existing product (with an established architecture). Three interviewees described the project as development of a new product with a new platform. One interviewee described the project as both.

B. Rating Continuous Integration

The developers generally had a very positive attitude towards continuous integration. The ratings (1-5) of how software development of large-scale systems benefits from the practices of continuous integration were generally 4 or 5, with an average of 4.13. One of the interviewees refrained from answering the question. Many of the developers added comments, which in most cases were very positive, for example "I see only benefits from it" or "I don't really see what the alternative would be". Some commented on the benefits of shorter feedback cycles, for example "Getting rid of uncertainty, how much merge work do we have ahead of us?" Several interviewees also commented on the difficulties to make continuous integration work ("extremely important, but it's difficult to make it work"). The most commented area was difficulties to establish test suites for regression test and test environments, for example "Potentially it could be great. [...] Our test environments are damn slow." Other voices, however, looked at it from a different angle, stating for example "Particularly the testing aspect I think I have experienced the greatest improvement from. You can always test on the latest."

Two of the interviewees rated the benefit from continuous integration practices as 3, and one interviewee as 2. They all provided comments on the relation between continuous integration and different aspects of system design, e.g. "You don't want to deliver so quickly that you don't have time to think. It's often the interfaces that aren't settled. I like to say that I think we kid ourselves. We think we move faster because we deliver frequently, but often it's an illusion, because what we do deliver isn't thought through."

C. Current Frequency of Commits

The interviewees' responses on frequency of commits were interpreted quite conservatively, e.g. "several times a day" was interpreted as three times a day. The values are also calculated with regards to that a week has seven days (not a five day work week). For example, "every six weeks" is interpreted as 42 days. The interview results on how the interviewees commit to branch and mainline, respectively, were:

- Committing to branch: 16 developers, of which 3 are mixed committers
- Committing directly to mainline: 7 developers, of which 3 are mixed committers

Interviewees who were *committing to branch* commit to a team or a feature branch with a frequency spanning from several times a day to every second day. In this group, two developers were divergent compared to other branch committers, and seem to not yet have established a way of working that could be related to as continuous integration. One of the outliers committed to a branch once every two weeks, a figure quite different from other interviewees. The interviewee commented: "We recently reorganized. Before we didn't do continuous integration [...]." The other outlier interviewee committed to the branch similar to other interviewees, but the branch was committed to mainline the first time after "four months" but was rejected and the next time "after a year".

Interviewees who were *committing directly to mainline* commit software with a frequently spanning from "several times a day" to "once a week". Three of the interviewees are *mixed committers*, meaning that they responded that they commit to both branch and directly to mainline. The frequency of their commits to both branch and mainline are similar to the other interviewees' responses, with the exception of one interviewee who committed the branch to mainline less frequent than other interviewees (70 days between commits). Several of the other interviewees commented that their response referred to development of new functions, adding that corrections is delivered "instantly". That is, during development phases that includes a lot of problem report and corrections, a larger part of the developers are probably also "mixed committers".

The commit frequency for all interviewees are shown in Fig. 1. The branch committers are shown as number 1-16, and mainline committers as number 14-20 (with number 14-16 as the mixed committers). The outliers are number 12 (committing much less frequently to branch) and number 13 (committing the branch much less frequently to the mainline).

A summary of the interviewees' responses is presented in Table I (with an exclusion of the two outliers that seemed not to have established a way of working that could be related to as continuous integration). The interviewees' (except the two outliers) commits to a branch with an average time of 0.84 days between commits, with a span from 0.33 days to 2 days. The branch was committed to the mainline on average time of 22.41 days between commits. The time between commits from branch to mainline differs more between the interviewees, spanning from 2.80 days to as much as 70 days. Mainline committers deliver directly to the mainline, with the most frequently committing interviewee on only 0.33 days between commits and the least frequently on 7 days between commits (on average 2.52 days).

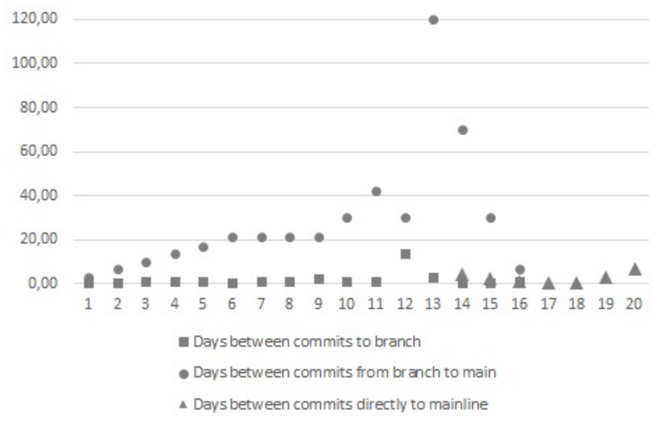

Figure 1. Commit frequency for each interviewee to branch and mainline.

Another observation is that 15 out of 20 of the interviewed developers commit to a branch or directly to the mainline daily, or even more frequently than daily. This can be interpreted as that as many as 75% of the interviewees integrate their software at a frequency that follows the guidelines in Martin Fowlers article [3] (shown in Fig. 2). If the two outliers are removed, the number is as high as 83% (15 out of 18 developers). The question is then why so many of the developers commit to a branch and not directly to the mainline.

D. *Commit to a Branch or Directly to the Mainline?*

There is no correlation between current ways of working (commit to branch or main) and how the interviewees responded that they want to work. Instead there were very mixed responses, where some of the developers wanted to change from committing to branch to committing directly to the mainline (or vice versa) and some wanted to remain in their current way of working. The results are summarized as:

- Prefer to commit to branch: 9 developers
- Prefer to commit directly to mainline: 11 developers

It shall be noted that eight of the nine developers that prefer to commit to branch come from the same case study company. The commit frequency that each of the interviewees would have preferred is shown in Fig. 3 (note that the interviewees are not listed in the same order in Fig. 3 as in Fig. 1).

TABLE I. TIME BETWEEN COMMITS TO BRANCH AND MAINLINE

	Interviewees	Time between commits		
		Min	*Average*	*Max*
Commits to branch (except outliers)	14	0.33 days	0.84 days	2 days
Commits from branch to mainline	14	2.80 days	22.41 days	70 days
Commits directly to mainline	7	0.33 days	2.52 days	7 days

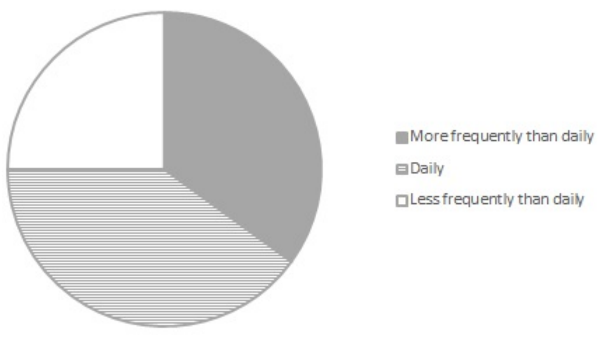

Figure 2. Fifteen of twenty interviewees responded that they commit to a branch or directly to the mainline "daily", or even more frequent (including the two outliers).

The major difference between the data shown in Fig. 3 (preferred way of working) and Fig. 1 (the present situation) is that only nine developers prefer to deliver to a branch, compared to 16 developers which currently deliver to branch (of which three are mixed committers). There are also no longer any outliers similar to what we found in the data that is related to the present situation. A summary of the interviewees' responses is presented in Table II. The interviewees who want to commit to a branch responded that they want to commit with on average 1.24 days between commits, with a span from 0.33 days to 3.5 days. They want the branch to be committed to the mainline on with on average 21.78 days between commits, spanning from 7 days to 63 days. The interviewees who want to commit directly to the mainline want to commit on average every 2.08 days, spanning from 0.06 days (every 30 minutes) to 7 days. The values (min, average and max) are very similar to the corresponding values in Table I, sometimes a bit higher and sometimes a bit lower. This is rather surprising as only two of the twenty interviews responded with the same values on the questions how they work now and how they want to work.

The responses on how often other developers' software should be committed to the mainline were generally the same values as how often the interviewees want their own software to be committed. There is one exception though, where a platform developer responded that updates of the platform should be committed every 4-8 week, but a "functional branch should commit daily".

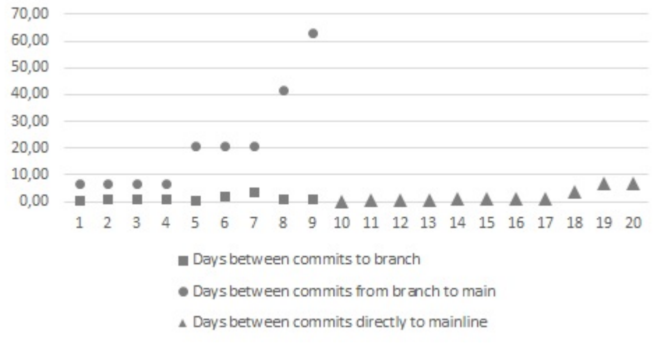

Figure 3. How often do the interviewees want to commit to a branch or to the mainline?

TABLE II. HOW OFTEN DO THE INTERVIEWEES WANT TO COMMIT TO A BRANCH OR TO THE MAINLINE?

	Interviewees	**Time between commits**		
		Min	***Average***	***Max***
Commits to branch	9	0.33 days	1.24 days	3.5 days
Commits from branch to mainline	9	7 days	21.78 days	63 days
Commits directly to mainline	11	0.06 days	2.08 days	7 days

The interviewees were quite generous with comments on pros and cons regarding committing to branch or directly to mainline. The comments can be sorted into four groups, each group supported by statements from 6-9 of the interviewees:

- In a branch you can test your functionality before committing to mainline
- Sometimes it's complicated to break down your work into small pieces that can be integrated directly to the mainline
- If you commit directly to the mainline, you won't have so much merge problems as if you work on a branch
- Working on a branch is more stable than working directly to the mainline
- If you commit directly to the mainline you get feedback faster

Developers from both case study companies described that they had test activities running on the branch, which were run either regularly or before committing from the branch to the mainline. Testing before commit builds confidence in that the software will not brake anything when it's on the mainline. As it seems, in both case study companies the test activities that are run as part of the integration process are not considered to be sufficient by the developers. The developers then chooses to run tests on the branch.

Another problem described by the interviewees from both case study companies is that it is sometimes hard to break down the work into small pieces that can be committed separately to the mainline, e.g. as one developer stated: "The important thing is that you are done with your work. You cannot always chop up things in small pieces and deliver half a function." A difference between the two case study companies is that in one of the companies the developers talk about committing complete functions, in the other case study company the developers separate the terms working software and complete software.

Problems with merge (or rebase) when working with a branch were brought up mainly by developers from one of the companies, e.g. "All the merge conflicts you accumulate. You build up a technical debt. […] But if you make small deliveries to the mainline you know what you have and what to expect." However, another developer from the same company commented that merge and rebase problems also exists when working directly on the mainline: "They batched commits and they failed, and you ended up rebasing for multiple days, so it took a long time to get even small commits in." We can only speculate around the factors behind that this problem was brought up by the developers from only one of the companies. One explanation could be that in an organization working with functional teams, many developers make changes in the same software components (causing merge conflicts), to be compared with an organization that works with component-based teams. Another explanation to that this problem was not described by the developers from one of the companies is that other issues is more in focus right now.

The opinion that a branch is more stable than the mainline was brought up by developers from mainly one of the companies. One aspect was about changing interfaces: "[I want] stable functions around me when I develop my software. I don't want other functions to change which makes me update my interfaces often". The other aspect was to prevent that new commits break existing functions on the mainline, e.g. "People must deliver working functionality that does not destroy other people's work". A developer from the other case study company stated in a similar way: "There have been situations where all others have been banned from integrating, because we had to keep a stable mainline when developing our [critical] feature". The opinion that a branch is more stable than mainline (supported mainly by the developers from one of the case study companies) is mirrored by the opinion that an advantage from committing directly to mainline is faster feedback, which was commented by mainly developers from the other company. As said before, we can only speculate on the factors behind this differences between the companies. One explanation is that one of the companies has invested more in test activities that both provides faster feedback to the developers and keeps the mainline more stable.

Finally, there were also some interesting comments by single interviewees. One interviewee (platform developer) commented on that major changes in the platform should be done in major steps, in order to provide stability for other teams. This corresponds well with the comments on that a branch is more stable than the mainline (above). One interviewee commented on that it's better to work in a branch when you are working with prototyping: "If you work with prototyping where the system changes a lot [on the mainline], you must update your automated tests a lot."

In a strict continuous integration context as described in literature [2,3], developers should commit to the mainline at least once a day. Instead of dismissing the developers that commit to a branch as not practicing continuous integration, we argue that the definition of continuous integration must be seen in a wider perspective. Our previous work [17] as well as related work [4] shows that even though a large-scale organization is integrating and delivering in a very rapid pace, this does not mean that every developer is "continuously integrating" as described in the practice. As we see it, the developers who commit to a branch look at the branch as "their mainline", where they implement and test their function step by step. As eight of the nine developers that prefer to commit to branch come from the same case study company, it is plausible that the preference to commit to branch or mainline for the most part is related to company culture and tradition. However, we believe that product architecture can also be a plausible explanation. The developers in projects within Company B that commit directly to the mainline are often working on a product that is integrated as a binary with other products in a wider context. The overall system (including thousands of developers) is integrated at a frequency far from "every day". On the other hand, Company A develops a large product that consists of several subsystems with different purposes and characteristics, but share the same mainline due to architectural limitations. We argue that continuous integration could not be simplified to discussing how often the developers commit to the same mainline, but also include integration of a subsystem on a branch or integration of binaries built from several different mainlines.

E. *Continuous Integration Impediments*

In the last question in the interviews, we asked about what needs to be changed so that the interviewees can commit software directly to the mainline every day. The responses included a large amount of statements and comments. Extracts from the interview responses were coded and collated into themes. A thematic network were constructed, resulting in a thematic map with four main themes, which in turn consist of several sub-themes. The four main themes and their sub-themes are shown in Table III, together with information about of how many interviewees that provided statements that supported each theme. We consider this summary not only to be valid for an organization that aims for the developers to commit every day, but in general to be the main factors that could enable more frequent integration of software.

As many as 19 of the 20 interviewees talked about different aspects of *activity planning and execution*. The theme "Activity planning and execution" includes "Work breakdown", "Teams and responsibilities" and "Activity sequencing".

The main concern is that in order to commit code more often, *work breakdown* must be on a more detailed level. Several interviewees described this as much of a question of culture ("the attitude of the developers"). There is however also an aspect of how much the work can be broken down into smaller pieces, where the product architecture often sets the limit. The interviewees gave mixed comments on weather commits shall only include complete functions or not (which also is discussed in Section III-D). Some of the interviewed developers described positive experiences from using feature toggles, where the feature is gradually built in the mainline and then unlocked. However, this also requires quite a bit of overhead and problems with architecture and design. Other voices are more skeptical. One developer stated "Is there a value in delivering half-done functions? Especially when you build functions in complex, large-scale systems?" There were also mixed comments on if a commit should include documentation (requirements, design etc) and test cases for the committed software, or if this could be delivered later. We argue that each project must make a decision on which approach for work breakdown that best fits both the system architecture and the development scope.

TABLE III. THE MAIN FACTORS THAT COULD ENABLE MORE FREQUENT INTEGRATION OF SOFTWARE, AND THE NUMBER OF INTERVIEWEES THAT PROVIDED COMMENTS RELATED TO THESE FACTORS

	Total	**Case study A**	**Case study B**
Activity planning and execution	**19**	**10**	**9**
- Work breakdown	15	7	8
- Teams and responsibilities	6	1	5
- Activity sequencing	13	6	7
System thinking	**17**	**8**	**9**
- Modular and loosely coupled architecture	12	5	7
- Developers must think about the complete system	8	5	3
Speed	**19**	**9**	**10**
- Tools and processes that are fast and simple	15	8	7
- Availability of test environments	9	2	7
- Test selection	7	1	6
- Fast feedback from the integration pipeline	9	5	4
Confidence through test activities	**16**	**9**	**7**
- Test before commit	9	6	3
- Regression tests on the mainline	9	5	4
- Reliability of test environments	6	3	3

An adjacent question is how to best handle *teams and responsibilities*. The key factor is that the project organization must both support working with small changes and at the same time take responsibility for the architecture. Some of the developers argued for cross-functional teams, which can efficiently implement a feature from end to end. One developer meant that the organization is more important than the architecture: "components aren't the problem, but that you have component teams". Others argue that a problem with cross-functional team is that the teams don't have ownership, but instead make cross-cutting changes. Component teams would, according to this reasoning, better address product ownership and architectural responsibility. One voice proposes a solution in-between: "if you're working agile with cross functional teams, the system department is even more important".

The comments around *activity sequencing* shows that a majority of the interviewed developers experience a need for more synchronization between teams. One example is the comment "if you don't communicate you can't deliver, because you're not synchronized". This experience come from developers with experiences from component teams as well as developers with experiences from cross-functional teams. Even if a team can work with all components, work must still be synchronized with other teams. "Too many teams, too many chefs in the kitchen", to quote one of the interviewees. Two of the interviewees talked about the importance of systemization, and to decide early on the interfaces between components. The interviewees were however not in favor of a huge systemization phase before starting any software development. One interviewee commented on how to handle prototyping, and said that developers tend to focus on software in a prototyping phase (not work in a structured way with requirements and testing). It is then tempting to stay in the prototyping phase, and postpone work with integration with other systems, robustness, test coverage, documentation etc. We believe that activity sequencing should include not only software development on the mainline, but also activities concerning overall design (architecture) and prototyping activities. The question if and how this should be handled opens up an interesting area of further work.

Factors related to s*ystem thinking* were supported by comments from 17 of 20 of the interviewees. The theme "System thinking" includes "Modular and loosely coupled architecture" and "Developers must think about the complete system".

As stated above, work breakdown can be limited by the architecture. A *modular and loosely coupled architecture* makes it easier to break down the work into small chunks, and work in parallel. Several of the comments from the interviews are related to a modular architecture (with small components) and that this would avoid problems where many teams want to make changes in the same component. One of the interviewed developers even suggests that "every class or whatever would need to be an independent component with a proper interface". Another voice asks for the software to be restructured to smaller components, as large components means that "a huge set of tests must be rerun" before committing the software. Many of the developers also talked about the benefits with loosely coupled architecture, as few dependencies means that you can commit changes independently. Otherwise, as one interviewees said "if you changed one thing it pulled in other stuff as well". Another interviewee proposed a solution with small components in "an architecture that splits the product to components in layers", where the layers represent functional applications, common services and infrastructure.

A bit surprisingly, many of the interviewed developers talked about that *developers must think about the complete system*. This aspect of system thinking is about understanding the functions of the complete product, and not just of your subsystem. If you don't see the whole picture, as one interviewee said, your tests will pass "but other systems will not work". Several ways of getting to understand the whole system was proposed by the interviewees: One interviewee meant that a part of the goals (set by project management) for every team must be "to work on the whole system". Another idea was to let as many developers as possible a chance to test the entire system. Generally, the interviewees were having the perspective that the developer should integrate small deltas with the complete system as often as possible. However, the quite opposite view was stated by one of the interviewees who meant that "there is no value in having feedback from the whole product every day". A more efficient way of working would be, according to this approach, to focus on stabilizing each subsystem first and then integrate the complete product.

Speed was also discussed by 19 of the 20 interviewees. The theme "Speed" includes "Tools and processes that are fast and simple", "Availability of test environments", "Test selection" and "Fast feedback from the integration pipeline".

In previous work [7] we have found that *tools and processes that are fast and simple* are important for the developers. Developers tend to commit to the mainline less frequently if it is time-consuming or complicated. This view is shared by many of the interviewees in this study, who claims that it must be "easy to deliver, and deliver fast". Another voice explicitly states "since the overhead on making a small commit is too big, you end up doing large commits instead". The request for fast tools does not only involve the build itself but also tools that support fast handling of merge conflicts. A few voices also talk about the delivery process as time-consuming, and especially issues related to change management boards. One interviewee suggest that committing should be extremely simple – "just press one button", which sounds similar to the idea of the "integrate button" presented by Duvall [2]. Some of the interviewed developers show a lot of frustration regarding some of their tools, for example "it should be hard not to do things right, but now it's the opposite – it's hard to do things right". Generally, problems with tools seems to steal time from the developers and contribute to irritation and stress. To quote one of the interviewees: "it's strange that it should be like that".

Availability of test environments is also described as an important factor. If you can't get access to sufficient test resources, you cannot commit and the software update is either placed in queue to be tested, or will be part of a larger commit. As one interviewee said: "if you get the [test] resources you need, you can do it more often – one time slot per week is not enough". Some interviewees also describe that although you get a time slot for a test resource, a significant part of the time goes to loading of software and handling problems with the test environment. Availability of test environments can also be limiting the integration tests run after that the developer has committed. One way of working is then batching of multiple changes before testing, which is a more efficient use of test resources. The problem with this way of working is that troubleshooting is much more difficult, compared to if every commit is tested separately.

Test selection means that the suite of regression tests must be constantly updated, where new tests are added and old tests are removed. There were comments from several interviewees like "we often run a lot of unnecessary tests", "we keep test cases too long" or "we executed a lot of test cases, but necessarily not the ones we should have executed". Two of the interviewed developers discussed the usefulness of unit tests. One interviewee meant that "unit tests probably aren't worth the trouble", and that focus instead should be on component tests. Another interviewee expressed the opinion that "organizations that emphasize 100% code coverage in unit tests have a much harder time getting frequent commits to work".

Fast feedback from the integration pipeline is perhaps the main motivator to commit frequently. Feedback from the test activities builds confidence that the software is working, or points out deviations or problems that can be fixed shortly after that they were introduced in the code. An integration pipeline consist of a sequence of test activities that is executed during integration of the committed software, and as scheduled test activities on the mainline. Different test suits are often executed nightly, weekly etc on the different test resources, according to a schedule that aims to provide fast feedback to the developer as well as an efficient use of the most production-like test environments. The interviewees ask for fast feedback, many of them using the word "immediate". One interviewee described that he got the first feedback after 15 minutes, which according to the interviewee was not fast enough. Another interviewee asks for "quick feedback", as otherwise additional problems are introduced on the mainline which makes it harder to isolate the problems.

Sixteen of 20 interviewees talked about different aspects of *confidence through test activities*. The theme "Confidence through test activities" includes "Test before commit", "Regression tests on the mainline" and "Reliability of test environments".

To *test before commit* is described as a prerequisite for a stable mainline, as the integration tests cannot include all test cases for a large-scale product. One interviewee wants everything that is committed to be "stable functions 100% tested". Others asks for "much more testing before commit to main" or that "everyone must test in their branch before delivering". One interviewee emphasize that in order to test efficiently, the test environment available for test before commit must be the same as running tests in mainline. Another voice argues that it must be easier for the developers themselves to set up test jobs in the test scheduling tool.

Regression tests on the mainline is by the interviewees seen as a way to guard mainline quality and stability. To quote one interviewee: "[everyone] must be able to trust mainline stability, we must have tests which protect the mainline". Regression testing includes tests in different levels: unit, component, subsystem etc. One interviewee brings up the problem of maintaining all test cases in the regression test suites, as well as the problem with test selection (described above). Several interviewees talk about problems with intermittent faults, which might originate from both the test environment and the production code. Problems on the mainline that aren't related to the latest commit is described as a large problem which makes troubleshooting more difficult. Better tests is described as the solution for this problem: "it's important to be able to find which commit that broke the build".

Some of the interviewees also describe *reliability of test environments* as a problem. That is, if the test environment does not fully represent the production environment, deviations or problems found during testing can be related to the test environment (and not only to the system under test), or that problems slip through and are found in other test activities. One voice described that problems derive from that different versions of hardware are used, and tests are not always run on at set of hardware where all components has the correct version. Another interviewee talked about differences between simulators (where bespoke hardware is represented by a simulator model that runs on a standard computer) and a rig which uses the real hardware. It is then important ensure that the simulators have the same behavior as the rig, or to specify for the tester which type of tests that can or cannot be run on the simulator.

F. Generalizability

Generally, the themes (the result from the thematic coding analysis) are built from comments and statements that more or less equally come from both case study companies. However, we find that comments within the following sub-themes to a significantly greater degree come from interviewees from one of the companies:

- Teams and responsibilities
- Availability of test environments
- Test selection

Based on our collected data, we can only speculate about the reasons for why these particular areas were not discussed by interviewees from both companies in the same way as all other topics. One explanation could be that issues around teams and responsibilities are more in focus in the company that we know quite recently have changed the organization and the ways of working. In the same way, availability of test environments and test selection might not be hot issues in one of the companies due to recent contributions to handle these type of problems. Another explanation is that the test scope for automated system testing differs between the two companies, which affects the need for test resources. Yet another explanation to that these problems were not described by the developers from one of the companies is that other issues is more in focus right now. However, most of the themes are equally supported by interviewees from both case study companies and there are no major differences to be found in our top-level themes. We argue that this supports the generalizability of the results of this study (external validity).

We argue that the factors in Table III generally are valid for of all types of large-scale and complex software systems. However, development of software-intensive embedded systems (as our two case study companies) adds extra complexity in the following areas:

- The factors that are related to test environments are more complicated to handle if the product runs on bespoke hardware.
- The factors in the theme "System thinking" is more complex if the product covers many technology fields (for example an electronic power system, a navigation system, a communication system etc).
- Tools and processes are more controlled if the product operates in a highly regulated environment (such as medical devices).

IV. Threats to Validity

This section discusses threats to construct validity, internal validity and external validity.

A. *Threats to Construct Validity*

One must always consider that a different set of questions and a different context for the interviews can lead to a different focus in the interviewees' responses. In order to handle threats against construct validity, the interview guide was designed with a terminology that we see as most commonly used (see Section II-B). Open questions were purposely designed to fit both developers who commit to a branch and those who commit directly to the mainline. We also describe the interview guide used in the study and the background of the interviewees and the case study companies.

B. *Threats to Internal Validity*

Of the 12 threats to internal validity listed by Cook, Campbell and Day [1], we consider Selection, Ambiguity about causal direction and Compensatory rivalry relevant to this work:

- *Selection*: The interviewees from both case study companies were purposively sampled in line with the guidelines for qualitative data appropriateness given by Robson [11] (see Section II-A). Due to this, the interviewees represent a wide range of sub-systems (with different characteristics) and project types (size and development context). Considering the rationale of these samplings and the fact that Robson considers this type of sampling superior for this type of study in order to secure appropriateness, we consider this threat to be mitigated.
- *Ambiguity about causal direction*: While we in this study discuss correlation, we are very careful about making statements regarding causation. Statements that include cause and effect are collected from the interview results, and not introduced in the interpretation of the data. Due to this, we consider this threat to be mitigated.
- *Compensatory rivalry*: When performing interviews and comparing scores or performance, the threat of compensatory rivalry must always be considered. Therefore, the questions in our interviews (see Section II-B) were deliberately designed to be as value neutral as possible, rather than judging the interviewee's performance or skills. However, our experiences from previous work is that we found the interviewed engineers more prone to self-criticism than to self-praise.

C. *Threats to External Validity*

The study is based on interviews from two case study companies. It is conceivable that the findings from this study are only valid for these companies, or only for companies that develop large-scale and complex software products which also include a significant amount of mechanical and electronic parts (software-intensive embedded systems). However, related work [4] also describes that large-scale continuous integration is challenging, which shows that the problems described in this paper is evidently not isolated to our two case study companies.

As described in Section III-F, most of the themes are equally supported by interviewees from both case study companies and there are no major differences to be found in our top-level themes. We argue that this supports the generalizability of the results of this study (external validity). Based on the analysis in Section III, we argue that the factors in Table III generally are valid for of all types of large-scale and complex software systems.

V. CONCLUSION

In this paper we have presented interview results from 20 developers with an experience of 4-32 years (on average more than 12 years) working in large-scale software development projects. The interviewees come from two case study companies, which both develop large-scale and complex software systems for products which also include a significant amount of mechanical and electronic parts. We describe the developers' integration behaviors, as well as how they look at pros and cons regarding committing to branch or directly to the mainline.

The interviewed developers currently commit either to a branch or directly to the mainline, or in some cases both (described in Section III-C). The interviewees commit to a branch on a frequency spanning from multiple times a day to every second day. Commits directly to the mainline span from multiple times a day to once a week. Fifteen of 20 developers commit to a branch or directly to the mainline on daily basis, or even more frequently than daily. We find that the developers who commit to a branch *look at the branch as "their mainline"*, where they implement and test their function step by step.

The interviewed developers' attitudes to committing to a branch or directly to the mainline are mixed (as presented in Section III-D). Nine of the twenty interviewees prefer to commit to a branch, and the others prefer to commit directly to the mainline. Instead of dismissing the developers that commit to a branch as not practicing continuous integration, we argue that the definition of continuous integration must be seen in a wider perspective. We argue that continuous integration could not be simplified to discussing how the developers commit to the same mainline, but also includes integration of a subsystem on a branch or integration of binaries built from several different mainlines. Continuous integration is not necessarily black or white, but there is a grey zone where one must consider what the system which one integrates is, and that arguably there are levels of continuous integration. Integration within your component or sub-system, integration with the larger system and integration with any system-of-systems layer on top of that – and there may be different integration frequencies at different levels. In other words, one might be *practicing continuous integration at one level, but not at another*, and this may have different pros and cons depending on the context.

Based on results from the interviews (presented in Section III-E) we find that the main factors that can enable more frequent integration of software are:

- Activity planning and execution: *Work breakdown*, *Teams and responsibilities* and *Activity sequencing*

- System thinking: *Modular and loosely coupled architecture* and *Developers must think about the complete system*
- Speed: *Tools and processes that are fast and simple*, *Availability of test environments*, *Test selection*, and *Fast feedback from the integration pipeline*
- Confidence through test activities: *Test before commit*, *Regression tests on the mainline* and *Reliability of test environments*

Our findings are based on case study companies that are developing software-intensive embedded systems (on a large scale). The areas that are especially related to the characteristics of software-intensive systems are presented in Section III-F. We argue that if these issues are taken into account, our findings on which factors that enable more frequent integration can be applied to all types of development of large-scale and complex software systems.

A. Further Work

In addition to the results presented in the analysis and the conclusions, we believe that this study also opens up several interesting areas of further work.

In Section III-C we found that during development phases that include a lot of problem reports and corrections, a larger part of the developers are probably also "mixed committers". An area of further work is to examine how the developers' continuous integration behaviors change during different development phases.

In Section III-D we discuss explanations to why developers see their branch as "their mainline". A large area of further work is to examine how this is related to company culture and tradition, the product architecture or other factors. We also believe that further work should examine how the practice of continuous integration should not only discuss how often the developers commit to the same mainline, but also include integration of a subsystem on a branch or integration of binaries built from several different mainlines.

Section III-E opens a wide range of fields for further work, regarding solution approaches to handle the problems and impediments that are described. We believe that one of the most interesting topics is to find efficient ways for work breakdown that take into account the limitations of the architecture. Another topic is to find models or ways of working for activity planning that include both system design and implementation in software. Third, simulators are in many cases used instead of expensive bespoke hardware, but questions still need to be answered about how to handle for example real-time aspects or testing of hardware characteristics.

Acknowledgment

The authors would like to thank all the participating engineers for their insights, patience and willingness to share their experiences and data with us.

References

[1] T. D. Cook, D. T. Campbell and A. Day, "Quasi-experimentation: Design & analysis issues for field settings", Houghton Mifflin Boston, vol. 351, 1979.

[2] P. Duvall, Continuous Integration, Addison Wesley, 2007.

[3] M. Fowler, "Continuous Integration", http://www.martinfowler.com/articles/ContinuousIntegration.html, 2006, Accessed online August 12 2016.

[4] J. Humble, "The Case for Continuous Delivery", https://www.thoughtworks.com/insights/blog/case-continuous-delivery, 2014, Accessed online November 30 2016.

[5] J. Humble and D. Farley, Continuous Delivery, Addison Wesley, 2011.

[6] I. Jacobson, G. Booch and J. Rumbaugh, The unified software development process, Addison Wesley, 1999.

[7] T. Mårtensson, P. Hammarström and J. Bosch, "Continuous integration is not about build systems", In review.

[8] T. Mårtensson, D. Ståhl and J. Bosch, "Continuous integration applied to software-intensive embedded systems – problems and experiences", The 17th International Conference on Product-Focused Software Process Improvement, Springer, 2016, pp. 448-457.

[9] Project Management Institute, A Guide to the Project Management Body of Knowledge (PMBOK Guide) – Fifth Edition, 2013.

[10] M. Roberts, "Enterprise continuous integration using binary dependencies", Extreme Programming and Agile Processes in Software Engineering, Springer, 2004, pp. 194–201,.

[11] C. Robson and K. McCartan, Real World Research, 4th ed, Wiley, 2016

[12] P. Rodriguez, et al, "Continuous deployment of software intensive products and services: A systematic mapping study", Journal of Systems and Software 123, 2017, pp. 263-291.

[13] R. Owen Rogers, "Scaling continuous integration". Extreme Programming and Agile Processes in Software Engineering, Springer, 2004, pp. 68–76.

[14] D. Ståhl, J. Bosch, "Experienced benefits of continuous integration in industry software product development: A case study", The 12th IASTED International Conference on Software Engineering, 2013, pp. 736–743.

[15] D. Ståhl and J. Bosch, "Automated software integration flows in industry: a multiple-case study", Companion Proceedings of the 36th International Conference on Software Engineering, ACM, 2014, pp. 54–63.

[16] D. Ståhl, K. Hallén and J. Bosch, "Achieving traceability in large scale continuous integration and delivery", Empirical Software Engineering, 2016, pp. 1-29, In press.

[17] D. Ståhl, T. Mårtensson and J. Bosch, "The continuity of continuous integration: correlations and consequences", Journal of Systems and Software, 2017, In press.