Chapter 7. Automated Software Integration Flows in Industry: A Multiple-Case Study

This chapter is published as:

Abstract
There is a steadily increasing interest in the agile practice of continuous integration. Consequently, there is great diversity in how it is interpreted and implemented, and a need to study, document and analyze how automated software integration flows are implemented in the industry today. In this paper we study five separate cases, using a descriptive model developed to address the variation points in continuous integration practice discovered in literature. Each case is discussed and evaluated individually, whereupon six guidelines for the design and implementation of automated software integration are presented. Furthermore, the descriptive model used to document the cases is evaluated and evolved.

7.1 Introduction

Continuous integration is an agile software development practice, involving automation and frequent integration of software changes. It was popularized in the late '90s as part of eXtreme Programming [Beck 2000], and since then numerous benefits to software development have been reported.

In previous work, we have investigated the effects of continuous integration, as experienced by software development project members [Ståhl 2013], as well as the rich variation of continuous integration interpretations and implementations in evidence in contemporary literature [Ståhl 2014a]. Given that continuous integration is a highly popular agile practice [West 2010] with steadily growing interest over the last decade13 we consider this to be an important area of research: while various benefits, such as improved release frequency and predictability [Goodman 2008], increased productivity [Miller 2008] and improved communication [Downs 2010] are reported by some, but not experienced by all, we know very little about what actually causes a particular benefit to manifest, or to not manifest. We have conducted a multiple-case study, documenting both software integration practice and experienced effects in each individual case, to improve our understanding of the choices facing industry professionals in designing their software integration flows, and the consequences of those choices.

The contribution of this paper is that it provides multiple examples of how the problem of software integration — from small to very large scale — is tackled in the industry today, using a descriptive model. Each case is individually evaluated based on previously established differences

13 Google trends: continuous integration.
in practice, and generalized guidelines for the design of automated software integration are derived from the studied cases. In addition, the model used to document the studied cases is evolved and validated.

The remainder of this paper is structured as follows. The next section provides a short background of our study, introducing the reader to the subject and the model used to document the cases. In Section 7.3 the research method is described. Then, in Section 7.4, the studied cases are detailed and the collected data is presented. In Section 7.5 generalized guidelines based on the multiple-case study are proposed. The paper is then concluded in Section 7.6.

7.2 Background

As software professionals with experience from developing, maintaining and supporting continuous integration in a number of industry cases and contexts, we have seen the term "continuous integration" used to describe very different practices. In previous work we have confirmed that industry professionals' experiences of continuous integration vary greatly [Ståhl 2013], and that there is a large number of variation points where continuous integration, as described in literature, varies [Ståhl 2014a]. This prompted us to develop a descriptive model to capture these differences, allowing us to more accurately study, document and compare different variants. This model builds on the concept of integration as a Directed Acyclic Graph (DAG), as suggested by [Beaumont 2012], with automated activities constituting interdependent nodes in that graph.

Based on our findings we are today hesitant to use the term "continuous integration", as it implies a degree of consensus and similarity which we do not find evident in industry practice. For instance, we tend to find ostensibly "continuous" cases where activities are carried out much more infrequently than some observers might consider to qualify as being continuous. Moreover, these frequencies vary from activity to activity in the same case. For the purposes of this study we therefore refrain from judging the "continuousness" of cases, and to emphasize this we instead use automated integration flows as an umbrella term. Automated, because automation is one of few aspects of continuous integration in which there is consensus in literature, with sources e.g. stating that "an automated integration server [...] guarantees that an integration build will happen" [Rogers 2004] or even considering it a criterion for success that "all [continuous integration] steps pass without error or human intervention" [Rasmusson 2004]. Flows, because source changes pass through the graph of interconnected integration activities, in sequence or in parallel, to reach some desired state, typically involving some degree of verification. In this way one's level of confidence in these source changes, or derivatives thereof (e.g. versions of the product), may successively increase as they progress through the system.

Consequently, though all the cases in our study are considered continuous integration by their respective companies, we describe them as automated integration flows consisting of interconnected automated activities, each with its respective scope and characteristics. Indeed, given the great degree of variation in literature previously established, we would expect a detailed industry multiple-case study to reveal a number of differences between these cases. Such differences include, but aren't limited to, the anatomy of the integration flow itself, communication methods and consequent effectiveness in communication, and the level of ambition of the integration flow. Furthermore, we would expect divergent scopes, not just including various forms of regression testing, but numerous other activities, e.g. performance and stability testing, static and dynamic analysis, as well as deployment activities. We would also anticipate differences in the speed and
frequency of the automated activities, ultimately affecting the speed at which the project members receive feedback from them, as well as in the degree of parallelism and modularity of the integration flow.

7.3 Research Method

The research was conducted as a multiple-case study of the integration flows of industry software development projects, with subsequent analysis of the results gathered from that multiple-case study.

7.3.1 Multiple-Case Study

The multiple-case study included five separate software development projects from four different companies: Ericsson AB, Jeppesen, Saab AB and Volvo Cars. The research was carried out as an Associated Project of the Software Center\textsuperscript{14} initiative at Chalmers University of Technology, with all participant companies being Software Center partners. Each of these companies runs a multitude of software development projects, from which we selected a total of five as cases for the study. In selecting the cases we aimed for the best possible spread in both size (as in number of individuals active in the project) and longevity (as in how long the project has been under active development), while at the same time having to take factors such as accessibility and particular wishes of the participating companies into account. The headcount of the studied cases ranges from 7 to 1200, and their longevity from 2 to 20 years.

Apart from attempting to achieve a good distribution, our only qualifying criterion in the case selection process was that of automation: only automatically triggered and executed flows were included in the case studies. Apart from the reasons discussed in Section 7.2, we also expect automated activities to be more consistent in their behavior over time than corresponding manual activities, thereby making the automated integration flow as a whole a more consistent and reliable case to study.

It shall be noted that the term "project" does not necessarily correspond to the studied companies' own terminology or classification. They may be referred to "programs", rather than "projects", or consist of multiple smaller projects, or constitute only a small part of a much larger project. Instead, for the purposes of this research, "project" shall be understood to mean a clearly delineated integration flow, with one or more sources and one or more sinks, through which changes made to a product or set of products are processed and/or verified. This integration flow may itself be a subset of a larger development context and, if so, the study and all its questions shall be understood to limit their scope to that smaller subset.

Each case study was separated into two distinct parts: the project's integration flow model and the effects of its software integration flow as experienced by the project members. These parts are described in detail below.

\textsuperscript{14} \url{http://www.software-center.se}
7.3.1.1 Integration Flow Model

The aim of the first part of each case study was to document the actual anatomy and characteristics of the project's integration flow, and for this purpose we applied the descriptive model developed by us in previous work (see Section 7.2), with minor refinements (see Section 7.4.1). It represents automated activities (e.g. testing, analysis and packaging), external triggering factors and input as nodes in a graph, with triggering and input relationships connecting them. The data used to populate the model was gathered through a combination of workshops with software integration engineers and analysis of available data, such as configuration files and logs. Some of the data (e.g. building the anatomy and result handling information) was drawn exclusively from discussions in such workshops, while other characteristics (i.e. build-duration, build-frequency and trigger-frequency) would be gathered by analyzing data if available, and if not, from responsible engineers' testimony. Conversely, due to its inherent unpredictability and presumed irregularity, fault-duration would only be gathered from data and, if such data would be unavailable, omitted from the study.

For further details on the descriptive model, its data fields and their meaning, please see [Ståhl 2014a].

7.3.1.2 Experienced Effects

The second part of each case study consisted of individual interviews with six project members: two developers, two testers, one project manager and one line manager (or as close equivalents thereof as possible, would any of these roles not be present in the project), totaling 30 interviews for the multiple-case study. The interviewees were asked to rate on a scale from -7 to +7 to what extent they had experienced the project's integration flow contributing to various effects, with -7 and +7 representing a very negative and a very positive contribution, respectively. A score of 0 would represent either no experienced contribution, or absence of the capability in question. In case the interviewee failed to understand the question or considered it to be inapplicable, he or she would also be given the option to not respond. Apart from the scoring of their experiences, additional information and testimony volunteered by the interviewees provided qualitative context. In total, each interviewee was asked fourteen questions: To what extent have you experienced that your project’s software integration flow affects...

- E1. ... the adoption of agile practices, other than continuous integration?
- E2. ... agile testing, in the sense of automated customer acceptance tests?
- E3. ... agile testing, in the sense of writing unit tests in conjunction with new production code?
- E4. ... communication in general within the project?
- E5. ... communication within the individual teams?
- E6. ... communication between teams?
- E7. ... developer productivity in general?
- E8. ... the developers' ability to check in with a minimum of overhead?
- E9. ... the developers' ability to effectively work in parallel in the same source context?
- E10. ... the developers' ability to effectively deal with rebasing and merging?
- E11. ... the developers' ability to effectively troubleshoot faults?
These questions were derived from those used by us in previous work [Ståhl 2013], which in turn were based on continuous integration benefits reported in literature. They can be grouped into four topics: supporting agile practices (questions E2-E3), supporting communication (questions E5-E6), improving productivity (questions E8-E11) and improving predictability (questions E13-E14). However, we considered it plausible that within each topic there may be effects other than those covered by the literature on which the questions were based. Consequently, for each topic we added a more generalized question in order to capture any such effects (questions E1, E4, E7 and E12).

Within each category of project members, in case there were multiple candidates (as would typically be the case for developers and testers), the company's contact person for the study was asked to choose randomly, with the only limitation that the interviewees be competent and experienced enough to understand and answer the questions.

7.3.2 Analysis

With each of the case studies completed, the gathered data was analyzed for the following purposes:

- **Model Analysis.** The validity of the model used to document the cases and any need for refinement was analyzed based on the gathered data along with comments and suggestions from the interviewed engineers.

- **Case Evaluation.** The models, scores and comments by interviewees and engineers were analyzed against the background outlined in Section 7.2. After individual evaluations, the complete set of cases was then analyzed, based on which guidelines for the design of automated integration flows were proposed.

7.4 Case Study

This section presents the participant companies and details the multiple-case study, including the application and evolution of the model.

7.4.1 Model Evolution

As described in Section 7.3.1.1, a model for describing software integration flows proposed by us in previous work [Ståhl 2014a] was used in the case study. Part of applying the model to the studied cases was to also look for insufficiencies and areas of improvement. To this end, input and change proposals from the interviewed engineers were welcomed. As a result, already in the very first case it became apparent that even though the integration engineers claimed to "really like the model" and that its application gave a sobering and educational perspective on their day-to-day work, there was a problem with ambiguity in how the triggering of and input to activities was represented in the original version of the model. Consequently, the model was updated to include two separate
relationships: trigger and input, with a new node type representing external triggering factors and the \textit{trigger-frequency} attribute moved from the activity node to the trigger relationship (see Figure 21).

The evolved model was successfully applied to all the studied cases, with no further cause to revise it being encountered. The only situation where we experienced some difficulty was in representing the extreme parallelism of case E. We do not consider this to be so much a problem with the model, however, as with comprehending and documenting such a vast number of activities in a limited amount of time. Indeed, it was repeatedly commented by employees in case E that a major issue with their integration flow was the heavy and obscure management of the very large number of components and their corresponding automated activities (see Section 7.4.2.5).

Figure 21: The evolved meta-model.

### 7.4.2 Cases

The multiple-case study included five independent industry software development projects from four companies (see Section 7.3.1). Each case is presented below, with the gathered data, and discussed from the perspective of previously established variations in software integration practice (see Section 7.2). It should be noted that the integration flow models provided below are heavily compressed: all attributes have been omitted, with noteworthy examples highlighted and discussed in text, rather than included in the figures. For each case, however, with the exception of \textit{trigger-frequency} (see Section 7.4.1), available data for all 14 attributes proposed by [Ståhl 2014a] was collected for every activity node. The intent of providing the compressed models is to provide the
reader with an understanding of the overall anatomy, and a reference while reading the descriptions and evaluations of the cases, rather than a visualization of the complete data set. We have in past experience found that merely providing a brief description of a given integration flow is insufficient, as there is too much confusion in terminology and too many possible variation points. In contrast, a graphical representation showing the precise paths taken by software artifacts in the integration flow gives a more solid base for meaningful comparison and experience exchange. It is for this reason the model was first developed, and compressed representations of the studied integration flows have been included in Figures 22, 24, 26, 28 and 30. It should be noted that all the node names in these figures have been changed, both for reasons of reader comprehension and of confidentiality.

7.4.2.1 Case A

Case A is a product development project at Ericsson AB. The product has been under development for approximately six years, with in excess of 1200 employees actively developing its software. The foundations of the project's current software integration flow were laid in an early and much smaller part of the project's life-cycle, with the intent of supporting test automation and rapid integration, although the term "continuous integration" wasn't used at the time. A compressed model of the integration flow is shown in Figure 22.

Figure 22: A compressed model of the integration flow of case A.

The product is built daily, with the resulting package going into acceptance testing. In parallel with this, a suite of unit tests and static and dynamic analysis activities (e.g. code coverage, memory consistency and code complexity checks) are performed. In order to achieve faster feedback, a basic
subset of the acceptance tests are executed in an activity which runs on an hourly schedule (the full 
Acceptance Test activity takes approximately 7 hours). This Hourly Basic Acceptance Test activity 
uses the latest product build, assuming it passes the Product Smoke Test Screening, with delta 
packages automatically added on top as parts of the product are changed. A typical work day starts 
out with a "fresh" product version running in the Hourly Basic Acceptance Test activity. Then, as 
changes are introduced during the day, delta packages are accumulated, only to be wiped clean 
when a new version, incorporating the day's changes, is verified by the Product Smoke Test Screening.

There are a few things especially worthy of mention in this integration flow. One is the process of 
actually making a change to the SCM Mainline, i.e. the developer pushing his or her code to the 
common development branch. Apart from the developers being expected to execute relevant test 
cases before pushing, each change (or "delivery", as the project members refer to it) must be 
manually approved by a project manager before it is allowed onto the SCM Mainline. The 
consequence of this is a queuing situation, with an elaborate ticket system having sprung up to 
support it, where low priority "deliveries" can be put on hold for extended periods of time. 
Furthermore, the average size of a "delivery" is much larger than what a developer produces in a 
few hours or even days of work, which is not surprising given that everything produced by such a 
large number of developers must pass through this needle's eye, partitioned into a limited set of 
"deliveries". Consequently, the frequency of the hourly and daily activities is deceptive: the actual 
speed and frequency at which the average project member receives feedback is dramatically lower, 
and varies depending on how their changes fare in the queue.

It should also be noted that the SCM Mainline node affords no parallelism: if there is a blockage, 
as interviewees testify is frequently the case, it effectively halts the entire project. This is coupled 
with a stated lack of communication: several interviewees profess little knowledge of the status or 
content of the automated activities, e.g. saying that there is "lots of valuable information in there, 
but we have failed at communicating it". This would seem to be, at least in part, related to the fact 
that several of the automated activities do not yield a clear "pass or fail" result. Instead, they 
generate logs, which are then inspected in order to determine whether there were any problems — 
something only a small minority of project members actually do, or are even capable of doing.
In combination, these factors have resulted in different parts of the project having developed their own integration processes prior to integrating with SCM Mainline, i.e. additional integration flows upstream of what is depicted in Figure 22, with some of these tributary flows being quite elaborate. This is interesting in that it represents a different approach to parallelism than that of modularization, as sometimes proposed in literature [Roberts 2004, Rogers 2004, v. d. Storm 2007]: what we would describe as parallelism by organization, as opposed to parallelism by architecture. While the exact consequences of this approach could not be established in our study, interviewees stated that a strong "us and them" mentality prevailed in the project. They also suggested that this parallelization approach wasn't the result of any conscious decision, but rather believed that it had spontaneously emerged as a response to the lack of modularization, as the architecture hadn't evolved to accommodate the scaling up of the project.

As seen in Figure 23, the experienced effect responses from case A project members are mixed, with particularly negative response to questions E1 (adoption of agile practices), E7, E8, E9, E10 (various aspects of developer productivity) and E14 (performing integration off the critical path). In their comments and explanations, the interviewees consistently returned to the fact that the automated integration was too far downstream — by the time changes actually made it into SCM Mainline all the "important" activities, the things that the project members felt actually had an impact (e.g. integrating and testing locally within their team or sub-project), had already taken place. With regards to communication, one interviewee opined that the integration flow "draws...
attention away from what really matters" and is misleading as to what is actually going on in the project. With regards to developer productivity it was described how, while in queue to "deliver" to SCM Mainline, developers would be caught in perpetual cycles of re-basing and re-testing as the SCM Mainline was updated, yet not being afforded a slot to "deliver" themselves.

### 7.4.2.2 Case B

Case B is an environment tracking and management system for military aircraft, developed by Saab AB. It has been under development for approximately 14 years, with 10 employees currently involved. The code depends on a project developed by another in-house group (Dependency1), and is in turn integrated by two other projects (Dependent1 and Dependent2). This integration flow can be seen in the model in Figure 24.

One of the striking features of this integration flow is the inconsistent trigger and input relationships. All activities can be triggered manually, but some are also triggered by changes in their respective code repositories, while others rely on scheduled triggering. Similarly, some are triggered by a successful build of one of their dependencies, yet others are not. Furthermore, while Product Instant Build integrates output generated by Dependency1 Instant Build, Dependent1 Nightly Build instead accesses the Dependency1 code repository for its input. The same behavior is displayed by Dependent2 Build, which is triggered by the successful execution of Product Instant Build, but accesses the Product SCM system instead of any output generated by the upstream build activity. Unsurprisingly, there is little central control or management of the integration flow. Rather, a number of projects share the same Jenkins\(^\text{15}\) continuous integration server, in which they configure their own jobs and depend on each other as they see fit.

\(^{15}\) [http://jenkins-ci.org](http://jenkins-ci.org)
The scope of case B's integration flow is limited: Product Instant Build performs functional testing, while the other activities do not test at all, but merely perform compilation. The results are communicated via web pages generated by Jenkins, with some activities taking less than five minutes and others averaging 100 minutes. There is a limited degree of parallelism: one activity can run independent of failures in the others, but since they (with the exception of Product Instant Build) do not consume the output of the latest successful dependency build, but rather access its code repository directly, any faults introduced in one component propagate to its dependents unchecked.

Figure 25 displays interviewee responses for experienced effects in case B. As can be seen in the figure, responses were almost exclusively either indifferent or positive, though not exceedingly so. The general attitude among the interviewees was that the integration flow is a benefit and support, though making use of it is clearly optional: one developer, despite having been involved in the project for some time, was only vaguely aware of its existence and hadn't experienced any effects whatsoever, giving a score of zero to every single question. In other words, the automated integration flow constitutes an appreciated, but highly optional, part of the development environment of case B, developed and maintained by individual enthusiasts.
7.4.2.3 Case C

A visualization system for military aircraft developed by Saab AB, case C employs 7 people and has been in development for 10 years. Compared to other cases in the study, its integration flow is simple, with only two activities (see Figure 26).

The build activities, *Product Build Configuration A* and *Product Build Configuration B*, are triggered by changes in the *Product SCM* system. While they can in theory be triggered manually, this never happened during the period studied. The builds also bring in dependencies deployed to a separate *Dependency Repository*, though updates to it do not cause any new builds. Both build activities are short, with a duration of 2-3 minutes, with a very limited scope: the product is only compiled, without any automatic testing or analysis taking place. Status is communicated via web pages generated by Jenkins and mails sent out to committing developers upon failure, meaning that they themselves do not have to actively find out if they caused a fault.

Figure 25: Distribution of interview responses for each experienced effect (see Section 5.3.1.2) in case B, with the horizontal bars representing quartiles zero through four.
The interviewee responses to experienced effects in case C, shown in Figure 27, were largely positive, though not unanimously so. Negative experiences were chiefly voiced by individuals with experience from other companies and projects, who were frustrated by the very small scope of the integration flow, in two senses. First, with no testing or analysis taking place the automated activities aren’t comprehensive enough, and may possibly even be harmful, as they can cause a false sense of security and even hamper much needed improvements in multiple areas. Second, the automated integration is limited to the product in isolation, and does not include other products in the larger system environment, meaning that only local and relatively trivial faults are caught by it. Other project members, on the other hand, explicitly stated that they didn’t have outside experience and were therefore unable to make comparisons, but were happy with the current setup as it was an improvement over having no automated integration at all, as had until recently been the case.
7.4.2.4 Case D

The youngest project in the study, case D has been running for two years and employs approximately 50 personnel. It is responsible for developing part of the on-board software for Volvo Cars’ electric vehicles. The integration flow of case D is to a great extent shaped by the fact that the software is developed using AUTOSAR\textsuperscript{16} modeling. As can be seen in Figure 28, it is relatively elaborate.

There are two main entry points in the integration flow depicted in Figure 28. One is the manual triggering of Baseline Definition, creating a baseline of the current source models, which will be used in subsequent activities. The other is the scheduled nightly execution of Model Variant Adjustment and Test Model Generation. These aren't triggered by Baseline Definition, but use its output. Downstream activities are then triggered by Model Variant Adjustment. Noteworthy is that in order to actually compile the software, an XML file describing the interfaces of all the components of the system, of which case D is only one part, must be created. The responsibility of XML Processing and Repair is to prepare an XML document describing the parts developed within

\textsuperscript{16} http://www.autosar.org
case D, so that it may be combined with that of all the other components, generating a Flattened XML. This is done manually by a third party, outside of case D’s control, causing a discontinuity in the integration flow.

Figure 29 displays the interview responses for experienced effects in case D. The general perception is clearly positive, $E_{13}$ (finding faults early) in particular receiving high scores. An interesting fact is that, as pointed out above, the automated integration is limited in scope, performing little more than generating and compiling the project code once a night. The positive responses, in spite of this, suggest to us that a significant amount of the project's efforts are expended on the transformation of models into compiled code. This is supported by interviewee comments, with developers claiming that the majority of their time is spent on overhead in model handling, rather than on actual development.

Furthermore, the entire integration flow is serialized: all activities act on the AUTOSAR Repository once a night. While this may be considered infrequent, the Baseline Definition activity only executes slightly more frequently, on average every 17 hours. This would indicate that the pace of development is low enough that the build frequency isn't a major concern.
Something that is not evident in Figure 29 is the very strong segregation between interviewee categories in their experienced effects. While developers were somewhat positive, and managers very much so, testers were nearly indifferent: the third quartile of all tester responses combined was zero. This adds further testimony that the integration flow focuses on code generation and compilation, rather than testing and verification.

Figure 29: Distribution of interview responses for each experienced effect (see Section 5.3.1.2) in case D, with the horizontal bars representing quartiles zero through four.

7.4.2.5 Case E

The case E development project produces a number of Jeppesen's products for airline fleet and crew management including optimization algorithms. The development involves some 140 people, and builds on a product that was initially created some 20 years ago. In case E one has striven for a very modularized architecture: products consist of "features", consisting of components. Apache Maven\(^\text{17}\) has been used to implement this setup, but with a home grown "Baseline" system for controlling dependency versions, instead of Maven's native dependency versioning. The modularization is extremely fine grained — there are hundreds of components, several times more than there are employees working on the project. As all these components come with their own

\(^\text{17}\) http://maven.apache.org
respective automated activities, the entire integration flow of case E is, in terms of number of activities, enormous. Consequently, three components, deemed representative by the project's configuration management team, constituting part of one "feature" of one product, were chosen to be included in the model shown in Figure 30.

Figure 30: A compressed model of the integration flow of case E. Out of hundreds, three representative components were chosen to be included in the model.

Changes made to the components’ repositories cause "snapshot" builds, i.e. transient component builds which do not constitute releases. Instead, when a new version of the product is to be built, its "baseline", identifying all its constituent parts' versions, is modified, which in turn triggers release builds of components and then features as required. Status web pages are generated for each build activity, and upon failure mails are also sent to "Build Masters", configuration managers and committing developers, who are expected to give high priority to correcting the fault. The scope and duration varies from activity to activity, and depending on whether it’s a "feature" or component level activity.

The interviewee responses for experienced effects in case E, shown in Figure 31, are mostly positive, with scores pertaining to developer productivity (E7 to E10) being somewhat mixed. This was also reflected in the interviewees' comments as they elaborated on their responses: all of the interviewed project members were highly aware of the automated integration flow, both its design and its purpose. They were also largely satisfied with it, and happy with its ability to support the project. The exception to this was that it was seen as adding overhead to the development process,
particularly as a consequence of the very large number of components. It was described how they have moved from a monolithic build and integration system to a modularized one. In retrospect, however, there was consensus that they had taken the modularization much too far. Therefore, they were now developing a third system, attempting to strike a balance between the two opposites.

7.4.3 Case Comparison

As anticipated (see Section 7.2), we find a high degree of disparity in the approaches to and implementations of automated integration flows. Not least, as is evident from the compressed models (see Figures 22, 24, 26, 28 and 30), the flow anatomies themselves are very different. We argue that this in itself is an important result, even though it makes it difficult to define any meaningful integration flow taxonomies based on the currently available data.

Figure 31: Distribution of interview responses for each experienced effect (see Section 5.3.1.2) in case E, with the horizontal bars representing quartiles zero through four.
7.5 Guidelines

This section presents six guidelines pertaining to the design of automated integration flows, based on our analysis of the multiple-case study.

7.5.1 Comprehensive Activities

Some of the studied cases — particularly case A (see Section 7.4.2.1) — have a large scope in their automated integration flow: they employ extensive arrays of tests and static and dynamic analyses. The information derived from this is, difficulties in effectively communicating it aside (see Section 7.5.2), highly valued. At the other end of the scale is case C, where the only automated verification is whether the code compiles or not (see Section 7.4.2.3). Case C interviewees with experiences from automated integration flows elsewhere considered this to reduce its value, or possibly even making it harmful in that it instills a false sense of achievement and security. Another symptom of a small scope is seen in case D, where testers see little to no value in the integration flow (see Section 7.4.2.4).

Our conclusion from this is that a large scope of test and analysis activities can to great effect be included in an automated integration flow, and that not doing so is at best under-utilization of its potential. Thus, we propose comprehensive activities as our first guideline: construct the automated activities such that their scope affords a sufficient level of confidence in the artifacts processed by them.

7.5.2 Effective Communication

We can see that the way in which the information generated by the integration flow is communicated differs between the studied cases, and that these differences impact the project members. Whereas some notify developers by mail when they have caused faults (see Sections 7.4.2.3 and 7.4.2.5), others leave it up to the individual to check the status themselves, which they may or may not actually do. In case B one of the developers was only barely aware that automated integration even existed (see Section 7.4.2.2) — a strong indication in a project of ten people that there is a communication deficit — preventing that person to derive any benefit from it.

Even if information is available it may not be understandable to everyone. Case A interviewees were frustrated by their inability to decipher the information generated by the integration flow (see Section 7.4.2.1), also preventing them from benefiting from it.

We conclude that even a very comprehensive integration flow can lose its value unless its results are received and understood by the project members. Thus, our second guideline is effective communication: ensure that the integration flow itself and its output are easily accessible and understandable by all its stakeholders.

7.5.3 Immediacy

Not only does the speed and frequency of the automated activities in our study differ — both between and within cases — but so does their accessibility. In particular, being allowed onto the SCM Mainline of case A can be a lengthy and demanding process (see Section 7.4.2.1), whereas in all the other cases the integration flow either picks up changes whenever the developer chooses to submit them, or includes them in daily builds.
During interviews, case A employees consistently returned to this complaint: the integration flow is "too far down-stream", any feedback from it comes much too late and the impression that it's not for their benefit, but for "someone else's", sentiments clearly reflected in their scoring of experienced effects (see Figure 23). It should be noted that at a glance this is not apparent. From the perspective of SCM Mainline (see Figure 22) there is a "continuous" in-flow of changes, but from the developers' point of view the integration is anything but. That being said, we believe that this situation in case A is strongly related to its dimensioning (see Section 7.5.4).

To conclude, we propose that project should strive for immediacy: make the integration flow easily and quickly accessible for the project members. We would also like to stress that lack of immediacy may be caused by a number of factors, such as the duration and frequency of the automated activities in the flow, or by process related factors as seen in case A.

7.5.4 Appropriate Dimensioning

Something that sets case A apart in our study is its size in combination with the many project member statements describing problems caused by the lack of immediacy (see Section 7.5.3). We can also conclude that at the point of SCM Mainline, the flow is completely serialized, with ensuing queuing and "traffic jams" (see Section 7.4.2.1).

In this sense, case E is at the other end of the scale (see Section 7.4.2.5). In their setup, with hundreds of largely independent components, there is a much higher degree of parallelism: failure in one component doesn't necessarily affect development in another, and almost certainly not in all at once. On the other hand, this comes at a price, as case E interviewees almost unanimously point out: managing all the components and their build activities becomes unwieldy and demanding, causing them to attempt to create a new system with just the right amount of parallelism.

This intuitively makes sense: maintaining an extensive highway network is only cost effective to the extent there is traffic to utilize it, and this issue is clearly related to size, or more precisely to "traffic" in the integration flow. Case B displays only a small degree of parallelism (which is also lessened by the lack of accuracy, see Section 7.5.5), and case C none at all. In both of these cases the in-flow of changes is much smaller than the duration of the automated activities allows for, and there is no congestion, but then cases B and C, with their 10 and 7 employees respectively, are dramatically smaller than cases A and E.

Consequently, our fourth guideline is appropriate dimensioning: adjust the capacity of the automated integration flow according to the traffic it must handle. Based on the limited number of cases in our study, it is impossible to say exactly when this becomes an issue. We would argue, however, that an expected frequency of input to an automated activity does carry clear implications for its execution frequency (and consequently duration) and/or the extent to which input must be batched, suggesting that pro-active dimensioning is feasible.

7.5.5 Accuracy

As previously discussed (see Section 7.2), in a software integration flow where an artifact (e.g. a product version) passes through a number of activities, designed to test and analyze it, our level of confidence in that artifact successively increases. This requires, however, that all of the activities actually act upon the same artifact, as we see examples of in cases A and E (see Sections 7.4.2.1 and 7.4.2.5). In case B, however, while some activities are triggered by the successful execution of their dependencies, they do not use its output: instead they check out code from the dependency's repository (see Section 7.4.2.2). The consequence of this is that the dependent isn't acting upon the
previously tested artifact, thereby building confidence in it, but on a source revision close in time to a revision that has been tested — possibly the same, but possibly not, making the concept of increasing levels of confidence inapplicable. In addition, this situation also risks introducing inconsistencies in compile-time, caused by e.g. differences in compiler settings and environment, which may affect the behavior of the resulting binary.

A similar issue arises in case D, where the actual compiled code is dependent on input created by a third party, which in turn is based on the output of case D's integration flow at some earlier point in time (see Section 7.4.2.4).

Based on these findings, we propose the guideline of **accuracy**: for all activities, know well what the input is, its history and consequently what level of confidence may be assigned to it.

### 7.5.6 Lucidity

One observation of the modeled cases is that even though in each case there is a flow of connected activities, the exact nature, as in which paths an artifact may take, isn't always lucid. Indeed, where this was discussed with the engineers during workshops, they stated that they weren't used to thinking about their integration as a flow. Recognizing the difficulties in communicating complicated integration activities to developers and other project members — e.g. stating that "only a handful of people actually know how this works" — they suggested that "thinking in flows" may be helpful. There is also reason to believe that adoption of the flow concept can help avoid discontinuities and inconsistencies, such as discussed in Section 7.5.5, by raising their visibility: by tracing these flows during the workshops the engineers were in several cases surprised by certain aspects of their own system and became aware of problems that had previously been obscure to them. Our final guideline is therefore **lucidity**: keep the flow of changes through the integration system clear and unambiguous. Not only does this help the status of the system to be more easily communicated, but inherent problems in its design may be more easily spotted.

### 7.6 Conclusions

In this section our conclusions from the study and the subsequent analysis are presented.

#### 7.6.1 Model Validation

In the early stages of applying the integration flow model to the five cases of our study we deemed it necessary to refine it, as described in Section 7.4.1. Following this we found no further need to revise the model. Therefore we consider the updated version, with additional node and relationship types (see Figure 21), to be, to the best of our knowledge, generally applicable to industry software development.

#### 7.6.2 Guidelines

Based on the multiple-case study we propose six guidelines for the design and implementation of automated software integration flows (see Section 7.5). The complete list of guidelines is shown in Table 19.
7.6.3 Future Work

While our case study has furthered our understanding of the nature and variants of industry software integration flows, many questions are still left unanswered. We still do not have sufficient data to connect particular integration flow variants to particular experienced effects. One obstacle to this is the great diversity among the cases: each one is distinct from all the others, making it impossible for us to categorize them and then compare the categories (see Section 7.4.3). Whether any such meaningful categories at all exist could be answered by further studies increasing the number of documented cases, and if so, comparative analysis may yield valuable results.

Finding unobtrusive ways of obtaining quantitative data from integration flows would also be a valuable contribution. While project members' estimation of experienced effects gives some insight, it is always subjective and limited by the interviewees' experiences and frames of reference. Consequently, we argue that complementing such information with e.g. measurements of integration flow throughput and product defect rates would constitute a valuable contribution to the field.

<table>
<thead>
<tr>
<th>Guideline</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comprehensive Activities</td>
<td>Construct the automated activities such that their scope affords a sufficient level of confidence in the artifacts processed by them.</td>
</tr>
<tr>
<td>Effective Communication</td>
<td>Ensure that the integration flow itself and its output are easily accessible and understandable by all its stakeholders.</td>
</tr>
<tr>
<td>Immediacy</td>
<td>Make the integration flow easily and quickly accessible for the project members.</td>
</tr>
<tr>
<td>Appropriate Dimensioning</td>
<td>Adjust the capacity of the automated integration flow according to the traffic it must handle.</td>
</tr>
<tr>
<td>Accuracy</td>
<td>For all activities, know well what the input is, its history and consequently what level of confidence may be assigned to it.</td>
</tr>
<tr>
<td>Lucidity</td>
<td>Keep the flow of changes through the integration system clear and unambiguous.</td>
</tr>
</tbody>
</table>

Table 19: Proposed guidelines.

Acknowledgments

We wish to extend our sincere gratitude to all of the participant companies and their employees for their time, patience and insights.