Large scale continuous integration and delivery

Stahl, Daniel

Chapter 6. Modeling Continuous Integration Practice Differences in Industry Software Development

This chapter is published as:
Ståhl, D., & Bosch, J. (2014). Modeling continuous integration practice differences in industry software development. *Journal of Systems and Software*, *87*, 48-59.

Abstract
Continuous integration is a software practice where developers integrate frequently, at least daily. While this is an ostensibly simple concept, it does leave ample room for interpretation: what is it the developers integrate with, what happens when they do, and what happens before they do? These are all open questions with regards to the details of how one implements the practice of continuous integration, and it is conceivable that not all such implementations in the industry are alike. In this paper we show through a literature review that there are differences in how the practice of continuous integration is interpreted and implemented from case to case. Based on these findings we propose a descriptive model for documenting and thereby better understanding implementations of the continuous integration practice and their differences. The application of the model to an industry software development project is then described in an illustrative case study.

6.1 Introduction

Continuous integration has, not least as one of the extreme programming practices [Beck 1999], become popular in software development. It is reported to improve release frequency and predictability [Goodman 2008], increase developer productivity [Miller 2008] and improve communication [Downs 2010], among other benefits. In previous work we found that the proposed benefits of continuous integration are disparate not only in literature: there are also great differences in the extent to which practitioners in industry software development projects have experienced those benefits [Ståhl 2013]. Consequently, we asked ourselves whether this disparity might be because of differences in the way the continuous integration practice itself had been implemented in different projects, be it because the concept had been interpreted differently or because the project context restricted the freedom of that implementation. Indeed, among the four projects included in the study there were indications that this may be the case, but as that study was not intended for this new research question it did not contain sufficient data to satisfactorily answer whether such differences manifest in software development at large.

Consequently, we decided to establish whether there are also differences in continuous integration descriptions found in the literature, and if so, in which regards the described implementations differ. In this paper we show the results of the systematic review conducted in order to answer this question, along with a proposed descriptive model for continuous integration practice variants based on its findings.

In this work we have focused on process related differences, rather than differences in tooling. While we recognize that tooling may improve or otherwise affect a continuous integration implementation, the practice of continuous integration itself requires no particular tools at all [Fowler 2006]. Consequently we regard tools to be of secondary importance, but not of primary

interest. Furthermore, we have not included contextual factors such as the size and longevity of the projects, the business environment or similar parameters. While they may conceivably correlate with variations in continuous integration practice – indeed, we consider the investigation of such correlations an important field of study in itself – they are not themselves aspects of continuous integration.

The contribution of this paper is twofold. First, it shows that there is not one homogeneous practice of continuous integration. Rather there are variation points – those evident in literature are presented and discussed individually in this article – with the term continuous integration acting as an umbrella for a number of variants. This is important, because when consequences of continuous integration are reported and discussed, it must be understood that such consequences potentially may not apply to the practice of continuous integration as a whole, but rather be related to a particular variant of it. Therefore, the second contribution of this article is that a descriptive model that addresses all the variation points uncovered in the study is proposed. Such a model enables better study and evaluation of continuous integration and can thereby bring a finer granularity to our understanding of the practice.

The remainder of this paper is structured as follows. In the next section the research method is described. Then the aspects of continuous integration described in literature, and the statements pertaining to those aspects, are presented and analyzed in Section 6.3. In Section 6.4 the proposed model is described, and then applied to a software development project in an illustrative case study in Section 6.5. The paper is then concluded in Section 6.6.

6.2 Research Method

The research was conducted by first reviewing existing articles on continuous integration to find differing descriptions of the practice, with the purpose of identifying aspects where there is contention in published literature. In other words, we searched for aspects (represented by clusters of statements, see Section 6.2.2) where different attributes or characteristics of the practice are evident, as such areas can then be considered to constitute potential variation points. To exemplify, some sources describe how checks and barriers are implemented to prevent non-correctional changes to be integrated on top of a broken build, whereas others relate how anyone is able to contribute anything at any time (see Section 6.3.2.8). As these are clearly differing views, this area is considered a variation point in the practice of continuous integration. In contrast, aspects where differing views are either not evident (see e.g. Section 6.3.1.4) or only addressed by a single source (see e.g. Section 6.3.1.5) are not regarded as potential variation points, the reasoning being that there appears to be consensus in the industry or that there is insufficient source material to reliably assess them. Based on this literature review a model for the description and documentation of continuous integration implementations was then constructed, intended as a guide to help ensure that the variation points discovered in the literature review are covered.

6.2.1 Systematic Review

As a result of observations of dramatically different experiences of continuous integration benefits [Ståhl 2013], and our assumption that this may be caused by differences between industry software development projects in how the concept of continuous integration is interpreted and implemented, we wanted to find an answer to the question of "Is there disparity or contention evident in the descriptions of various aspects of the software development practice of continuous integration found in literature?". To answer this question we conducted a systematic review

[Kitchenham 2004], where a review protocol was created and informally reviewed by colleagues. The protocol described the research question above, the sources to be searched (the IEEEXplore and Inspec databases), the exclusion and inclusion criteria of the review (see Table 17) and the method of extracting and clustering descriptive statements found in the publications (see Section 6.2.2). Following this the sources were searched (October 2012), with ACM subsequently being added for completeness, for publications relating to the software practice of continuous integration.

The search terms yielded 64, 79 and 45 results in IEEE, Inspec and ACM respectively. Combined, these result sets contained 112 unique items. Exclusion criteria EC1, EC2 and EC3 (see Table 17) were applied to this set, and the abstracts of the remainder were studied to determine whether they dealt with the software practice of continuous integration, or pertained to other fields of research (exclusion criterion EC4). This left a set of 76 publications.

Finally, these 76 publications were reviewed in full in search of descriptions of continuous integration practices (exclusion criterion EC5). Such descriptions were found in 46 of the reviewed articles.

Inclusion criteria	
IC1	Papers, technical reports, theses, industry white papers and presentations with the terms "continuous integration" and "software" in their titles or abstracts.
Exclusion criteria	
EC1	Where studies were published multiple times (e.g. first as a conference paper and then as a journal article) only the most recent publication was included.
EC2	Material not available to us in English or Swedish.
EC3	Posters for industry talks lacking content beyond abstract and/or references.
EC4	Material that does not address the software practice of continuous integration, or only mentions it in passing.
EC5	Material that does not describe one or more aspects of how continuous integration practices are, can or should be implemented.

Table 17: Inclusion and exclusion criteria of the literature review.

6.2.2 Analysis of Literature

Statements as to the nature of continuous integration found in the 46 publications of the literature review were extracted and clustered in groups addressing similar aspects, where one statement may be included in more than one cluster. This yielded 180 discrete, descriptive statements pertaining to one or more aspect of continuous integration and 22 clusters (see Table 18). Following this, any group not containing any disparity in their statements were culled. In other words, only groups of statements describing aspects of continuous integration where contention was evident were preserved. This could either manifest as multiple statements in disagreement, or as statements themselves identifying disparity. Additionally, clusters containing statements from only one unique source were culled.

Cluster		Statements	Unique sources	Contention	Claimed disparity
C1	Build duration	10	9	Yes	Yes
C2	Build frequency	10	8	Yes	No
C3	Build triggering	32	29	Yes	Yes
C4	Build version selection	2	2	No	No
C5	Component dependency versioning	6	3	No	No
C6	Definition of failure and success	8	6	Yes	No
C7	Fault duration	5	5	Yes	Yes
C8	Fault frequency	1	1	No	Yes
C9	Fault handling	9	9	Yes	No
C10	Fault responsibility	6	5	No	No
C11	Integration frequency	7	7		Yes
C12	Integration on broken builds	6	6	Yes	No
C13	Integration serialization and batching	6	6	Yes	Yes
C14	Integration target	8	6	Yes	Yes
C15	Lifecycle phasing	1	1	No	Yes
C16	Modularization	17	11	Yes	Yes
C17	Pre-integration procedure	16	12	Yes	Yes
C18	Process management	1	1	No	No
C19	Scope	50	40	Yes	No
C20	Status communication	19	16	Yes	Yes
C21	Test separation	11	9	Yes	Yes
C22	Testing of new functionality	10	9	Yes	Yes

Table 18: Clusters of descriptive statements extracted from literature, shown alongside the number of constituent statements, the number of unique sources from which those statements were extracted, whether there exists contention between the sources and whether there are within the cluster single sources claiming disparity of implementations, respectively.

It shall be noted that determining what in this context constitutes an aspect of continuous integration practice – and thereby a cluster – is ultimately a call of judgment. Particularly, automation is not included, even though it is frequently brought up by papers discussing continuous integration, e.g. stating that "test cases [...] will be folded into the automated regression test suite" [Sturdevant 2007], that "an automated integration server not only makes integration easy, it also guarantees that an integration build will happen" [Rogers 2004], "the build process has to be fully automated" [Dösinger 2012] or that "the build process is initiated automatically" [Pesola 2011] to mention a few. For the purposes of this study, we have taken the position that the practice of continuous integration is by definition automated, as described by Martin Fowler: "Each integration is verified by an automated build" [Fowler 2006]. Indeed, from the literature included in this study, we have not found reason to reconsider this position. One source goes so far as to consider it a

criterion for success that "all [continuous integration] steps pass without error or human intervention" [Rasmusson 2004], and so questions of e.g. whether test cases are included in automated test suites rather becomes a matter of the scope of continuous integration, which is covered by its own statement cluster (see Section 6.3.2.13).

6.2.3 Proposing a Model

Based on the analysis of the literature review, a model for documenting continuous integration was created. The purpose of this model was to cover all the statement clusters displaying contention or disparity, thereby answering all the relevant questions that may set one particular instance of continuous integration apart from another, yet at the same time being practical to use.

The benefit to researchers from using such a model is that it may help focus attention on the aspects of continuous integration that act as differentiators, and it provides a method for managing the multitude of continuous integration variants in existence. The benefit to software development professionals – to practitioners of continuous integration – is that it lists choices that they can make – and possibly have already made, consciously or unconsciously – with regards to implementing continuous integration. Such information can be an important factor in successfully improving one's development process.

6.3 Statement Clusters

This section describes each of the clusters of statements found in the literature (see Section 6.2). In Section 6.3.1, those clusters that were culled from the set are presented. Then, in Section 6.3.2, the preserved clusters are discussed.

6.3.1 Culled Clusters

Clusters either not containing more than one unique source, or not found to display disparity or contention were culled from the set. These are described and discussed below.

6.3.1.1 Build Version Selection

It is pointed out in [v. d. Storm 2008] that "the continuous integration system should always attempt to build the latest version of the software", while [v. d. Storm 2007] states that "if the latest build of a component has failed [...] an earlier successful build is used instead". While these perspectives seem to differ, it shall be understood that they deal with different contexts: one discusses source code revision, while the other concerns itself with handling component failures in a modularized environment. Therefore there is no contention between them – indeed, they are entirely compatible with each other – and this cluster was therefore culled.

6.3.1.2 Component Dependency Versioning

One topic found in several of the papers is that of modularization of the product, and whether to rebuild the entire product upon integration of new changes, or only those components affected by the change (see Section 6.3.2.11). In the latter case, some articles discuss version handling of such components. It is stated that each component shall be built individually, with new versions made

available for every such build [Roberts 2004]. It is also said that component dependencies shall be on the latest available version and that only one version of any one component may occur in a system configuration [v. d. Storm 2008]. Furthermore, if a component build fails, then the latest working version shall be used for dependencies in its stead [v. d. Storm 2007].

There is only a small number of articles addressing this aspect – which is not altogether surprising, as it is only relevant in a component oriented continuous integration setup – and no direct contention between them.

6.3.1.4 Fault Frequency

The issue of fault frequency in continuous integration is discussed by [Brooks 2008], where two development teams have been compared. Among other findings, it is pointed out that one team suffered much more frequent build failures than the other. It is readily conceivable that, while being a complex factor dependent on multiple parameters, the frequency of errors in continuous integration can have an impact on the development effort. Lacking additional sources, however, this cluster was culled from the set.

6.3.1.4 Fault Responsibility

Multiple sources describe how developers causing faults in continuous integration are also responsible for correcting those faults [Miller 2008, Yuksel 2009], e.g. stating that it is "the responsibility of [the last person who checked in code] to ensure that a reported bug is resolved immediately" [Ablett 2007]. Additionally, the practice of developers not leaving work until their changes have been successfully verified is mentioned [Miller 2008, Rogers 2004]. No statements contradicting these stated practices were found in the study, with the exception of [Janus 2012], describing how a "Quality Manager" interprets the produced code analysis metrics. This, however, is only for non-critical violations of coding standards, and so we find that this cluster does not display any contention or disparity.

6.3.1.5 Lifecycle Phasing

It is stated in [Holck 2003] that continuous integration may be performed "during a phase of integration and tests; or it may be part of iterative methods". While other sources in the study do not explicitly discuss this – causing this cluster to be culled – it shall be noted that, in our understanding, it is implicit in many of the studied publications that continuous integration takes place during, if not necessarily limited to, development.

6.3.1.6 Process Management

In [Holck 2003] it is noted that the degree of control over continuous integration processes differs, giving examples of projects using "less structured processes" as opposed to "central management of the build process". This aspect was, however, not highlighted by other sources in study.

6.3.2 Preserved clusters

Statement clusters containing more than one unique source and either displaying contention between sources, or containing sources themselves acknowledging diversity in implementations were preserved. These clusters are presented and discussed below.

6.3.2.1 Build Duration

Some publications in the study give more or less exact figures for the duration of "builds" (where the scope of what a build entails varies, see Section 6.3.2.13); time from check-in to notification of verdict can be "several minutes" [Downs 2010] or "a few minutes" [Woskowski 2012], or the time required to compile and run tests can be over an hour [Yuksel 2009]. Some articles highlight that build duration does vary from project to project [Brooks 2008] and that this can be of some importance, as a too long duration means that "continuous integration starts to break down" [Rogers 2004] and the build time must be quick enough to "allow the CI server to keep up with the changes and return feedback to the software engineers while their memory of the changes is still fresh" [Dösinger 2012]. Others discuss separating quick test suites from slower ones to provide incremental feedback [Roberts 2004], removing slow tests altogether from the regular continuous integration builds and instead running them on a separate schedule [Holmes 2006], or how parallelism caused by modularization can affect build durations [v. d. Storm 2008], as it is remarked that "the primary factor influencing the build time is the increasing number of tests" [Rogers 2004].

From this it is clear that not only is build duration a variation point for continuous integration and considered to be of great importance; it is also not an independent variable. Instead, it is highly dependent on what is included in the build. This would pose a problem if one were to attempt classification and comparison of continuous integration implementations, as any measurement of build duration would also have to take into account what is actually achieved in that duration. The abstract concept as such, represented by this statement cluster, however, is preserved for the purposes of this study.

6.3.2.2 Build Frequency

It shall be noted that for the purposes of this study, we make a distinction between build frequency and integration frequency (see Section 6.3.2.7). By the former we mean the frequency at which continuous integration "builds" (regardless of the scope of those builds) are performed, while the integration frequency refers to how often changes are brought into the product development mainline (typically in the form of source code changes). We consider these to be two crucially different activities which may or may not be synchronized.

The build frequencies described in literature vary. Some mention "multiple builds per day" [Stolberg 2009] or "every few hours" [Rogers 2003], in contrast to the daily builds practiced by others [Holck 2007]. Yet the frequency does not just vary between projects, it may also vary within the same project. Separation of slow activities into more infrequent cycles is described [Holmes 2006, Downs 2010, Woskowski 2012, Long 2008] as well as performing "weekly integration" builds while modules are tested in isolation "several times a day" [Tingling 2007].

6.3.2.3 Build Triggering

The vast majority of statements on how continuous integration builds are triggered describe how source code changes cause a build to start [Liu 2009; Bowyer 2006, Holmes 2006, Ablett 2007, Rogers 2003, Rogers 2004, Sturdevant 2007, Stolberg 2009, Kim 2009a, Kim 2009b, Pesola 2011, Cohan 2008, Janus 2012, v. d. Storm 2007, Woskowski 2012, Dösinger 2012, v. d. Storm 2005, Matsumoto 2012, Gatrell 2009, Gestwicki 2012]. This is not always the case, however: builds can be executed at a certain time each day [Holck 2007], or a mixed approach where some activities run on a fixed schedule while others are triggered by source code changes can be used [Tingling 2007, Hoffman 2009, Downs 2010, Woskowski 2012].

Another setup is where multiple activities are chained together in a sequence. A source code change triggers the first activity [Yuksel 2009], while subsequent activities are triggered by "successful execution and evaluation of prior events" [Hill 2008]. There is reason to believe that this may be more common than the number of explicit statements suggests, as it is sometimes hinted at or implied, e.g. stating that when "a step in the process fails, further process steps are skipped" [Dösinger 2012]. Though such a statement is not unambiguous enough to be counted to the statement cluster total, it does imply that there are process steps which are triggered by the success of upstream steps.

In a modularized scenario, where each component has its own continuous integration, a build can also be triggered by a new version of a component dependency being made available [Roberts 2004, v. d. Storm 2007].

6.3.2.4 Definition of Failure and Success

What is considered a failure in a continuous integration build is touched upon by several sources. Commonly, if any test fails during the build, then the build as a whole is failed [Ablett 2007, Rasmusson 2004], with some sources explicitly mentioning that compilation must also succeed [Downs 2010, Janus 2012, Yuksel 2009] (although it may be argued that compilation errors are implicitly not allowed, even where the source does not explicitly state it).

This zero tolerance toward test failures is not ubiquitous, however: it is put forward that for most teams "it is fine to permit acceptance tests to break over the course of the iteration as long as the team ensures that the tests are all passing prior to the end of the iteration" [Rogers 2004]. Yet others impose additional requirements before they will consider a build to be successful, such as a certain level of test coverage [Yuksel 2009] or the absence of severe static code analysis warnings [Janus 2012].

6.3.2.5 Fault Duration

Fault duration – that is, how long a detected fault persists before it is successfully addressed – is not extensively discussed in the studied sources, but there are statements indicating differences. One example is a comparatively strict approach where "if any compilation errors or any test failure occurs, the relevant developer should solve the problem in less than 30 min or revert the check-in" [Yuksel 2009], whereas another notes that "the great majority of build breaks were fixed within an hour" [Miller 2008], without explicitly stating any similarly precise rules. Much longer fault durations are also described: "there were several occasions when [...] the code was broken for up to two weeks" [Tingling 2007]. Indeed, [Brooks 2008] recognizes that build failure length differs: while some would be "stuck for hours because of a broken build", others have "very few of these problems".

Yet the question of fault duration is dependent on that of the definition of failure and success, as demonstrated by one source claiming that typically not all types of test failures need to be fixed immediately, but can be left until "the end of the iteration" [Rogers 2004]. Consequently, the question of fault duration only really becomes meaningful in a context where "fault" is well defined. That being said, there are clear differences in fault duration as it is described by the sources in our study, and consequently the cluster is preserved.

6.3.2.6 Fault handling

How faults, once detected, are handled (e.g. given what priority by whom) is touched upon by a number of sources. Several describe a policy of fault fixing being given top priority, either as the personal responsibility of the developer committing the fault [Yuksel 2009] (a policy which would arguably presume that the offending commit can always be identified), as the responsibility of the developers "that have committed Source Code since the last successful integration" [Janus 2012], or as the responsibility of the last developer to check in code, who is then responsible "to ensure that [the fault] is resolved immediately, for example by reverting to an older version or by fixing the problem in another way" [Ablett 2007]. Another source describes how "after an initial investigation, one developer would fix the build while the rest of the team continued with their work" [Miller 2008], hinting at a more flexible delegation of work.

Some sources display a slightly more relaxed attitude with regards to faults, however. It is stated that there's a difference between types of tests, some of which may not be a priority to fix [Rogers 2004]. One source describes a development team displaying a certain extent of ambivalence: while fixing faults is not a top priority, broken builds cause a number of problems unless fixed quickly – e.g. promoting a "laissez-faire attitude" and hiding other problems – and it is suggested that perhaps "a team policy could be instituted to make the broken build the highest priority, ahead of any other work items" [Downs 2010]. In a multi-step integration setup it is furthermore stated that if one step fails, then "further steps are skipped" [Dösinger 2012].

In other words, while sources take a similar position on the question of responsibility per se (see Section 6.3.1.4), there are different views on the methods used and priority given to addressing these faults.

As something of a special case in this context, if one uses a modularized continuous integration implementation with separate builds per component, it is mentioned that if a component build fails, then the latest successful build of that component is used by downstream dependencies [v. d. Storm 2007].

In conclusion, there are clear differences in how one reacts to and handles a continuous integration fault once it has been detected, even though this, as is indeed the case with other aspects, is not an independent variable. As the definition of a fault (see Section 6.3.2.4) or whether one uses modularized continuous integration (see Section 6.3.2.11) varies, so the very meaning of fault handling is not constant.

6.3.2.7 Integration Frequency

Integration frequency, as opposed to build frequency (see Section 6.3.2.2), is described by a number of sources. It is claimed that "on average developers check in once a day" [Miller 2008], and that while the integration frequency "will vary from project to project, from developer to developer, and from modification to modification [...] a good rule of thumb [is that] developers should integrate their changes once every few hours and at least once per day" [Rogers 2004]. Other sources conclude that this implies that there will be multiple integrations per day [Watanabe 2012],

relating how "on average, a version was submitted to the source control system every hour over the period studied" [Gatrell 2009], although presumably this would depend on the number of developers co-existing in the same source context.

While not mentioning any figures, other sources simply claim that integration should be "frequent and timely" [Baumeister 2011] or "[performed] regularly and early" [Dösinger 2012]. Meanwhile, [v. d. Storm 2008] states that even though current usage of the term continuous integration often does not consider the frequency at which check-ins are made, "continuous integration proper [...] includes the continuous checking in of changes, however small they may be".

To conclude, we do not consider contention to be evident in this cluster – not least because several of the sources are very vague – but it still fulfills the preservation criteria, as disparity is claimed.

6.3.2.8 Integration on Broken Builds

There are different approaches as to whether commits on top of revisions that failed in continuous integration are acceptable or not. Several sources describe how commits are not allowed unless the latest continuous integration build was successful [v. d. Storm 2008], saying that "once the build is broken other developers cannot check in their work" [Miller 2008], that "all merge requests [are refused] unless they contain a special tag [identifying them as fixes]" [Lacoste 2009] and that "developers not working directly on fixing the problem are not permitted to commit their changes [because] it could greatly complicate the problems for the people engaged in fixing the build" [Rogers, 2004].

Others are less concerned. In "decentralized" continuous integration, developers are allowed to "add contributions to the development version at any time" [Holck 2007]. In another case, check-ins on broken builds were not prevented, even though it was suggested that this sometimes caused problems, since "new code could conceivably be problematic too, but the confounding factors would make it difficult to determine exactly where the problem was" [Downs 2010].

To summarize, the consensus appears to be that code commits on broken builds can be problematic, but whether enough so to actively try to prevent it (by process or by automated blocking of unwanted check-ins) is contended.

6.3.2.9 Integration Serialization and Batching

While it is common to let committed changes trigger new builds (see Section 6.3.2.3), it's an open question how one handles a situation where multiple changes are made during the time span of a single build. This can be particularly relevant in a context of slow build times and high integration frequency. This is discussed by [Rasmusson 2004], pointing out that there are different approaches to serialization and batching: the check-in process can be serialized in order to minimize failures and "avoiding all integration conflicts", as opposed to "the more normal free flowing practice whereby any developer can optimistically check-in as soon as they have run the build locally and all tests pass". One source states that polling for changes implies "batching the revisions to be tested" [Lacoste 2009], while another claims that "every commit should build the mainline on an integration machine" [Stolberg 2009], with references to tooling used for achieving this.

It shall also be noted that in a situation where activities (particularly tests, see Section 6.3.2.15) are separated into stages it is possible that those stages are not executed at the same frequency [Tingling 2007, Hoffman 2009, Downs 2010], in effect batching changes in between different integration activities.

6.3.2.10 Integration Target

The integration target aspect concerns where developers check in their changes. Most of the publications in the study do not explicitly deal with this, but rather state that, for example, a change in the source code repository triggers a build (see Section 6.3.2.3) without specifying where in the repository those changes are made. There are, however, some that go into details.

One method is that of every commit resulting in a new build on the product "mainline" [Stolberg 2009] and letting developers check in to the development version at any time [Holck 2007]. It is described how "all the development teams [were moved] into one common code branch – no private branches" and "each code check-in is now immediately integrated" [Goodman 2008].

Multiple branches are also used, however. One variant is the pattern of a single development branch into which new changes are merged, and a "stable" branch into which "all the revisions vetted by [the continuous integration] are pushed", the latest version of which is used for deployment [Lacoste 2009]. Another variant is to let each team in a multi-team project integrate internally, using their own integration server, before integrating with the project at large (as a response to problems encountered when attempting to scale continuous integration) [Rogers, 2004]. However, the same source continues, this "creates the problem of cross-team integration" where "teams are potentially building up an integration debt". Indeed, another source relates how, when implementing continuous integration, the initial decision was made "that each individual Scrum team should have a dedicated and private server", but "as integration issues were being discovered very late" they put "all teams onto a single server environment again" [Sutherland 2011].

6.3.2.11 Modularization

While there are sources making explicit statements that their continuous integration is not modularized [Rasmusson 2004], e.g. claiming that "the entire software is built [and] tested" upon changes [Ablett 2007] or that "testing in the CI process focuses only on 'local' projects" [Dösinger 2012], it is our understanding that sources where this topic is not discussed generally presume a non-modularized approach. For the most part, the sources that do deal with modularization are positive examples, in that they either explain how continuous integration can be modularized or describe examples of such modularization.

In such sources, it is related how products can be composed of "hundreds of components with complicated dependency relationship[s]" and "the source code of each [component] is controlled independently" [Kim 2008]. Expanding on this concept, another source describes how components rely on pre-built artifacts of their dependencies, and "integrating the whole application then means building the topmost component in the dependency hierarchy" [v. d. Storm 2008]. Furthermore it is stated that each component has its own continuous integration cycle, following which it is published to be tested in combination with other components [Roberts 2004], that "continuous integration can be seen as a [directed acyclic graph], where nodes correspond to package builds and edges correspond to dependencies among packages" [Beaumont 2012] and that components are rebuilt if they themselves are changed, or one of their dependencies change [v. d. Storm 2007]. Similar concepts are also discussed by [Rogers 2004].

Additionally, a modularized approach to continuous integration is claimed to impact feedback times. It is described how "modules were tested in isolation and embedded into the program" [Tingling 2007] and that such a practice enables faster feedback, because "instead of building the complete system on every change", only the components that have affecting changes are rebuilt [v. d. Storm 2008]. In a similar vein, it is claimed that while a single source repository is often assumed

in continuous integration, this in fact scales poorly, thus motivating a modularized approach [Roberts 2004]. To facilitate testing of components in such an environment, "surrogates" can be used "to simulate the behaviors of unavailable components" [Liu 2009], and it is described how tests can be executed in several steps: first by component, then for the system at large [Kim 2008].

It is noteworthy how, unlike most continuous integration aspects in this study, statements pertaining to modularization are mostly found in sources explicitly focused on that very problem. In contrast, few other sources mention it at all. This leads us to the conclusion that in many cases, non-modularized continuous integration is the default alternative, possibly even adopted without being consciously chosen.

6.3.2.12 Pre-Integration Procedure

The pre-integration procedure of continuous integration refers to which actions are prescribed prior to performing an integration, e.g. by checking in source code. In some cases, these procedures can be practically non-existent, with one source arguing that the benefit of continuous integration can be measured as the time saved by developers not compiling and testing before checking in [Miller 2008]. Others offer the option without prescribing any mandatory process, with developers running small subsets of tests rather than waiting for the centrally executed test suite [Holmes 2006]. It is also related how developers "typically [...] run tests before checking in changes" [Brooks 2008] and that "developers could ensure their check-in [...] both by manually compiling the code [...] and by executing the set of unit tests, [but] few did so" [Downs 2010]. These examples appear to share the sentiment that "fundamentally, it needs to be acceptable to break the build" [Rogers 2004].

Other sources provide examples or mandatory pre-integration procedures, where developers are obliged "to integrate their own contributions properly" [Holck 2007]. This can take various forms, such as reviews before checking in [Downs 2010, Janus 2012], running light-weight "developer builds" [Rasmusson 2004], performing "a pre-check [...] before committing" [Woskowski 2012] or "[ensuring that] all corresponding unit tests are successful" [Alyahya 2012]. One source stresses the importance of testing before integration, fearing that the alternative "would be a nightmare" [Lacoste 2009], while [Yuksel 2009] describes a checklist of mandatory activities, where before committing any code, the developers must compile the whole system, design and code the needed unit and integration tests, and finally execute the entire unit and integration test suites.

Clearly, there are stark contrasts in what procedures are required before integrating. One source remarks on this, stating that it is "a common approach [to] institute a strict and thorough pre-commit procedure that will make it as hard as possible for developers to break the build", but that such procedures also have negative side effects [Rogers 2004]. The automation of pre-integration procedures is also discussed, since the developers "may forget [or] may not follow the practices" [Alyahya 2012].

6.3.2.13 Scope

By scoping of continuous integration we refer to the amount and type of activities included in the practice, either as part of a single "build", or separated into several stages (see Section 6.3.2.15). Typically the compilation (where applicable) of source code followed by testing is included [Alyahya 2012, Miller 2008, Liuet 2009, Bowyer 2006, Kim 2008, Holmes 2006, v. d. Storm 2008, Ablett 2007, Hill 2008, Holck 2007, Rogers 2003, Rogers 2004, Sturdevant 2007, Stolberg 2009, Roberts 2004, Tingling 2007, Sunindyo 2010, Kim 2009a, Kim 2009b, Pesola 2011, Goodman 2008, Rasmusson 2004, Beaumont 2012, Cannizzo 2008, Hoffman 2009, Moreira 2010, Downs

2010, Cohan 2008, Huang 2008, Brooks 2008, Janus 2012, v. d. Storm 2007, Yuksel 2009, Matsumoto 2012, Lier 2012] – even in sources that do not explicitly state this, it is to our understanding often implicitly the case. The tests are frequently combined with static and/or dynamic code analysis [Miller 2008, Holmes 2006, Hill 2008, Kim 2009a, Cannizzo 2008, Moreira 2010, Woskowski 2012], even though this can be considered to be "Continuous Measurement" and therefore not part of the scope of continuous integration itself [Janus 2012].

The types of tests executed varies: some only run automated unit test suites [Miller 2008, Bowyer 2006, Hill 2008, Moreira 2010], others also run integration tests [Alyahya 2012, Liu 2009, Holmes 2006, Downs 2010, Huang 2008, Woskowski 2012, Baumeister 2011, Watanabe 2012], functional and/or non-functional system tests [Holmes 2006, Sturdevant 2007, Cannizzo 2008, Baumeister 2011] and/or acceptance tests [Holmes 2006, Stolberg 2009, Roberts 2004, Cannizzo 2008, Watanabe 2012]. Continuous integration can also involve creating installation packages [Miller 2008, Pesola 2011, Cohan 2008, Gestwicki 2012], so that a release "boils down to selecting the desired build" [v. d. Storm 2007], and deploying the project [Ablett 2007, Rogers 2003, Stolberg 2009, Sunindyoet 2010, Pesola 2011, Goodman 2008, Moreiraet 2010, Cohan 2008, Dösinger 2012, Lier 2012, Gestwicki 2012]. It shall be noted, however, that the term "deployment" in this context is loosely defined and may refer to different activities.

In conclusion, we consider merely compiling and unit testing to be the basic continuous integration "build" activities. Then, other activities can be added on top of this, including but not limited to various types of more advanced testing, code analysis, packaging and deployment. Indeed, it's essentially possible "to chuck everything into [the build], including the kitchen sink" [Rogers 2004].

6.3.2.14 Status Communication

There are various approaches to communicating the continuous integration status, e.g. sending notifications of build failures [Dösinger 2012, Matsumoto 2012], in development projects. Dispatching e-mails is common [Sturdevant 2007, Stolberg 2009, Kim 2009a, Kim 2009b, Hoffman 2009, Downs 2010, Gestwicki 2012], either notifying the last person to check in [Ablett 2007], "relevant developers" [Yuksel 2009] or "the whole development [project]" [Lacoste 2009]. Other communication methods can be used, such as RSS [Stolberg 2009] web pages [v. d. Storm 2005] or dashboards [Baumeister 2011]. This may then be displayed on "information radiators" [Ablett 2007, Goodman 2008, Hoffman 2009, Downs 2010], making the current status visible to all in the vicinity. Other methods include differently colored lava lamps and robotic dogs walking up to the responsible developers, "[displaying] to the team that it is not happy with that developer, in a friendly, funny and playful way" [Ablett 2007].

One source extensively discusses and evaluates differences in notification methods, and concludes that a combination of multiple communication channels can have a great impact on awareness of and responsiveness to broken builds [Downs 2012].

6.3.2.15 Test Separation

Test separation refers to the practice of segmenting test suites into multiple parallel or sequential activities. Similar to the case of modularization (see Section 6.3.2.11), sources that touch upon this issue tend to be positive examples, and it is difficult to find explicit statements to the effect that testing is not separated, although to our understanding this is the case in a number of the articles in the study. That being said, one source argues that even though it's common to have "a single integration process that compiles the code, runs the unit tests and the acceptance tests, builds

deployment packages for QA and the customer, validates code coverage and checks coding standards amongst other things" [Rogers 2004], this is not necessarily a good thing, as they increase the build duration (see Section 6.3.2.1) and thereby delay feedback. Consequently, tests are sometimes separated into multiple activities.

One separation approach is to "segment tests by functional area and to only run those tests thought to be affected by the code change" [Brooks 2008] or to split test suites by components [Kim 2008]. Commonly tests are separated into sequential stages based on the time it takes to execute them and the context in which they run [Sturdevant 2007, Sunindyo 2010, Brooks 2008, Yuksel 2009], e.g. "one an 'express build' that just runs unit tests to give a basic idea of the success of an integration; another a longer 'full' build that actually runs database processes, acceptance tests, deployments, etc." [Roberts 2004], or slower tests are performed on a different schedule altogether [Tingling 2007, Woskowski 2012]. Another source states in passing that continuous integration "is the automation of sequential build process steps" [Dösinger 2012], which could be interpreted as implying that automated steps are linked together in a chain of sequential stages, but is ultimately too ambiguous to be included in the statement cluster.

6.3.2.16 Testing of New Functionality

Some sources in the study describe testing in continuous integration as primarily safe-guarding legacy functionality [Downs 2010] by "[testing] against a suite of automated regression tests" [Ablett 2007]. Continuous integration can, however, be used for validating new functionality as well, by creating the automated test cases before the production code is implemented [Liu 2009, Holmes 2006, Yuksel 2009], or in parallel with the implementation [Sturdevant 2007, Stolberg 2009, Tingling 2007, Goodman 2008]. Some sources discuss the use of test-driven development, e.g. stating that "writing failing unit tests prior to writing any production code, then writing only enough production code to make the test pass" is required practice [Gestwicki 2012], yet do not explicitly describe the practice in relation to continuous integration and could therefore not be included in the statement cluster.

6.4 A Descriptive Model

It is apparent that continuous integration implementations vary in a multitude of ways. Consequently, we conclude that to derive more value from studies and discussions on continuous integration and its effects, more comprehensive information about the actual particular implementation or implementations at hand is required. In this section we propose a model, or guide, for how to better document the practice, that is designed to address every one of the variation points discovered in the systematic review (see Section 6.3.2). The model consists of two parts: the Integration Flow Anatomy – depicting activity and input nodes and their relationships (see Section 6.4.1) – and the node attributes applying to those nodes (see Section 6.4.2). Both of these parts, together forming the complete descriptive model, are detailed in turn below, as well as how they were designed and which variation points they cover. Following this there is a discussion on how to use a subset of the attributes and possible constraints (see Section 6.4.3).

As an alternative to defining a new model, existing ways of representing variability were also considered, with particular attention paid to the COVAMOF framework [Sinnema 2004]. We consider the problem of representing activities, their scope, relationships and characteristics in a software integration process, however, to be a much simpler one than that of modeling e.g.

variability, dependencies and interactions of software components. In addition, not all concepts (e.g. dependencies and realizations) necessarily translate well across the problem domains. Consequently, we have opted for the model described in this section.

6.4.1 Integration Flow Anatomy

A number of statements found in the systematic review touch upon how a "build" may consist of large numbers of interconnected steps, performing various tasks, which conditionally trigger each other. These steps may be executed in parallel or in sequence, or run on different schedules altogether. They may concern themselves with the entire product, or with separate components. As one of the articles in the study explains, this can be thought of as a directed acyclic graph (DAG) [Beaumont 2012].

We find that by using such a DAG to depict the steps, or activities, of an integration process, several questions can be answered. Therefore, a meta-model was constructed with the aim of being able to accurately reflect all the variants possible from the variation points discovered in the systematic review. This meta-model is shown in Figure 18. It consists of two types of nodes: Input (e.g. source code) and Activity (e.g. execution of test cases). Activities may be triggered by either input or activity nodes, with the conditions under which the trigger is activated (e.g. the source activity succeeded or failed) documented. Furthermore, both activity and input nodes contain a set of attributes describing their scope and characteristics.

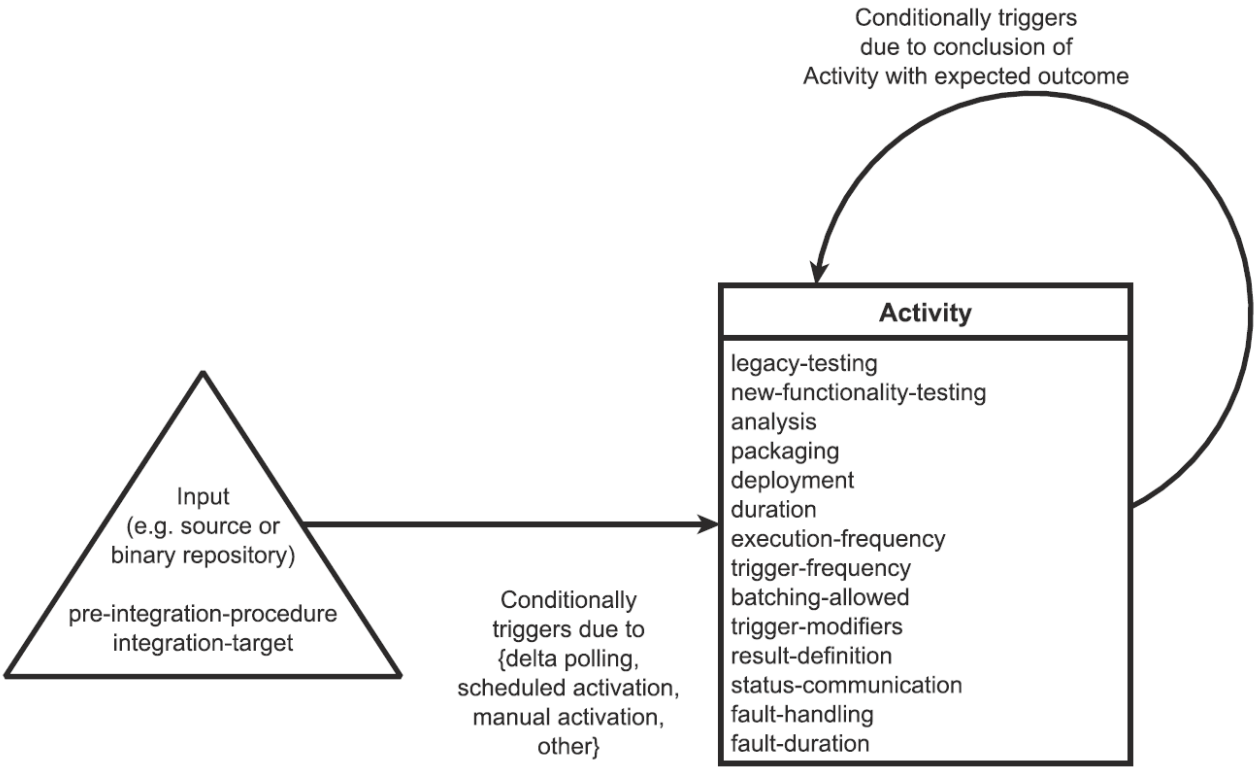

Figure 18: The meta-model of the descriptive model. Note that the attribute set of either node type is flexible: the depicted model contains all attributes required to address the uncovered variation points (see Section 4.4.2), but both sub-sets and super-sets are conceivable.

The nodes themselves and their triggering relationships can be used to answer questions pertaining to the variation points of modularization and build triggering (see Sections 6.3.2.11 and 6.3.2.3, respectively). Section 6.4.2 describes how the remainder of the variation points are covered by adding attributes to the nodes.

6.4.2 Node attributes

This section presents a set of attributes, grouped into themes, for the activity and input nodes of the Integration Flow Anatomy (see Section 6.4.1). These attributes are derived from the variation points uncovered in the literature review (see Section 6.3.2). As each group of attributes is presented below the reasoning behind it and the variation points addressed by each attribute is explained.

It shall be noted that, when applying the model, the actual attribute set used may vary – with the information conveyed by the descriptive model varying accordingly – depending on the scope and purpose of the application of the model. A study focusing, for example, solely on the communication aspects of a particular integration flow may choose to exclude attributes deemed irrelevant to that purpose. This is further discussed in Section 6.4.3.

6.4.2.1 Scope Attributes

The scope theme of attributes applies to the activity nodes and addresses the scope (see Section 6.3.2.13), test separation (see Section 6.3.2.15) and testing of new functionality (see Section 6.3.2.16). The following attributes are designed to fully cover these variation points:

- **legacy-testing:** a list of testing activities applied to legacy code. Different nomenclature is used by different sources – testing activities mentioned by articles in the study include e.g. unit, acceptance, system, integration, performance and function tests.
- **new-functionality-testing:** a list of testing activities applied to functionality that is not yet fully implemented or considered legacy. Definitions of what constitutes legacy may vary.
- **analysis:** a list of analysis activities carried out, e.g. static code analysis or test coverage measurements.
- **packaging:** a boolean signifying whether the product is packaged and made ready for deployment.
- **deployment:** a list of environments (e.g. a lab environment or live customer systems) to which the product is deployed as part of this activity.

The **legacy-testing** and **new-functionality-testing** attributes are derived from both the test separation and testing of new functionality variation points. Since test activities may be split across multiple different steps, it's important to document in the DAG which nodes contain which types of testing. Also, since it's evident that projects treat testing of new functionality differently, the test activities need to be documented in two separate attributes for legacy and new functionality, respectively.

The **analysis**, **packaging** and **deployment** attributes all address the scope variation point. Apart from testing, these are the three areas where the systematic review shows that the scope differs, and so these attributes are included in order to clearly show which, if any, of the tasks of analysis, packaging and deployment are performed in any given activity node.

6.4.2.2 Build Characteristics Attributes

The build characteristics theme contains questions pertaining to build duration (see Section 6.3.2.1), build frequency (see Section 6.3.2.2), integration frequency (see Section 6.3.2.7), integration on broken builds (see Section 6.3.2.8) and integration serialization and batching (see Section 6.3.2.9). To answer these questions, we propose that the following attributes shall be applied to the activity nodes:

- **duration:** the average duration of the activity.
- **execution-frequency:** the execution frequency of the activity.
- **trigger-frequency:** the triggering frequency of the activity.
- **batching-allowed:** a boolean signifying whether integrations may be batched into single builds.
- **trigger-modifiers:** a list of descriptions of possible modifiers to the activity's triggering behavior.

The **duration** attribute reflects the time required to execute an activity and addresses the build duration variation point. Similarly to the scope of the entire continuous integration being equal to the union of its constituent parts, its duration is then equal to the total duration of the activity nodes on its critical path.

Furthermore, though seemingly similar, **execution-frequency** and **trigger-frequency** are treated as separate attributes, corresponding to the separate variation points of build frequency and integration frequency. The former documents how often an activity is executed, whereas the latter how often it is triggered. Depending on the type of trigger this metric obviously has different meanings: in a situation where the trigger is a source code change it shows the frequency at which new content is integrated, whereas if it's a new version of a component being published it shows the frequency at which that component is being made available for integration with the larger system. Regardless it's informative – in particular, it's relevant to the **batching-allowed** attribute (corresponding to the variation point of integration serialization and batching): where the integration frequency is higher than the build frequency, does one batch those integrations into a single build or not?

Finally, the **trigger-modifiers** attribute is derived from the variation point of integration on broken builds. Here any impact of the activity's state or context on the trigger, e.g. failures blocking new incoming changes unless they are flagged as fixes, should be documented.

6.4.2.3 Result Handling Attributes

Like the build characteristics theme of attributes (see Section 6.4.2.2), result handling attributes apply to each individual activity node (see Section 6.4.1). This theme covers the definition of failure and success (see Section 6.3.2.4), fault handling (see Section 6.3.2.6), fault duration (see Section 6.3.2.5) and status communication (see Section 6.3.2.14). In order to address these variation points, we propose the following attributes:

- **result-definition:** a list of possible results and their definitions.
- **status-communication:** a description of when, how and to whom the activity's status is communicated.
- **fault-handling:** a description of how discovered faults are addressed: by whom, and given what priority.

- **fault-duration:** the average duration of unbroken faulty status of the activity.

The **result-definition** attribute describes what is considered e.g. a faulty or successful execution of the activity. As possible outcomes may vary, a description shall be given per outcome. The **status-communication**, **fault-handling** and **fault-duration** attributes all address their corresponding variation points.

6.4.2.4 Input node attributes

This section describes the proposed attributes that apply to the input nodes of the model. The relevant variation points in this context are pre-integration procedure (see Section 6.3.2.12) and integration target (see Section 6.3.2.10). From these the following two attributes are derived:

- **pre-integration-procedure:** a description of the procedure required before integrating changes.
- **integration-target:** a description of where the integration takes place (e.g. which branch and the rules governing it).

The **pre-integration-procedure** attribute describes what, if anything, the developer must do in order to integrate, and thereby create the change-set that serves as input to the activities of the integration flow. The **integration-target** attribute, on the other hand, describes whether the context of that integration is e.g. a team branch or a "mainline" branch.

6.4.3 Attribute Selection and Constraints

We recognize that it is not always desirable, practical or even possible to assemble all the data required by the full list of attributes proposed above. This is the reason why the meta-model does not prescribe any mandatory attributes. Obviously, the more complete the model, the more information and potential value it brings, but none of the proposed attributes explicitly requires any of the other in order to be valid or meaningful. A hypothetical descriptive model containing only a sub-set of attributes is shown in Figure 19 to serve as a simple illustrative example.

On a further note, we have not identified any definite constraints in the sense of invalid or impossible attribute combinations creating invalid areas in the attribute space. However, this does not rule out that such areas, or areas that in practice are unpopulated, exist. In future work, we intend to investigate this further by means of gathering empirical data through multiple-case studies (see Section 6.6.4.3).

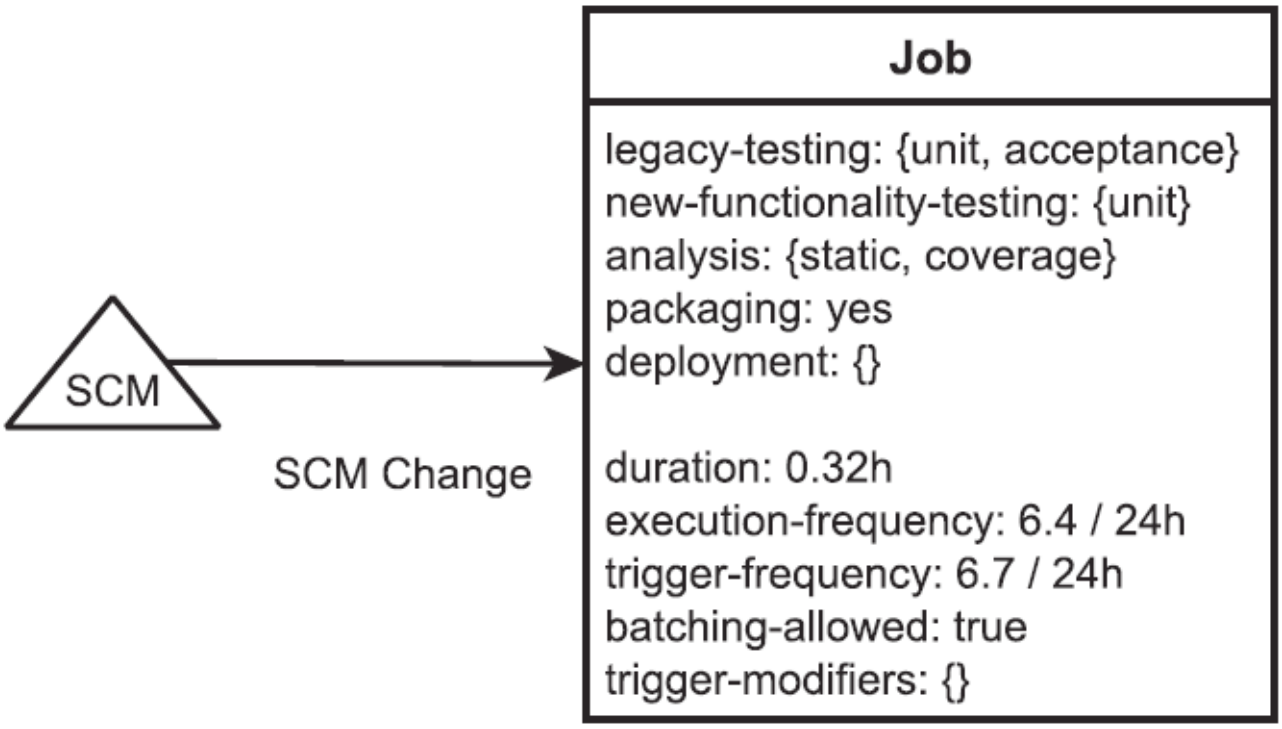

Figure 19: A description model, containing all proposed scope and build characteristics attributes, of a hypothetical simple integration flow.

6.5 Illustrative Case Study

This section describes how the model proposed in Section 6.4 was applied to a development project, in this article referred to as Project A, within Ericsson AB. The assembled model and its data is not presented in its entirety in this article – instead, the purpose is to illustrate the steps involved in assembling the model and to present an example of how those steps may play out and the insights such an exercise may provide.

It shall be understood that the descriptive model (see Section 6.4) is not based on this case study, but on a systematic review (see Section 6.2.1). Neither is it intended as a complete validation of the model, beyond demonstrating that it can be applied with positive results. Furthermore, conducting the case study did not give cause for revising the model.

6.5.1. Project A

In Project A, multiple development teams are responsible for developing one of the components of a network node, with non-trivial integration dependencies to the other components of the node. The project was chosen as a case study candidate when one of the authors was invited to assist in improving its continuous integration implementation. In that situation, the model was used in order to establish the baseline implementation, and as a basis for identifying and planning improvement

activities. Based on the multiple continuous integration case studies conducted in previous work [Ståhl 2013] and our experience of industry software development, we deemed the project representative of industry practice and therefore suitable to serve as an illustrative example of application of the model.

6.5.2. Sketching the Integration Flow Anatomy

The first step of building the model was to sketch the Integration Flow Anatomy. This was done in front of a whiteboard, in collaboration with engineers working in the project, by analyzing the actual activities configured in Jenkins, their continuous integration tool. By studying how the Jenkins activities, or Projects, related to each other and their contents we were quickly able to create a graph of the component's entire flow (see Figure 20), including its internal delivery to another part of the organization, and also determine the scope attributes of the activity nodes. During this work it soon became evident that the emerging anatomy had not been entirely clear to the project members themselves beforehand, which shows that this in itself can be a rewarding exercise from an educational point of view. It was also discovered that the delivery from D1 (see Figure 20) to the receiving organization was done manually and that not much data on this was available. Consequently this was identified by the engineers as a prioritized area of the project's integration to improve.

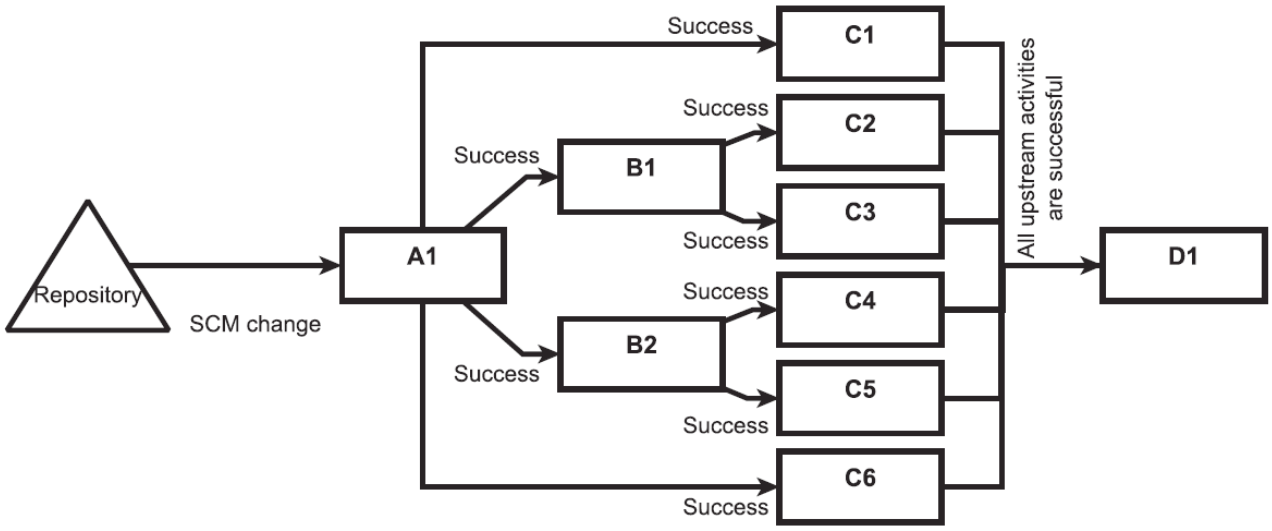

Figure 20: The Integration Flow Anatomy of Project A. Activity D1 was followed by a manual delivery to another part of the organization, which was outside the scope of the case study. The figure only shows the anatomy itself, with all node attributes deliberately left out.

6.5.3 Determining the Input Node Attributes

Following the Integration Flow Anatomy, the attributes of the input node was discussed. This took the form of an unstructured interview between the authors and the project's engineers, with the authors asking for descriptions of the current **pre-integration-procedure** and **integration-target** (see Section 6.4.2.4) in Project A. With regards to the **integration-target** attribute, it was suggested by the engineers themselves that the current implementation, where a "develop" branch was used both as the integration branch for all the developers and as the release branch from which deliveries

were picked, might prove to be unsuitable in the future. It was agreed, however, that in the project's current early stage of development this solution was adequate, but that it may have to be revised in the future.

6.5.4 Determining the Activity Node Attributes

All the build characteristics attributes (see Section 6.4.2.2) for the activity nodes could be gathered by the authors themselves by analyzing the configuration of the project's continuous integration tool, Jenkins. One insight gained while collecting the **trigger-modifiers** information was that the first activity, A1, was in Jenkins configured as being part of the same "Project" (the entity used to configure an activity in Jenkins) as D1, meaning that A1 would not be ready to start work on new changes until all other activities had finished. The consequence of this was that the entire flow became serial, in the sense that it was unable to work on more than a single change-set at any given time. It was concluded that this was a problem that should be addressed, not least because it caused under-utilization of available hardware.

Furthermore, this example illustrates how activities in the Integration Flow Anatomy may not necessarily map directly to e.g. Jenkins Projects – rather, the activities in the model should correspond to how the project members themselves would conceptually describe their integration flow. It may be argued, then, that if that description does not easily translate into how the activity entities of one's continuous integration tool are configured, then this should be seen as an indication that there may be a problem with said configuration, as indeed turned out to be the case in the studied project.

Finally, in a second session, the result handling attributes (see Section 6.4.2.3) of each activity in the integration flow was discussed with the project's engineers. In this project, the result handling of all the activities in the flow were similar – for instance, the status of all activities considered relevant were visualized in real time using a special information radiator functionality in the continuous integration tool, thereby made available to all stakeholders. Also, while failure in any activity would cause mails to be sent out to all developers and project support staff, it was a dedicated team's responsibility to act on those failures, and then escalate to developers if deemed necessary. Though it was said that this may not be the best solution, and that it would be better if the developers themselves had the mindset to take that responsibility directly, the result handling in the project was considered satisfactory and no urgent improvement needs were identified.

6.6 Conclusion

This section presents the conclusions of the conducted literature review, the model proposal, the illustrative case study and remaining questions left unanswered.

6.6.1 Disagreements in Related Work

It is clear from the conducted literature review that there is currently no consensus on continuous integration as a single, homogeneous practice. Out of the 22 statement clusters synthesized from statements extracted from the included articles, differences and/or disagreements were evident in all but six (see Section 6.3). Not only does this mean that, in order to make a meaningful comparison of software development projects, simply stating that they use continuous integration is insufficient information as we instead need to ask ourselves what kind of continuous integration is used. It also

means that, taking into account the dramatic differences in experienced continuous integration effects [Ståhl 2013], we need to ask which aspects or variants of continuous integration any proposed benefit (or disadvantage) is an effect of. For this purpose, based on the findings in our study, we have proposed a descriptive model for better documentation of continuous integration variants.

6.6.2 Model Proposal

In this paper we have proposed a descriptive model of continuous integration implementations, based on variation points evident in literature. We believe that using this model will enable us to better understand implementations of continuous integration and how they compare to each other. This in turn is a prerequisite for studying correlations between differences in those implementations and differences in their experienced effects. If we are to reach a level of understanding where a development project is able to pro-actively and confidently design its continuous integration according to the benefits it wishes to maximize, then this is an important step on that path. That being said, it is also clear that applying the model to one's own integration flow, with or without comparing it to others, can be a valuable and educational experience, as it may provide insights into one's own development process.

6.6.3 Model Validation

We have demonstrated through an illustrative case study the applicability of the model proposed in this article to an industry development project, and that tangible benefits can be derived from it. Not only did the model bring to light the actual anatomy and characteristics of the integration flow in the project – something that had thitherto been an opaque part of the environment for a number of the project members – but it was also able to indicate areas where opportunities for improvement could be found and served as a basis for the planning of activities to pursue those improvements.

That being said, however, while the research presented in this article demonstrates that disparity in multiple areas exists, the sample size is insufficient to fully understand their distribution and consequently the actual space of variations. Also, we do not assume that we have identified every possible variation point in this research. There may still be important differentiators not yet included in the model, particularly as continuous integration as exercised by practitioners may well evolve in the future. Therefore, additional data provided by further case studies would help in improving our understanding of not only the value ranges and statistical distribution of the model's attributes, but could also uncover attributes that are as of yet missing. Based on the case studies conducted in previous work [Ståhl 2013] and our professional experience we expect such case studies to allow refinement of the proposed model, but not lead to major disruption.

In conclusion, we would consider future case studies applying the model to a larger number of industry software development projects to be an important contribution, both in that they would serve to further validate and refine the model and at the same time provide additional data points in the study of continuous integration itself – in particular, we find that comparative case studies of multiple implementations are lacking in contemporary literature.

6.6.4 Open Questions for Further Research

A number of questions still remain unanswered in the field of continuous integration.

6.6.4.1 Correlations Between Differences in Practice and Differences in Experience

In previous work [Ståhl 2013] we have identified disagreement among software development professionals as to the benefits of continuous integration experienced in their projects. In the research presented in this article we have further demonstrated that continuous integration implementations themselves may differ. Consequently, what we ask ourselves is whether there is a correlation between these differences in experience on the one hand, and differences in implementation on the other. Would it be possible to improve our understanding in such a way that we cannot only present a model for describing variants of the practice, but also demonstrate that these variants allow for different effects? If so, could that be used to allow industry professionals to decide which flavor of continuous integration they should strive for, based on the benefits they prioritize?

We are still far from such an understanding of the practice. In order to get there we need a sufficient body of data detailing both experienced effects and the variation points of continuous integration in a number of projects utilizing continuous integration. We therefore propose that case studies be performed in this area.

6.6.4.2 Contextual Differences

Software development projects obviously differ in more ways than in how they have chosen or been able to implement continuous integration. They may be of varying size, longevity, budget, organizational structure, competence setup, geographic distribution etc. While the number of such conceivable variation points may be nearly infinite, it is nevertheless possible that some of them interact with the variation points of continuous integration. It may be that certain contextual factors enable or are enabled by particular variants of continuous integration, or that they influence the very interpretation of the continuous integration concept.

We believe that case studies investigating such relationships in the industry would be a valuable contribution in this area.

6.6.4.3 Internal Constraints and Correlations of the Model

We have proposed a model containing a number of attributes, grouped into themes, covering the variation points where we have shown that continuous integration implementations can and do differ. What is not understood is how, if at all, these variation points correlate. It is conceivable that some variants enable or disable each other, or certain variants tend to manifest together, allowing us to cluster them an identify "typical species" of continuous integration. Conversely, it is possible that constraints exist such that certain areas of the combinatorial space created by these attributes are invalid, or unpopulated in practice.

It could also be that some of the variation points identified in this research are so tightly coupled that they can be better understood as different manifestations of the same underlying practice, thereby allowing variation points to be merged and the model simplified.

Our understanding of these questions would be furthered by access to larger sets of empirical data. To this end, we propose multiple-case studies to be conducted, where the proposed model is applied and any constraints and correlations are searched for.

Acknowledgments

We would like to thank Ericsson AB for allowing us to study one of their software development projects and publishing our findings. We also want to thank our colleagues for all their feedback and the patience they have demonstrated during our research.

Part II:
Continuous Integration and Delivery Modeling and Architecture