Empir Software Eng (2017) 22:967–995
DOI 10.1007/s10664-016-9457-1

Achieving traceability in large scale continuous integration and delivery deployment, usage and validation of the eiffel framework

Daniel Ståhl[1] · Kristofer Hallén[1] · Jan Bosch[2]

Published online: 24 October 2016
© Springer Science+Business Media New York 2016

Abstract The importance of traceability in software development has long been recognized, not only for reasons of legality and certification, but also to enable the development itself. At the same time, organizations are known to struggle to live up to traceability requirements, and there is an identified lack of studies on traceability practices in the industry, not least in the area of tooling and infrastructure. This paper presents, investigates and discusses Eiffel, an industry developed solution designed to provide real time traceability in continuous integration and delivery. The traceability needs of industry professionals are also investigated through interviews, providing context to that solution. It is then validated through further interviews, a comparison with previous traceability methods and a review of literature. It is found to address the identified traceability needs and found in some cases to reduce traceability data acquisition times from days to minutes, while at the same time alternatives offering comparable functionality are lacking. In this work, traceability is shown not only to be an important concern to engineers, but also regarded as a prerequisite to successful large scale continuous integration and delivery. At the same time, promising developments in technical infrastructure are documented and clear differences in traceability mindset between separate industry projects is revealed.

Communicated by: Patrick Mäder, Rocco Oliveto and Andrian Marcus

✉ Daniel Ståhl
daniel.stahl@ericsson.com

Kristofer Hallén
kristofer.hallen@ericsson.com

Jan Bosch
Jan@JanBosch.com

[1] Ericsson AB, Datalinjen 3, Linköping, Sweden

[2] Chalmers University of Technology, Lindholmspiren 5, Göteborg, Sweden

Keywords Continuous integration · Continuous delivery · Traceability · Very-large-scale software systems

1 Introduction

Traceability in software engineering is widely recognized as an important concern, with substantial research and engineering efforts invested in solving its challenges (Gotel and Finkelstein 1994; Ramesh and Jarke 2001; Wieringa 1995). Traceability can serve multiple purposes, such as internal follow-up, evaluation and improvement of development efforts (Dömges and Pohl 1998), but it is also crucial in order to demonstrate risk mitigation and requirements compliance — either to meet customer expectations on transparency or to adhere to specific regulations, as is particularly the case for safety-critical systems (BEL-V BfS CSN ISTec ONR SSM STUK 2013; QuEST Forum 2015; European Cooperation for Space Standardization 2015; Radio Technical Commission for Aeronautics 2011).

As defined by the Center of Excellence of Software and Systems Traceability (CoEST) (Center of Excellence of Software Traceability 2015), traceability is "the ability to interrelate any uniquely identifiable software engineering artifact to any other, maintain required links over time, and use the resulting network to answer questions of both the software product and its development process", with this network consisting of *trace links* — "a specified association between a pair of artifacts, one comprising the source artifact and one comprising the target artifact [with] a primary trace link direction for tracing" — which may also carry semantic information.

Despite its recognized importance — indeed, something as fundamental as determining which changes have been made and which requirements have been verified in which configurations, in order to make a qualified release decision and then generating satisfactory documentation for that release, requires some form of traceability — organizations often struggle to achieve sufficient traceability (Mäder et al. 2009), not least due to lacking support from methods and tools (Bouillon et al. 2013). Indeed, related work indicates the importance of traceability tooling that is "fully integrated with the software development tool chain" (Rempel et al. 2013) and calls for further research into technical infrastructure (Cleland-Huang et al. 2014). Furthermore, there is a gap between research and practice in the traceability area, with considerable valuable work on development and comparison of e.g. traceability recovery techniques (Antoniol et al. 2002; Oliveto et al. 2010) but few studies on traceability practices in the industry (Mäder et al. 2009).

A crucial part of the traceability problem is one of generating accurate and consistent trace links (Center of Excellence of Software Traceability 2015) which can then be used to answer questions such as which product releases a code change has been included in, which bug fixes or features were implemented in a given version, or which tests have been executed for a given product version, which requirements they map to and in which configurations they have been executed. As difficult as it demonstrably is to achieve this, there are several factors which complicate it further.

Scale The larger the project, the more engineers and engineering artifacts — and thereby trace links — are involved. Large scale development can involve many thousands of artifacts (e.g. source code, bug reports, requirements, backlog items, test cases), created and modified over time by hundreds or thousands of engineers, adding further difficulty to manual approaches (Asuncion et al. 2010). Returning to the example of release decisions and

documentation, with a handful of people involved it may be feasible to manually gather the required information, but with thousands of engineers contributing to a single release this turns into a both lengthy and costly practice. This becomes particularly problematic in contexts — also related to the overall scale of the project — where the system is split into multiple tiers of completeness. To exemplify, components being integrated into sub-systems, which are integrated into complete solutions, with internal delivery interfaces across which traceability must be maintained.

Geographical Diversity Particularly in large multinational companies, the engineers generating the artifacts may be spread across multiple sites located in different time zones and on different continents, with all of the implied communication difficulties, such as culture and language barriers.

Technology and Process Diversity Also related to scale (particularly systems split into many components and tiers), when numerous different tools are used in development, generating heterogeneous engineering artifacts, this poses additional traceability challenges, both in automated and manual approaches. Both human and automated agents must have access to and be able to parse all the used formats and standards involved in order to generate consistent and accurate trace links — a problem particularly pronounced where integrated components are delivered by external companies with completely separate processes and technology stacks.

Continuous Integration and Delivery Agile methodologies in general may pose additional challenges by encouraging reduced overhead, particularly documentation (Nerur et al. 2005). Continuous integration and delivery — studied by us in previous work (Ståhl and Bosch 2013; 2014a; 2014b; Ståhl et al. 2015) — in particular can be problematic, however. For the purposes of this work we define continuous integration and delivery as the practice of all developers committing their changes frequently, and that each change is considered a release candidate to be validated by the build and test process (Humble and Farley 2010). The implication is that the traceability requirements under which one operates must also be — at least potentially — satisfied for every single change made to the product during its life cycle. This scenario is dramatically different from the paradigm of relatively infrequent releases planned well in advance. Adding to this, the higher frequency of integrations, test executions and deliveries implies a dramatically increased amount of traceability data to be analyzed — in a similarly dramatically reduced time frame. As (Wang et al. 2015) points out, while providing explicit requirements traceability for every release is rare, “in continuous releases [it] is much rarer”.

Furthermore, the time dimension is an important factor in traceability. Not only is “continuous and incremental certification” crucial in avoiding the “big-freeze” problem of safety-critical software (The Open-DO Initiative 2015), but there are additional benefits to a prospective approach to traceability. For instance, the concept of *in situ* real time generation of trace links — as opposed to the *ex post facto* approach generally taken by automated tools (Asuncion et al. 2010) — affords the ability to chronologically structure and analyze trace links and to provide engineers with immediate feedback (Asuncion and Taylor 2009). This, combined with the scaling and speed challenges outlined above, imply that an optimal solution to software traceability is to be sought in tool and framework support for automated *in situ* generation of trace links and subsequent analysis of those trace links — as opposed to a solution solely based on process or development methodology.

In other words, while many fundamental traceability needs are the same regardless of continuous integration and delivery practice, the reduced time frames, increased frequencies and enlarged amounts of traceability data pose severe challenges *in practice*, calling for novel automated solutions to aid industry practitioners. That being said, as will be discussed in Section 4, these practices also prompt additional needs.

Consequently the research question driving the work reported from in this paper is: *How can traceability needs of large scale software development in industry be effectively addressed in a continuous integration and delivery context?*

The contribution of this paper is twofold. First, an industry developed framework addressing the problem of achieving traceability in the paradigm of continuous integration and delivery is presented and discussed. Second, it validates the studied framework by investigating traceability needs in the industry and the extent to which they are addressed by it.

The remainder of this paper is structured as follows. In the next section the research method is explained. In Section 3 the Eiffel framework is presented. The results of an investigation into traceability needs in the industry are presented in Section 4, followed by validation in Section 5. Threats to validity are discussed in Section 6 and the paper is then concluded in Section 7.

2 Research Method

The work reported from in this paper consists of three parts: an investigation into the industry developed continuous integration and delivery framework Eiffel, identification of prominent traceability needs in the industry and validation of the framework.

2.1 Framework Investigation

Eiffel (not related to the programming language (Meyer 1988)) — a framework for supporting continuous integration and delivery developed by Ericsson in response to a lack of adequate available solutions to the problems outlined in Section 1 — was investigated through perusal of documentation, informal discussions with continuous integration architects and *in situ* demonstrations, as well as drawing on our findings as participant observers (Robson 2011). The results of these findings are presented in Section 3.

2.2 Needs Identification

To identify the traceability needs most prominent among engineers in large scale industry projects, interviews with 15 software development professionals spread across three cases in two companies (cases A, B and C, see Section 4.1) were conducted.

2.3 Validation

The Eiffel framework was validated using the following methods to achieve data triangulation (Runeson and Höst 2009):

Eiffel Experience Interviews with software development professionals in a very-large-scale project having deployed Eiffel (case A, see Section 4.1), to capture their experiences and reflections from using the framework.

Feedback Interviews with software development professionals in two very-large-scale projects not using Eiffel (cases B and C, see Section 4.1), to investigate whether similar and/or alternative solutions could be found.

Comparison of traceability data gathering process with and without Eiffel, by repeating an analysis carried out in case A several years earlier, before the framework was developed.

Systematic Literature Review seeking to answer "Which solutions to the traceability problem in a continuous integration and/or delivery context have been proposed in literature?" by searching for solutions comparable to the Eiffel framework.

Observer Participation as continuous integration subject matter expert and product owner, respectively: two of the researchers have been involved in the development of the framework and its deployment in large parts of Ericsson.

All interviews conducted in this work were carried out in a similar fashion. Interviewees were purposively sampled to achieve the best possible coverage of stakeholder roles and perspectives — and thereby triangulation (Runeson and Höst 2009). They were semi-structured and conducted individually, with a small set of prepared behavior/experience and knowledge questions (Hove and Anda 2005) as guide, and the interviewees were explicitly encouraged to provide spontaneous reflections and/or elaborations. During each interview two researchers were present — achieving observer triangulation (Runeson and Höst 2009) — who then compared notes and, where the interviews were conducted in Swedish, translated the transcript to English. Where necessary, specific terms, e.g. *trace links*, with which the interviewees were unfamiliar, were explained.

3 Eiffel

In response to the needs outlined above and the lack of adequate available solutions, Ericsson has developed the Eiffel framework. This framework has been deployed in case A (see Section 4.1), among many other projects and units within the company, in order to address traceability in a continuous integration and delivery context. Some space has been devoted to this section, as we believe that a description of the framework and some discussion on its usage and deployment is not only an important contribution of the paper, but also required for understanding and interpretation of the collected data.

3.1 Overview

Ericsson, with many years of experience from developing very-large-scale software systems and an ambitious continuous delivery agenda, has long recognized the problems outlined in Section 1. Following careful analysis of its enterprise level needs for both cross-organizational traceability and collaboration in continuous integration and delivery, a

framework named Eiffel was developed, in which two of the authors of this paper participated as architects and developers. This framework has since been successfully deployed and used in production in large parts of the company for several years leading up to the study reported from in this paper. It has been published as open source and is currently available at GitHub[1]. While in-depth descriptions of the framework, the messaging protocol, trace links and document types are available online, this section presents a brief summary.

Eiffel is frequently presented as four things: a concept, a protocol, a toolbox, and an inner source (Fitzgerald 2006) community; the concept, briefly put, is that continuous integration and delivery activities in real time communicate through and are triggered by globally broadcast atomic *events*, which reference other events as well as design artifacts via semantic trace links, and are persistently stored, thereby forming traversable event graphs. In this sense, the Eiffel strategy is crucially different from the Information Retrieval approach of establishing trace links between code and free text documents, as described by e.g. (Antoniol et al. 2002), in that it only operates on structured data and does not attempt probabilistic analysis. The querying, analysis and visualization of these graphs can then be used to answer traceability questions. The inner source community supports and evolves the protocol and the toolbox, which realize the concept.

In developing the Eiffel framework, Ericsson wanted to not just address requirements traceability, but traceability across various types of "artifacts that are of varying levels of formality, abstraction and representation and that are generated by different tools", in the words of (Asuncion and Taylor 2009); Ericsson's work also takes a similar approach, but focuses on the provisioning of real time traceability throughout the automated processes of compilation, integration, test, analysis, packaging etc., rather than in "development time". In its current state the implementation of the Eiffel framework within Ericsson has primarily focused on what is termed *content traceability*: which artifacts constitute the *content* of a given version of a component or product, with artifacts being e.g. implemented requirements, completed backlog items or source code revisions. Through this focus it is able to address the concerns outlined in Section 1. At the same time, Eiffel is designed to act as a thin abstraction layer, providing a communication interface between products and organizations that maintains traceability yet hides divergence in processes and tools such as software control systems, artifact repositories and continuous integration servers.

In addition, in its work on the Eiffel framework, Ericsson has placed great emphasis on the bi-directionality of trace links. In order to satisfy multiple needs with the same traceability solution, the generated event graphs must be traversable both forward and backward through time, or "downstream" and "upstream" in the words of Ericsson's engineers, borrowing terminology sometimes used in dependency management (Humble and Farley 2010).

3.2 The Messaging Protocol

Whenever an event occurs during software build, integration and test — e.g. an activity has commenced, an activity has finished, a new component version has been published, a test has been executed or a baseline has been created — an atomic message detailing that event is sent on a global, highly available RabbitMQ message bus.

[1] https://github.com/Ericsson/eiffel

The payload of the message is a JSON document representing the *event*, containing information such as time stamp, sender identity and semantic links. These links may be either other events (e.g. an event reporting the publishing of a new artifact referencing the message reporting the creation of the baseline on which it was built, which in return references events reporting the creation of the source revisions included in the baseline) or external documents (e.g. backlog items, bug reports, requirements, test cases and source code revisions). These events messages are typically generated as the result of automated activities, but may also result from manual actions (e.g. a tester reporting the execution and outcome of a test case).

Once sent, the event messages are not only used for documentation of trace links, but to drive the continuous integration system itself. To exemplify, a listener receiving an event saying that a new artifact has been published may trigger a test activity as a consequence, which in turn dispatches more events which are received by yet another listener, triggering further actions. For more in-depth information on e.g. trace link types, please see the online Eiffel documentation[2].

An important principle in the Eiffel framework is that these event messages are not sent point-to-point, but broadcast globally, and are immutable. This means that the sender does not know who might be interested in a certain message, does not need to perform any extra work on behalf of her consumers, and is geographically independent of them — all important features in light of the challenges discussed in Section 1. Any consumers simply set up their listeners and react to received messages as they see fit. Furthermore, the immutability means that the historical record never changes.

The end product of this framework is a system of real time generated trace links which both document and drive the production of software, allowing every commit to be traced to every resulting release — and every release to be traced to every included commit — across all intermediate steps of compilation, integration and test, as well as referencing e.g. requirements and planning artifacts, as exemplified in Fig. 1.

3.3 The Toolbox

Apart from the messaging protocol itself, there is a number of software components that enable the Eiffel framework. Several of these are plugins to various tools, such as Jenkins, Nexus, Artifactory, Gerrit et cetera, for sending and/or receiving Eiffel messages.

Other components are built from the ground up in order to persistently store, query, aggregate, analyze and visualize the messages — a task that requires specialized solutions, as the steadily growing number of messages generated on a global scale is on the order of billions. The analysis of this data can serve many purposes for different stakeholders, such as tracking individual commits, tracking requirements, providing statistics on project performance, monitoring bottlenecks, analyzing release contents etc. To provide the reader with an example of day-to-day use, a screen shot from such a visualization tool — Follow Your Commit — can be seen in Fig. 2. As one interviewee explains, “The individual developer often begins with Follow My Commit [...] There I actually get the information of how what I did has been moved forward in the chain.”

From informal discussions with the engineers working on development and deployment of the Eiffel framework we find a consensus that substantial potential remains untapped in this area, and that emerging technologies for big data aggregation and

[2] https://github.com/Ericsson/eiffel

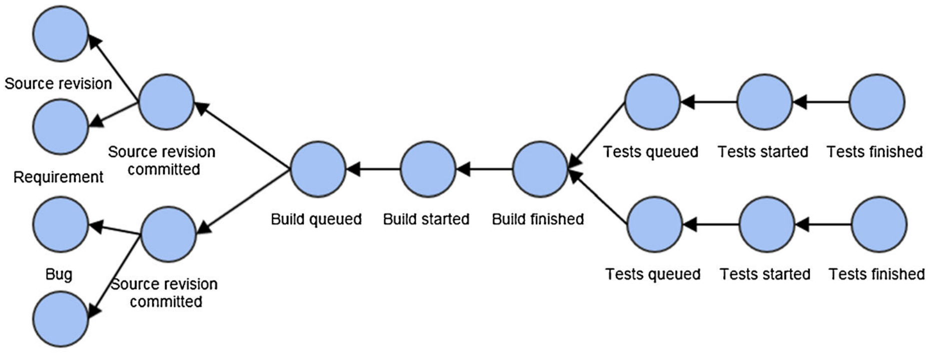

Fig. 1 A simplified example of an Eiffel event graph. Even based on such a simple graph, a number of questions relating to the identified challenges (see Section 1) can be addressed in real time. These include code churn, commit frequency, implemented requirements and bugs per build, lead time from commit to various stages in the pipeline, where a feature or a commit is in the pipeline at any given time, build and test delays due to queuing and resource scarcity, and more. Furthermore, all these metrics are collected and analyzed independently of organization, geographical location, time of event and development tools used

analysis hold great promise for even more advanced data driven engineering based on Eiffel.

3.4 Deployment

How to successfully deploy a framework such as Eiffel in a large existing development organization is a pertinent question in the context of this research, particularly in light of

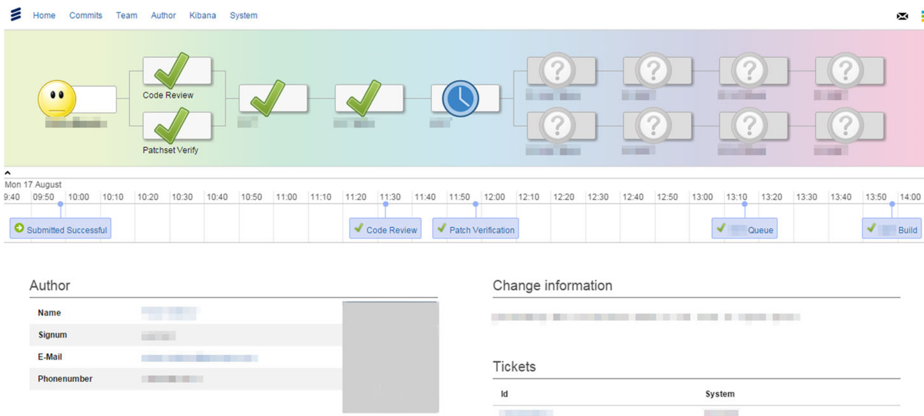

Fig. 2 A developer-centric "Follow Your Commit" visualization of Eiffel generated trace links, providing developers with a real time view and time line of their recent commits, aggregating engineering artifacts from a large number of systems, including multiple continuous integration servers on multiple sites. All represented artifacts offer the option to drill down into further details and/or follow links to applicable external systems. Note that the visualization shows a flow of continuous integration and delivery activities spanning both a component, and two independent systems into which that component is integrated, in line with Software Product Line principles (Clements and Northrop 2002; van der Linden et al. 2007). Sensitive information has been deliberately blurred

the aggravating factors of technology and process diversity (see Section 1): If achieving alignment on common tooling and infrastructure is so difficult, how can the solution to traceability be alignment on a common framework?

As participant observers, we have witnessed several Eiffel deployments in Ericsson, and find that the deciding factor is not necessarily decisions or directives from high level management; arguably as with any adoption of new technology. Rather, notable examples of large scale Eiffel adoption can be found in multi-tiered system contexts (as discussed previously, see Section 1), where the top tier defines its delivery interface in terms of the Eiffel protocol, thereby forcing the lower tiers to comply. To exemplify, where system integration requires incoming deliveries to be communicated and documented using certain Eiffel events (see Section 3.2), constituting a minimum viable level of traceability, the delivering components need to produce those events somehow. They *can* choose to generate them manually at the time of delivery, simply as a different format of delivery report (see Section 3.5). We find, however, that in the face of such requirements the easiest solution is not only for the components to instead generate the events *in situ*, automatically and in real time, but then to propagate the practice downward through the integration chain. Following this initial alignment, the system tier is then able to successively raise the bar by requiring greater detail and scope of the Eiffel events, allowing improved traceability capability to trickle down through the system hierarchy (see Fig. 3).

In this context, identifying the *minimum viable level of traceability* is important, in order to minimize the adoption threshold. It should also be noted that this is in line with the intent of the Eiffel framework: to function as a thin layer, abstracting away lower level divergence (see Section 3.1). In the example above, that thin layer allows the lower tier components to hide both their internal technology stacks and their internal processes, affording them a level of autonomy while at the same time isolating their dependents from the nitty-gritty details of their build, test and integration system.

3.5 Manual Activities Compatibility

While a high degree of automation is a common denominator in continuous integration implementations (Ståhl and Bosch 2014b), in the case studies performed by us both in this and previous work (Ståhl and Bosch 2014a) we find a significant amount of manual

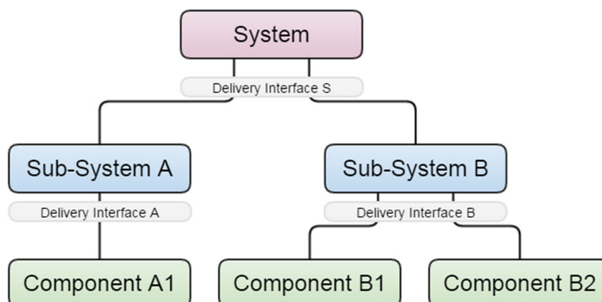

Fig. 3 A hypothetical example of a multi-tiered integration system aligning top-to-bottom on Eiffel based traceability through the definition of delivery interfaces. When Delivery Interface S is defined as an Eiffel interface, this provides an incentive for Sub-Systems A and B both to deploy Eiffel internally and to push that responsibility downward, also defining Delivery Interfaces A and B in terms of Eiffel events. Thus the deployment trickles down through the integration hierarchy

activities in parallel with or subsequent to continuous integration and delivery systems, e.g. manual release decisions or manually executed tests. Traceability is no less of a concern for such manual activities than it is for automated ones, and the same requirements apply — e.g. the importance of cohesive data storage and presentation, as opposed to trace links being scattered across numerous disjointed systems. Despite this, in our experience both as software engineers and researchers, we frequently observe projects where manual testing is conducted using separate processes and tools, e.g. following manually defined test project plans and producing manually authored test reports on ad-hoc formats intended for manual interpretation. Automated testing, on the other hand, is typically executed as part of the continuous integration and delivery system and reported on standardized formats to be visualized and interpreted by generic software packages.

Consequently, even though the Eiffel framework is intended to be used as a part of the continuous integration and delivery system itself, it is designed to allow manually authored events to be communicated identically to automatically generated ones. Indeed, the concept of a thin layer abstracting away underlying details (see Section 3.1) applies once more: by reporting both manual and automated test executions and results using Eiffel events, the same level of traceability can be ensured and the consumer of those events is not exposed to the mode of testing, unless such information is explicitly sought. This ability is particularly important when, as is very often the case, organizations strive to automate as much as possible of their existing manual test cases; by communicating via standardized events their dependents are insulated from the impact of such changes in processes and methods.

4 Identification of Traceability Needs

Prompted by the challenges outlined in Section 1 and striving to understand the most prominent traceability needs as perceived by engineers in the industry, thereby providing context and an opportunity for subsequent validation of the Eiffel framework, two series of interviews were conducted. The first, to identify the most prominent traceability needs in industry, with software professionals in multiple roles (see Section 2) in case A (see Section 4.1 below for a description of the studied cases). The second, to establish whether those needs were unique to that case or shared by others in the industry, with professionals of corresponding roles in cases B and C. This approach was chosen in order to focus on the needs stated by engineers already familiar with and using Eiffel, as such needs may well be different from non-users and possibly reflect the characteristics and capabilities of the framework — encouraging its adoption and/or being influenced by it — and then compare that to the perspectives of engineers not using it.

4.1 Case Descriptions

Three very-large-scale industry cases have been studied in this work.

Case A is a development project in Ericsson AB, a multinational provider of telecommunication networks, networking equipment and operator services as well as television and video systems. This project develops several network nodes containing custom in-house developed hardware. The software is developed by several thousand software engineers spread across numerous sites on three continents and belonging to separate organizations, all with their unique backgrounds, competences, processes, technologies and tools. To accomplish this, considerable

effort has been devoted to implementing a continuous integration and delivery system which integrates on the order of tens of thousands of commits and thousands of tested product builds per month (as of early 2015, and increasing). It automatically integrates components — developed and delivered at a very high frequency by separate and independent organizations — as binaries, similar to the practice described by (Roberts 2004). Despite this, it is stated as formal strategy that “developers shall be responsible for monitoring their contributions in the integration flow and acting upon any faults they introduce”, not just within their components, but also on a system level.

While the software itself is highly configurable it may also be deployed to a wide array of hardware configurations and network topologies. As discussed in Section 1, this poses a formidable test traceability challenge: when, where and how has each requirement been verified? This challenge is exacerbated by the fact that there is, with a few exceptions, no central control or inventory of employed tools, and that they change over time, severely restricting efforts to achieve traceability based on e.g. a single continuous integration server, revision control system or application life-cycle management tool. Furthermore, the product exists in a market subject to both security regulations and national legislations which may place demands on traceability.

Case A has deployed the Eiffel framework, and is continuously expanding the scope of that deployment.

Case B is a similarly large and distributed development project of reusable components integrated into numerous products within Ericsson AB. They pursue an ambitious continuous integration and delivery agenda, and are planning to adopt the Eiffel framework, but are not currently using it.

Case C is the development of software components for Volvo Trucks — components developed both in-house and by third party suppliers and integrated into a highly hardware dependent system solution. They have a history of implementing continuous integration on a component level, and are looking to expand that way of working to the system level. Case C has not deployed or had any access to the Eiffel framework.

4.2 Case a Interviews

Following the reasoning above, the case A interviewees — in the nomenclature of the studied case: test manager, track manager, developer, project manager, continuous integration architect, configuration manager and continuous integration system developer — were asked the interview question shown in Table 1. All of the interviewees were seasoned engineers with at least eight years of industry software development experience from multiple projects, many of them recognized as leaders or experts in their fields within their organization. Their responses were transcribed, translated from Swedish into English and then thematically coded by two of the researchers. During the thematic coding, statements were first categorized into themes and then sub-themes in an iterative process, where themes were restructured and statements reclassified in search of the best possible fit. Certain statements were coded as pertaining to more than one theme. The result of this process was a set of themes representing functional and non-functional needs — not necessarily related to any particular tool or framework, but to traceability in general — allowing analysis of the emphasis placed by the interviewed professionals on these needs (see Fig. 4) through the number of coded statements and the number of interviewees making those statements. While

Table 1 Single interview question posed to interviewees in cases A in order to identify their traceability needs

	Case A Interview Questions
IQ_{a1}	Which types of trace links to do you consider to be important?

Section 4.3 provides explanations of each individual theme, Table 2 shows an overview with the number of included statements and the number of roles making those statements.

As shown in Fig. 4, test result and fault tracing (Trf), upstream tracing (Ust) and content downstream tracing (Dst) are the three dominant functional needs for the interviewees. We find that this fact, together with the examples provided by the interviewed engineers in this area, strongly supports the findings in related work (Asuncion and Taylor 2009) stressing the fact that traceability can play an important role in supporting the development effort itself, rather than only serve purposes of satisfying e.g. legal requirements and certifications.

4.3 Themes

Test Rresult and Fault Tracing (Trf) represents the ability to trace test results and/or faults to particular source or system revisions. This is exemplified by the need to "follow [trace links] in order to understand why we can't build a [system version]". As one developer puts it, what he wants to know if whether his source code changes has caused any faults or not — "my main concern is whether everything is green" — and will when troubleshooting "drill down" through trace links.

Content Upstream Tracing (Ust) refers to the ability to establish the contents of system revisions — modules, libraries, individual source changes, implemented requirements et cetera. Interviewees describe using "trace links for documentation purposes" and how it's

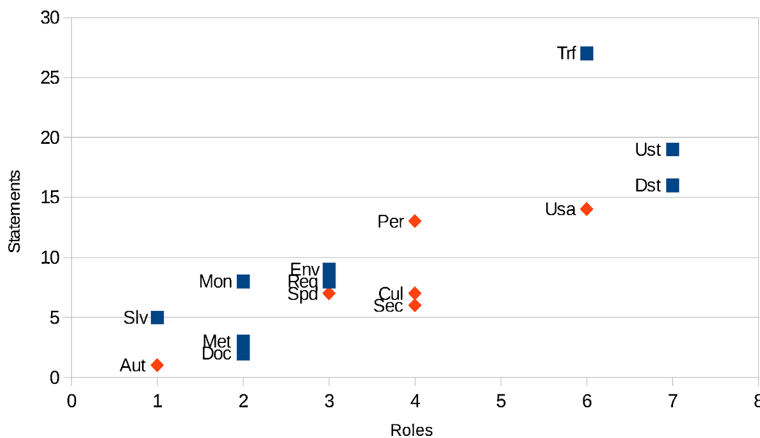

Fig. 4 A scatter plot of the functional (blue squares) and non-functional (red rhombuses) themes representing needs highlighted by interviewee statements (none of which were negative, i.e. stating a lack of need), showing number of discrete statements versus number of roles making those statements. The most emphasized functional needs are Test result and fault tracing (Trf), Content upstream tracing (Ust), Content downstream tracing (Dst). The most emphasized non-functional needs are Usability (Usa) and Persistence (Per). Further explanation of the themes is provided in Section 4.3

Table 2 Overview of interviewee statements coded as pertaining to functional and non-functional needs, showing the number of statements and the number of interviewee roles from which those statements were gathered. See Fig. 4 for a graphic representation

Functional Needs Theme	#roles	#statements
Test result and fault tracing (Trf)	6	27
Content upstream tracing (Ust)	7	19
Content downstream tracing (Dst)	7	16
Context and environment tracing (Env)	3	9
Requirement and work item tracing (Req)	3	8
Monitoring (Mon)	2	8
Metrics and Reports (Met)	2	3
Documentation Tracing (Doc)	2	2
Salvageability (Slv)	1	5
Non-functional Needs Theme		
Usability (Usa)	6	14
Persistence (Per)	4	13
Culture and process (Cul)	4	7
Security and stability (Sec)	4	6
Speed (Spd)	3	7
Automation (Aut)	1	1

important for “knowing that things hang together, that I haven’t lost anything” and to track the development, integration and test progress of features.

Content Downstream Tracing (Dst) is similar to Content upstream tracing (Ust), but takes the opposite viewpoint: where did a given artifact end up? To exemplify, in which system revision was a particular source change first integrated or a certain requirement implemented? As one engineer puts it, “when I’m following my commit from start to finish [I want to] see its status through various [test activities]”, an ability which implies an unbroken chain of trace links from the source code commit through the entire integration and test pipeline of every system that code is integrated into.

Context and Environment Tracing (Env) refers to the conditions under which build and test activities were conducted. This was mentioned by 3 of the interviewed roles, in 9 statements. In the words of one interviewee, you need to know “exactly which tests and test tools [we used] to achieve this result” and you need “serious version control of what’s [building the software]”.

Requirement and Work Item Tracing (Req) is specifically concerned with the ability to connect artifacts such as requirements, user stories and tasks to e.g. source and system revisions. It was e.g. stated that “to us it’s important that you tie a [commit] to requirements, work packages, [high level system requirements] or trouble reports, so you can see what kind of work this is”.

Monitoring (Mon) represents the ability to monitor the health of the integration system, e.g. populating information radiators with real time data on bottlenecks, availability and throughput. It’s also described as a question of traffic management: “We make use [of trace links] to see when queues grow, if we get queue overload [and then to] drill down into the flow to troubleshoot”.

Metrics and Reports (Met) describes how trace links can be used in statistical analysis for purposes of key performance index (KPI) measurements, e.g. "to see how quickly [commits] can pass through [the system]".

Documentation Tracing (Doc) captures how various types of documentation, e.g. trade compliance and product characteristics "should be natural to connect to the product [via trace links], instead of maintaining in large systems and Excel sheets on the side", but no concrete plans or use cases for this have been formulated in the studied case.

Salvageability (Slv) was mentioned only by the configuration manager. It concerns adherence to legal requirements, stating e.g. that "a configuration [deployed at a customer site] shall be possible to correct, and that shall be possible to do perhaps ten years after the development organization that created it has ceased to exist".

Usability (Usa) captures the fact that adequate traceability in theory is not enough — for it to be of any value, the various stakeholders must also be able to make effective use of it. To exemplify, adequate archiving and "being able to search quickly" is highlighted along with the importance of easily track and analyze chains of trace links for e.g. troubleshooting purposes.

Persistence (Per) means that trace link data must be permanently stored and it must be "guaranteed that no data is lost". That being said, however, several of the interviewees reflect that not all traceability data is equally important to keep, that some "is not critical long term, even if it's useful" but that deciding what to keep and what to discard "is tricky".

Culture and Process (Cul) represents how tooling and infrastructure are intimately connected to culture and process. Downstream traceability is highlighted to ensure awareness "of the fact that others are pushing code as well; if you get that gut feeling in every developer it's much easier to do [development and integration of large systems]". One interviewee also considers the fact that improving traceability and visibility by itself may even be counter-productive unless one is able to interpret and respond to it appropriately: "as soon as things start flashing red every manager will ask why it's red — you have to keep calm".

Security and Stability (Sec) represents the need for traceability infrastructure to be resilient both to changes in the development environment and to security threats, as the traceability data is highly sensitive information. One interviewee explains how this invalidates traceability infrastructures on top of e.g. source code management systems: "the connection to repositories comes with a dependence [where] if anyone changes the repository or modifies it you become very vulnerable to that: the traceability breaks if anyone modifies the [repository layout or scheme]".

Speed (Spd) partly overlaps with usability, but focuses particularly on the speed at which one can be notified when a particular commit, system revision or requirement has failed a test is highlighted, stating that "it's really the negative feedback I want as quickly as possible if it exists". It's also explained that this is not just a concern for any one group of stakeholders, but that "the entire organization wants quick feedback on what they do".

Automation (Aut) represents the need for continuous integration and delivery traceability infrastructure to be automated. We find the fact that it was only mentioned explicitly in a single statement highly interesting, particularly in the light of the established prevalence of automation in continuous integration systems (Ståhl and Bosch 2014b) and the fact that the studied case is highly automated in this regard — one interpretation being that the interviewed engineers consider the need for automation implicit.

4.4 Case B and C Interviews

In the second series of interviews, a total of eight engineers from cases B and C (see Section 4.1) were included. The purpose of these interviews was to establish whether the most prominent concerns revealed in case A (see Section 4.2) are unique to that case — conceivably influencing or influenced by the adoption of Eiffel — or shared by engineers in other industry projects. These interviews were conducted using the same methodology, but focusing on the three most prominent functional needs of case A (see Table 3 for interview questions) and inviting the interviewees to elaborate and expand on these attitudes, explicitly asking them to add any of their own reflections or to themselves phrase and/or answer questions they felt were lacking from the interview.

In the case of **Test result and fault tracing (Trf)** interviewees stated e.g. that “absolutely”, this is “particularly important” and “super crucial”. Several explain that while tracing faults to individual source code revisions is not something they have considered, it’s crucial to “at least [trace faults] to the product version”. One interviewee touches upon usability as he relates how his project keeps millions of test results “traced in every direction”, but that “this is a problem for us, [because] it creates an illusion” and that “it doesn’t help if [system verification teams] kill themselves with traceability if [we don’t use it]”, leading them to consider “removing certain trace links [...] because they only produce the impression of a level of control which does not exist”.

In the case of **Content upstream tracing (Ust)** interviewees stating e.g. that “that is absolutely the case, this is something we consider important” and “from my point of view it is incredibly important”.

In the case of **Content downstream tracing (Dst)** interviewees stated e.g. that it’s “extremely important”, “especially when problems occur”, but also that even though

Table 3 Interview questions posed to interviewees in cases B and C in order to assess the extent to which engineers in other cases share the most prominent traceability needs documented in case A, along with the number of positive responses per question

	Case B and C Interview Questions	#positive
IQ_{bc1}	In your development efforts, is the ability to trace product contents from a commit to product releases — downstream traceability — considered important?	8
IQ_{bc2}	In your development efforts, is the ability to trace product contents from a product release to included commits and related life cycle documents and work items — upstream traceability — considered important?	8
IQ_{bc3}	In your development efforts, is the ability to trace test results and faults to relevant source revisions and/or product versions considered important?	8

they themselves found it essential, “I think the interest in this varies [from individual to individual]” and that “it depends on who you ask”.

Consequently, we find that these concerns are not isolated to a single case, but considered highly important by multiple large scale projects and — in our professional experience — also in the industry at large. Interestingly enough, we find that achieving these types of traceability functionality is not *only* considered a crucial challenge in the context of continuous integration, but *also* an absolute and non-negotiable prerequisite for making large scale continuous integration work in the first place. As two of the interviewees explain, the current methods of securing traceability “are not sustainable in the long run”, which “is our greatest impediment to achieve speed [in integration and test]” and that “If we succeed with this, then we’ll succeed. If we fail, then we’ll fail at continuous integration, delivery and everything.”

In summary, based on the statements made by practitioners we conclude that traceability — particularly of content, faults and test results — are key challenges for the industry in achieving large scale continuous integration and delivery. This is because, as demonstrated by the interviewee statements above, insufficient traceability is not only a concern from a certification and legality perspective, but because it hampers effective development practices in a number of areas:

- Troubleshooting faults discovered in continuous integration and delivery.
- Discovering problems such as congestion or resource scarcity in the continuous integration delivery system.
- Providing developers with relevant feedback with regards to the source code changes they commit.
- Determining the content of system revisions.
- Monitoring the progress of features, requirement implementations, bug fixes et cetera.

Of these, the first two are unique to continuous integration and delivery contexts. The third is not, but is arguably particularly relevant in such contexts: in previous work we have often found that the ability of developers to easily determine the status of commits is of great importance in continuous integration and delivery (Ståhl and Bosch 2014a). The final two needs are clearly relevant regardless continuous integration and delivery, but as discussed in Section 1, the ability to satisfy them is affected by these practices.

5 Validation

This section presents validation of the Eiffel framework as a solution to traceability in continuous integration and delivery. This is accomplished through interviews with professionals working both in projects using and not using the framework, as well as through a comparison of the process to gather traceability data in case A with and without Eiffel and a systematic literature review. In addition, the results of these methods are, where applicable, discussed in the context of our own findings as participant observers.

5.1 User Interviews

In order to investigate the experiences of software professionals using the Eiffel framework, further semi-structured interviews were conducted in case A. Following the interviews, the discrete interviewee statements were thematically coded (Robson 2011) and analyzed, and a thematic network (see Fig. 5) was created using the same method and the same themes

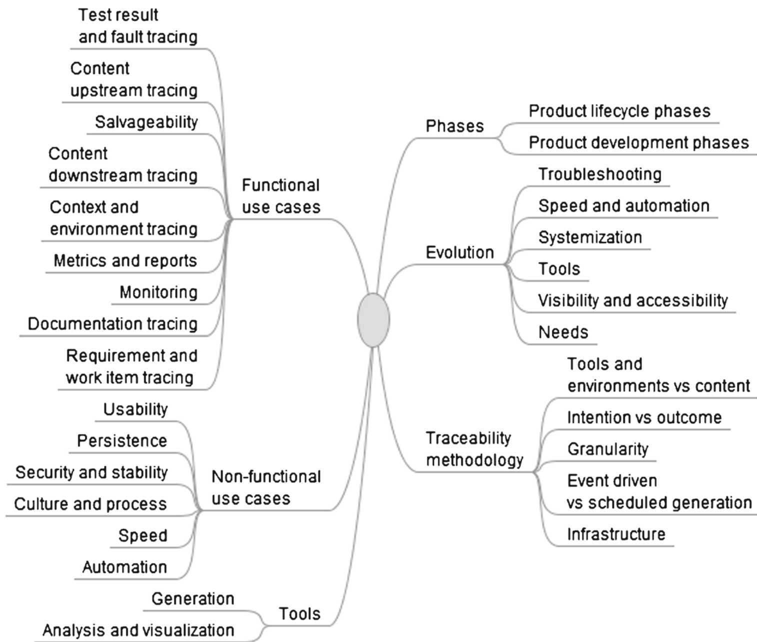

Fig. 5 The thematic network resulting from the individual semi-structured interviews in case A

for functional and non-functional needs as described in Section 4. As in all interviews, the interviewees were encouraged to provide spontaneous input outside of the questions of the interview guide (see Table 4): *How are trace links generated in your development process?*, *How are trace links analyzed in your development process?*, *What use do you make of generated trace links in your role?* and *How have the traceability capabilities of your development process evolved over time?*

A recurring topic throughout all seven interviews was the continuous integration and delivery framework Eiffel, along with the various tools for aggregating, analyzing and presenting trace links generated by it, which has been extensively deployed in the studied project. It is not nearly the only tool in use, but while “Eiffel is a systemized product [...] the others have arisen from needs in the project without any thought to traceability or integration with other systems [...] and that has caused problems”. It is also clear that Eiffel addresses most of the highlighted use cases and is seen as the way forward, and intended to replace many of the other solutions currently used in parallel. The reasons for this include

Table 4 Interview questions posed to engineers in case A using the Eiffel framework

	User Interview Questions
IQ_{u1}	How are trace links generated in your development process?
IQ_{u2}	How are trace links analyzed in your development process?
IQ_{u3}	What use do you make of generated trace links in your role?
IQ_{u4}	How have the traceability capabilities of your development process evolved over time?

performance, data consistency, usability and alignment, stating e.g. that “Eiffel is a platform I think we will rely on more and more [and] it feels like a very solid foundation” and “we are very much headed towards gathering data via [Eiffel] events”. Indeed, it is formal strategy in the project that “the primary method of collecting integration flow data shall be via Eiffel events”, supporting the position of the framework as an important contribution in satisfying continuous integration and delivery traceability.

In line with this, six out of seven interviewees described how they use Eiffel to address test result and fault tracing, downstream content tracing, and/or upstream content tracing. This is not least the case across the delivery interface between components and systems, where they e.g. state that with Eiffel “when we create [release candidates] they sometimes fail [but] using just that [Eiffel] data we can actually in 95 % of cases [locate] the fault, revert that and the next build will be ok”, “modules can tell me which [bugs] they have tried to correct in a given delivery”, “you can tell what is provided”, “[the project manager] can understand the developer’s point of view” and “[I can see] how what I did has been moved forward in the chain”. A project manager also explains how, using this data, one can “know where a feature is in our development chain [...] for instance, ’It’s in a [system revision] now, and now it has achieved [a specified quality threshold]’; or a [change request], a [work item] or a [bug report]”. In a similar vein, other engineers relate how Follow Your Commit (see Fig. 2) “is good for understanding ’What is going on, I have done something and want to understand where it is going”’ answering “What is the status of a [commit] right now?”.

These statements strongly support the feasibility of Eiffel as a solution to large scale traceability, particularly for purposes of integrating across tiers of system completeness (e.g. components into systems), addressing the concerns outlined in Section 1. Taking that one step further, we have also witnessed it facilitating the integration and testing of networks consisting of multiple systems such as the one studied in case A, where, in the words of a test manager, it strips away most of the coordination and alignment otherwise required, as the communication protocol and trace link data is already given.

The one interviewee who did not support this view was a configuration manager in one of the components, who favored a source code management based approach to traceability, as his primary concern was salvageability and reproducibility of software and configurations, stating that Eiffel long term data persistence was not satisfactorily assured in the current deployment.

One interviewee in particular — the continuous integration architect — was eager to discuss methodological and philosophical questions with regards to traceability. For instance, the importance of separating *intended* content from *actual* content — effectively invalidating planning documents for purposes of traceability analysis — and the positive effects of event driven *in situ* trace link generation and the difficulties in achieving sufficient granularity in trace link references.

5.2 Non-User Interviews

In order to investigate whether alternatives to Eiffel are employed in industry, four engineers from cases B and C, respectively, were interviewed. These were purposively sampled to as closely as possible match the roles in case A expressing the strongest personal needs — as opposed to describing the needs of others — in other words, the primary stakeholders of continuous integration and delivery traceability. None of the interviewees were shown Eiffel prior to the interviews; while plans in case B were being made to adopt Eiffel (see Section 4.1) only one of the interviewees was familiar with the framework at the time. The interviews were focused on how various traceability needs are addressed (see Table 5).

Table 5 Interview questions posed to engineers in cases B and C, not using the Eiffel framework

	Non-User Interview Questions
IQ_{nu1}	How, if at all, do you address the need for downstream traceability?
IQ_{nu2}	How, if at all, do you address the need for upstream traceability?
IQ_{nu3}	How, if at all, do you address the need for test result and faults traceability?
IQ_{nu4}	What is your assessment of the usability of your traceability solutions?
IQ_{nu5}	What is your assessment of the persistence of traceability data in your development effort?

Through these interviews, stark differences compared to case A became evident. In both case B and C, delivery from component to system level is manually handled, with manually authored delivery reports and little visibility from component to system, or vice versa. This lack causes several problems — e.g. from a system point of view "we don't know which new test cases [to] introduce for this release [because we] don't have release notes from these small deliveries [so] it's hard to start testing" and testing has "a lot of waste [because we] test with the wrong input data [so] approximately 40 % of our work is wasted due to poor traceability". On the other hand, lack of downstream traceability (from component to system) causes other problems: the developers "don't understand that it's a long chain, they don't understand that they're part of a truck" and it takes too long to get feedback on faults found beyond the component scope to the developer.

In both cases, interviewees state that they do not have adequate solutions for these types of traceability. The traceability they do have is largely based on their source control systems, unstructured documents, requirement management tools and/or ticket management tools, with limited integration between these, so while "there is no tool where I can see that my commit has been included here and verified [at system level]" the data exists "indirectly, but you have to dig for it and I don't think anybody does that" and "right now it's heavy work". In other words, while traceability data exists in e.g. source code repositories, text documents and spreadsheets, "that doesn't mean it is used", confirming the concerns about usability highlighted in Section 4. The manual traceability handling is also identified as a major blocker for more frequent deliveries of software, both at a component and a system level: "today, without continuous integration, it's nearly overbearing, deliveries are very costly", "sometimes it scares me, thinking about how many people fill out various documents and simply create deliveries and sub deliveries, because you want traceability", and therefore work is ongoing to implement "tooling to handle the amount of data and make a different kind of reporting possible".

Statements such as these show that not only is functionality comparable to that of Eiffel lacking, but also that this lack is identified as a problem. Furthermore, they reinforce the position that satisfactory solutions must be sought in tooling and infrastructure, rather than (exclusively) in processes, and that such solutions are a non-negotiable prerequisite for successful large scale implementation of continuous integration and delivery, as discussed in Section 1.

An observation made by us during the interviews is that the concept of downstream traceability, so important to the case A interviewees, was very difficult to convey to the engineers in case B and C: in each of the eight interviews we had to resort to concrete examples and story-telling to explain the question. Even then, while confirming its importance, several interviewees discussed it in terms of what we would label upstream traceability, or traceability limited to the component context. The majority, however, grasped the concept and

in both of the cases explained that this was something they were hoping to achieve. We find this difficulty to communicate the concept highly interesting. Despite the claims of the interviewees to the contrary, it is conceivable that the actual need is small or non-existent, but one may also speculate that there's a hierarchy of needs: in what may be called a *reactive mindset*, where developers delivering to a higher system tier "throw their changes into a black hole, and then get angry mails back a few weeks later", in the words of an engineer interviewed by us in previous work, understanding the system level is not the developer's concern — if there's a problem they'll be told and then react, whereas a lack of "angry mails" is presumed to mean everything is fine. In the *proactive mindset* of case A, however, developers are expected to understand the system and how they affect it (as evident from interviewee statements and formal strategy, see e.g. Section 4.1), allowing them to act on faults they introduce before they cause major disruption to the larger development effort. This requires adequate traceability to support that expanded responsibility — or possibly the other way around: the availability of information has lead to the expanded responsibility. Both as researchers and software professionals ourselves, we consider this a highly interesting question worthy of further investigation (see Section 7.1.2).

As for the non-functional concern of persistence, highlighted as important in the original case study, all of the control case engineers in both cases either stated that their data persistence was adequate, or were uncertain but had no reason to believe it was insufficient.

5.3 Traceability Process Comparison

A recurring topic in the case A interviews was how the Eiffel framework has made traceability data more readily accessible, reducing the time spent "playing detective". As a track manager put it, previously there were "lots of different systems, so if you tried to trace something from a [product release] you had to go look inside [it] to see what's in that, then perhaps in some other system, because that particular organization had their data in another system, and if you had access there you could go in and see which sub-groups there were, and so on and on. Now when I go into [a tool presenting Eiffel data] and look it's fairly simple. It's practically clickable, so I can drill fairly far down." Such statements represent an opportunity to quantitatively assess the improvement, and thereby verify the claims made by the interviewees.

During the case study we found that four years earlier, in 2011, high level management requested an investigation into lead times and frequency of integrations of a platform — developed by a different organization — in the studied case, the background being that no reliable data existed on who used which versions of the platform when. That investigation was conducted by mailing and phoning project managers and integration engineers to ask them about their integrations. Using the technology now available in the same project, we repeated the exercise to fetch the same data and thereby benchmark the tooling now available. A comparison of the two data collection procedures is presented in Table 6.

The difference in data acquisition method and lead time is very clear, and corroborates the statements made by interviewees that with the current tooling "you can tell what is provided [in a given baseline]", "in my role it's [now] much, *much* easier to comprehend the situation" and that "we have achieved transparency in things: [whereas] previously there were a few individuals who could see the status of builds, now [...] you have access to data — before it was very difficult, you had to be a UNIX expert to get to the data". Consequently, we find that this comparison supports Eiffel as a valid contribution by demonstrating its ability to provide content traceability (see Section 4).

Table 6 A comparison of the data collection procedures in 2011 and 2015 to measure the integration frequency of a platform product into the studied case. The data acquisition lead time refers to the total time from beginning to search for the data until that data is acquired — crucially different from effective time spent. Similarly, the integration lead time refers to the time from a source change being committed until it is integrated into a system revision. Note the differences in both data acquisition method and the acquired data, discussed in Section 5.3

	2011	2015
Data acquisition method	Asking engineers by mail and phone	Web based database interface
Data acquisition lead time	15 days	10 minutes
Integration lead time	Varies, estimated average of 30 days	Varies, up to 11 hours
Integration frequency	Varies, 0-2 times a month	Once every 5 days

Apart from this, there is also a striking difference in the actual data: integration lead times have decreased, and frequency has increased. The interviewees stress that no single tool can claim the credit for that, however, but that it's caused by not only a number of improvements on a broad front, but also changes in process and mindset. As a project manager puts it, while he wants to "give credit" to Eiffel, "the big thing is the Lean thinking that's emerged and pushed us to start talking about flows". In our own experience from observing the deployment of Eiffel in multiple contexts, there is a synergy between culture and tooling: the needs that Eiffel addresses are pushed to the fore by Lean and Agile ways of working, and those ways of working are at the same time promoted and encouraged by the visibility it affords. Such observations and interviewee statements like the one above lead us to believe that this synergy, where new technologies have the potential to change the way engineers approach their work, may explain the differences in mindset observed between those using Eiffel and those who do not (see Section 5.2).

5.4 Systematic Literature Review

To investigate whether solutions comparable to Eiffel have been presented in published literature, a systematic literature review (Kitchenham 2004) was conducted. A review protocol was created, containing the question driving the review ("Which solutions to the traceability problem in a continuous integration and/or delivery context have been proposed in literature?") and the inclusion and exclusion criteria (see Table 7). The protocol was informally reviewed by colleagues and then applied.

As shown in Table 7, the initial search in four databases yielded a total of 26 unique results. These were then culled through subsequent application of exclusion criteria to five papers, which are discussed in detail below.

In (Dabrowski et al. 2012) a graph based description of related software artifacts is proposed, and a case is made for its usefulness in ensuring traceability from e.g. requirements to code. This is highly interesting work, but does not address the identified challenges. First, this work focuses on describing the relationships between individual source code files at a given time, functioning as a detailed snapshot of the project's anatomy, rather than automatically recording how the software and artifact dependencies change over time and tracking e.g. real time content (see Section 4). Second, it does not consider the problem of obtaining the required information in a heterogeneous, dispersed and changing tooling landscape.

Table 7 Inclusion and exclusion criteria of the systematic literature review, along with the number of papers yielded from the performed searches and remaining after application of exclusion criteria, respectively

Inclusion Criteria		Yield
IC_1	Scopus search on 2016-02-16 for `TITLE-ABS-KEY(("continuous integration" OR "continuous delivery") AND ("traceability" OR "trace link" OR "trace" OR "artifact")) AND SUBJAREA(COMP)`	21
IC_2	Web of Science search on 2016-02-16 for `TS=("continuous integration" OR "continuous delivery") AND TS=("traceability" OR "trace link" OR "trace" OR "artifact") AND SU=Computer Science`	3
IC_3	IEEE Xplore search on 2016-02-16 for `(("continuous integration" OR "continuous delivery") AND ("traceability" OR "trace link" OR "trace" OR "artifact"))`	7
IC_4	Engineering Village search on 2016-02-20 for `(("continuous integration" OR "continuous delivery") AND ("traceability" OR "trace link" OR "trace" OR "artifact"))`	8
Exclusion Criteria		Remaining set
EC_1	Union of results, excluding duplicates (in which case the most recent publication is used) but not extensions	26
EC_2	Exclusion of conference reviews, posters for industry talks or workshop proceedings, or other items lacking available content beyond abstract and/or references	22
EC_3	Exclusion of publications that, through review of abstracts, are shown not to address any of the software engineering concerns of continuous integration, continuous delivery or traceability	13
EC_4	Exclusion of publications that, through full review, are shown not to propose any tooling or framework based solution to the traceability problem	5

In other words it does not address the problems outlined in Section 1. Even then, keeping such a graph up to date in a project with hundreds or thousands of commits every day would require complete automation — and therefore some form of integration with every employed tool — to be feasible.

This work is expanded upon by (Dabrowski et al. 2013), where a tool set that supports the above graph approach is described. The tool integration aspect is still missing, however, and in its current form the computation times call its applicability to continuous integration into question: following the trend lines of the reported times for detecting and storing source code relationships alone shows that it will soon run into days for an enterprise scale project. Consequently, while representing promising work in an exciting area of software engineering, it does not satisfy the needs of the studied cases.

TRICA, a collection of integrated open source tools for revision control (Subversion), issue tracking (Trac) and continuous integration (Hudson) is introduced by (Ki and Song 2009). Unfortunately, the very fact that it focuses on alignment of a selection of tools invalidates it from the point of view of the outlined challenges (see Section 1). Not only do our cases (see Section 4.1) encompass organizations using dozens of tools in these three categories alone, but this inventory is constantly in flux. As a case in point, at the time when the paper proposing TRICA was published all three of its tools saw use in Ericsson, but have since been largely abandoned in favor of more recent alternatives.

CodeAware (Abreu et al. 2015) proposes a system of static and dynamic probes inserted in a continuous integration environment, as well as *coordinators* and *actors* listening and responding to these probes. This concept is similar to the Eiffel approach in that it is a distributed system providing real time data to stakeholders and potentially driving the behavior of the continuous integration system, and also mentions the possibility of emitting events to subscribers. While early and highly interesting work, in its current form it focuses on facilitating test and verification, rather than traceability per se. Consequently, it is altogether conceivable that a CodeAware-like system and Eiffel could coexist and even complement one another.

Finally, (Rylander 2008) discusses the approach of tracking built artifacts in the context of a "pipeline" of stages. In the years since this article was published, similar solutions have become standard features of popular continuous integration servers such as Jenkins, Bamboo or TeamCity, but fail to address the problems of dispersed, multi-server, multi-process and multi-tool contexts described in Section 1.

Consequently, we find that solutions to the identified problems are lacking in literature, strengthening Eiffel's position as a valid contribution.

5.5 Validation Summary

In this section we have shown that the Ericsson developed Eiffel framework for continuous integration and delivery traceability is a valid solution supporting the critical software engineering capabilities discussed in Section 4:

- **Troubleshooting faults discovered in continuous integration and delivery** is enabled by generating trace links to test results, system revisions, baselines and source code revisions, so that "when we create [release candidates] they sometimes fail [but] using just that [Eiffel] data we can actually in 95 % of cases [locate] the fault, revert that and the next build will be ok".
- **Discovering problems such as congestion or resource scarcity in the continuous integration delivery system** is enabled by "specially tailored" visualizations of Eiffel data that present a "view of how we're faring right now, across the entire flow".
- **Determining the content of system revisions** is enabled by generating trace links between source code revisions, baselines, system revisions and work items et cetera, so that "modules can tell me which [bugs] they have tried to correct in a given delivery" and "you can tell what is provided" in a given system revision. The improvement brought by Eiffel is confirmed by a quantitative comparison of traceability data gathering before and after its deployment.
- **Monitoring the progress of features, requirement implementations, bug fixes et cetera** is enabled by the automated real time generation of trace links as these artifacts progress through the continuous integration and delivery system, whereby one is able to "know where a feature is in our development chain [...] for instance, 'It's in a [system revision] now, and now it has achieved [a specified quality threshold]'; or a [change request], a [work item] or a [bug report]".
- **Providing developers with relevant feedback with regards to the source code changes they commit** is enabled by visualization of real time Eiffel data, allowing developers to answer the question "What is the status of a [commit] right now?".

In addition, the Eiffel framework is applicable to the aggravating circumstances also discussed in Section 1:

- **Scale**, **Geographical diversity** and **Technology and process diversity** through the demonstrated *in vivo* efficacy of the system in case A — a project encompassing thousands of software engineers spread across three continents and multiple organizations, each with their unique processes, nomenclatures, backgrounds and technology stacks (see Section 4.1).

- **Continuous integration and delivery** through actually being a part of and utilizing the continuous integration and delivery system itself, rather than struggling to cope with its consequences in terms of accelerated speeds and frequencies. In this aspect it is crucially different from the (particularly manual) traceability solutions of other projects (see Sections 4 and 5.2).

Furthermore, we have been unable to find comparable alternatives available to the studied industry cases or described in published literature.

6 Threats to Validity

External validity must always be considered when drawing conclusions from a case study. In this work we have mitigated the threat by confirming that the traceability concerns raised are also relevant in other projects, one of them in a separate company, with the caveat that they are all large scale project with multiple tiers of system completeness and that the concerns and needs of the interviewees mostly related to the larger system level, as opposed to the smaller component level. We have also shown that technology similar to what has been developed in case A in order to address those needs is not available in the other cases or in literature, supporting the generalizability of that solution.

Threats to internal validity include selection and compensatory rivalry (Cook et al. 1979). Selection, because interviewees were not randomly selected. Instead, purposive sampling was used — in case A in order to cover as many roles and perspectives as possible, and in cases B and C to find perspectives correlating to those having the most articulate functional needs in case A. Compensatory rivalry, because it is conceivable that interviewees feel compelled to put their project into the best light. It should be noted, however, that there were no incentives in the study to provide positive answers and, in our view, most interviewees tended towards self-criticism, rather than self-praise.

Due to the involvement of two of the researchers in the development of the studied framework, researcher bias is a concern. To defend against this, apart from participant observation, three additional methods of validation have been employed to maximize triangulation (see Section 5).

7 Conclusions and Further Work

In this paper we have studied three very-large-scale, geographically dispersed software development projects. Driven by the traceability challenges identified in related work, we have confirmed through interviews that traceability is a serious concern to software professionals of multiple roles, not least in the context of continuous integration and delivery, where it is in fact regarded as a prerequisite to successful large scale implementation of

these practices. Particularly, based on these findings, we conclude that the lack of adequate traceability solutions poses a threat to several important software engineering abilities:

- Troubleshooting faults discovered in continuous integration and delivery.
- Discovering problems such as congestion or resource scarcity in the continuous integration delivery system.
- Determining the content of system revisions.
- Monitoring the progress of features, requirement implementations, bug fixes et cetera.
- Providing developers with relevant feedback with regards to the source code changes they commit.

In addition, we have established through a systematic review of related work that satisfactory solutions for achieving traceability in a continuous integration and delivery context, particularly in large scale development efforts, are lacking.

In response to this situation, Ericsson has developed Eiffel — a framework "fully integrated with the software development tool chain", as called for in related work (Rempel et al. 2013) and designed to generate *in situ* trace links throughout the continuous integration and delivery process.

In this paper we have presented the Eiffel framework and discussed its design, characteristics and deployment. Furthermore, we have validated the solution using multiple methods:

- Seven software professionals of multiple roles using the Eiffel framework in their day-to-day work have been interviewed to capture their experiences, concluding that the Eiffel framework not only addresses the software engineering abilities listed above, but is also regarded as a long term solution and "a very solid foundation" to build upon.
- Eight software professionals of corresponding roles in projects *not* using the Eiffel framework have been interviewed to establish whether they have access to traceability capabilities comparable to those offered by Eiffel, concluding that not only do they not have access to comparable functionality, but that this lack is identified as a problem to the extent that a satisfactory solution to traceability in continuous integration and delivery is regarded as a prerequisite for the large scale implementation of those practices.
- The methods of gathering relevant traceability data in the same project before and after the deployment of Eiffel have been quantitatively compared, concluding that using the framework the time and effort required are dramatically reduced, as shown in Table 6.
- Published literature has been reviewed in search for comparable solutions, concluding that none are available.

Our conclusion is that this industry developed solution to continuous integration and delivery traceability is — to the best of our knowledge — unique and addresses several critical concerns of large scale software engineering, which otherwise lack adequate solutions.

Furthermore, while carrying out this work we have observed differences in mindset with regards to developer responsibility to understand and monitor their impact in a larger system context, as opposed to a limited component context, between investigated industry projects. We hypothesize that this difference represents an example of positive synergy between tooling and process in that it correlates with the availability of sufficient traceability data to enable such an expanded developer responsibility.

7.1 Further Work

We find that the work reported from in this paper opens up several promising avenues of further research.

7.1.1 Development Activity Extension

As the interviewed engineers state themselves, their work on Eiffel is an ongoing process, with a number of foreseen challenges and opportunities. One of particular interest for further research is the possibility to extend the framework to development activities, e.g. automatically generating trace links to relevant requirements and bug reports as the developer writes his or her code, similar to what is described by (Asuncion and Taylor 2009). Such an extension has great potential for complementing current Eiffel functionality, which currently relies on developers entering references to artifacts outside of the continuous integration and delivery system — e.g. bug reports, work items and requirements — with consequent risks of inconsistent formatting, forgetfulness and other human error factors. Automated real time analysis of the developer's work, as proposed in related work, would address these vulnerabilities.

7.1.2 Developer Responsibility Mindset

As discussed in Section 5.2, we find the observed differences in developer responsibility mindset to be highly interesting and worthy of further investigation, not least from a large scale software development methodology perspective. Intriguing questions include the interaction between available infrastructure and mindset, as well as how that difference in responsibility and mindset affects actual behavior and outcome.

7.1.3 Improved Trace Link Analysis

Many of the use cases for analyzing and visualizing traceability data observed in the studied cases requires information to be aggregated from large sets of individual trace links. To exemplify, the Follow Your Commit visualization (see Fig. 2) mines various trace links — to builds, automated activities queuing, starting and stopping, tests executions, work items et cetera — and populate data objects representing the life-cycle of a source code revision as it progresses through the integration pipeline. Another example is statistical analysis of traceability data, e.g. to monitor the system and identify congestion or resource scarcity issues (see Section 4).

The implementations for such analyses and visualizations that we have witnessed are, while based on the shared Eiffel protocol, still custom ad-hoc implementations. At the same time, the possibility for conducive generic solutions exists, e.g. in the form of distributed real time computing systems as Apache Storm and Apache Spark. We consider research into the application of such systems to the problem of processing real time traceability data to be highly interesting.

Acknowledgments We wish to extend our sincere gratitude to the insightful, generous and helpful engineers at Ericsson AB and AB Volvo.

References

Abreu R, Erdogmus H, Perez A (2015) Codeaware: sensor-based fine-grained monitoring and management of software artifacts. In: Proceedings of the 37th international conference on software engineering-volume 2, pp 551–554. IEEE Press

Antoniol G, Gerardo C, Gerardo C, De Lucia A, Ettore M (2002) Recovering traceability links between code and documentation. IEEE Trans Softw Eng 28(10):970–983

Asuncion HU, Taylor RN (2009) Capturing custom link semantics among heterogeneous artifacts and tools. In: Proceedings of the 2009 icse workshop on traceability in emerging forms of software engineering, pp 1–5. IEEE Computer Society

Asuncion HU, Asuncion AU, Taylor RN (2010) Software traceability with topic modeling. In: Proceedings of the 32nd acm/ieee international conference on software engineering-volume 1, pp 95–104. ACM

BEL-V BfS CSN ISTec ONR SSM STUK (2013) Licensing of safety critical software for nuclear reactors

Bouillon E, Mäder P, Philippow I (2013) A survey on usage scenarios for requirements traceability in practice. In: Requirements engineering: Foundation for software quality. Springer, pp 158–173

Center of Excellence of Software Traceability (2015). http://coest.org[Online; accessed 11-June-2015]

Cleland-Huang J, Gotel OCZ, Hayes JH, Mäder P, Zisman A (2014) Software traceability: trends and future directions. In: Proceedings of the on future of software engineering, pp 55–69. ACM

Clements P, Northrop L (2002) Software product lines, Practices and patterns. The sei series in software engineering. Addison-Wesley

Cook TD, Campbell DT, Day A (1979) Quasi-experimentation: Design & analysis issues for field settings, vol 351. Houghton Mifflin, Boston

Dabrowski Robert, Stencel K, Timoszuk G (2012) Improving software quality by improving architecture management. In: Proceedings of the 13th international conference on computer systems and technologies, pp 208–215. ACM

Dabrowski R, Timoszuk G, Stencel K (2013) One graph to rule them all software measurement and management. Fundamenta Informaticae 128(1-2):47–63

Dömges R, Pohl K (1998) Adapting traceability environments to project-specific needs. Commun ACM 41(12):54–62

European Cooperation for Space Standardization (2015) ECSS-E-40C

Fitzgerald B (2006) The transformation of open source software. Mis Quarterly

Gotel OCZ, Finkelstein ACW (1994) An analysis of the requirements traceability problem. In: Requirements engineering, 1994., proceedings of the first international conference on IEEE, pp 94–101

Hove SE, Anda B (2005) Experiences from conducting semi-structured interviews in empirical software engineering research. In: Software metrics, 2005. 11th ieee international symposium, 10. IEEE

Humble J, Farley D (2010) Continuous delivery: reliable software releases through build, test, and deployment automation. Pearson Education

Ki Y, Song M (2009) An open source-based approach to software development infrastructures. In: Proceedings of the 2009 ieee/acm international conference on automated software engineering, 525–529. IEEE Computer Society

Kitchenham B (2004) Procedures for performing systematic reviews. Keele UK Keele University 33(2004):1–26

Mäder P, Gotel O, Philippow I (2009) Motivation matters in the traceability trenches. In: Requirements engineering conference, 2009. re'09. 17th ieee international, pp 143–148. IEEE

Meyer B (1988) Eiffel: A language and environment for software engineering. J Syst Softw 8(3):199–246

Nerur S, Mahapatra R, Mangalaraj G (2005) Challenges of migrating to agile methodologies. Commun ACM 48(5):72–78

Oliveto R, Malcom G, Poshyvanyk D, De Lucia A (2010) On the equivalence of information retrieval methods for automated traceability link recovery. In: 18th international conference on program comprehension (icpc), pp 68–71. IEEE

QuEST Forum (2015) TL9000 Quality Management System. www.tl9000.org. [Online; accessed 11-June-2015]

Radio Technical Commission for Aeronautics (2011) DO-178C, Software Considerations in Airborne Systems and Equipment Certification

Ramesh B, Jarke M (2001) Toward reference models for requirements traceability. IEEE Trans Softw Eng 27(1):58–93

Rempel P, Mader P, Tobias K (2013) An empirical study on project-specific traceability strategies. In: Requirements engineering conference (re), 2013 21st ieee international, pp 195–204. IEEE

Roberts M (2004) Enterprise continuous integration using binary dependencies. In: Extreme programming and agile processes in software engineering. Springer, pp 194–201

Robson C (2011) Real world research: A resource for users of social research methods in applied settings 3rd edition. John Wiley & Sons, West Sussex

Runeson P, Höst M (2009) Guidelines for conducting and reporting case study research in software engineering. Empir Softw Eng 14(2):131–164

Rylander S (2008) Building a robust development environment. Dr Dobbs Journal

Ståhl D, Bosch V (2013) Experienced benefits of continuous integration in industry software product development: A case study. In: The 12th iasted international conference on software engineering,(innsbruck, austria, 2013), pp 736–743

Ståhl D, Bosch J (2014a) Automated software integration flows in industry: a multiple-case study. In: Companion proceedings of the 36th international conference on software engineering, pp 54–63. ACM

Ståhl D, Bosch J (2014b) Modeling continuous integration practice differences in industry software development. J Syst Softw 87:48–59

Ståhl D, Hallén K, Bosch J (2015) Continuous integration and delivery traceability in industry: Needs and practices (in review). In: Submitted for review

The Open-DO Initiative (2015) http://www.open-do.org[Online; accessed 12-June-2015]

van der Linden FJ, Schmid K, Rommes E (2007) Software product lines in action, The best industrial practice in product line engineering. Springer. https://books.google.se/books?id=PC4LyoSNNAkC

Wang W, Gupta A, Wu Y (2015) Continuously delivered? periodically upyeard? never changed? studying an open source project's releases of code, requirements, and trace matrix. In: Just-in-time requirements engineering (jitre), 2015 ieee workshop on IEEE, pp 13–16

Wieringa R (1995) An introduction to requirements traceability

Daniel Ståhl is continuous integration subject matter expert and architect at Ericsson AB. He has a background of nine years of software development, integration and architecting in the telecom industry, where his work primarily revolves the application of continuous integration and delivery practices to multinational enterprise scale organizations. He received a MSc degree from Linköping University, Sweden, in 2007.

Kristofer Hallén is a Continuous integration architect at Ericsson AB. He has a background of ten years of software development, integration, testing and software architecture in the telecom and automotive industry. He has focused on the continuous integration and delivery of large scale systems including both hardware and software. He recieved a Licentiate of Engineering degree from Linköping University, Sweden in 2006.

Jan Bosch is professor of software engineering and director of the software research center at Chalmers University Technology in Gothenburg, Sweden. Earlier, he has worked as Vice President Engineering Process at Intuit Inc and as head of the Software and Application Technologies Laboratory at Nokia Research Center, Finland. Before joining Nokia, he headed the software engineering research group at the University of Groningen, The Netherlands, where he holds a professorship in software engineering. He received a MSc degree from the University of Twente, The Netherlands, and a PhD degree from Lund University, Sweden.