2016 42th Euromicro Conference on Software Engineering and Advanced Applications

Continuous Integration and Delivery Traceability in Industry: Needs and Practices

Daniel Ståhl
and Kristofer Hallén
Ericsson AB
Linköping, Sweden
Email: {daniel.stahl,kristofer.hallen}@ericsson.com

Jan Bosch
Chalmers University of Technology
Göteborg, Sweden
Email: jan@janbosch.com

***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 identifies traceability as a key challenge in achieving continuous integration and delivery and documents an industry developed framework — Eiffel — designed to provide real time traceability in continuous integration and delivery. The efficacy of the Eiffel framework is then investigated through comparison with previous traceability methods.**

***Index Terms*—continous integration; continuous delivery; traceability; eiffel**

I. Introduction

Traceability in software engineering is widely recognized as an important concern, with substantial research and engineering efforts invested in solving its challenges [1]–[3]. Traceability can serve multiple purposes, such as internal follow-up, evaluation and improvement of development efforts [4], 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 [5]–[8].

As defined by the Center of Excellence of Software and Systems Traceability (CoEST) [9], 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, organizations often struggle to achieve sufficient traceability [10], not least due to lacking support from methods and tools [11] — indeed, related work indicates the importance of traceability tooling that is ”fully integrated with the software development tool chain” [12] and calls for further research into technical infrastructure [13]. Furthermore, there is a gap between research and practice in the traceability area, with few studies on traceability practices in the industry [10].

The challenges of achieving traceability do not only increase with project size — more engineers, often geographically dispersed, generating more engineering artifacts to be linked and more communication overhead [14] — but also with tooling diversity, causing artifacts to be created and stored in diverse formats, requiring additional integration efforts.

A crucial part of the traceability problem is one of generating accurate and consistent trace links [9] 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, several factors, including scale, geographical diversity and technology and process diversity further complicate the problem. Agile methodologies in general may pose additional challenges, e.g. by encouraging reduced overhead [15]. Continuous integration and delivery in particular can be problematic, however: with very frequent deliveries, all of which are potentially deployable to customers, all must also comply with traceability requirements. This precludes the option of manual tracing of engineering artifacts through the development process for every release, and places even greater emphasis on the need for effective tool support.

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 [16], 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 [17] — affords the ability to chronologically structure and analyze trace links and to provide engineers with immediate feedback [18]. This implies 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.

2376-9505/16 $31.00 © 2016 IEEE
DOI 10.1109/SEAA.2016.12

Following this, the research question investigated in this paper is *What is an effective infrastructure based traceability solution for very-large-scale industry projects in a continuous integration and delivery context?*

The contribution of this paper is twofold. First, it presents an industry developed open source framework designed to achieve continuous integration and delivery traceability in very-large-scale development. Second, it investigates its efficacy by quantitatively comparing traceability process in an industry case before and after adoption of the framework.

The remainder of this paper is structured as follows. The next section presents the employed research method, whereupon the studied case is described in section III. Following this the Eiffel framework, a proposed solution to the identified challenges of traceability in continuous integration and delivery is presented in section IV, whereupon traceability process with and without Eiffel in the studied case is compared in section V. The paper is then concluded in section VI.

II. Research Method

To investigate the research question a very-large-scale software development project in Ericsson AB and its adoption of the open source Eiffel framework (not related to the programming language [19]) was studied. Eiffel, originally developed in Ericsson in response to a lack of adequate available solutions, was documented through perusal of documentation, informal discussions with continuous integration architects and *in situ* demonstrations, as well as drawing on our findings as participant observers [20].

To document the efficacy of the Eiffel framework in practice, the effort required to answer particular traceability related questions was then compared using and not using the framework. Furthermore, findings from observer participation as continuous integration subject matter expert and product owner, respectively, were documented: two of the researchers have been involved in the development of the framework and its deployment in large parts of Ericsson.

III. Case Description

The studied case 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 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, similarly to the practice described by [21]. 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. In order to achieve continuous integration and delivery traceability, the project has deployed the Eiffel framework and is continuously expanding the scope of that deployment.

IV. 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 the studied case (see section III), among many other projects and units within the company, in order to address traceability in a continuous integration and delivery context. It is now in wide-spread use within the company and promoted as a global framework for enterprise level continuous integration and delivery. It has since been licensed as open source and is available at GitHub[1], where more in-depth documentation can be found along with an overview of implementations available as open source.

The core concept of Eiffel 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 directed acyclic graphs (DAGs) of event. A simplified example of such a graph is shown in figure 1. The querying, analysis and visualization of these graphs can then be used to answer traceability questions. It is worth noting that 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. [22], in that it only operates on structured data and does not attempt probabilistic analysis.

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 [18]; 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.

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”

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

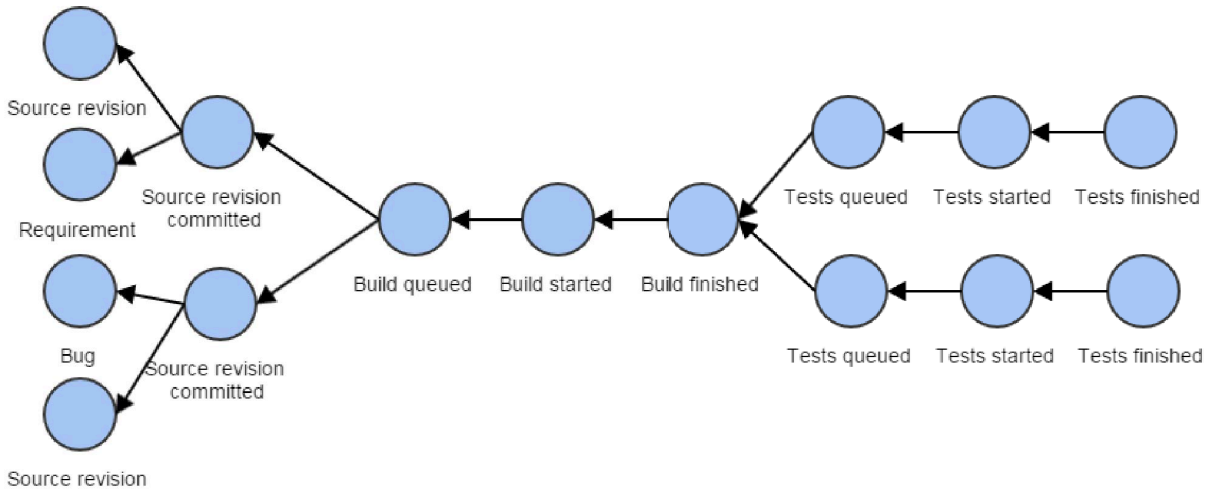

Fig. 1: A simplified example of an Eiffel event graph. Even based on such a simple graph, a number of questions relating to the identified challenges can be addressed in real time. These include code churn, commit frequency, implemented requirements and bugs per build, lead time from commit to various 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.

and ”upstream” in the words of Ericsson’s engineers, borrowing terminology sometimes used in dependency management [23].

A. Implementation

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 message bus. The payload of the message is a JSON document containing semantic links to other uniquely identified events (e.g. stating why an activity triggered, or identifying the item under test) as well as information regarding the event itself (e.g. a test verdict or the location of a published artifact). These messages are typically generated automatically (e.g. by plugins to various tools, such as Jenkins, Nexus, Artifactory, Gerrit et cetera) and 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 messages are not only used for documentation of trace links, but to drive the continuous integration system. To exemplify, a listener receiving a message saying that a new artifact has been published may trigger a test activity as a consequence, which in turn dispatches more messages which are received by yet another listener, triggering further actions.

An important principle in the Eiffel framework is that these 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. 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. Furthermore, it is worth noting that there is no synchronization or central management of events: Eiffel represents a federated, decentralized approach where every producer and consumer is independent from every other, with the exception that producing valid trace links (in the form of event references) may require database lookups — a task which can be greatly simplified by dedicated event assembly and dispatch services.

B. Analysis

The analysis of Eiffel 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 can be seen in figure 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 analysis hold great promise for even more advanced data driven engineering based on Eiffel.

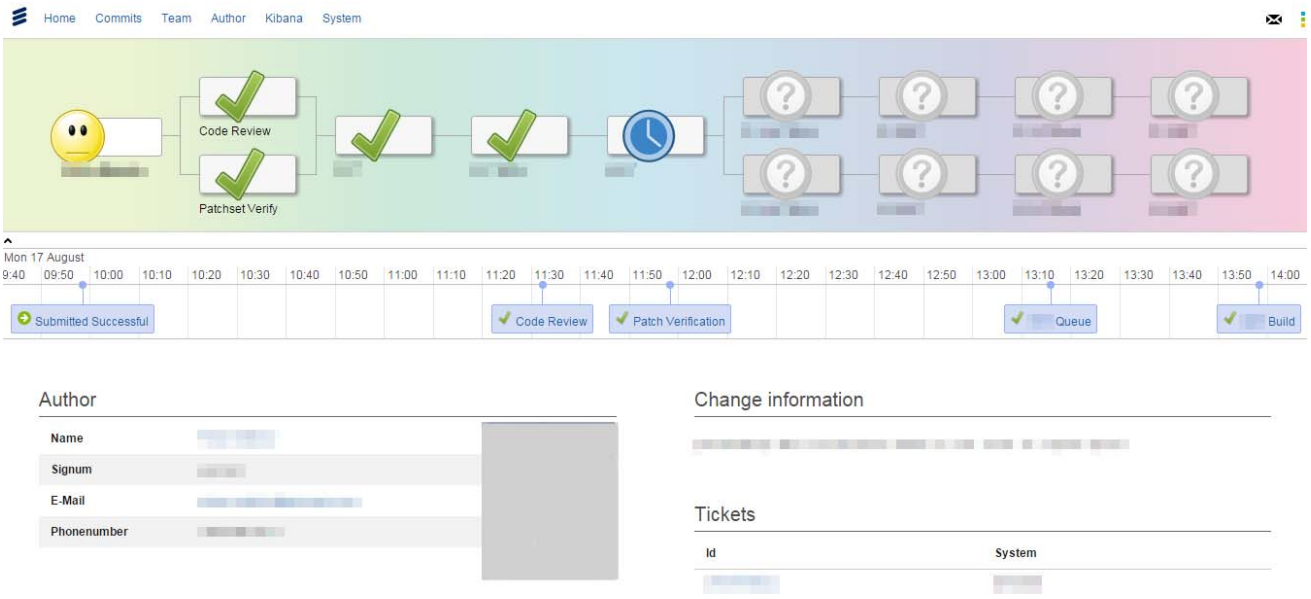

Fig. 2: A developer-centric "Follow Your Commit" visualization of Eiffel generated trace links, providing developers with a real time view and time line of their recent commits, aggregating engineering artifacts from a large number of systems, including multiple continuous integration servers on multiple 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 [24], [25]. Sensitive information has been deliberately blurred.

V. Traceability Process Comparison

A recurring theme in interviews with engineers in the studied case 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 brought by Eiffel adoption.

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 I.

	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

TABLE I: Comparison of traceability data collection procedures in 2011 and 2015.

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".

While it may be argued that any comparison between an automated tool and a manual process should be a foregone conclusion, we point to the fact that what we describe here is not a hypothetical exercise, but the de facto employed processes to address traceability concerns in the studied case before and after adoption of the Eiffel framework — a framework born from the identified lack of automated solutions

to replace those manual processes in a satisfactory way. Consequently, we find that this comparison supports Eiffel as a valid contribution by demonstrating its ability to provide content traceability.

VI. Conclusion

In this paper we have identified software traceability as a key challenge in building continuous integration and delivery capabilities in large scale industry projects. We have also documented Eiffel — an Ericsson developed open source framework that is "fully integrated with the software development tool chain", as called for in related work [12], addressing the challenges of automated *in situ* trace link generation from within a continuous integration and delivery system and providing real time feedback to a wide range of stakeholders. This framework has been deployed in multiple projects and units within the company, including the studied case. In that case we have found that a wide array of tools used to address traceability needs, but that Eiffel is foremost among these: it is identified as the solution of the future and with much greater ambitions than addressing a specific problem encountered by a "frustrated developer" who proceeded to "hack something together", which is otherwise described by engineers in the studied case as being the rule.

This framework is presented in detail in section IV, its main characteristics being that it automatically generates trace links for engineering artifacts created in the continuous integration and delivery system. This results in a graph of *events* with semantic references, which are traversed, analyzed, aggregated and presented by a number of tools compatible with the same protocol, thereby addressing a number of needs, including the ones discussed above. Furthermore, the traceability data is broadcast on a highly available and scalable global message bus, providing a shared communication protocol to express a set of core concepts, regardless of underlying tools and methods.

The efficacy of the Eiffel framework is subsequently investigated through quantitative comparison of traceability methods with and without the framework.

As the traceability concerns addressed by Eiffel are not unique to the studied case we argue that the solution is generalizable to other continuous integration and delivery projects — particularly very-large-scale projects. That being said, we consider further validation of the Eiffel framework and its ability to address traceability concerns in an industrial continuous integration and delivery context to be a highly relevant area of future work.

Acknowledgment

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

References

[1] O. C. Gotel and A. C. Finkelstein, "An analysis of the requirements traceability problem," in *Requirements Engineering, 1994., Proceedings of the First International Conference on*. IEEE, 1994, pp. 94–101.

[2] B. Ramesh and M. Jarke, "Toward reference models for requirements traceability," *Software Engineering, IEEE Transactions on*, vol. 27, no. 1, pp. 58–93, 2001.

[3] R. Wieringa, "An introduction to requirements traceability," 1995.

[4] R. Dömges and K. Pohl, "Adapting traceability environments to project-specific needs," *Communications of the ACM*, vol. 41, no. 12, pp. 54–62, 1998.

[5] BEL-V, BfS, CSN, ISTec, ONR, SSM, STUK, "Licensing of safety critical software for nuclear reactors," 2013.

[6] QuEST Forum, "TL9000 Quality Management System," www.tl9000.org, 2015, [Online; accessed 11-June-2015].

[7] European Cooperation for Space Standardization, "ECSS-E-40C," 2015.

[8] Radio Technical Commission for Aeronautics, "DO-178C, Software Considerations in Airborne Systems and Equipment Certification," 2011.

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

[10] P. Mäder, O. Gotel, and I. Philippow, "Motivation matters in the traceability trenches," in *Requirements Engineering Conference, 2009. RE'09. 17th IEEE International*. IEEE, 2009, pp. 143–148.

[11] E. Bouillon, P. Mäder, and I. Philippow, "A survey on usage scenarios for requirements traceability in practice," in *Requirements Engineering: Foundation for Software Quality*. Springer, 2013, pp. 158–173.

[12] P. Rempel, P. Mader, and T. Kuschke, "An empirical study on project-specific traceability strategies," in *Requirements Engineering Conference (RE), 2013 21st IEEE International*. IEEE, 2013, pp. 195–204.

[13] J. Cleland-Huang, O. C. Gotel, J. Huffman Hayes, P. Mäder, and A. Zisman, "Software traceability: trends and future directions," in *Proceedings of the on Future of Software Engineering*. ACM, 2014, pp. 55–69.

[14] F. P. Brooks, *The mythical man-month*. Addison-Wesley Reading, MA, 1975, vol. 1995.

[15] S. Nerur, R. Mahapatra, and G. Mangalaraj, "Challenges of migrating to agile methodologies," *Communications of the ACM*, vol. 48, no. 5, pp. 72–78, 2005.

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

[17] H. U. Asuncion, A. U. Asuncion, and R. N. Taylor, "Software traceability with topic modeling," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 95–104.

[18] H. U. Asuncion and R. N. Taylor, "Capturing custom link semantics among heterogeneous artifacts and tools," in *Proceedings of the 2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering*. IEEE Computer Society, 2009, pp. 1–5.

[19] B. Meyer, "Eiffel: A language and environment for software engineering," *Journal of Systems and Software*, vol. 8, no. 3, pp. 199–246, 1988.

[20] C. Robson, *Real World Research: A resource for Users of Social Research Methods in applied settings 3rd Edition*. West Sussex: John Wiley & Sons, 2011.

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

[22] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Transactions on Software Engineering*, vol. 28, no. 10, pp. 970–983, 2002.

[23] J. Humble and D. Farley, *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[24] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*, ser. The SEI series in software engineering. Addison-Wesley, 2002. [Online]. Available: https://books.google.se/books?id=tHGFQgAACAAJ

[25] F. van der Linden, K. Schmid, and E. Rommes, *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer Berlin Heidelberg, 2007. [Online]. Available: https://books.google.se/books?id=PC4LyoSNNAkC