TimeTracer: A Tool for Back in Time Traceability Replaying

Christoph Mayr-Dorn
Johannes Kepler University Linz
Linz, Austria
christoph.mayr-dorn@jku.at

Michael Vierhauser
Johannes Kepler University Linz
Linz, Austria
michael.vierhauser@jku.at

Felix Keplinger, Stefan Bichler, Alexander Egyed
Johannes Kepler University Linz
Linz, Austria

ABSTRACT

Ensuring correct trace links between different types of artifacts (requirements, architecture, or code) is crucial for compliance in safety-critical domains, for consistency checking, or for change impact assessment. The point in time when a trace link was created, however, (i.e., immediately during development or weeks/months later) has a significant impact on the quality of these trace links. Assessing quality thus relies on obtaining a historical view on artifacts and their trace links at a certain point in the past which provides valuable insights on when, how, and by whom, trace links were created. This work thus presents *TimeTracer*, a tool that allows engineers to go back in time – not just to view the history of artifacts but also the history of trace links associated with these artifacts. *TimeTracer* allows easy integration with different development support tools such as Jira; and it stores artifacts, traces, and changes thereof in a unified artifact model: https://youtu.be/Av7NJqGQDuA.

CCS CONCEPTS

• **Software and its engineering → Software development process management**.

KEYWORDS

traces, history, replay, versioning, process, Jira

ACM Reference Format:
Christoph Mayr-Dorn, Michael Vierhauser, and Felix Keplinger, Stefan Bichler, Alexander Egyed. 2020. TimeTracer: A Tool for Back in Time Traceability Replaying. In *ICSE '2020: International Conference on Software Engineering – Tool Demonstration.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/xxxxxxxxxxxxx

1 INTRODUCTION

Quality assurance engineers, product managers, or team leads need to maintain trace links between different types of artifacts for various purposes. This is especially crucial in safety-critical domains to fulfill regulatory requirements such as the DO-178C standard for airborne systems, to facilitate consistency checks, or supporting change impact assessment. One problem engineers face is that for these activities the *current* set of trace links and artifacts' status is often insufficient to properly assess (process) quality or whether the relevant traces were available when engineers engaged in design, implementation, or testing activities. A quality assurance engineer, for example, needs to know how trace links were established and how the process of creating them progressed (when a specific trace link was created, updated, or reviewed). The point in time when a trace link was created, e.g., when developers create links between requirements and code immediately when working on new code or weeks later (when they barely remember precise details on what they have been working on), has a severe impact on the quality of these trace links. It might further make a difference whether requirement-to-architecture trace links were in place when development started, or whether these have been created after the fact. Obtaining a historical view on artifacts and their trace links, as well as the version of linked artifacts at a certain point in time, allows to gain insights when and how trace links were created (e.g., during design or just a the end before a quality gate) and thus can provide invaluable feedback whether engineers followed the intended process. Furthermore, when engineers detect errors in already reviewed trace links, they may return to a state of links and artifacts at the time of the last review to better understand why certain trace links were (not) set at that particular point in time. A quality assurance engineer may check for how long an artifact, or the process, were in an inconsistent state. Returning to a prior state and stepping through changes allows the realistic testing of new quality/trace constraints, for example, testing whether addition constraints are able to timely identify a newly emerged undesirable situation. Merely resetting an artifact to a previous version or inspecting its change log – as most tools support – is insufficient as trace links typically connect multiple different (types) of artifacts, thus requiring to simultaneously and consistently rewinding those artifacts to a previous version.

In order to address these challenges, we propose a novel approach and tool called *TimeTracer*, that provides a mechanism to go back in time across not just artifacts, but also traces. *TimeTracer* connects to various software development support tools such as Jira [1], e.g., used for issue tracking and (agile) project management, or Jama [8] for managing requirements, and stores artifacts, traces, and changes thereof in a unified artifact model. A quality assurance engineer, for example, selects the artifacts s/he is interested in, the trace scope (the extent to which additional artifacts connected via trace links are selected for reverting), and the timestamp in the past. *TimeTracer* then retrieves the directly and indirectly identified artifact from the database and then, step-by-step, applies changes until the provided timestamp is reached. The engineer can inspect the state of artifacts and trace links at that time and additionally step change-by-change through time (i.e., replay how artifacts and traces evolved over time). To show the feasibly of the approach, we conducted a preliminary evaluation of our tool using data from the Dronology [2] project.

ICSE'2020, May 2020, Seoul, South Korea

ACM ISBN 978-1-4503-XXXX-X/xx/xx…$15.00
https://doi.org/10.1145/xxxxxxxxxxxxx

2 USAGE SCENARIOS & RELATED WORK

An artifact's history (aka previous artifact and trace link states) can be useful for various different purposes and application scenarios. In the following we briefly describe three motivating scenarios in which *TimeTracer* can provide valuable support for analyzing and assessing the quality of artifacts and trace links.

1 – Sprint retrospective: While burndown-charts and other analyses, typically baked into agile project management tools, can provide interesting data, *TimeTracer* 's ability to step back and forth in time can provide insights that greatly exceed those that these analyses can provide. For example, when discussing what specific tasks and activities required more time or resources, or coordination than expected, engineers can precisely re-establish the process information and artifact status at the time of each situation. This could allow to, on a very fine-grained level, analyze what led to a certain problem by stepping back in time and revisiting the steps that lead to the issue, or stepping forward to asses the impact on other tasks or development artifacts.

2 – Developing Quality Constraints: When quality assurance engineers introduce new constraints (e.g., a requirement's validation method needs to trace to an appropriate test method), they need to test them on changing artifacts in realistic scenarios. Manually creating those scenarios and preparing a change history is tedious and potentially error-prone: important conditions could be overlooked or completely missing. Artifact and trace history replayed via *TimeTracer* facilitates the use of real data sets and ensures that new constraints are correctly evaluated and detect violations when changes occur. This procedure is particularly useful when an undesirable situation (e.g., releasing without completing a test coverage review) has been identified that wasn't covered by the existing set of constraints. When implementing and adding new constraints, the quality assurance engineer can use *TimeTracer* to check on the fly whether the new constraints would have caught the undesirable situation in the past right after it occurred.

3 – Software Engineering Research Evaluation: Evaluation of novel development support tools or recommendation algorithms, for example, for newly reported bugs often relies on capturing what decisions or results a prototype produces, given a particular (set of) input artifacts or process states. These then, in turn, need to be compared to a later/subsequent state, in order to assess the prototypes functionality and usefulness. For example, the ability of an algorithm to predict/recommend a trace link between a requirement and a test case based on textual description can be evaluated by running the algorithm on the artifact version before a trace has been established, and then comparing the recommendation with the replayed artifacts and traces at a later point in time. With *TimeTracer* , researchers can return an artifact to its initial version, execute their recommender, and then replay the artifact to its final version to compare it to their recommendations.

2.1 Related work and tools

Previous work on support for coordination in the software engineering domain includes Hipikat [3] which supports developers in retrieving relevant artifacts from a project's overall history. Its considers documents, tasks, commits, messages, and artifact changes but does not included detailed interaction history or artifact dependencies. Wolf [7] extracts artifact ownership and changes from source code repositories and generates traces between artifacts and engineers. The tool provides an organizational view for managers and an individual view for developers to support impact analysis activities. More recently, various approaches have been proposed that leverage information from different artifact repositories in order to create, maintain or augment incomplete trace links [9, 10] or use historical information [6] for predicting impacted classes. Design space [4] supports the integration of artifacts (and their traces) from multiple engineering domains and also maintains a version history. It, however, provides no dedicated replay mechanism nor mechanism to import an existing artifact history. One commercial tool "Replay for Jira" [1] is an add-on to Jira. It, however, solely focuses on visualizing how Jira issue properties changed over time. While this can provide valuable information in an agile context for scrum masters, or project managers, it does not provide an API for external data analysis outside of Jira and hence does not provide support for including artifacts and traces to other repositories or tools. *TimeTracer* in contrast supports arbitrary artifacts and traces, and provides a simple but powerful API for integration.

3 TOOL ARCHITECTURE

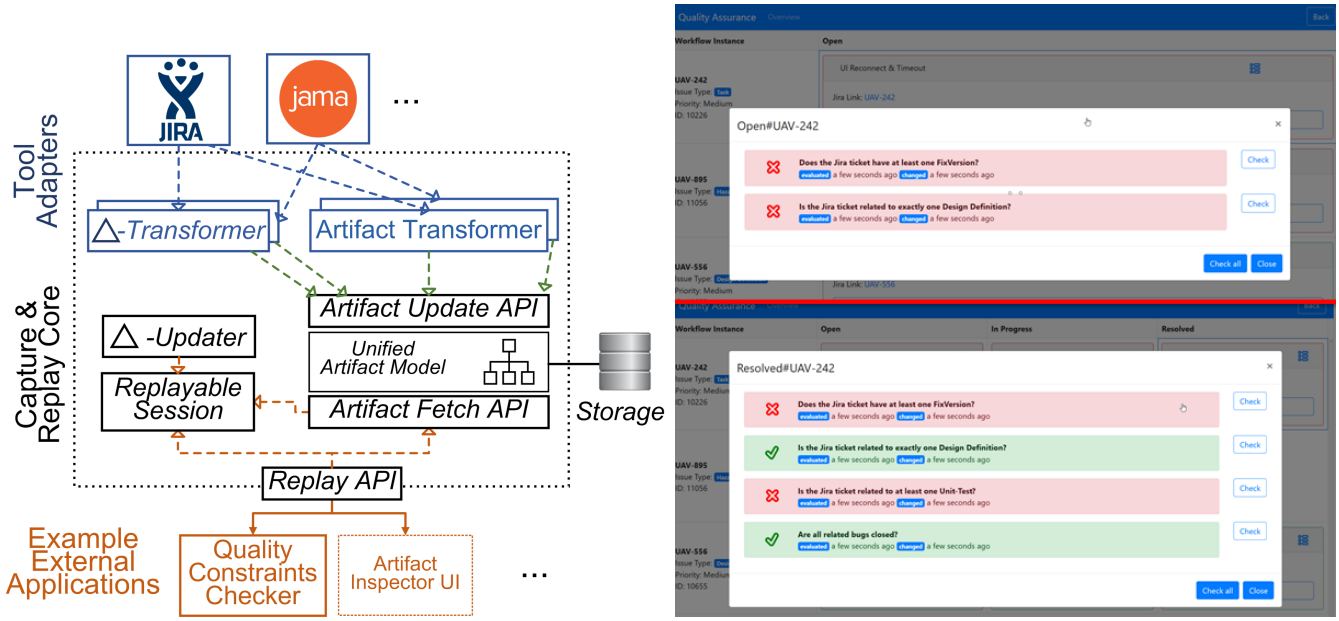
TimeTracer consists of artifact generic storage and replaying components, tool specific adapters for importing artifacts, their relations and changes, and provides a *Replay API* to external application.

Capture and Replay – C&R Core

The C&R Core provides a generic datamodel that is used to create a uniform representation of the different artifacts provided by the different tool adapters. In our prototype implementation the data model comprises three main elements: (1) *ReplayableArtifacts* and their properties, (2) *Relations* (e.g., traces), and (3) *ChangeEntries* associated with an artifact. A ReplayableArtifact is a generic representation of any element provided by a tool adapter.[2] For example, an issue entry, or a user story in Jira are fetched by the respective adapter and converted into our internal representation. ReplayableArtifacts can have arbitrary properties attached i.e., fields from the Jira issue (such as type, state, description, etc.). Dependencies as well as links between issues are represented as bidirectionally traversable *Relations*. For example, in Jira a *refines* dependency between a *user story* and a *task* is transformed into the respective relation with the source and target represented by the corresponding ReplayableArtifact. Finally, change/history information is collected and stored as separate *ChangeEntries* containing all information necessary to restore an artifact to its prior state. ChangeEntries describe the creation and deletion of artifacts, changes to their properties and relations. Note that artifacts are not completely deleted from the storage but rather just marked as "deleted at their origin". Otherwise, the replay mechanism would not be able to reconstruct an already deleted artifact. The unified representation allows the replay component to operate without any specific adapter code and without maintaining an active connection to the respective repository or data source. All artifacts are stored in a database and can

[1] https://marketplace.atlassian.com/apps/1213308/replay-for-jira

[2] In this paper, we use the term *Artifact* to refer to the artifact at its origin, and ReplayableArtifact to refer to its representation in the *TimeTracer* tool.



(a) *TimeTracer* architecture

(b) Quality Constraint Checker - Results Interface: replayed (top), current (bottom)

Figure 1: *TimeTracer* architecture and Quality Constraint Checker application

be retrieved when requested via the Replay API (e.g., all artifacts of a specific type for a specific time period)

Tool Adapters

The *Artifact Update API* provides an internal, clean decoupling of the capture and replay mechanism from the various data sources. Each tool adapter has to provide three main capabilities: a) retrieving data from a source and transforming them into ReplayableArtifacts and Relations, b) extracting change and history information and storing them as respective ChangeEntries, and finally c) providing updates should the artifact change at its authoritative origin (not shown in Figure 1(a)). The *Artifact Transformer* collects artifacts from a data source, e.g., all user stories for a specific project from a Jira server, converts fields to artifact properties and provides the C&R Core with the respective ReplayableArtifact and Relation elements. The converter decides whether properties such as Jira item's release field are modeled as a ReplayableArtifact's property or as a separate release ReplayableArtifact with a relationship to the parent ReplayableArtifact item. Secondly, the *Delta-Transformer* analyzes the change history of an artifact, for example, in the case of a Jira issue, all history items attached to an user story and creates a set of corresponding ChangeEntries which includes change timestamps, change author, changed elements, and fields etc. Additionally, the Delta-Transformer has to handle references to any artifact deleted prior to being tracked by *TimeTracer* . It might decide to drop any historical changes referencing a deleted artifact, (then *TimeTracer* remains completely unaware of the previous existence of that artifact) or creates a skeleton ReplayableArtifact without any properties beside its id. Note that in the latter case, the exact time of deletion cannot be established but rather only the last confirmed existence as indicated by a change entry. Finally, the tool adapter's update mechanism has to keep track of the original artifact (e.g., via periodically polling the Jira server for changed elements) and update the ReplayableArtifact in case a change is observed. The tool adapters only require read-only access to data sources as *TimeTracer* does not change any artifact at its origin and leaves real data in production environments untouched.

Extensibility - Build your own Tool Connector

Most software development tools such as Jira, Jama, or Github provide a REST API and even come with corresponding Java clients. Implementing additional connectors is thus a matter of implementing (a) the conversion of domain specific artifact representations into our generic artifact representation, (b) the conversion of artifact changes into our generic *ChangeEntries* format, and, where necessary, (c) the polling mechanism for continuously retrieving recent artifact updates. As described above, the tool connectors are strictly reading artifacts thus no logic for updating artifacts is foreseen nor required by our tool. *TimeTracer* is also suitable to include artifacts managed by external tools that provide no change history under the condition that the connector tracks artifacts already from the time of their creation and polls for updates sufficiently often (where "sufficiently often" depends on the expected artifact update frequency and the extent to which missing potentially intermediate artifact versions is acceptable).

3.1 Implementation Details and Usage

Our *TimeTracer* prototype is fully implemented in Java and the source code is available on Github [3] and the demonstration video on Youtube.[4] For storing ReplayableArtifacts and ChangeEntries we employ a Neo4J graph database. We have further implemented prototypes of an example connector for Jira. This connector is largely

[3] https://github.com/jku-isse/icse2020-demo-timetracer
[4] https://youtu.be/Av7NJqGQDuA

generic and can handle arbitrary types of artifacts (such as agile user stories, requirements, and custom types) and can easily be configured to collect issues and changes. The key element of the Replay API is a *ReplayableSession*. The user obtains a ReplayableSession from the C&R Core and can additionally specify the scope of relevant artifacts to be returned. This could, for example, comprise all artifacts or rather (for performance reasons) a set of one or more artifacts including a traversal depth. The traversal depth determines how far (i.e., how many hops) the session will continue along (past and present) relations to select artifacts for replaying. For example, a depth of 1 will select the initial artifacts and all of their directly linked artifacts.

All the replay actions affect only the artifacts in the session, thus avoiding the need to conduct replay actions for all stored artifacts. *Replay time* denotes the point in time (in the past) to which all artifacts in the session have been transformed to (i.e., the artifacts are in the state they were in at that particular time). *TimeTracer* provides following key functionalities: (1) going *forward* and applying the the next change from all changes across all artifacts in the session; (2) *forward to a specific timestamp* like *forward* but applying changes in temporal (oldest first) order that have a timestamp earlier or equal to the provided timestamp; (3) going *backward* from all changes across artifacts in the session and the current replay time, applies the next change of the session's past; (4) *backward to a specific timestamp* like backward, but applying changes in temporal order (newest first) that have a timestamp later than the provided timestamp; (5) *fastforward(artifactIds)*: repeatedly calls forward as long as neither of the provided artifacts are affected, i.e., stops after having applied the first change to one of these artifacts; (6) *fastreverse(artifactIds)*: repeatedly calls backward as long as neither of the provided artifacts are affected, i.e., stops after having applied the first change to one of these artifacts; (7) *latest/oldest*: reverts all artifacts to their most up to date version, respectively, earliest known version.

4 PRELIMINARY PROTOTYPE EVALUATION

We conducted a preliminary evaluation by integrating *TimeTracer* with a demo application that provides a development-process aware **q**uality **c**onstraint **c**hecker (QCC) in order to demonstrate the feasibility of the second scenario, and utilized data available from the Dronology dataset [2]. The Dronology project uses Jira to manage Hazards, Requirements, Design Definitions, Tasks, etc. (including trace links among the different artifacts) to manage the development of a framework for controlling and coordinating UAVs. QCC allows engineers to define and check constraints on traces and artifacts that need to hold true in the various issue states ("open", "in progress", ...). It integrates the Drools rule engine [5] to check any constraints encoded as Drools rules. We provide a simple command line interface (as shown in the accompanying video) to define the point in time to revert an Jira issue (and linked artifacts) and/or to step through the various changes of one or more issues. Replaying the issue from beginning to end facilitates checks for each issue state by repeatedly triggering consistency checks and analyzing if constraints are violated (cf. screenshot of QCC User Interface in Figure 1(b)). In the QCC, while replaying only some artifact would be possible, we replay the entire set of stored artifacts to demonstrate the performance of the C&R Core. Replaying approx. 10.000 changes for around 1.200 issues only takes a second. This demo application thus provides initial evidence that *TimeTracer* is sufficiently quick for this purpose.

Future Evaluation Plan: We plan on performing a thorough qualitative evaluation of *TimeTracer* through a case study with our industry partner in the domain of safety-critical systems to understand how their quality assurance engineers can use replaying of artifacts from Jama and Jira for inspecting the development process. We expect to learn to what extent the existing replaying capabilities are sufficient and which other features might be necessary in practice to support the engineers' tasks. We will also use the involved real industry artifacts, traces, and their changes to quantitatively evaluate performance. In parallel, we plan to utilize *TimeTracer* during student software engineering projects where students need to follow a prescribed process. With and without *TimeTracer*, students write quality constraints and document the extent to which they followed the process. Here, we will measure (and compare) the students' effort to write quality constraints and obtain an accurate assessment.

5 CONCLUSIONS

TimeTracer enables the replaying of artifacts and their respective relations (i.e., trace links) to other artifacts at different points in time, based on their change history. The underlying generic data model allows to integrate various tools and facilitates tool-agnostic replaying. *TimeTracer* supports, for example, engineers in understanding how a development process evolved, or in testing quality constraints, as well as software engineering researchers for validating recommendation tools. Future work includes in-depth evaluation and increasing the number of tool connectors.

REFERENCES

[1] Atlassian Jira Project Management. 2019. [last-accessed 01-12-2019]. https://www.atlassian.com/software/jira.

[2] Jane Cleland-Huang, Michael Vierhauser, and Sean Bayley. 2018. Dronology: an incubator for cyber-physical systems research. In *Proc. of the 40th Int'l Conf. on Software Engineering: New Ideas and Emerging Results*. 109–112.

[3] Davor Cubranic, Gail C. Murphy, Janice Singer, and Kellogg S. Booth. 2005. Hipikat: A Project Memory for Software Development. *IEEE Trans. Software Eng.* 31, 6 (2005), 446–465.

[4] Andreas Demuth, Markus Riedl-Ehrenleitner, Alexander Nöhrer, Peter Hehenberger, Klaus Zeman, and Alexander Egyed. 2015. DesignSpace: an infrastructure for multi-user/multi-tool engineering. In *Proc. of the 30th ACM Symp on Applied Computing*, ACM, 1486–1491.

[5] Drools Business Rules Management System . 2019. [last-accessed 01-12-2019]. https://www.drools.org.

[6] Davide Falessi, Justin Roll, Jin LC Guo, and Jane Cleland-Huang. 2018. Leveraging Historical Associations between Requirements and Source Code to Identify Impacted Classes. *IEEE Trans. Software Eng.* (2018).

[7] Mayara Figueiredo and Cleidson R. B. de Souza. 2012. Wolf: Supporting impact analysis activities in distributed software development. In *Proc. of the 5th Int'l WS on Co-operative and Human Aspects of Software Engineering*. 40–46.

[8] Jama Application Lifecycle Management Tool. 2019. [last-accessed 01-12-2019]. https://www.jamasoftware.com.

[9] Michael Rath, Jacob Rendall, Jin LC Guo, Jane Cleland-Huang, and Patrick Mäder. 2018. Traceability in the wild: automatically augmenting incomplete trace links. In *Proc. of the 40th IEEE/ACM Int'l Conf. on Software Engineering*. IEEE, 834–845.

[10] Hang Ruan, Bihuan Chen, Xin Peng, and Wenyun Zhao. 2019. DeepLink: Recovering issue-commit links based on deep learning. *Journal of Systems and Software* 158 (2019), 110406.