Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Actionable light-weight process guidance☆

Christoph Mayr-Dorn [a,*], Cosmina-Cristina Ratiu [a], Luciano Marchezan de Paula [a], Felix Keplinger [a], Alexander Egyed [a], Gala Walden [b]

[a] Institute for Software Systems Engineering, Johannes Kepler University, Linz, Austria
[b] Robert Bosch AG, Vienna, Austria

ARTICLE INFO

Dataset link: https://figshare.com/s/3cc1026d1abf5b1849e4

Keywords:
Process
Constraints
Quality assurance
Traceability
Engineering guidance
Repairs

ABSTRACT

Software engineering organizations in safety-critical domains require rigorous processes that include explicit software quality assurance measures (QA) to achieve high-quality and safe engineering artifacts. One major challenge for engineers is adhering to the correct process that is applicable in their specific working context, to understand which steps are ready to start, what actions are missing to complete their step, and when rework has happened. In this paper, we propose and evaluate ProGuide, a framework that provides actionable, light-weight process guidance by continuously assessing pre-conditions, post-conditions, and QA constraints. In case of a violation, it provides concrete repair actions. Evaluation on a safety-critical open source system and engineers from our industry partner Bosch showed that repairs are complete and small in number, and resulted in less frustration and fewer mistakes compared to being provided with no process guidance.

Editor's note: Open Science material was validated by the Journal of Systems and Software Open Science Board.

1. Introduction

Software engineering organizations in safety-critical domains require rigorous processes for their engineers to follow during software and system development to realize the benefits, such as high-quality engineering artifacts (Damian and Chisan, 2006). To this end, processes include explicit software quality assurance measures (QA) such as the creation of traces between artifacts (Kramer et al., 2014; Rempel et al., 2014), conducting artifact reviews, test planning, change management, and many more. Regulations, standards, and guidelines specify the required final traceability paths as evidence for system safety, for example, the U.S. Food and Drug Administration (FDA) principles in the medical domain (Food and Administration, 2022; McHugh et al., 2014), DO-178C/ED-12C for airborne systems (Brosgol and Comar, 2010), and Automotive SPICE (ASPICE) (Macher et al., 2017) or ISO 26262 (Henriksson et al., 2018; Oliveira et al., 2017) in the automotive industry. It is up to the organizations, however, to define suitable processes and operationalize them.

One challenge for engineers in these settings is adhering to the process (Mayr-Dorn et al., 2021), as there is rarely a one-size-fits-all process within an organization. Processes evolve and improve continuously, hence multiple versions of the same process will co-exist within an organization. Different customers demand following slightly different process variants (and traceability paths). Individual departments or teams responsible for different subsystems might have to adhere to different regulations.

In semi-structured interviews, engineers at our industry partner stated that they find it hard to identify which precise process is relevant to their current work context. Understanding the process definitions and their implications is a daunting task, especially for newcomers. When engineers confirm with colleagues what quality assurance activities they need to conduct—and when—to fulfill their process, they need to avoid accidentally obtaining inapplicable feedback from someone who is familiar with a different process variant. Engineers in our interviews expressed their desire to have immediate feedback on their specific process instance to understand which steps are ready to start, what actions are missing to complete their step, and when rework has happened. In industry, however, process definitions are typically not described in a machine executable format (Diebold and Scherr, 2017). As a consequence, there is little to no automation support for the developer to check whether processes are followed, the extent to which deviations occur during development, or what needs to be done (next) to adhere to the process.

In this paper, we propose and evaluate ProGuide, a novel framework to provide actionable, lightweight process guidance. Specifically, we use the Object Constraint Language (OCL)—widely used in different

☆ Editor: Dr. Christoph Treude.
* Corresponding author.
E-mail addresses: christoph.mayr-dorn@jku.at (C. Mayr-Dorn), cosmina-cristina.ratiu@jku.at (C.-C. Ratiu), luciano.marchezan_de_paula@jku.at (L. Marchezan de Paula), alexander.egyed@jku.at (A. Egyed), gala.walden@at.bosch.com (G. Walden).

https://doi.org/10.1016/j.jss.2024.112064
Received 27 September 2023; Received in revised form 12 February 2024; Accepted 12 April 2024
Available online 16 April 2024

domains (Ali et al., 2014)—to specify for each step in a process its pre-conditions, post-conditions, QA constraints, and output over the engineering artifacts such as requirements, change requests, reviews, test plans, and many more. These rules are incrementally reevaluated whenever the artifacts change to obtain an evaluation tree that always identifies all causes of a rule violation. We transform the evaluation tree into a repair tree, filter it, and then obtain a set of actionable repair options that identify all possible ways for the engineer to fulfill the process definition. This allows the engineer to understand not only whether a step is ready to be started or whether it is complete, but–more importantly–also **how** a process deviation can be fixed.

ProGuide is different from related work on traceability (e.g., Rempel et al. (2014) and Cleland-Huang et al. (2014)) which focuses primarily on the final audited traceability evidence (Watkins and Neal, 1994) but not on the process. Similarly, work on constraint checking (Egyed et al., 2018; Klare, 2018) primarily focuses on consistency among diverse engineering artifacts but not on process guidance. Compared to early work on process-centric environments (PCEs) such as PRIME (Pohl et al., 1999), our approach not only requires minimal integration efforts (as the actual engineering tools need not be adapted to provide feedback) but also provides concrete actions when engineers deviated from the intended process.

The salient contributions of this paper are:

(1) a constraint-based framework for providing actionable guidance.
(2) a prototype for evaluating process guidance.
(3) insights from industry on process repair suggestions.

We evaluated our approach with respect to two research questions: RQ1: How many repairs are generated, and how complete are they? and RQ2: How useful are the repairs and how much overhead does ProGuide cause?

To this end, we replayed over 400 lightweight process instances from a safety-critical open source system and conducted a small experiment with engineers from our industry partner in the automotive domain.

The results showed that ProGuide did not omit any repair options engineers have used. Repairs are typically small in number (hence won't overwhelm an engineer), and are considered useful and applicable but could contain more details. Finally, for fixing non-trivial constraints, ProGuide resulted in less frustration and fewer mistakes compared to no guidance support.

In summary, this work contributes to supporting engineers as it reduces perceived frustration when trying to understand which activities are currently applicable. Hence, we further reason that this will reduce the need to ask others for help as work-context-specific guidance is provided automatically. This promises to also reduce the time needed to adapt to new processes as the most recent, applicable constraints are quickly and directly accessible with timely feedback to immediately highlight deviations. Such timely feedback also is expected to reduce the effort of rework as engineers are informed about the need for repairs while still in being the relevant work context rather than receiving feedback late when they have already moved on to other tasks.

The remainder of this paper is structured as follows. We motivate ProGuide with an industry-derived scenario in Section 2 before introducing background aspects in Section 4. We describe our constraint-based process guidance framework in Section 5, followed by an outline of the evaluation method (Section 6). We present evaluation results in Section 7 and subsequently discuss them in Section 8 before concluding the paper with a comparison of related work (Section 3) and an outlook on future work (Section 9).

2. Motivating example

The following scenario is based on discussions with engineers at our industry partner Bosch and involves a diverse set of artifacts.

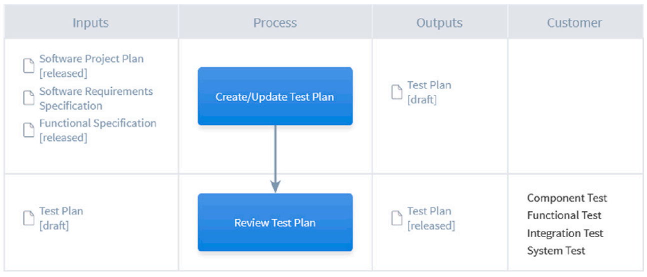


Fig. 1. Verification and validation process excerpt.

Suppose Alice needs to create, respectively update, an existing `Test Plan` to ensure that verification and validation can be carried out properly (see step "Create/Update Test Plan" in Fig. 1). This step typically has the `Software Project plan`, `Software Requirements Specifications` (SRSs), and `Functional Specifications` (FSs) as input. SRSs and FSs need to be in state released and need to have been reviewed to signal to Alice that these artifacts are finalized and of high quality.

The challenges here are manifold. First, Alice needs to recall exactly what are the applicable pre-conditions she would need to ensure before she starts her work. Similarly, she needs to understand that for her particular process context she needs to trace the test plan to all covered FSs, and in turn that all FSs needed to be traceable to a test plan.

Second, Alice might need to identify the extent to which her step's pre-conditions are not fulfilled yet to assess the risks of rework should she decide to start early. Realizing that FSs have been reviewed but one is not in the right state yet could present an acceptable risk. Obtaining such information requires time, especially when multiple other engineers are responsible for those artifacts.

Third, a customer demands a change with implications on upstream artifacts such as SRSs and FSs. The change is potentially relevant to the test plan and needs to be communicated to Alice in a timely manner. Understanding the impact of changes requires in-depth knowledge of the process in general, and the artifacts and their dependencies of the underlying process instance in particular.

Engineers at Bosch describe that addressing these challenges is very time-consuming (e.g., using reviews), having to inspect extensive process documentation often at a generic level. Contacting coworkers runs into the same inefficiencies as they need to understand the inquiring engineer's process context. Ultimately, feedback from QA arrives sometimes weeks later, leading to rework.

Instead, imagine a guidance mechanism that reduces these inefficiencies and rework by immediately determining the relevant process and its applicable constraints from the underlying artifacts, automatically evaluates these constraints, and even suggests the repair actions to fulfill them. This is the goal of ProGuide.

3. Related work

Supporting Engineering Processes Several research efforts in the 90's focused on process-centric software development environments (PCSDE) (Barthelmess, 2003; Gruhn, 2002). A number of frameworks (Fernstrom, 1993; Bandinelli et al., 1996; Grundy and Hosking, 1998; Geppert et al., 1998; Pohl et al., 1999) support engineers by determining which steps may be done in the engineer's current work context, automatically executing them where possible. While such research supports detailed guidance on the individual steps, little guidance is available on why steps are not ready or completed yet. Furthermore, deviations from the prescribed process are not well supported.

PRIME (Pohl et al., 1999), for example, depends upon a tight integration with the engineering tools to determine when and which process to invoke. It detects conditions to start/execute a process step, then makes these steps/actions available in the tool (which needs to be extended with addition interaction elements to this end). These steps are typically very fine grained. For example, they describe what element to change to refine a data flow diagram, but do not define what condition the outcome of a particular step should fulfill. In other words, PRIME does not inform what are the conditions and the respective alternative actions to fulfill them so that an engineer enables or completes a particular step. Deviating from a process needs to be explicitly modeled and merely results in process termination. Otherwise, if the user does something else so that some step's conditions no longer hold, no guidance/hint is given on what to do to (re)enable a particular process step. Process Weaver (Fernstrom, 1993) uses petri nets augmented with conditions. Steps/activities become active when preconditions are fulfilled and are considered closed when post-conditions are fulfilled. No feedback is provided what exactly needs to be done to complete a pre or post-condition. No redoing/reopening of completed steps is possible unless explicitly modeled as a loop. SPADE-1 (Bandinelli et al., 1996) also uses petri nets with guards (basically a condition) that determines transitions. Again as above, no guidance is available what concretely needs to be done to fulfill a guard. Neither can steps be re-done beyond what is modeled, hence no deviation is possible. Serendipity (Grundy and Hosking, 1998) focuses on which user/role in a particular step applies which artifacts. Progress is triggered by events which can be filtered (sort of a constraint). i.e., resulting in notifications to users that a step is available. No support is available what exact events are necessary/possible next to complete a step or start at step, nor is there the possibility to support constraint checking upon deviation. EvE (Geppert et al., 1998) follows the Event-Condition-Action paradigm: when a particular event occurs, and a conditions is fulfilled, an action/step is executed. No recommendations which conditions (how to) fulfill or what events to trigger to make progress as intended to be used like traditional workflow engine. Hence also no deviation supported.

Other engineering artifact-centric approaches (LaMarca et al., 1999; Cugola and Ghezzi, 1999; Junkermann et al., 1994) specify which actions and conditions are available and enforce their correct order. They thereby avoid violations, but also severely restrict engineers and hence are not practically relevant.

Other recent work utilizes process description for fine-granular processes for refactoring (Zhao and Osterweil, 2012) and rework (Zhao et al., 2013) support. General purpose engineering process modeling and execution (e.g., Dumas and Pfahl, 2016; Ellner et al., 2010; Alajrami et al., 2016; Winkler et al., 2019; Hachemi and Ahmed-Nacer, 2022) focus on step-centric languages such as SPEM and BPMN, which both imply active execution where engineers cannot deviate from the prescribed process. In general, however, we notice that research has moved away from explicit process support towards agile methods.

Traceability Support The QA constraints in our use cases highlight traceability concerns as a related and relevant sub-aspect of the overall engineering process. Indeed, several papers (Maro et al., 2018, 2022) identified support for the tracing process, i.e., how to establish and check traces, as an open challenge—missing or wrong traces are the result (Mäder et al., 2013). Along these lines, several researchers have proposed techniques for continuously assessing and maintaining software traceability (Cleland-Huang et al., 2014; Rahimi and Cleland-Huang, 2018; Mosquera et al., 2022). Rempel et al. proposed an automated traceability assessment approach for continuously assessing the compliance of traceability to regulations in certified products (Rempel and Mäder, 2016). These automated approaches are orthogonal to our work as they are process-unaware. They provide little guidance for which step in a process a trace link must be available, nor can they distinguish between different tracing regulations in different processes.

Business Process Compliance Checking In the business process management domain, compliance checking approaches determine whether complex sequences of events and/or their timing violate constraints. Ly et al. analyze frameworks for compliance monitoring (Ly et al., 2015) and highlight that the investigated frameworks have little or no inherent support for referencing data beyond the properties available in the respective events (hence no access to the actual artifact details and their traces/relations to other artifacts). Also, recent work such as (Cabanillas et al., 2020) or (Knuplesch et al., 2017), lacks this crucial support for defining constraints on artifact details. Ly et al. also show that hardly any of their analyzed approaches supports proactive violation detection or the ability to continue evaluations after a violation. Also, no repair recommendations can be provided as none of the approaches support root cause analysis in a manner useful for software engineering. A few other works focusing on repair include Kumar et al. (2013) who calculate repair plans for resource allocations but do not address continued reasoning support in the presence of violations. Van Beest et al. (2014) apply AI-centric planning for process repair. Their approach then generates a single repair plan for automatic execution immediately after detecting a deviation. Maggi et al. (2011) detect violations by checking events against a separate local finite-state automaton for each constraint. A violation then results in ignoring, resetting, or disabling that automaton but not in an actual repair.

Repairing Artifact Inconsistencies Consistency checking is applied in different domains to maintain different types of engineering artifacts (Torres et al., 2021; Colantoni et al., 2021). Once the inconsistency is identified, repairs may be generated to fix it. Thus, several approaches that deal with consistency checking, also propose the generation of repairs (Macedo et al., 2017). For instance, Nentwich et al. check distributed software engineering documents encoded in XML (Nentwich et al., 2003a,b). Their approach, however, suggests a complete set of repair alternatives which may include incorrect repairs. Ohrndorf et al. utilize the history of changes to trace unfinished modifications and identify the cause of inconsistencies, subsequently proposing possible repairs based on the history (Ohrndorf et al., 2018, 2021). Barriga et al. prompt users for their preferences regarding the generated repairs (Barriga et al., 2020). They apply a learning algorithm that finds the most relevant sequence of repairs for the inconsistencies detected. The user is required to give feedback each time a repair is proposed to improve the generation for the next repairs. Tröls et al. present a cloud infrastructure allowing a team to check if their models are consistent considering other engineers' changes (Tröls et al., 2022). Their work, however, is limited to inconsistency identification, not proposing repairs for identified inconsistencies. There are also approaches supporting change propagation of repairs, analyzing repair's side effects on the model's consistency (Khelladi et al., 2019; Kretschmer et al., 2021; Marchezan et al., 2022). Puissant et al. apply Prolog' automated planning to solve UML design model inconsistencies by generating resolution plans (Puissant et al., 2015). These approaches primarily focus on design artifacts only, e.g., UML, and additionally assume any property of every artifact can be changed. Additionally, being process-unaware, they cannot provide warnings to engineers in situations where a step preconditions is fulfilled, but an upstream step's post-condition fails as this requires understanding the dependencies between process steps.

Research Gap: Existing process support approaches that allow the engineer to deviate from the process do not offer any guidance on what are the concrete various alternatives (i.e., actions) that would bring the process back into a consistent state. They neither provide such guidance for ongoing work while the process is in a deviating state.

4. Background technologies

The work presented in this paper makes use of an experimental incremental constraint checker by Reder and Egyed (2012) that includes repair generation (Khelladi et al., 2019). The two key reason for choosing this checker is, first, its ability to re-evaluate only those rules that are affected by a change. Second, it identifies all the possible repair alternatives that make a violated constraint consistent again. We briefly describe the involved concepts in this section.

4.1. Incremental constraint checking

The constraints used in our framework are expressed in OCL, a language that is widely used in different domains (Ali et al., 2014). Constraint checking generates a hierarchical *evaluation tree* containing an evaluation node for every rule's OCL (sub)expression. Each node contains its expression's intermediate evaluation result and is annotated with the expression's *expected* and *evaluated* result. Distinguishing between the expected and evaluated result is crucial since a constraint violation is caused only if the two values are different. The expected result is obtained by applying the expression on the given artifacts in the scope of the rule.

Expression results are combined using the OCL logic operators, i.e, *and*, *or*, *not*, among others, to obtain the evaluated result of the evaluation tree. The expected result of the evaluation tree is always *true*. A difference between the expected and evaluated result is captured as a *false* value in the evaluation node. Any expression evaluated to *false* is then considered part of the *cause* of the violation.

Find in Listing 1 an example of how a QA constraint looks like encoded as an OCL rule for Azure DevOps Services. Note that the context (i.e., the starting point, marked by the term "self") of a rule in our approach is always the process step. Inputs are prefixed with "in_", and outputs are prefixed with "out_" to distinguish them from other step properties. Each input and output property is a set, hence, the constraint always needs to iterate over the set's elements, e.g., line 3, where we start checking that for all Test Plans (TPs), there exists at least one link to another workitem of type Functional Specification (SPEC) via a predecessor link.[1]

```
1 self.out_TPs->size() > 0
2 and
3 self.out_TPs->forAll( tp |
4  tp.relatedItems
5  ->select(link | link.linkType.name='Predecessor')
6  ->collect(link2 : <root/types/workitem_link> |
       link2.linkTo)
7  ->select( ref | ref.workItemType.name = 'SPEC')
8  ->asSet()
9  ->size() > 0)
```

Listing 1: Example QA Constraint in OCL for Azure DevOps Services that checks whether there is at least one Test Plan and whether it links to any Functional Specification. The constraint is defined over the following data structure: the step refers to a set of test plans via "out_TPs". A Test plan links to other work items via its property "relatedItems". This is a set of link objects that have a linkType such as Predecessor, Successor, Child, Parent etc. and the property "linkTo" that refers to the work item being linked to. The type of a work item (e.g., Test Plan, Requirement, Specification) is accessible via the "workItemType" property.

Showing the evaluation tree for such a complex rule would be visually overwhelming. Hence, lets analyze a simplified OCL constraint from the motivating scenario "`name: RequirementReady, context: StepA, inv: self.SRS.status = 'released' or self.SRS.status = 'accepted'`". This disjunction requires only one of its two expressions to be *true* for the disjunction to be *true*.

[1] The data schema used by all the listings in this paper is available as part of the online supporting material (Mayr-Dorn, 2023).

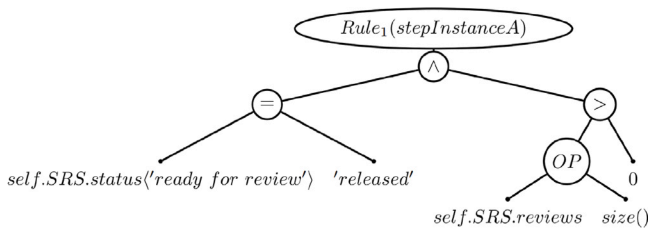


Fig. 2. Evaluation tree example.

Hence, if either `status` is `released` or `status` is `accepted`, the evaluation tree will result in *true*. If neither is the case, thus both expressions evaluate to *false*, they generated a *false* disjunction result, indicating a rule violation. In this case, both expressions are the cause of the violation.

For this rule, the evaluation tree in Fig. 2 takes a fictive stepInstanceA artifact in the scope of the rule (indicated by "self") which has a property "SRS", which points to an SRS artifact with status "ready for review" and has no reviews linked. Bold tree edges (in this example, all of them) indicate a mismatch between expected and evaluated results. Here, on the left self.SRS.status does not equal 'released'; on the right, the operator (OP) size() applied to self.SR.review returns 0, but needs to be larger than 0. Hence the overall rule evaluates to false.

Rule evaluation occurs incrementally, i.e., only those rules that are potentially affected by a change will be selected for re-evaluation. To this end, the constraint checker maintains a mapping from artifact instances to rule evaluation instances. Initially, this map contains only the rule's context instances, e.g., for the rule above only stepInstanceA will map to its rule "RequirementReady" evaluation tree. As the constraint checker navigates across instances and their properties it thereby extends the scope of the evaluation and add all those newly added scope elements (e.g., any SRS linked to the step) to the mapping. Once an update to a requirements is detected, the constraint checker looks up the instance (i.e., the SRS) and its rule evaluation, ultimately triggering re-evaluation.

Temporal Constraints: Invariant constraints such as the examples in this paper are insufficient to express in which order an artifact needs to exhibit a particular property value. To this end, temporal constraints (e.g., using linear temporal logic (LTL) operators) are needed. At the time of conducting the experiments, we did not support such operators but since then have integrated basic LTL operators into the incremental constraint checker (Ratiu et al., 2023). Temporal constraints are not necessary when the process model's control flow constructs are sufficiently expressive, and sometimes they might even be detrimental as they incur unnecessary repair actions.

Take for example the following constraint as a fictive precondition for the step "Requirement Review": a requirement must be linked to a change request before its status can be set to "Ready For Review" (defined as "`name: ReviewPrecondition, context: StepReview, inv: UNTIL(self.SRS.status <> 'Ready For Review', self.SRS.tracesToCR.size() > 0)`" when using the notation from Ratiu et al., 2023). When an engineer happens to set the requirement's status first and only then links to a change request, this would result in a constraint violation even though the essence of the precondition is fulfilled: the link is available and the status is correct. The repair would then cause unnecessary effort for the engineer as they need to set to status to something else and then again to "Ready For Review". Using the invariant (i.e., non-temporal) constraint version as precondition would achieve the same effect without additional work: "`name: ReviewPrecondition, context: StepReview, inv: self.SRS.status = 'Ready For Review' and self.SRS.tracesToCR.size() > 0`".

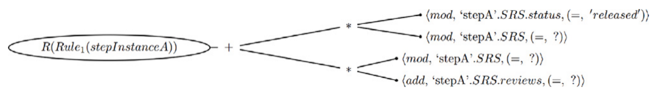

R(Rule1(stepInstanceA))
+
*
⟨mod, 'stepA'.SRS.status, (=, 'released')⟩
⟨mod, 'stepA'.SRS, (=, ?)⟩
*
⟨mod, 'stepA'.SRS, (=, ?)⟩
⟨add, 'stepA'.SRS.reviews, (=, ?)⟩

Fig. 3. Repairtree example: '+' nodes indicate sequential child repair actions; '*' nodes indicate alternative child repair actions.

4.2. *Repair generation*

The repair generation analyzes the evaluation tree to propose possible repairs for expressions that are part of the *cause* of the violation. Continuing with the example constraint above: when both expressions evaluate to *false* thus the overall rule evaluates to *false*. Here, the repair generation will propose repairs for both subexpressions "`self.SRS.status = 'released'`" and "`self.SRS.status = 'accepted'`". Repairs are structured in a tree with three types of nodes, namely, *sequence*, *alternative*, and *repair action* (leaf nodes only). A node's children are either some *repair actions* and/or further subnodes. In our example since repairing either subexpression is sufficient to fulfill the disjunction, the repair tree would contain an *alternative* node. This node type defines that one repair action is sufficient to fix the violation. It is up to the engineer to analyze and select the action to be performed. If we consider another constraint from the motivating scenario above ("`self.SRS.status = 'released' and self.SRS.reviews->size() > 0`"), however, the evaluation tree requires both sub-expressions to be fulfilled. In this case, if both are violated, the repair tree would have a *sequence* repair node. This node type implies that every repair action must be executed. In the example, at least two repair actions are required, one for fixing the `status` and one for fixing the `reviews` property. Notice, however, that is still possible to have *alternative* nodes under the *sequence* nodes, since there may be more than one way of fixing those properties. Repair action nodes describe a change by identifying the artifact, the artifact's property, and the change operator. Operators are either adding or deleting from collection properties, or modifying a single-valued property. Modifications are further described using the constraint operators =, <>, >, or <. Repair actions are by default *abstract*, i.e., the engineer needs to select the property's value. The repair mechanism, however, also generates *concrete* repair actions that suggest the removal of values from collections (identifying the entry), and repair actions that use the = constraint operator (i.e., identifying the value to set a property to).

For the example evaluation tree in Fig. 2, as both subexpression needs to be true, the repair tree (Fig. 3) consists of a sequence node, each offering an alternative between two options. The repair generator starts at the top level with ∨ and subsequently generates a sequence node ('+'). It then navigates each subtree independently: along the evaluation tree's left-hand side (or repair tree top), it suggests to set (i.e., *mod*) the value of SRS to something different, or to set (i.e., *mod*) the value of status to "released". Along the right-hand side of the evaluation tree (resulting in the repair tree bottom), the repair generator again suggests replacing (i.e., *mod*) the SRS value (i.e., referencing a different SRS), or to (i.e., *add*) something to the reviews property.

5. A constraint-based process guidance framework

5.1. *Approach*

We first outline the general flow through ProGuide (cf. Fig. 4) before describing the key aspects in more detail in separate subsections below.

Upon startup *Process Definitions* are loaded (A), the contained OCL rules are augmented (c.f. Section 5.4 below) and then stored in the *Artifact Store*. Whenever an engineer instantiates a process via the *Process Dashboard* (B), the *Process Engine* will first fetch the initial artifacts from the *Tool connectors* (C) and store the process instance in the *Artifact Store*. Any subsequent changes to the engineering artifacts are automatically pushed into the *Artifact Store* (D). Engineering artifacts may range from requirements, change requests, bug reports, reviews, code, test plans, to test results. The *Constraint Checker* obtains all artifact changes, including any process definitions and process instances, and determines which OCL rules to instantiate and further evaluate. The evaluation results are also stored in the *Artifact Store*, and changes thereof (i.e., when an OCL rule is no longer violated or newly violated) are used as a trigger for the *Process Engine* (F) to update the process instance (e.g., mark a step as complete, cf. Section 5.5). This includes ensuring that a step's output remains accurate (G) with the help of the *Repair Generator*. Updating the process instance (H) will, in turn, trigger rule re-evaluations (E), potentially leading to several iterations of rule result updates and process updates (E, G, F, H). Eventually, the updated process status is forwarded to the *Process Dashboard* for displaying. When an engineer inspects the process instance, the *Process Dashboard* obtains and filters any available repair options (J), cf. Section 5.6 below. The engineer can then decide when and which repair to execute by following the repair option's link into its respective tool (K). Any changes there are then immediately pushed back into ProGuide and result in reevaluation. Hence the engineer obtains rapid feedback on whether they have indeed successfully fixed a constraint.

From the perspective of the engineers, the whole procedure is highly automated. Once a process is manually instantiated by providing a reference to the entry artifacts (e.g., a change request), all updates from tools, rule re-evaluation, and repair generation happens without human interaction. Repair execution remains fully within the control of the engineers on purpose. Our approach assumes that a process definition is available (typically provided by a QA engineer or process engineer to be used by several projects or products within an organization) and that relevant tool connectors are configured (typically managed by an organization's IT department upon introducing a new tool).

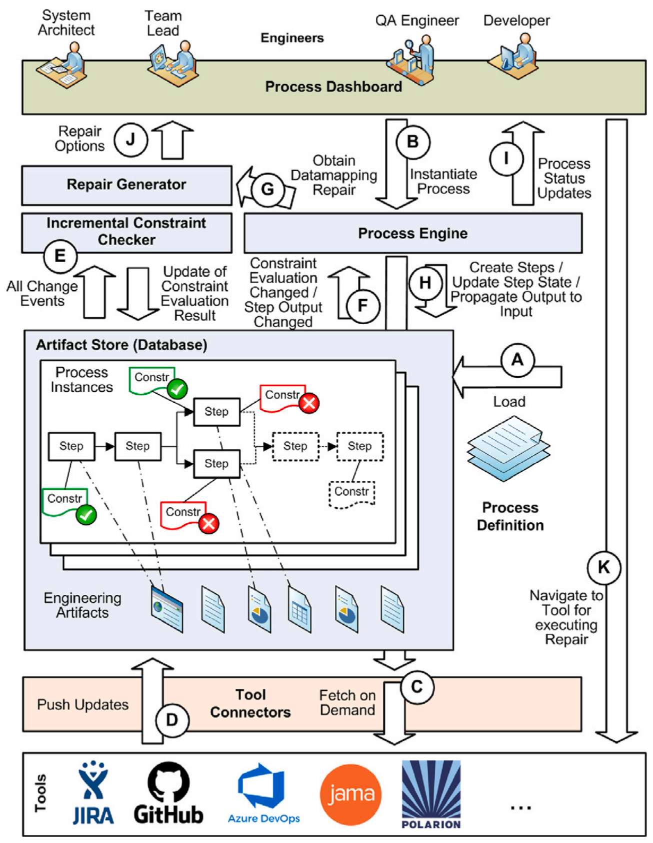


Fig. 4. ProGuide overview.

5.2. Process definition

A process definition describes the sequence, parallelism (AND), and alternatives (OR, XOR) of steps and the steps' artifact *inputs*, and the steps' artifact *outputs*. These are standard elements in most process models and software development companies typically make such information available to engineers in the form of high-level process documentation (Diebold and Scherr, 2017), e.g., via PDFs, slides, or tools such as Stages.

For our purpose, a process engineer further expresses in OCL (i) the pre-conditions over the steps' input, (ii) the step's post-conditions over the step's output, and optionally (iii) QA constraints over the step's output. *Pre-conditions* specify when a step is ready to start (e.g., the step "Create/Update Test Plan" may start when all the SRSs' state is "released"), while *Post-conditions* specify how the Process Engine determines that an engineer signals the step's completion (e.g., setting a `Test Plan` artifact to "ready for review"). *QA constraints* separately specify conditions that additionally need to hold upon a step's completion to ensure or document high software quality (e.g., a `Test Plan` must trace to an SRS).

In contemporary engineering environments, however, engineers create only tacit relations between engineering artifacts and the process (i.e., which artifact is needed for which step, which artifact is created or changed upon finishing a step). These tacit relations need to be made explicit to enable checking of pre-, post-, and QA constraints. In the case of input, the process definition's data flow allows to map the artifacts from the output of one step to the input of the subsequent step(s). For determining a step's output, however, there is a need for so-called data mappings. *Datamappings* define the navigation path starting at a step input artifact to locate the step's output. For example, the step "Create/Update Test Plan" has all `Test plans` as output that are linked from its input `Change Request` artifact (c.f. Listing 2).

Process Definition Meta Model

A process definition exists in the framework in two different forms: first, a generic process definition that describes the above listed elements (see also SOM Fig. 1 top). At this level, the OCL rules exist as mere strings. This generic process definition supports simple navigating across the process steps that make up a directed, acyclical graph. This generic process definition is then transformed into a dynamically created meta model that describes the process instance (c.f., SOM Fig. 1 bottom) - the process instance meta model. For each process definition there exists a distinct process meta model where step input and step output become first class fields (rather than just a map that associates field names to their type). For example, a generic process step named "Refine" in Process "SRSRefinement" with an input of name "requirements" is transformed to a unique step instance type called "Refine_SRSRefinement" that has a property "in_requirements". Any outputs become properties prefixed with "out_". SOM Section II.D provides a detailed discussion on the relation between definition data model and process instance meta model. The rationale behind a distinct process instance meta model for each process definition is grounded in simpler constraint writing as all the constraints have a process step as their context (i.e., the "self" will refer to the process step) and a rule can thus straightforward navigate to any inputs and outputs of that step. Section 5.2.3 discusses further process and rule engineering aspects.

5.2.1. Process model notation

Our deliberate choice was to avoid imposing a specific visual notation, leading us to disregard the use of existing languages like BPMN (Dumas and Pfahl, 2016), for three main reasons. First, our objective was to streamline the data model to its essential elements, eliminating any unnecessary complexity. For instance, BPMN introduces a multitude of elements beyond what our framework supports, which could complicate process design for engineers, requiring them to remember which elements are applicable and which ones have no impact.

Second, key aspects of our process model such as the DataMapping (from step input to output), Preconditions, Postconditions, Cancelation conditions, and QA constraints have no natural visual counterpart in BPMN and hence would need custom integration in any BPMN editor. There is no natural element for a step's post condition as in BPMN the assumption is that a step is complete once the user explicitly marks the step as done. In our process model, completion is determined by the post condition over an artifact's properties. The core purpose of a post condition is to relieve the engineer of having to interact with a process engine and is enabled to merely work on the underlying engineering artifacts.

Third, we expect that modeling in an existing language that comes with different execution expectations might lead to process engineers erroneously expect similar behavior enforcement from our framework. Foremost, languages such as BPMN imply that once a step is done, the output wont change unless there is an explicit loop, but once the loop is exited, there will not be any further changes possible. Also, the core assumption is that a task cannot be work on, and non of any data can be changed within that task unless all required prior steps are complete. In stark contrast, our process model describes that the modeled sequence is the ideal flow but must deal with the possibility that any artifact at any time can change, triggering a reevaluation of affected step pre- and postconditions, with subsequent step status updates.

Hence, we chose to define a process data model that supports only the essential process properties instead of reusing a standardized process notation. The canonical format for persisting and transferring a process definition is JSON. Processes definitions in that format are then transformed into the framework internal data model. A mapping from BPMN to our format is feasible but requires extra engineering effort beyond the scope of this paper. We do, however, support two modeling environments. The framework integrated visual, Blockly-based editor as well as an experimental transformation of processes models defined in Methodpark by UL's "Stages" product, a web based process modeling environment. The online supporting material describes the modeling procedure with the integrated editor in more detail.

5.2.2. Tool connectors and artifact meta model

Writing OCL rules for step pre/post conditions and QA constraints assumes availability of a data model that describes the properties and relations of the involved engineering artifacts such as Requirements or Test Cases. This data model is defined by the tool connectors. The tool connectors need to be written only once for each tool and hence can be reused across teams and even organizations. As part of our prototype we provide connectors for Azure DevOps Services, Jira, Jama, and GitHub.

The supporting online material provides the schema definition for Jira, GitHub, and Azure DevOps Services in the form of an XML schema definition (XSD).[2] Note that the schema is purely for informing the process engineer when writing OCL rules. All four supported tools provide their information in a different JSON format in a data structure that makes rule specification very inconvenient. When fetching a jira issue, for example, the JSON response uses numeric field ids for providing field values, e.g., a field named "isReleased" will only be accessible via id "10333" and not by its name. Alternative to inspecting an XSD, our prototype enables the inspection of available artifact meta models via its web interface. The prototype's *Artifact/Instance Inspector* provides insight for every artifact type (as well as process step type) into the available properties, their type (e.g., String, Boolean, a complex Artifact, etc.), and their cardinality (i.e., single, list, set, or map). This inspector also enables inspecting artifact instances and navigating across their links to other artifacts and even details which property is used in the scope of which OCL rule evaluation.

[2] We cannot provide the schema for Jama as we only have access to a Jama server with confidential schema at one of our other industry partners.

Azure DevOps Services, Jira, and Jama are examples of web tools that enable custom artifact types, custom fields, and custom relations while GitHub has a static schema. Subsequently, the precise artifact meta models for Azure DevOps Services, Jira, and Jama depend on the particular tool configuration. This implies that tool connectors cannot assume a predefined set of artifact types and their properties but need to fetch schema information from the tools and use that schema to dynamically create the artifact meta model and also to map the fetched artifact instance information (i.e., JSON) dynamically to artifact properties. Our connectors fetch the tool's schema information upon startup if that schema has not been locally cached from a prior run. We observed that Azure DevOps Services, Jira, and Jama use a different, proprietary schema description format. Our connectors support these formats, hence, process designers may write OCL rules accessing all the information that a tool makes available.

As hinted at above, tool connectors do not simply map the obtained artifact information directly onto the framework's internal data structure: i.e, the artifact meta model over which the OCL rules are defined. As we learned from preparing and running the experiment reported in this paper, the design of the artifact meta model determines to a large extent how complicated rules become and subsequently how understandable guidance actions become. Take for example the modeling of different link types between work items in Azure DevOps Services. For experiment we modeled this as a work item having a `relatedItems` property that contains `Link` artifacts that have a `LinkType` who's name contains the actual human readable name (e.g., "Predecessor" or "Realizes"), and a linkTo property that points to the work item on the opposite end of this link. Listing 1 provide an example OCL rule navigating over such a link. When such a rule is used as a post condition and needs fixing, the guidance alternatives might include the suggestion to change the `linkType` of the work item's `Link`. This might be confusing to engineers when the tool does not offer the ability to change the link type but rather requires to remove a link and create a new one. Hence, in the latest framework version, our tool connector dynamically defines properties for each available link type and thereby enables simpler navigation. In the case of Listing 1, lines 4–6 become a single expression `tp.predecessorItems`. We expect that such internal artifact meta model evolution becomes rare as the framework matures and subsequently the need to co-evolve the OCL constraints with the models (Khelladi et al., 2017) becomes negligible.

5.2.3. Process engineering

The main stakeholders defining processes and their constraints are typically dedicated process engineers and quality assurance engineers (or regular engineers taking on such responsibility).

These engineers typically describe processes and constraints in a non-executable manner, e.g., they document the process in a PDF or use a web-based modeling tool such as Stages to provide a reference to engineers on the necessary steps, the steps' readiness criteria, and steps' definitions of done. Such a description will also contain references to the applicable tools, how to use the tools, which artifact types to use, and often even which artifact fields in those tools engineers need to fill in.

Using the prototype's integrated editor or applying our experimental transformation of process models from stages enable the execution of such processes. The supporting online material contains a short user guide on using the framework and the integrated editor for specifying processes. The process engineer first defines the required steps, their sequence, and how engineering artifacts are used as step inputs and outputs.

Once a step's input (and optional output) is defined, the engineer may continue defining the step's preconditions, postconditions, and if the step may be skipped under certain conditions, the cancellation conditions. As outlined above, the engineer does not need to know the process definition meta model in detail, just what are the step's inputs and outputs when specifying how the OCL rule navigated from "self" (the step) to any input and output artifacts. The remaining process structure is irrelevant as an OCL rule navigating to other steps would interfere with the process engine's behavior (and would introduce a tight coupling between steps that is invisible in the process' control flow and data flow structure). For example, a requirements refinement step having a set of requirements in input "requirements" may come with a precondition that checks if there is at least one requirement provided: `self.in_requirements.size() > 0`.

At the moment, a process engineer needs to have a basic understanding of the OCL syntax and may inspect the provided example process definitions for inspiration for writing pre- and post conditions. The framework's *Artifact/Instance Inspector* supports the engineer in identifying which properties are available for use in an OCL rule. Specifying OCL rules across data models from different tools is possible. For a simple example, a step that takes a Jira issue as input "jira" and an Azure DevOps Services work item as input "workitem" may specify a constraint that checks if the issue's "fix date" is equal to the work item's "release date" in the form of: `self.in_jira.fixDate = self.in_workitem.releaseDate`.

Upon deploying the process, the engineer receives detailed error and warning feedback on rule syntax errors and process modeling errors. The framework's *OCL Playground* enables the engineer to rapidly develop OCL constraints and execute them against any desired artifact including the ability, in the case of a constraint violation, to inspect what guidance actions would be available for fixing.

On how to use preconditions vs post conditions: A step's precondition is intended to ensure the minimally necessary conditions on the step's input. Its purpose is not to repeat all the preceding steps' post conditions as these conditions are checked by the process engine and need to be fulfilled before the step is marked as "save to start". This has the intended side effect to foster re-usability of steps. Take a review step for example. It may only need at least one artifact to be reviewed, resulting in a simple precondition along the lines of `self.in_artifacts.size() > 0`. Any additional conditions on the artifact under review, such as being in a particular state or having a trace to another artifact, are defined as the postconditions and/or QA constraints of a preceding step. Such a review step could then be reused, for example, for requirements reviewing and test case reviewing.

On reusing constraints: Reusing a process with a different tool and hence different data model requires manual effort. Support for reuse has not been our focus yet for two main reasons: on the one hand, rules are very organization specific, often considered key intellectual property, and thus would only be reused at most within an organization. This is a rare scenario as this implies that different tools are used for the same purpose. On the other hand, different tools are likely to exhibit different field names and relationship names, and might even engineers use similarly sounding fields for quite different purposes.

5.3. Process engine

The Process Engine determines which steps of an engineering process have started, which ones are completed, which ones should not be done, and which QA constraints are fulfilled, respectively violated, solely triggered by changes to engineering artifacts (incl. issues/tickets/workitems). Hence, engineers are not restricted in their activities and can deviate from the process as they prefer.

Each change to an artifact used within the scope of a process instance leads to constraints reevaluation and potential subsequent process step status updating. The engine thereby does not just check if a step's pre-, post-, and QA constraints are fulfilled but also traverse the process instance to prior steps (i.e., upstream) to ensure these conditions remain fulfilled upstream as well.

Steps are `Ready` to be worked on when pre-conditions are fulfilled and all prior steps are `Completed`. In the case of deviations, a step can be `Prematurely started` or `Prematurely in progress` when the engine detects step output (via the datamapping) and any of the prior steps is not completed (or no longer completed). For example, an engineer decides to start reviewing a Test Plan although the Test Plan's state is not "ready for review". This enables feedback to the engineer that the prior step may yet cause changes to its output (i.e., the current step's input) and hence rework might be likely. Likewise, a step can be `Unsafe started` or `Unsafe in progress` when the engine detects that the QA constraints of a prior step are not fulfilled. In our example, suppose a Test Plan is in state "ready for review" but does not trace to an SRS. This conveys the information to the reviewing engineer that the Test Plan is considered correct but the lack of a trace to the corresponding SRS might signal that the Test Plan does not absolutely cover the intended scope and therefore should be reviewed with extra scrutiny. Note that a step can be `Unsafe started/in progress` and `Prematurely started/in progress` at the same time.

5.4. Constraint analysis and rewriting

Before a process definition is deployed and available for instantiation (A), however, constraints analysis and rewriting is necessary as directly using the post-conditions and QA constraints for producing repair actions has drawbacks. Taking the motivating example step to "Create/Update Test Plan" that has at least one Test Plan as output. Recall that the QA constraint demands that the test plan needs to trace to all covered Functional Specifications as specified in Listing 1. The problem is, that as long as an engineer has not produced any artifacts that are detected as output, ProGuide would not be able to provide actionable guidance on how to fulfill post-conditions or QA constraints as there is no output available to repair. In our example, the QA constraint would fail if there is no test plan but it could not give meaningful guidance on how to mitigate this violation as merely recommending to add some test plan as output is not very actionable.

Hence, the *Process Engine* analyses the rules and automatically rewrites them by replacing every occurrence of an artifact output in a rule with its corresponding datamapping rule.[3] Listing 2 provides the datamapping rule for our example, describing that test cases are considered output if they can be reached from the steps input change request via a successor link.[4] The resulting augmented QA constraint for our example is depicted in Listing 3. The data schema underlying these listings is available as part of the online supporting material (Mayr-Dorn, 2023).

Subsequently, the repair generator is also able to suggest all the properties on the navigation path from the input artifact(s) to the output artifact(s) as potential repair locations, e.g., in our example to add a `Test Plan` artifact to the `successor` list of the step's change request so that the Test Plan will be considered as the step's output. The beneficial side effect of augmenting constraints is that engineers automatically receive also guidance on what link types to use to establish traceability among artifacts.

```
self.in_CRs->any().relatedItems
 ->select(link | link.linkType.name='Successor')
 ->collect(link2 : <root/types/workitem_link> |
     link2.linkTo)
 ->select( ref | ref.workItemType.name = 'TP')
 ->asSet()
 ->symmetricDifference(self.out_TPs) ->size() = 0
```

Listing 2: Example DataMapping in OCL for Azure DevOps Services that identifies all workitems of type test plan reachable from the input change request via a successor link as the steps's output.

[3] When post-conditions and QA constraints are defined over the step's input, then no rewriting needs to occur for these rules.

[4] Specifically, the rule ensures that the set of Test plans in the output is always identical to the set of test plans reachable from the change request.

```
self.in_CRs->any().relatedItems
 ->select(link | link.linkType.name='Successor')
 ->collect(link2 : <root/types/workitem_link> |
     link2.linkTo)
 ->select( ref | ref.workItemType.name = 'TP')
->size() > 0
and self.in_CRs - >[...]
->forAll( tp |
 tp.relatedItems
 ->select(link3 | link3.linkType.name='Predecessor')
 ->collect(link4 : <root/types/workitem_link> |
     link4.linkTo)
 ->select( ref2 | ref2.workItemType.name = 'SPEC')
 ->asSet()
 ->size() > 0)
```

Listing 3: Augmented QA constraint having replaced the step's output with the output's data mapping.

The same mechanism enables detection of prematurely started steps. Suppose the steps for creating requirements and functional specifications are not complete yet. By default, this will not result in activation of the "Create/Refine Test Plan" step as per process definition the necessary inputs are not available in the correct state (i.e., "released"). Subsequently, the process engine would not detect an engineer prematurely starting on test plan creation, and hence, neither be able to warn that engineer that rework might be necessary, nor to conduct QA constraint checks on any prematurely created test plans.

To this end, ProGuide additionally creates premature step detection rules by taking any QA constraint and post-condition for a step and replacing the step's output parameter (i.e., self.out_XXX) with the datamapping constraint as described above. The augmentation procedure continues with replacing the step's input parameter (i.e., self.in_YYY) in the datamapping constraint with any output of the step's upstream steps that are used as input as defined by the process' dataflow. Recursively navigating the steps upstream eventually reaches the process's input, thereby creating a premature step detection rule that directly depends only on the process's input and no longer on the step's input.

5.5. Event driven process updating

The *Process Engine* reacts to two main events (i) rule evaluation changes, and (ii) step output changes. In the former case, it checks whether pre-/post-condition or QA constraint changes should lead to a step state change and also whether new downstream steps should be instantiated. When a datamapping rule is no longer fulfilled (e.g., a `Change Request` now links to an SRS that is not listed as the output yet), the *Process Engine* asks the *Repair Generator* to find and execute a concrete repair (which ultimately will lead to an output change). Those output change events, in turn, trigger data propagation to any subsequent steps. This ensures that even when engineers conduct rework on an upstream step previously considered done, any output changes are immediately propagated downstream to any other step having this artifact as input. In turn, this propagation may result in a cascade of downstream reevaluation of pre-/post-conditions, and QA constraints. As all OCL rules in a process definition have only step input and output in their scope, we can guarantee that no infinite loop (E, G, F, H) occurs.

5.6. Repair filtering and translation

The repair generator produces repair options based on the underlying artifact data model in the *Artifact Store* (which in turn is defined by the tool connectors based on a tool's API). The repair generator (Kheladi et al., 2019) we reuse assumes any property can be changed and any property is a potentially useful change. This assumption does not hold for processes. Without filtering, the repair alternatives for a violated constraints repairs will include the deletion of the process step, directly adding to or removing from step input or output, and changing type information. For the latter case consider, for example, a `Change Request` that traces to a SRS via an "affects" relation. Tools like Jira typically mode this as an issue with a property named `outgoing links` that have a set of `links`, where one link entry is of `type` "affects" and has a reference to an issue of `type` SRS. The `types` here are not primitive strings but rather complex artifacts with one property containing their human-readable name. A rule that mandates the existence of such a trace potentially would result in repair options that suggest changing the human-readable property of the `type` artifact. This property typically cannot be changed and also should not be changed as all links referring to this type would then be impacted as well. We address this challenge by removing all repair options from the repair tree that suggest changing artifacts which do not have a human-accessible endpoint in their tool. The remaining repair options are translated into human-readable text via a simple template mechanism.

5.7. Differences to ProCon

In our previous work (Mayr-Dorn et al., 2021) we introduced ProCon, a framework for process-centric quality assurance support that defined basic concepts the work in this paper builds upon. ProCon specifically focused on supporting engineers in their work by providing frequent feedback to what extent quality assurance checks are violated. To this end, a first version of a process engine was devised. We therefore refrained from repeating the basic concepts and data models in this paper (which are available as SOM) and focus on the novel parts of rule augmentation, analysis, and repair generation.

The initial version of the Process Engine required significant manual rule specification efforts. The quality assurance checks, and process step pre- and post-conditions had to be written in a java dialect for the Drools rule engine. Doing so, QA engineers had to explicitly encode for which QA violations which artifacts were most likely the cause. Not only do such rules miss edge cases as the QA engineer has to be aware of them explicitly, but also, there was no hint given on how to repair a constraint violation.

A further shortcoming was the focus on QA checks, while pre and post conditions were only used to decide whether QA checks should be carried out or not. Engineers just received information on whether steps are ready or complete but neither any information on what the actual pre- or post-conditions were, nor how to repair them if they were not fulfilled. Hence, also as long as no output was detected, also the QA constraints were not evaluated and no explanation given. In contrast, this work explicitly focuses on supporting process step pre- and post conditions as a means to assist engineers in understanding when they can start with work, respectively are done.

In the same prior work, process designers also had to encode not just how a step's output is obtained from a step's input (i.e., the *Datamapping*), but also write code that ensured that any change to the input was reflected in the output. This is error-prone and may lead to a mismatch between the constraint states—as displayed by the *Process Dashboard*—and the actual process state—as visible in the engineering tools. With ProGuide, these datamappings are expressed as OCL rules that are automatically kept consistent. Additionally, we now provide constraint rewriting and augmentation as outlined in sub Section 5.4. To this end, the prior Process Engine version required process designers to manually encode the conditions of a prematurely started step, did not distinguish between premature and unsafe step starts, and did not establish for a step whether any upstream changes affected the pre-/post-conditions or QA constraints of prior steps.

Another major difference to our prior work is that it has not been evaluated with actual engineers. In this paper, a major aspect is the evaluation the usefulness of actionable guidance with our industry partner.

5.8. Prototype availability

We host a publicly accessible prototype of ProGuide at http://140.78.115.5:7171/home[5] which is able to instantiate a demo process. Note that this public version showcases recent improvements that were not available at the time of the experiment. Fig. 5, however, shows a screenshot of how our experiment participants received guidance during the experiment. The supporting online material provides further details of how to use the online prototype and also provides access to an executable jar for self-hosting the ProGuide prototype with connectors integrated to connect to arbitrary Jira projects (hosted on the Atlassian cloud) and projects on Azure DevOps Services.

6. Evaluation approach

We evaluate ProGuide along the following two main research questions (RQs).

RQ1: ***How many repairs are generated, and how complete are they?*** Correctness (generating useful repairs without overwhelming the engineers) and coverage (recommending all plausible repairs) are two key recommendation metrics (Avazpour et al., 2014). Specifically, we aim to avoid overloading the engineer with too many repairs while supporting the engineers also in edge cases. We evaluate this RQ primarily via a replay of artifact changes from the Dronology open source system.

RQ2: ***How useful are the repairs and how much overhead does ProGuide cause?*** Overly unsuitable, confusing repairs, or significant mental overhead will result in engineers rejecting the guidance. Murphy-Hill and Murphy highlight the importance of understandability for recommendation delivery (Murphy-Hill and Murphy, 2014). We evaluate this RQ in the scope of a controlled experiment with engineers from our industry partner.

The following two sections describe the mixed methods evaluation design of the two RQs with an more detailed description including data and prototype availability as supporting online material (SOM) (Mayr-Dorn, 2023). We chose a combination of case study and human experimentation to obtain quantitative insights from real processes in the past and qualitative insights from actual user observations.

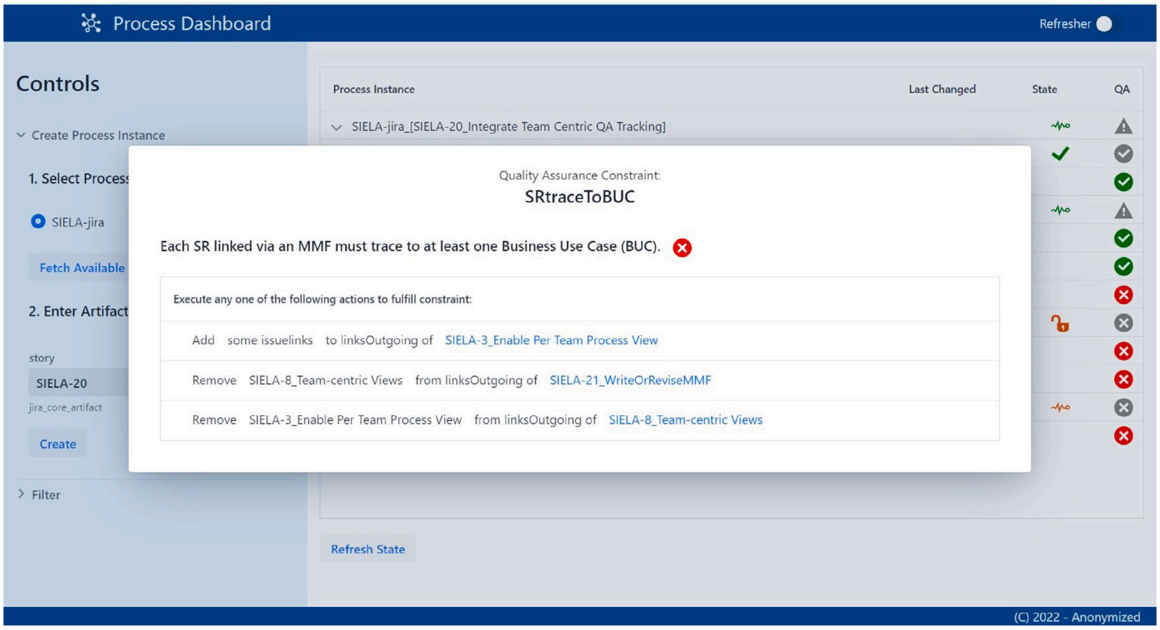
6.1. Dronology case study

6.1.1. Data set and process description

Dronology is a UAV management and control system, providing a full project environment for managing, monitoring, and coordinating the flights of multiple UAVs. It is a research incubator with various development artifacts publicly available (Cleland-Huang et al., 2018; Cleland-Huang and Vierhauser, 2023), developed by both students and professional engineers over several years. We obtained permission to use data from multiple sprints maintained in Jira from April 2017 to December 2019 consisting of 1201 issues amongst which are 111 Bugs, 237 Requirements, 267 Design Definitions, 169 Tasks, and 256 Sub-Tasks.

We treated each Task and Sub-Task as a small process instance (yielding in total 425 process instances) each consisting of two steps for "Preparing" and "Executing" the issue, using the issue's status for the steps' pre- and post-conditions. We identified five QA constraints applicable to Tasks and Subtasks and confirmed their relevance with lead developers of the Dronology project (see also Table 1). Overall, we have one precondition and one post condition for each step, together with the five QA constraints for an overall nine constraints. Encoding all these constraints in OCL took less than one hour.

[5] Use "dev" for username and password.



Quality Assurance Constraint:
SRtraceToBUC

Each SR linked via an MMF must trace to at least one Business Use Case (BUC).

Execute any one of the following actions to fulfill constraint:

Add some issuelinks to linksOutgoing of SIELA-3_Enable Per Team Process View

Remove SIELA-8_Team-centric Views from linksOutgoing of SIELA-21_WriteOrReviseMMF

Remove SIELA-3_Enable Per Team Process View from linksOutgoing of SIELA-8_Team-centric Views

Fig. 5. A screenshot of an unfulfilled QA constraint with details and suggested repair options, as seen on the web-based dashboard.

Table 1
Pre-/Post-conditions and QA constraints for the Dronology case study.

ID	Description
	Step prepare
Pre-condition	The process input is an issue of type Task or Subtask
Post-condition	The Task/Subtask is in state "In Progress"
QA_Assignee	A Task/Sub-Task needs to have an assignee.
QA_TraceDD	The Task/Subtask traces to exactly one Design Definitions directly, or via its parent
QA_NoTraceReq	The Task/Subtask must not directly trace to a requirement
	Step execute
Pre-condition	The Task/Subtask is in state "In Progress"
Post-condition	The Task/Subtask is in state "Resolved" or "Closed"
QA_Subtasks	All subtasks are all in state "Resolved" or "Closed"
QA_RelatedBugs	All related bugs are all in state "Resolved" or "Closed"

6.1.2. *Data processing*

We used our trace link replay tool to reset all issues and their trace links to the earliest change event and then replayed every single change in the correct temporal order.

This enables us to start from the beginning of the development process and, step-by-step, simulate (i.e., "replay") changes made by engineers (e.g., modify the state of artifacts in Jira, add trace links, etc.) then automatically trigger QA constraint checks, step pre- and post-conditions, and subsequently provide repair options the same way as in a "live" environment. We deployed a repair analysis component in ProGuide which determines every change's impact on any constraint that might be affected by it:

- *Positive* impact if the change repairs a constraint. If so, we additionally record whether we had suggested that change as part of prior repair options. If not, this would identify an incomplete repair option list.
- *Negative* impact if the change violates a so far fulfilled constraint. If so, we additionally record how many repair options ProGuide now generates for the newly violated constraint.
- *Neutral* impact if the constraint remains fulfilled or remains unfulfilled. In the latter case, we regenerate the repair tree and record again the number of repair options.

Across the 425 process instances, replaying almost 11.500 change events resulted in 3337 recalculations of repair options across all pre-/post-conditions and QA constraints.

6.2. *Controlled experiment*

In highly processes driven engineering environments, engineers need to be aware of the process progress and any constraints that need to be fulfilled, respectively repaired.

The *goal* of this study is to observe to what extent engineers are able to determine process progress with and without tool assistance, and whether the provided constraint repairs are useful.

The *motivation* behind this study is to determine what process aspects engineers find challenging, to what extent guidance is unclear, and what other support they might benefit from to minimize their effort to follow the process and its QA constraints.

The *perspective* is of researchers and practitioners at large process-driven software and systems development organizations and those at vendors providing support tools to these organization to take novel insights as a basis for developing appropriate support mechanisms for the developer stakeholder group.

The *context* of this study comprises six engineers tasked with evaluating and repairing process step constraints with and without automatic constraint evaluation support in the scope of a controlled experiment.

Table 2
Pre-, Post-conditions, and QA constraint excerpts from the controlled experiment.

ID	Description
	Requirements management process
QA_AllReq-Released	Ensure that all requirements are in state “Released”.
QA_FuncSpec-ReadyForReview	Ensure that all function specifications are in state “Ready for Review”.
QA_FuncSpec-Released	Ensure that all function specifications are in state “Released”.
QA_OneResolved-Review	Ensure that all function specifications have exactly one review linked as successor in state “Resolved”.
	Project management process
Precondition CreateOrUpdateSoftwareProjectPlan	Ensure all feasibility studies are in state “Released” and all risk analysis are in state “Released”.
Post-condition CreateOrUpdateSoftwareProjectPlan	Ensure all software project plans are in state “Ready for Review”.
	Verification and validation process
QA_FuncSpec-HasReview	Ensure each released function specification has a review linked as successor.
QA_FuncSpec-TracesToTestPlan	Ensure each function specification is traced to one of the change request’s test plans.

6.2.1. *Experiment design*

For the evaluation of the passive process engine, we have set up the experiment following the recommendation of Ko et al. (2015). The participants were six function developers from Bosch, selected based on their familiarity with the basic engineering process steps. Each participant went through an experiment session that adhered to the following procedure. (Participant demographics are reported at the end of this subsection.)

(1) Introduction: the participants were presented the aim of the experiment and the purpose and development state of the prototype to be tested. This included signing an informed consent form (provided as SOM) and explaining to them that no personal information will be collected during the experiment (except for voice recordings kept for one month for analysis purposes) and that participation is voluntary, unpaid, and may be aborted at any time without reason and repercussions.
(2) Demographic details: we enquired about the background information of each participant, with respect to their experience in their current position and their familiarity with the process and the different tools used in the experiment.
(3) Initial expectations collection: the participants were invited to describe their preferred guidance methods and the aspects of the process where they feel guidance would be especially useful based on their daily work experience.
(4) Tool demonstration: we have selected an example process which is used to demonstrate the basic features of Azure DevOps Services and the ProGuide dashboard, which the participants will use during the course of the experiment.
(5) Warmup task: after the demonstration, the participants receive an example process and a set of yes/no questions, which allow them to explore the tools and familiarize themselves with the experiment environment for 10 min.
(6) Experiment tasks: the participants are given four tasks, of which two focus the understandability of the environment (Task 1a + 1b) for 5 min each, and two require them to edit engineering artifacts to reach a consistent process state (Task 2a + 2b) within 10 min each. For each category of tasks, the participants carry out one task using only Azure DevOps Services and another with the additional support of the dashboard. We have ensured that all the possible permutations of the setup were covered across all participants at least once and not more than twice.
(7) Perception collection: after each task, the participants fill in the NASA TLX (Hart, 2006) complexity score form.
(8) Feedback collection: after all tasks have been completed, the participants were asked to give their perception of the impact of the Dashboard on their performance comparing the two similar tasks, were invited to express their overall feedback on the dashboard, and how it could enhance their established daily work processes. We then applied thematic analysis independently by three authors (including the industry contact) of the paper and then compared and condensed the results.
(9) System usability scores (Brooke, 1996): at the end, each participant filled out the usability score form consisting of the 10 standard questions which capture their impressions after their first experience with the prototype.

The experiment was carried out across two days at the end of July 2022, with three sessions each day. Each session lasted around 1 h 30 min, taking into consideration the feedback sessions as well. The experiment took place in a meeting room at Bosch. The meeting room was fitted with a large display that was connected to a laptop. The tools involved in the experiment were running in a browser on the laptop, while ProGuide was connected through a local network. The participant screen and their voice were recorded during the tasks.

The experiment setup was tested with a Bosch function developer that was not involved as participant in the experiments. We have run two test runs with them in order to adapt and fine-tune the task descriptions and duration, and improve the tasks to match realistic scenarios. Due to the highly complex tool environment at Bosch we opted for an experiment environment and example processes that did not entirely match the usual Bosch process steps but are very close in terms of process steps, artifact types, and tool capabilities.

6.2.2. *Scenario processes*

For the purpose of the user study, we have selected three processes which are part of and similar to the development processes used by our industry collaborator. These were presented in a familiar visual notation exported from the process support tool “Stages”.[6]

The Requirements Management process, pictured in Fig. 6, is used twice in different process instances. It contains 4 steps and a total of 10 constraints. The second selected process is the Project Management, as seen in Fig. 7, which contains 5 steps and a total of 8 constraints. Lastly, the second process repair task concerns the Verification and Validation process in Fig. 1, featuring 2 steps and 8 constraints (see also Table 2).

Based on these processes, we have created an example setup consisting of 20 Azure work items. The types and link structure of these

[6] https://www.methodpark.com/stages.html.

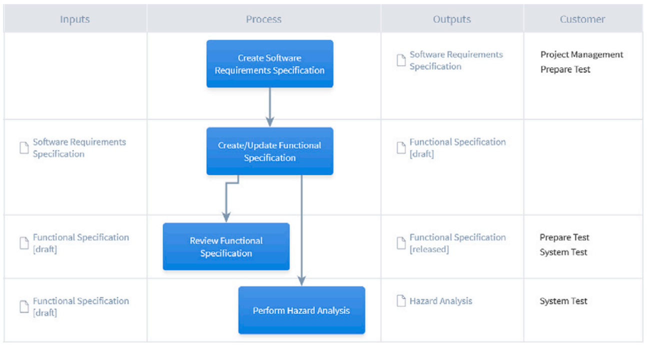

Inputs
Process
Outputs
Customer
Create Software Requirements Specification
Software Requirements Specification
Project Management
Prepare Test
Software Requirements Specification
Create/Update Functional Specification
Functional Specification [draft]
Functional Specification [draft]
Review Functional Specification
Functional Specification [released]
Prepare Test
System Test
Functional Specification [draft]
Perform Hazard Analysis
Hazard Analysis
System Test

Fig. 6. Requirements management process overview.

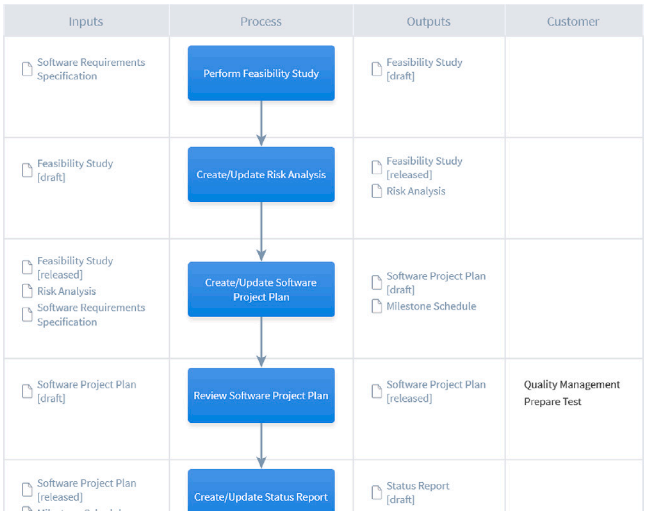

Inputs
Process
Outputs
Customer
Software Requirements Specification
Perform Feasibility Study
Feasibility Study [draft]
Feasibility Study [draft]
Create/Update Risk Analysis
Feasibility Study [released]
Risk Analysis
Feasibility Study [released]
Risk Analysis
Software Requirements Specification
Create/Update Software Project Plan
Software Project Plan [draft]
Milestone Schedule
Software Project Plan [draft]
Review Software Project Plan
Software Project Plan [released]
Quality Management
Prepare Test
Software Project Plan [released]
Create/Update Status Report
Status Report [draft]

Fig. 7. Project management process overview.

artifacts are provided in the Traceability Information Model as SOM, which was made available to the users during the experiment introduction. Each of the two task pairs (1a, 1b, 2a, 2b) has a corresponding change request (CR). We also provided 10 software requirement specifications (SRS), three functional specifications (SPEC), two feasibility studies (FS) and one software project plan (SPP). The software project plan and the feasibility studies are represented by Documentation type work items, which have titles with specific prefixes to signal the exact type of the document. This will also be the case for one of the work items the users will create, a test plan which is a Documentation work item with the prefix “TP”. The other existing work items, as well as the review work item the users will have to create, have separate types created to describe them. For the purpose of the experiment, we have set the titles, states and descriptions for each work item, and the users were instructed to only consider this information and the links in their experiments.

6.2.3. *Task 1a*

The “Create/Update Functional Specification” step of the Requirements Management process is contained in a process status-checking task. The process instance starts with a change request, which is linked to six software requirement specifications via “Affects” links. It is required that all these linked SRSs are in the state “Released”, as is the change request. The users must notice that two of the requirements are actually in different states, and as such, they are not ready to proceed with the process step.

6.2.4. *Task 1b*

The second process status checking task addresses step 3 of the Project Management process, “Create/Update Software Project Plan”, in which a change request should have a set of linked software requirements, at least one feasibility study and at least one risk analysis. Additionally, all the feasibility study and risk analysis documents must be in state “Released”. In our model, the feasibility studies and risk analysis documents are artifacts of the type Documentation, with specific prefixes defined to differentiate the specific documentation type. The setup presented to the user during the experiment features a change request with three linked requirements and two feasibility study documents in the correct state, but lacks a risk analysis document. The user is expected to signal the missing risk analysis which would prevent them from safely proceeding with the process step.

6.2.5. *Task 2a*

Further, the third step of the Requirements Management process, that is “Review Functional Specification”, is part of the first process repair tasks. This process step starts again with a change request, that has two linked SRSs. Each SRS has a Functional Specification linked as a Successor, both of which need to be reviewed in this step. In order for the review to proceed, each Functional Specification must be in the state “Ready for Review”. Firstly, the users have to find this is not the case, as one of the SPECs is actually in state “Draft”. Upon the user’s signal, the moderator will ensure this pre-condition is fulfilled, and the user can proceed with the reviews. By the end of the task, the user must create a Review artifact, that is in state “Resolved” and linked to their corresponding SPEC via a “Successor” link. Additionally, each SPEC must have exactly one review in the correct state, and must have its own state set to “Released”.

6.2.6. *Task 2b*

Lastly, the second process repair task concerns the Verification and Validation process, specifically Step 2, “Create/Update Test Plan”. Here, the change request is linked to a set of requirements, each having linked functional specifications in the state “Released”. Each functional specification should also have its review linked further, through a successor link. Additionally, the change request is linked to a software project plan, which is a documentation type artifact with a prefixed title. At the beginning of this task, the change request has one requirement linked, which further links to two functional specifications, one of which does not link a review. The user must signal this missing artifact, which is then added by the moderator, resulting in the pre-conditions being fulfilled for the process step to proceed. The user is then to create a test plan work item, of the type Document with a “TP” prefix, which should have its state set to “Ready for Review” and should be linked via “Related” links to all the functional specifications it covers, and to the change request directly.

6.2.7. *Task order*

All participants executed these tasks in this particular order but alternated which of the tasks were supported by ProGuide and which ones were only done using Azure DevOps Services. Hence in total there were 4 possible permutations. Across the six participants, we ensured that each permutation occurred at least once and not more than twice.

6.2.8. *Participant demographics*

The experiment involved six participants from the group of function developers at our industry partner Robert Bosch AG. The Robert Bosch AG is part of the Bosch Group which is one of the leading automotive suppliers and employs roughly 421,000 associates worldwide. The participants were recruited by our industry author directly to obtaining a diverse selection in terms of customer project involvement. The industry author had no managerial authority over the participants but was rather situated in a comparable employment situation.

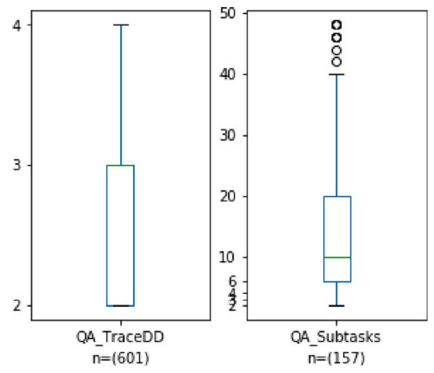


Fig. 8. Range of repair options (excerpt) with n reporting how often the repair options were recalculated.

The participants' experience in professional software development at Bosch spanned between 6 months and 11 years, out of which up to 8 years were spent in their current position in the company. In their daily work, most participants find themselves consulting the process documentation or discussing process requirements with other developers several times a week, whether for personal purposes or to help other colleagues out. Two other participants reported checking the process documentation more rarely, once a month or even less often, both reporting themselves towards the upper limit of experience in their current position. None of the participants has any previous experience with Azure DevOps Services, but all have extensively used other ticketing systems during their careers. They also never worked with the Object Constraint Language (OCL).

7. Results

7.1. Answering RQ1

7.1.1. Results from replaying Dronology

Across the 425 process instances, replaying almost 11.500 change events resulted in 3337 recalculations of repair options across all pre-/post-conditions and QA constraints. Fig. 8 details for two of them the range of repair options (all other repair suggestions always consisted of a single option or two options, see SOM Mayr-Dorn, 2023). Fixing the trace to a Design Definition (QA_TraceDD) required choosing between two and three options depending on whether there was a parent task present. Ensuring that all subtasks are closed (QA_Subtasks) yielded the most options – specifically, two for every subtask – as this depends directly (i.e., linearly) on the number of subtasks. Hence the maximum of 48 repair options did not describe 48 different ways of fixing the constraint but rather mandated that for up to 24 subtasks the engineer has to decide whether to close them or unlink them.

ProGuide's repair analysis component detected no situation where no repair options were provided, and also detected no actual eventual repair by an engineer that was not initially suggested. To this end, we focus on the 294 (out of 425) process instances that were signaled as completed (i.e., the issue state was 'closed' or 'resolved'). Table 3 details for how many process instances we suggested the repair of a violated QA constraint, how many of those were repaired, and how many were fulfilled at the end (QA Constraint IDs reference the constraints in Table 2). Note that the least repaired constraint QA_TraceDD requires the most mental work as the correct Design Definition needs to be identified in order to trace to it.

7.1.2. Results from the user study

In the experiment, the amount of constraints (counting pre-conditions, post-conditions, and QA constraints) varied with the concrete tasks the participant was asked to solve with the help of ProGuide. From the overall 36 constraints across four process definitions, the *Process Dashboard* presented between 18 and 20 constraints to a participant. The amount of repair options per constraint instance ranged from 0 to 6, with a median of 2. Note, that it was not a failure of ProGuide when no repair option was provided, as in all cases this was caused by a process step not having the necessary input.[7] In such cases, we provided a hint that input is required instead of a repair option.

When asked whether there were any repairs not suggested they would have found useful, participants did not identify any additional repair options but commented on the level of detail provided (see Section 7.2.5).

RQ1 Key Observation: The suggested repair options are complete and typically small in numbers, respectively growing linear with the number of constraint violating, linked artifacts.

Table 3
QA repairs for completed process instances: displaying number of **vio**lated, number of **rep**aired, percentage of violated (**V%**), number of fulfilled at end (**Ok at end**), and percentage of fulfilled at end constraints (**%**).

QA constraint ID	Vio.	Rep.	V%	OK at end	%
QA_Assignee	221	206	93%	279	95%
QA_TraceDD	294	44	15%	44	15%
QA_NoTraceReq	3	2	67%	293	99%
QA_Subtasks	8	6	75%	292	99%
QA_RelatedBugs	0	0	n/a	294	100%

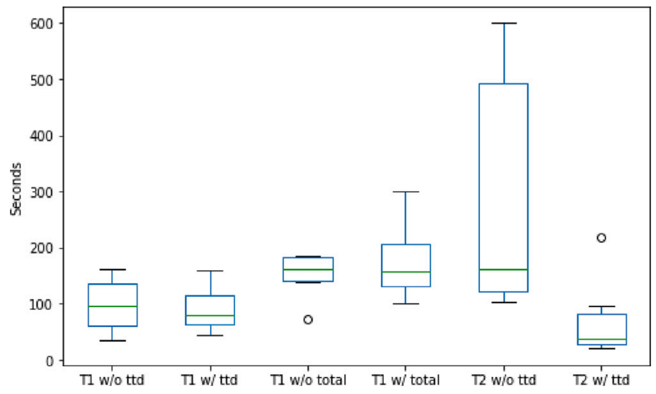


Fig. 9. Duration for executing tasks T1 and T2 measuring time to detection (ttd) and total time, with (w/) and without (w/o) ProGuide.

7.2. Answering RQ2

7.2.1. Task duration differences

In order to assess the mental overhead that ProGuide causes, we measured the time how long participants took to detect that the pre-conditions in Task 1a/b and Task 2a/b were not fulfilled, and also how long it took participants to explain why the preconditions were not fulfilled in Task 1a/b. We then determined the difference in duration when participants were using ProGuide to solely using Azure DevOps Services.

Fig. 9 left depicts the duration until participants noticed the unfulfilled precondition. The range of time needed is roughly equal, with three participants requiring more time with ProGuide, while three participants required more time without.

Fig. 9 middle depicts the duration until participants could explain the unfulfilled precondition or reached the 5-minute time limit (one

[7] We do not recommend repairing the step's input, as the input is derived from the output of a prior step, and only there any repairs make sense.

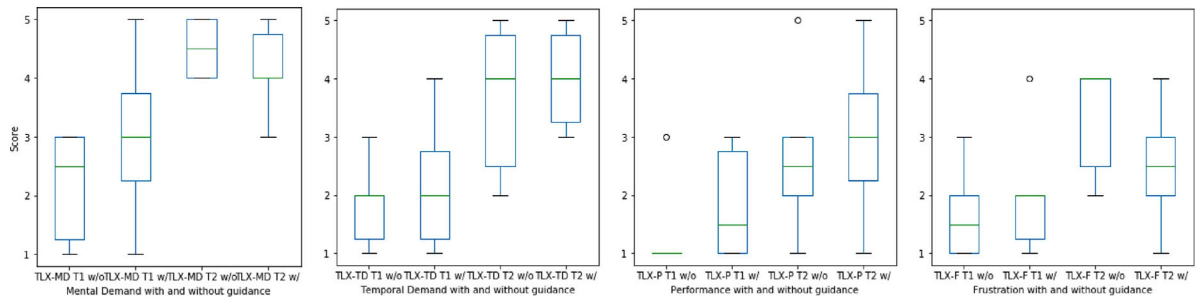


Fig. 10. NASA TLX: mental demand, temporal demand, performance, and frustration for T1 and T2 with and without ProGuide for all six participants.

participant). The median duration is similar, with some participants taking much longer with ProGuide.

During Task 1, we observed some participants hold back on the quick assessment until they had identified the constraint violation reason in Azure DevOps Services when they were using ProGuide. This was not the case in Task 2 where participants were only asked to quickly assess the precondition before continuing with the remainder of their task. The obtained duration measurements are depicted in Fig. 9 right. Average and median duration is much lower when participants used ProGuide compared to without ProGuide. Note that in the latter case, two participants failed to detect that the preconditions were violated and hence their duration was considered the task's time limit of 10 min.

7.2.2. Task correctness differences

In order to quantify the assistance that ProGuide provides, we analyzed the constraints that participants managed to fulfill within 10 min. Out of the 12 tasks 2a and 2b instances, participants reached the time limit four times (twice with and twice without ProGuide). For comparison of Task 2a and 2b, we clustered their constraints into two groups: constraints that ensured the involved artifacts were in the correct state, and those that checked the use of correct traces between artifacts. In the former group, two constraints remained unfulfilled independently of ProGuide usage. In the latter group, only one constraint was not fulfilled when using ProGuide, while four constraints remained unfulfilled using only Azure DevOps Services.

Note that we refrain from analyzing overall task time in detail, as most participants used up the allotted time to recheck their constraints. Two participants that finished their task without ProGuide around a minute early had some constraints violated. The single participant that finished their task with ProGuide around half a minute before the deadline had all constraints fulfilled.

7.2.3. Task perception

To understand the impact of ProGuide on perceived performance, we compare the NASA-TLX (Hart, 2006) sub-scales for tasks with and without ProGuide. Fig. 10 compares the participant rating from mental demand (MD), temporal demand (TD), performance (P), and frustration (F) for Task 1 and Task 2. For the reporting of the numbers below, recall that Task 1 was easier and shorter than Task 2.

We observe that mental demand is higher during Task 1 when using ProGuide, while in Task 2 it is the opposite.

Taking the median, temporal demand is roughly equal with and without ProGuide. Also, participants judged their allocated tasks 1a and 1b equally demanding. Task 2 put more time pressure on the participants, with two participants judging their task with ProGuide less temporally demanding, and four participants more temporally demanding.

For perceived performance we notice an interesting phenomenon: for Task 1, participants were not quite satisfied when using ProGuide even all were equally successful to detect the reason for the violated preconditions. Similarly for Task 2, where participants actually made fewer mistakes with ProGuide. Most noteworthy, the individual participants' self-perceived performance score only correlated for one participant with their actual performance. In other words, four participants believed they did better without ProGuide even they made fewer mistakes while using it, and one participant believed they did better with ProGuide but actually made more mistakes.

Finally, measuring frustration, we noticed that participants reported to have felt slightly more frustrated with ProGuide in Task 1 but definitely less frustrated with ProGuide in Task 2. Especially, when looking at individual participants' scoring, we observe two participants being equally frustrated and four participants being less frustrated when having ProGuide available.

7.2.4. System Usability Score (SUS)

We also collected and aggregated SUS scores (Brooke, 1996) from all participants (n = 6) (graph available in the SOM). SUS scores range from 0 (extremely difficult to use) to 100 (extremely easy to use). The primary goal of obtaining SUS scores was to ensure the prototype is usable enough so as not to negatively impact the result by hindering the participants in completing their tasks. The mean SUS score is 55 (std 20.1). Bangor et al. (2009) determined that SUS results highly correlate with adjective ratings on a Likert scale of size seven from "worst imaginable" to "best imaginable". In their analysis, the fourth Likert scale entry (here "Ok") correlates with a SUS scale of 50.9. Hence, our prototype can be considered usable from this perspective, however, with room for improvement. We did not collect SUS score for Azure DevOps Services as it is a mature service platform provided by Microsoft. Its self-hosted predecessors Microsoft Azure DevOps Server, Team Foundation Server, and Visual Studio Team System have a long history, being first released in 2005.

7.2.5. Qualitative observations

Overall participants stated that the repairs are correct but could be more detailed to facilitate the selection and execution of the repair thereby reducing cognitive overhead. While, for example, the human-readable QA constraint description stated that all `Requirements need to be in the state ‘‘released’’`, they would have preferred to have this precise state also repeated in the repair option instead of just "`Change state of Requirement SRS-2`".

Participants also commented on the repair options that suggested removing a trace, e.g., between a *Change Request* and an SRS. While they generally believed that such a repair might be useful they stressed that this case is rather rare and should not be presented at the top of the listed repair options. This kind of repair option however became relevant during the experiment task when one participant used the wrong link type to trace a *Review* to an FS. Again, participants preferred to have more detailed repair options; in this case the type of link to create.

When participants reflected on their task execution with and without ProGuide, they clearly see the benefit of having the prototype support them in establishing what the current process progress is. While they found Azure DevOps Services easy to use, especially when the corresponding artifact web page displayed all the details they needed for constraint assessment. When asked, whether they would have found it as easy without the experiment task documentation telling them what to check on what artifact (i.e., an Azure DevOps Services workitem), they responded no. Some commented it would be hard to establish the process status just from the Azure DevOps Services artifact alone as one would have to memorize all and every relevant constraint to know what to look out for.

When using ProGuide, they liked the immediate feedback given upon changes in Azure DevOps Services and that they obtain an overview of the process with explicit process step readiness and completion status. Aspects that made ProGuide harder to work with was the unclear differentiation between step completion constraints and QA constraints in the *Process Dashboard*. They also noted that all artifact details including links to other artifacts (aside from artifact title and repair options) have to be obtained from Azure DevOps Services.

RQ2 Key Observation: Participants deem ProGuide's repair options as useful but suggest including more details to reduce the cognitive overhead.

8. Discussion

8.0.1. Repair amount and completeness

The two evaluation settings (Dronology and user study) have shown that the suggested repair options are typically small in number. Occasionally, many repair options may be provided when a number of linked artifacts are the source of a constraint violation. In this case – and given the type of underlying constraints – the repair options grow linearly with the violating linked artifacts. Ultimately, the complexity of a constraint in combination with the number of artifacts that are within the scope of the constraint determine the number of repair options.[8] However, as the constraints (also in human-readable form) need to be understandable by engineers, there exists an implicit boundary on how complex those constraints will become in practice. Likewise, the number of artifacts in the constraint scope is primarily determined by trace links and hence also will be limited to an amount engineers are able to manage without any guidance support. We thus can conclude that the number of repair options will not overwhelm engineers but rather offer precise guidance on how to fix a violation.

The two evaluation settings have also shown that the repairs are complete, i.e., no possible repair option was omitted. The feedback from the user study has identified that all generated repair options are plausible, but vary in their likelihood to be applicable. Overall this implies that engineers are made available the full range of options without limiting them to a subset that might not be a suitable choice in their occasional edge case. Indeed, participants mentioned in the introductory discussion of the user study, that guidance needs to be able to consider edge cases to be truly useful. We are aware, however, that engineers will benefit from more sophisticated repair details and repair option ranking (see below).

Repair Usefulness The qualitative feedback from the participants provides a clear indication on the usefulness of process guidance, with all of them seeing the benefit of obtaining a process progress overview, establishing an explicit status (rather than having to infer it from artifacts), receiving automated, timely feedback, and being provided with actionable repairs. Beyond this assessment, the qualitative feedback was invaluable to better understand what factors influence the uptake and acceptance of a guidance tool. The more details the repair option provides, the easier an engineer finds it to execute even as such details can be obtained from the constraint description. We hypothesize that merely informing the engineer that a constraint is violated and suggesting repair options creates a mental gap, namely the explicit description of the violation's cause is missing and has to be inferred from the constraint and the repair. In the next prototype iteration, we plan to include such an explanation to simplify the interpretation and assessment of the repair option (the information is readily available in the constraint evaluation tree). This explanation would then be the right location to provide artifact details that are relevant to the constraint evaluation and thus further reduce the need to inspect the affected artifacts in their tools.

Ranking repairs would further improve usefulness. The number of repairs is typically small, hence in the case of multiple options, the best suitable repair will be among the top five and easily found. Participants, nevertheless, preferred to have the most likely applicable repair option listed first. As ProGuide collects suggested and actually executed repair options, the data basis for determining the most used repairs in general, as well as in a particular process context, is available. We list further participant-provided feature suggestions that go beyond repair-centric aspects in Section 8.0.1 below.

Guidance Overhead Unsurprisingly, introducing another tool on top of an engineer's regular work environment adds some overhead as we can see in the duration of Task 1 as well as the mental and temporal demand measured via the NASA TLX scale when comparing task execution with and without ProGuide. Note, however, that the experiment task description stated the precise constraints that the participant should check. In their regular working environment, engineers would have to establish the relevant (i.e., currently applicable) constraints from their context and memory. These two aspects are automatically established by ProGuide. Hence we argue that the mental effort without any guidance would be much higher in reality. Additionally, we observe that as participants get more familiarized with (and gain trust in) the ProGuide and as the task complexity increases, participants require less time with ProGuide than without, make fewer mistakes, and also feel less frustrated (see time-to-detection (ttd) for Task 2 in Fig. 9 right and NASA TLX frustration score Fig. 10).

Guidance beyond repairs During the concluding discussion, participants were invited to suggest improvements and to comment on the overall approach.

Most participants noted that for following the process they have to have a good understanding of why the readiness conditions, completion conditions, and QA constraints are relevant. Providing a rationale for why a constraint is in place supports the engineer, especially newcomers, in choosing the right repair and also motivating them to follow the process.

All participants stressed the importance of being able to deviate from the process, especially when rework is necessary. Displaying the effect of rework, e.g., some QA constraint being no longer fulfilled, is very useful and could be further improved by details on what has changed, and an explicit assessment of who and what is affected by that change including notification of the affected engineers.

Some participants mentioned their desire to reduce the amount of tool switching and micro-actions needed to execute a repair and suggested the possibility of automatically executing the repair from the *Process Dashboard*. All participants did not share this idea, as some noted that such a feature could lead to engineers not properly reflecting on the repair, perhaps even choosing the repair that causes the least effort.

8.1. Discussion on approach prerequisites

While our approach is not limited to supporting engineering processes of safety-critical systems, in practice these highly regulated environments are the ones most likely to benefit from our approach as other domains rarely need to provide evidence of having followed a particular engineering process. Consequently, in these engineering environments the integration of tools is a key aspect for demonstrating traceability across each engineering artifact's life-cycle. We make use of

[8] Repairs could grow more quickly when a constraint requires pairwise comparisons among a large number of artifacts. However, we have not encountered any such constraint in discussions with our industry partners.

this need for integration as our approach relies on navigability amongst artifacts to track an engineering process from its very start with user requirements all the way to validation and verification. From our experience during tool connector prototyping we are aware that existing engineering tools are not as easily to integrate yet as each defines its own mechanism to describe, fetch, and listen to engineering data (changes). Yet there is a clear trend towards improved observability of engineering activities.

As the main focus of this paper is on actionable guidance and how it supports engineers in their process, we have not addressed effort and difficulty of writing process constraints by process engineers. Our approach requires a basic understanding of how to use OCL and we are aware that current tool support (in general) for writing OCL rules is not comparable to, for example, writing SQL queries. On the one hand, we believe that with more sophisticated in-IDE support such as syntax highlighting and code completion (e.g., suggesting only valid properties), writing OCL rules will become easier. On the other hand, we are also working on limiting the extent and complexity of OCL constraints by generating large parts automatically from process modeling environments. We are currently investigating how the Stages modeling environment can be used to enable the process engineer to only define multiple but minimal OCL constraints (e.g., just defining the required SRS status) and subsequently augment and combine these fragments into process-centric constraints. Further investigations are continuing in how to integrate the complex tool infrastructure at Bosch with our approach. All these efforts are, however, outside the scope of this paper.

8.2. Threats to validity

Internal Validity. We address researcher bias by modeling processes and constraints utilized in industry or are followed in open source development (i.e., Dronology). Especially for the experiment, we obtained frequent feedback from our industry partners to ensure the fictive task settings match those in the daily work of the participants. During the user study, we addressed selection bias by having engineers engage in tasks with and without ProGuide rather than creating an experimental group and a control group.

External Validity. The process and constraints used in the experiment at Bosch might not reflect the challenges and concerns in other companies. Hence we cannot make the claim that ProGuide in its current state will match the guidance needs of most software and/or systems engineering companies. Our past experience with other industry evaluation partners, however, has shown that similar concerns exist among companies operating in safety-critical or regulation-intensive domains. The current data modeling, constraint checking, and repair generation capabilities are evaluated against the needs observed by these industry partners. We expect to identify additional features and challenges from further workshops and friendly user testing. We thus make no claims that ProGuide offers support of every and all situations but definitely covers a significant relevant foundation.

Along these lines, the repair analysis in the Dronology use case might not reflect the complexity of industrial applications. Hence the number of repair options and completeness of repair options can only be generalized partially to industrial settings. Yet the constraint complexity of the Dronology use case is comparable to that found in the Bosch experiment.

Construct validity. The tight time limit for experiment Task 2 that was reached by most participants might make a clear differentiation between guidance and no guidance difficult. We nevertheless decided to impose a strict limit to observe how much participants accomplish in that short time. This mimics a situation where the engineer is interrupted (e.g., from a phone call, or having to leave for a meeting). Such an interruption requires the engineer to reestablish their working context in memory to understand what still needs to be fulfilled while an engineer with guidance merely needs to consult ProGuide to understand where they left off.

9. Conclusions

We presented an approach to assist engineers in understanding readiness and completion of process steps, and how to fix violated constraints. Our evaluation demonstrated that the generated repairs are complete and small in number. Engineers in our user study made fewer mistakes and reported lower frustration when using ProGuide.

Our future work is two-fold. We will focus on the improvements suggested by engineers at Bosch: mainly ranking repair actions by relevance, including constraint rationale, and change impact assessment along the process. At the same time we will investigate how to best support process engineers in writing and testing process definitions.

CRediT authorship contribution statement

Christoph Mayr-Dorn: Conceptualization, Data curation, Funding acquisition, Investigation, Methodology, Resources, Software, Validation, Visualization, Writing – original draft, Writing – review & editing. **Cosmina-Cristina Ratiu:** Data curation, Investigation, Software, Writing – original draft. **Luciano Marchezan de Paula:** Investigation, Methodology, Writing – original draft. **Felix Keplinger:** Software, Visualization. **Alexander Egyed:** Resources, Supervision. **Gala Walden:** Data curation, Validation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The part of the data that could be disclosed is available as SOM https://figshare.com/s/3cc1026d1abf5b1849e4. Any industry related data cannot be disclosed due to confidentiality restrictions.

Acknowledgments

This work was funded by the Austrian Science Fund (FWF) under the grant numbers 10.55776/P34805 and 10.55776/P31989, by the state of Upper Austria via LIT-2019-8-SEE-118 and the LIT Secure and Correct Systems Lab, and by the FFG, Contract No. 881844: $Pro^2 Future$ is funded within the Austrian COMET Program Competence Centers for Excellent Technologies under the auspices of the Austrian Federal Ministry for Climate Action, Environment, Energy, Mobility, Innovation and Technology, the Austrian Federal Ministry for Digital and Economic Affairs and of the Provinces of Upper Austria and Styria. COMET is managed by the Austrian Research Promotion Agency FFG.

References

Alajrami, S., Gallina, B., Sljivo, I., Romanovsky, A.B., Isberg, P., 2016. Towards cloud-based enactment of safety-related processes. In: Skavhaug, A., Guiochet, J., Bitsch, F. (Eds.), Computer Safety, Reliability, and Security - 35th International Conference, SAFECOMP 2016, Trondheim, Norway, September 21–23, 2016, Proceedings. In: Lecture Notes in Computer Science, vol. 9922, Springer, pp. 309–321. http://dx.doi.org/10.1007/978-3-319-45477-1_24.

Ali, S., Yue, T., Zohaib Iqbal, M., Panesar-Walawege, R.K., 2014. Insights on the use of OCL in diverse industrial applications. In: International Conference on System Analysis and Modeling. Springer, pp. 223–238.

Avazpour, I., Pitakrat, T., Grunske, L., Grundy, J., 2014. Dimensions and metrics for evaluating recommendation systems. In: Robillard, M.P., Maalej, W., Walker, R.J., Zimmermann, T. (Eds.), Recommendation Systems in Software Engineering. Springer, pp. 245–273, [Online]. Available: http://dx.doi.org/10.1007/978-3-642-45135-5_10.

Bandinelli, S., Di Nitto, E., Fuggetta, A., 1996. Supporting cooperation in the SPADE-1 environment. IEEE Trans. Softw. Eng. 22 (12), 841–865. http://dx.doi.org/10.1109/32.553634.

Bangor, A., Kortum, P., Miller, J., 2009. Determining what individual SUS scores mean: Adding an adjective rating scale. J. Usability Stud. 4 (3), 114–123.

Barriga, A., Heldal, R., Iovino, L., Marthinsen, M., Rutle, A., 2020. An extensible framework for customizable model repair. In: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems. MODELS '20, Association for Computing Machinery, New York, NY, USA, pp. 24–34, [Online]. Available: http://dx.doi.org/10.1145/3365438.3410957.

Barthelmess, P., 2003. Collaboration and coordination in process-centered software development environments: a review of the literature. Inf. Softw. Technol. 45 (13), 911–928, [Online]. Available: http://dx.doi.org/10.1016/S0950-5849(03)00091-0.

Brooke, J., 1996. SUS: a "quick and dirty" usability scale. In: Jordan, P.W., Thomas, B., Weerdmeester, B.a., McClelland, a.L. (Eds.), Usability Evaluation in Industry. Vol. 189, Taylor and Francis, London.

Brosgol, B., Comar, C., 2010. DO-178C: A New Standard for Software Safety Certification. Tech. Rep., ADA CORE TECHNOLOGIES NEW YORK NY.

Cabanillas, C., Resinas, M., Ruiz-Cort's, A., 2020. A mashup-based framework for business process compliance checking. IEEE Trans. Serv. Comput. http://dx.doi.org/10.1109/TSC.2020.3001292, 1–1.

Cleland-Huang, J., Gotel, O., Hayes, J.H., Mäder, P., Zisman, A., 2014. Software traceability: trends and future directions. In: Proc. of the on Future of Software Engineering. FOSE 2014, ACM, pp. 55–69. http://dx.doi.org/10.1145/2593882.2593891.

Cleland-Huang, J., Vierhauser, M., Dronology Public Datasets. [Online]. Available: https://dronology.info/research/datasets.

Cleland-Huang, J., Vierhauser, M., Bayley, S., 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. In: ICSE-NIER '18, ACM, pp. 109–112. http://dx.doi.org/10.1145/3183399.3183408.

Colantoni, A., Horváth, B., Horváth, Á., Berardinelli, L., Wimmer, M., 2021. Towards continuous consistency checking of DevOps artefacts. In: 2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion. MODELS-C, pp. 449–453. http://dx.doi.org/10.1109/MODELS-C53483.2021.00069.

Cugola, G., Ghezzi, C., 1999. Design and implementation of PROSYT: a distributed process support system. In: Proc. of the IEEE 8th Int'l Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises. WET ICE'99, IEEE, pp. 32–39.

Damian, D., Chisan, J., 2006. An empirical study of the complex relationships between requirements engineering processes and other processes that lead to payoffs in productivity, quality, and risk management. IEEE Trans. Softw. Eng. 32 (7), 433–453.

Diebold, P., Scherr, S.A., 2017. Software process models vs descriptions: What do practitioners use and need? J. Softw.: Evol. Process 29 (11), http://dx.doi.org/10.1002/smr.1879.

Dumas, M., Pfahl, D., 2016. Modeling software processes using BPMN: When and when not? In: Kuhrmann, M., Münch, J., Richardson, I., Rausch, A., Zhang, H. (Eds.), Managing Software Process Evolution: Traditional, Agile and beyond – How to Handle Process Change. Springer International Publishing, Cham, pp. 165–183, [Online]. Available: http://dx.doi.org/10.1007/978-3-319-31545-4_9.

Egyed, A., Zeman, K., Hehenberger, P., Demuth, A., 2018. Maintaining consistency across engineering artifacts. IEEE Comput. 51 (2), 28–35. http://dx.doi.org/10.1109/MC.2018.1451666.

Ellner, R., Al-Hilank, S., Drexler, J., Jung, M., Kips, D., Philippsen, M., 2010. eSPEM – A SPEM extension for enactable behavior modeling. In: Kühne, T., Selic, B., Gervais, M.-P., Terrier, F. (Eds.), Modelling Foundations and Applications. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 116–131.

Fernstrom, C., 1993. Process weaver: Adding process support to UNIX. In: Proc. of the 2nd Int'l Conf. on the Software Process-Continuous Software Process Improvement. IEEE, pp. 12–26.

Food, U., Administration, D., 2022. Code of federal regulations - title 21 - part 820 quality system regulation. https://www.ecfr.gov/current/title-21/chapter-I/subchapter-H/part-820?toc=1. Accessed: 2021-12-01.

Geppert, A., Tombros, D., Dittrich, K.R., 1998. Defining the semantics of reactive components in event-driven workflow execution with event histories. Inf. Syst. 23 (3–4), 235–252. http://dx.doi.org/10.1016/S0306-4379(98)00011-8.

Gruhn, V., 2002. Process-centered software engineering environments, a brief history and future challenges. Ann. Softw. Eng. 14 (1–4), 363–382.

Grundy, J.C., Hosking, J.G., 1998. Serendipity: Integrated environment support for process modelling, enactment and work coordination. Autom. Softw. Eng. 5 (1), 27–60, [Online]. Available: http://dx.doi.org/10.1023/A:1008606308460.

Hachemi, A., Ahmed-Nacer, M., 2022. POEML: A process orchestration, execution, and modeling language. J. Softw.: Evol. Process 34 (6), e2456.

Hart, S.G., 2006. NASA-task load index (NASA-TLX); 20 years later. In: Proceedings of the Human Factors and Ergonomics Society Annual Meeting. Vol. 50, Sage publications Sage CA: Los Angeles, CA, pp. 904–908, no. 9.

Henriksson, J., Borg, M., Englund, C., 2018. Automotive safety and machine learning: Initial results from a study on how to adapt the ISO 26262 safety standard. In: 2018 IEEE/ACM 1st International Workshop on Software Engineering for AI in Autonomous Systems. SEFAIAS, pp. 47–49.

Junkermann, G., Peuschel, B., Schäfer, W., Wolf, S., 1994. MERLIN: Supporting cooperation in software development through a knowledge-based environment. In: Software Process Modelling and Technology. Research Studies Press Ltd., GBR, pp. 103–129.

Khelladi, D.E., Bendraou, R., Hebig, R., Gervais, M.-P., 2017. A semi-automatic maintenance and co-evolution of OCL constraints with (meta)model evolution. J. Syst. Softw. 134, 242–260. http://dx.doi.org/10.1016/j.jss.2017.09.010, [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016412121730198X.

Khelladi, D.E., Kretschmer, R., Egyed, A., 2019. Detecting and exploring side effects when repairing model inconsistencies. In: Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering. In: SLE 2019, Association for Computing Machinery, New York, NY, USA, pp. 113–126, [Online]. Available: http://dx.doi.org/10.1145/3357766.3359546.

Klare, H., 2018. Multi-model consistency preservation. In: Proc. of the 21st ACM/IEEE Int'l Conf. on Model Driven Engineering Languages and Systems: Companion Proceedings. MODELS '18, ACM, pp. 156–161. http://dx.doi.org/10.1145/3270112.3275335.

Knuplesch, D., Reichert, M., Kumar, A., 2017. A framework for visually monitoring business process compliance. Inf. Syst. 64, 381–409. http://dx.doi.org/10.1016/j.is.2016.10.006, [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0306437915301770.

Ko, A.J., LaToza, T.D., Burnett, M.M., 2015. A practical guide to controlled experiments of software engineering tools with human participants. Empir. Softw. Eng. 20 (1), 110–141.

Kramer, D.B., Tan, Y.T., Sato, C., Kesselheim, A.S., 2014. Ensuring medical device effectiveness and safety: a cross–national comparison of approaches to regulation. Food Drug Law J. 69 (1), 1–23, i.

Kretschmer, R., Khelladi, D.E., Egyed, A., 2021. Transforming abstract to concrete repairs with a generative approach of repair values. J. Syst. Softw. 175, 110889. http://dx.doi.org/10.1016/j.jss.2020.110889, [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016412122030279X.

Kumar, A., Yao, W., Chu, C.-H., 2013. Flexible process compliance with semantic constraints using mixed-integer programming. INFORMS J. Comput. 25 (3), 543–559.

LaMarca, A., Edwards, W.K., Dourish, P., Lamping, J., Smith, I., Thornton, J., 1999. Taking the work out of workflow: mechanisms for document-centered collaboration. In: ECSCW'99. Springer, pp. 1–20.

Ly, L.T., Maggi, F.M., Montali, M., Rinderle-Ma, S., van der Aalst, W.M., 2015. Compliance monitoring in business processes: Functionalities, application, and tool-support. Inf. Syst. 54, 209–234. http://dx.doi.org/10.1016/j.is.2015.02.007, [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0306437915000459.

Macedo, N., Jorge, T., Cunha, A., 2017. A feature-based classification of model repair approaches. IEEE Trans. Softw. Eng. 43 (7), 615–640. http://dx.doi.org/10.1109/TSE.2016.2620145.

Macher, G., Much, A., Riel, A., Messnarz, R., Kreiner, C., 2017. Automotive SPICE, safety and cybersecurity integration. In: International Conference on Computer Safety, Reliability, and Security. Springer, pp. 273–285.

Mäder, P., Jones, P.L., Zhang, Y., Cleland-Huang, J., 2013. Strategic traceability for safety-critical projects. IEEE Softw. 30 (3), 58–66.

Maggi, F.M., Montali, M., Westergaard, M., van der Aalst, W.M., 2011. Monitoring business constraints with linear temporal logic: An approach based on colored automata. In: International Conference on Business Process Management. BPM, Springer, Clermont-Ferrand, France, pp. 132–147.

Marchezan, L., Assuncao, W.K.G., Kretschmer, R., Egyed, A., 2022. Change-oriented repair propagation. In: Proceedings of the International Conference on Software and System Processes and International Conference on Global Software Engineering. ICSSP '22, Association for Computing Machinery, New York, NY, USA, pp. 82–92, [Online]. Available: http://dx.doi.org/10.1145/3529320.3529330.

Maro, S., Steghöfer, J.-P., Bozzelli, P., Muccini, H., 2022. TracIMo: a traceability introduction methodology and its evaluation in an agile development team. Requir. Eng. 27 (1), 53–81.

Maro, S., Steghöfer, J.-P., Staron, M., 2018. Software traceability in the automotive domain: Challenges and solutions. J. Syst. Softw. 141, 85–110.

Mayr-Dorn, C., 2023. Supporting online material. https://figshare.com/s/3cc1026d1abf5b1849e4. Accessed: 2023-09-27.

Mayr-Dorn, C., Vierhauser, M., Bichler, S., Keplinger, F., Cleland-Huang, J., Egyed, A., Mehofer, T., 2021. Supporting quality assurance with automated process-centric quality constraints checking. In: 43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021. IEEE, pp. 1298–1310, [Online]. Available: http://dx.doi.org/10.1109/ICSE43902.2021.00118.

McHugh, M., McCaffery, F., Casey, V., 2014. Adopting agile practices when developing software for use in the medical domain. J. Softw.: Evol. Process 26 (5), 504–512. http://dx.doi.org/10.1002/smr.1608, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.1608. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1608.

Mosquera, D., Ruiz, M., Pastor, O., Spielberger, J., Fievet, L., 2022. OntoTrace: A tool for supporting trace generation in software development by using ontology-based automatic reasoning. In: International Conference on Advanced Information Systems Engineering. Springer, pp. 73–81.

Murphy-Hill, E.R., Murphy, G.C., 2014. Recommendation delivery - getting the user interface just right. In: Robillard, M.P., Maalej, W., Walker, R.J., Zimmermann, T. (Eds.), Recommendation Systems in Software Engineering. Springer, pp. 223–242, [Online]. Available: http://dx.doi.org/10.1007/978-3-642-45135-5_9.

Nentwich, C., Capra, L., Emmerich, W., Finkelstein, A., 2003a. Xlinkit: a consistency checking and smart link generation service. ACM Trans. Internet Techn. 2 (2), 151–185, [Online]. Available: http://doi.acm.org/10.1145/514183.514186.

Nentwich, C., Emmerich, W., Finkelsteiin, A., 2003b. Consistency management with repair actions. In: Proceedings of the 25th International Conference on Software Engineering. In: ICSE '03, IEEE Computer Society, Washington, DC, USA, pp. 455–464, [Online]. Available: http://dl.acm.org/citation.cfm?id=776816.776871.

Ohrndorf, M., Pietsch, C., Kelter, U., Grunske, L., Kehrer, T., 2021. History-based model repair recommendations. ACM Trans. Softw. Eng. Methodol. 30 (2), http://dx.doi.org/10.1145/3419017.

Ohrndorf, M., Pietsch, C., Kelter, U., Kehrer, T., 2018. Revision: A tool for history-based model repair recommendations. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings. ICSE '18, Association for Computing Machinery, New York, NY, USA, pp. 105–108, [Online]. Available: http://dx.doi.org/10.1145/3183440.3183498.

Oliveira, P., Ferreira, A.L., Dias, D., Pereira, T., Monteiro, P., Machado, R.J., 2017. An analysis of the commonality and differences between ASPICE and ISO26262 in the context of software development. In: European Conference on Software Process Improvement. Springer, pp. 216–227.

Pohl, K., Weidenhaupt, K., Dömges, R., Haumer, P., Jarke, M., Klamma, R., 1999. PRIME—toward process-integrated modeling environments: 1. ACM Trans. Softw. Eng. Methodol. 8 (4), 343–410. http://dx.doi.org/10.1145/322993.322995.

Puissant, J.P., Van Der Straeten, R., Mens, T., 2015. Resolving model inconsistencies using automated regression planning. Softw. Syst. Model. 14 (1), 461–481.

Rahimi, M., Cleland-Huang, J., 2018. Evolving software trace links between requirements and source code. Empir. Softw. Eng. 23 (4), 2198–2231.

Ratiu, C.-C., Mayr-Dorn, C., Egyed, A., 2023. Defining and executing temporal constraints for evaluating engineering artifact compliance. arXiv:2312.13012.

Reder, A., Egyed, A., 2012. Incremental consistency checking for complex design rules and larger model changes. In: France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (Eds.), Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30-October 5, 2012. Proceedings. In: Lecture Notes in Computer Science, vol. 7590, Springer, pp. 202–218, [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33666-9_14.

Rempel, P., Mäder, P., 2016. Continuous assessment of software traceability. In: Dillon, L.K., Visser, W., Williams, L. (Eds.), Proc. of the 38th Int'l Conf. on Software Engineering. ICSE 2016, ACM, pp. 747–748, [Online]. Available: http://dx.doi.org/10.1145/2889160.2892657.

Rempel, P., Mäder, P., Kuschke, T., Cleland-Huang, J., 2014. Mind the gap: assessing the conformance of software traceability to relevant guidelines. In: Proc. of the 36th Int'l Conf. on Software Engineering. ICSE '14, ACM, pp. 943–954. http://dx.doi.org/10.1145/2568225.2568290.

Torres, W., Van den Brand, M.G., Serebrenik, A., 2021. A systematic literature review of cross-domain model consistency checking by model management tools. Softw. Syst. Model. 20 (3), 897–916.

Tröls, M.A., Marchezan, L., Mashkoor, A., Egyed, A., 2022. Instant and global consistency checking during collaborative engineering. Softw. Syst. Model. 1–27.

Van Beest, N., Kaldeli, E., Bulanov, P., Wortmann, J.C., Lazovik, A., 2014. Automated runtime repair of business processes. Inf. Syst. 39, 45–79.

Watkins, R., Neal, M., 1994. Why and how of requirements tracing. IEEE Softw. 11 (4), 104–106.

Winkler, D., Kathrein, L., Meixner, K., Staufer, P., Pauditz, M., Biffl, S., 2019. Towards a hybrid process model approach in production systems engineering. In: Walker, A., O'Connor, R.V., Messnarz, R. (Eds.), Systems, Software and Services Process Improvement. Springer International Publishing, Cham, pp. 339–354.

Zhao, X., Brun, Y., Osterweil, L.J., 2013. Supporting process undo and redo in software engineering decision making. In: Münch, J., Lan, J.A., Zhang, H. (Eds.), Proc. of the Int'l Conf. on Software and System Process. ICSSP '13, ACM, pp. 56–60. http://dx.doi.org/10.1145/2486046.2486057.

Zhao, X., Osterweil, L.J., 2012. An approach to modeling and supporting the rework process in refactoring. In: Jeffery, D.R., Raffo, D., Armbrust, O., Huang, L. (Eds.), Proc. of the 2012 Int'l Conf. on Software and System Process. IEEE Computer Society, pp. 110–119, [Online]. Available: http://dx.doi.org/10.1109/ICSSP.2012.6225953.

Christoph Mayr-Dorn Is a senior researcher at the Institute for Software Systems Engineering at the Johannes Kepler University Linz, Austria. He holds a Ph.D. in Computer Science from the Technical University Vienna. His current research interests include software process monitoring and mining, change impact assessment, and software engineering of cyber–physical production systems.

Cosmina-Cristina Ratiu Is a Ph.D. student at Johannes Kepler University and researcher at the Pro2Future GmbH. She received her Master's diploma in Computer Science at the same university, after completing a Bachelor's degree in Computer Science at Babes-Bolyai University in Cluj Napoca, Romania. She is currently involved in researching flexible guidance for software development processes.

Luciano Marchezan de Paula is currently a University Assistant at the Institute of Software Systems Engineering (ISSE) at the Johannes Kepler University Austria. He received his Ph.D. in Computer Science from the Johannes Kepler University Linz - Austria - in 2023. and his master's degree in Software Engineering from the Federal University of Pampa (Unipampa - Brazil). His research interests include Model-Driven Software Engineering, Automated Software Engineering, Software Reuse, and Empirical Software Engineering.

Felix Keplinger Finished his Master Degree in Software Engineering at the Johannes Kepler University Linz, Austria.

Alexander Egyed Is a Full Professor and Chair for Software-Intensive Systems at the Johannes Kepler University, Austria (JKU). He received a Doctorate degree from the University of Southern California, USA in 2000 and then worked for industry for many years before joining the University College London, UK in 2007 and JKU in 2008. He is most recognized for his work on software and systems design – particularly on variability, consistency, and traceability.

Gala Walden: Is a process engineer in the Process, Methods & Tools department of Robert Bosch AG, Vienna and has multiple years of experience in function development.