2021 IEEE/ACM 43rd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER) | 978-1-6654-0140-1/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICSE-NIER52604.2021.00010

Inconsistency-tolerating guidance for software engineering processes

Christoph Mayr-Dorn, Roland Kretschmer, Alexander Egyed
Johannes Kepler University
Linz, Austria
firstname.lastname@jku.at

Ruben Heradio, David Fernandez-Amoros
Universidad Nacional de Educacion a Distancia
Madrid, Spain
rheradio|david@issi.uned.es

***Abstract*—Software processes, together with software quality assurance, focus on ensuring and attesting that the engineering processes result in the appropriate software quality. Complex processes and regulations (e.g., in safety-critical systems), time pressure, or coordination needs, often cause engineers to deviate from prescribed processes, producing a cascade of inconsistencies whose repair is typically troublesome. Accordingly, guidance is needed to help engineers to fix the inconsistencies and understand the implications of postponing inconsistency resolution until engineers reach a consensual agreement of the most convenient repair. To this end, we bring together techniques and methods from process engineering, model inconsistency checking, and formal methods. Preliminary evaluations with real industry data have demonstrated the ability of our early prototype to track process inconsistencies across time and the potential for automated repair.**

***Index Terms*—software engineering process, inconsistency, developer guidance, repair, constraints**

I. Introduction

Software processes together with software Quality Assurance (QA) focus on ensuring and attesting that the engineering processes result in the appropriate software quality. In safety-critical domains, various regulations, standards, and guidelines stipulate stringent traceability paths [1], but do not prescribe the corresponding detailed software engineering process. Research as early as the 90s identified rigid, active process enactment as detrimental to engineers' flexibility. Indeed, the current practice in industry is using semi-formal descriptions to specify processes [2], rather than rigidly enforced processes. Informal studies at our industry partners have revealed that engineers deviate temporarily from the intended process. Thus, there exists a tension between the need to follow regulations and the process on the one hand, and the need to be able to deviate on the other hand.

As regulations and processes are complex and potentially different for each project, it becomes infeasible for QA engineers to manually track deviations and guide engineers back in a timely manner. The problem is then how to provide automated process guidance to engineers in the presence of violated process and quality constraints.

This is non-trivial as a process deviation typically affects not only a single engineer but has impact on other engineers as well. Without awareness of process deviation and its impact on others, a deviation may go unnoticed or not be completely corrected. Inconsistencies then may propagate to subsequent process steps and their engineering artifacts, ultimately leading to costly rework at a later time or lower software quality.

We argue that guidance needs to come in two forms. First, supporting engineers in determining which activities are needed to return to a consistent process state, and second, identifying what are the affected process steps and their responsible engineers. The latter aspect is especially important as inconsistencies may not be fixable immediately. For example, a consensual agreement among the implicated engineers on the best repair alternative may not have been achieved yet. A guidance mechanism will then notify other engineers about the extent to which their work is affected by an inconsistency and thus might incur rework in case they continue. The same mechanism can be employed for what-if scenarios, e.g., helping to estimate the impact of starting a step too early.

This paper takes our current inconsistency-tolerating process engine [3] as a basis. This engine signals any deviation from the process without enforcing its immediate correction. However, it neither estimates the deviation impact nor assists engineers to fix it. To overcome these limitations and support the repair delay when needed, we propose to encode the process instance as a Boolean formula that is then examined with formal method techniques to identify and isolate the affected process steps and artifacts. Our new approach aggregates multiple repair templates that suggest localized fixes into repair plans, which are checked for their overall consistency, and then presented to the engineer for enactment.

Ultimately we expect such guidance to reduce the time in an inconsistent state, and thus the potential for unintended inconsistency propagation, which in turn reduces the amount of errors and subsequent rework that cascades beyond an initial deviation.

A preliminary evaluation with real process instances at our industry partner has demonstrated our prototype's basic ability to detect when process deviations occur and when they are (manually) fixed, thereby highlighting the significant duration process steps remain in an inconsistent state.

II. Motivating Example

Our industry partner ACME-ATC (anonymized) is a world-leading voice communication provider for air-traffic control and command-control centers.

978-1-6654-0140-1/20/$31.00 ©2021 IEEE
DOI 10.1109/ICSE-NIER52604.2021.00010

Engineers are organized in teams dedicated to specific subsystems. Engineers' tasks are arranged into work packages and Sub-Work Packages (SubWPs). Figure 1 shows the process ACME-ATC enforces to accomplish a SubWP, starting with High-Level Requirements (HLReq) and design Specifications (HLSpec), and concluding with the requirement implementation and the successful execution of the corresponding test cases. Once that HLReq and HLSpec are reviewed, HLReq is refined into Low-Level Requirements (LLReq), causing updates to the Low-Level design Specifications (LLSpec). The implementation starts after the trace links between HLReq and LLReq, and between LLReq and LLSpec are reviewed.

Even in this straightforward process, multiple deviations may occur. For example, when HLReq and HLSpec reviewing (S1) is taking more time than expected, engineers may decide to prematurely start refining HLReq into LLReq (S2) before the review results are completed to gain time. Afterwards, the set of engineers responsible for updating the design definitions (S3) start working on LLReq, ignoring that they are incomplete. Suppose that a review (S1) highlights some shortcomings of HLReq, which cascade to LLReq (S2) and thus onto design decisions (S3). Such an impact propagation across multiple engineers or even teams may occur with a delay (potentially requiring rework in S2 and S3) or may not occur at all (potentially leading to bugs).

Feedback between process steps also may lead to inconsistencies. For example, imagine that engineers detect ambiguity in HLReq when they are updating LLSpec (S3). As a result, S1 and S2 artifacts need to be partially reworked with further impact in already started ulterior steps.

Such a simplified process and artifact structure already comes with a large number of control and data-flow dependencies. Obtaining an accurate set of affected artifacts, process steps, and their responsible engineers, is therefore vital to quickly identify who needs to coordinate in order to handle a process deviation (i.e., who should fix what and when).

III. Approach

Our approach consists of four main elements depicted in Figure 2. An inconsistency-tolerating *Passive Process Engine* compares the engineering activities with a prescribed process by analyzing changes in the artifacts (A). An *Incremental Deviation Checker* analyzes the process state and artifact changes (B) to highlight the violations that may have occurred. The detected deviations are made available via the *Process Dashboard* (C). Additionally, a Boolean formula encoding the process instance is passed to the *Impact Scope Identification* (D) for determining the inconsistency causes and then computing the affected steps and artifacts. This scope alerts engineers on what steps and artifacts may be subject to rework due to the process deviation (E). Also, the inconsistency causes are matched with *Repair Templates* in the *Repair Recommender* (F) for guiding engineers (G) to bring steps and artifacts back in line with process specifications and other (safety-related) regulations (H).

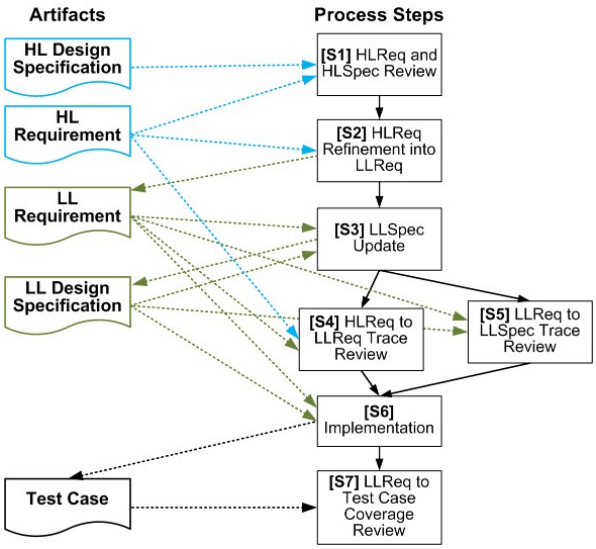

Fig. 1: Motivating scenario: ACME-ATC's simplified process model excerpt depicting a selection of artifacts (left) that are input to and output from (dashed lines: data flow) process steps (right, full lines: control flow).

A. Inconsistency-tolerating Process Execution

The initial version of our *Passive Process Engine* [3] tolerated limited deviations, such as violations of artifacts quality and traceability constraints. This paper presents a new version that also deals with deviations from the process step sequences. For example, when a step starts prematurely, or when some step output artifacts are updated after it is already marked as completed.

Artifact changes are used to trigger step progress (e.g., modifications on requirement status, new bug assignee, etc.), and decision nodes between steps (not shown in any Figures) are employed for synchronization purposes (e.g., for observing whether S4 and S5 in Figure 1 are completed before marking S6 as *Enabled*). The state diagram in Figure 3 depicts the correct transitions a step may have, together with its possible deviations. Accordingly, a premature start would happen when the step passes directly from *Available* to *Active*.

B. Incremental Deviation Checking

Engineering processes for safety-critical systems involve a plethora of diverse engineering artifacts where changes in one artifact affect the consistency of other artifacts. In a preliminary investigation at our industry partner, we studied >100 process instances that involved around 14,000 intertwined artifacts. Accordingly, a highly scalable inconsistency-checking mechanism is needed.

Our approach satisfies this demanded scalability by splitting any process or artifact modifications into atomic changes. Then, the *Incremental Deviation Checker* incrementally examines the atomic changes, focusing exclusively on the process model areas affected by the changes. For example, as the status

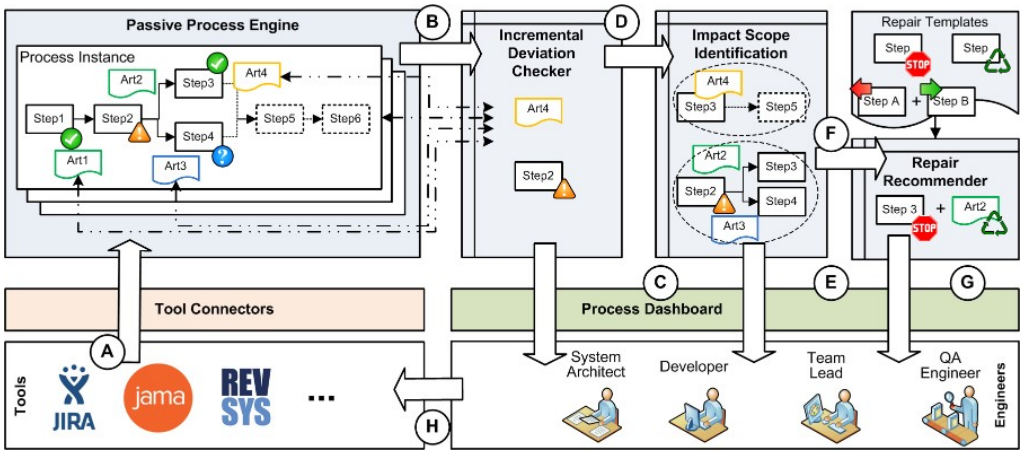

Fig. 2: Architectural overview: the blue boxes represent our approach's fundamental components, which monitor the process activities, detect possible deviations, inform engineers on their impact, and propose alternative ways to fix them.

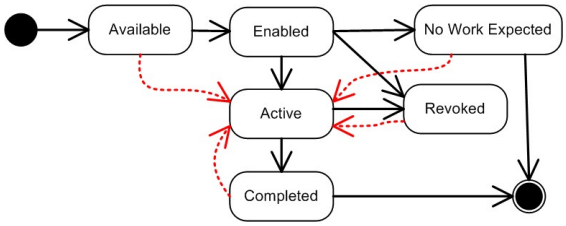

Fig. 3: Step state transitions: desirable transitions (full lines) and inconsistency inducing transitions (red, dotted lines).

property of a HL Requirement changes, only the constraints in tasks S1, S2, or S3 that make indeed use of that property need to be re-evaluated rather than all constraints in the process.

C. Inconsistency Isolation

We use a Boolean logic engine to compute the corresponding impact scope. For example, when the change of a HL Requirement status property leads to a constraint violation, we identify which other constraints make use of any of this artifact's properties and potentially flag these constraints as impacted. To calculate this scope, *Impact Scope Identification* first encodes the process model as a Boolean formula using a procedure similar to Kuhlmann et al.'s [4] translation of UML/OCL models into Boolean logic. Identifying the root of an inconsistency and its potential fixes is then a matter of identifying Minimal Unsatisfiable Sets (MUSes) and Minimal Correction Sets (MCSes), respectively [5]. A MUS is a subset of the formula constraints that is both unsatisfiable and cannot be made smaller without becoming satisfiable. A MCS is a subset of the constraints, whose removal makes the formula satisfiable and cannot be made smaller without losing its correcting capability. The inconsistency low-level impact boundary is determined from the union of all MUSes [6], which is then translated back into properties of the process and/or artifacts.

MUSes are typically computed by repeatedly calling a SAT-solver [5], [7]. Although our engine currently follows this procedure, we plan to explore alternative knowledge compilation techniques for achieving higher scalability, such as Binary Decision Diagrams (BDDs) [8] or Sentential Decision Diagrams (SDDs) [9]. In this regard, we have already demonstrated the efficient synthesis of BDDs even for very large engineering models [10].

D. Inconsistency Repair Generation

The *Repair Recommender* component supports repairs at the process level (e.g., repeating a step, waiting for a step, undoing step changes, etc.) and at the artifact level (e.g., modifying a property value, creating properties, establishing traces, etc.). Repair trees consist of repair instances, which range from fine-granular fixes (e.g., suggesting specific values or aborting a step) to coarse-granular recommendations (e.g., switching to another process step or creating a new artifact). Our repair component is extensible, thus supporting repairs tailored to different artifacts; currently it produces recommendations for UML/OCL models. In the HL requirement status property example: one possible repair tree would suggest to (i) set the property to a valid value and (ii) to postpone execution of Step S4, respectively, if already executed, to repeat it.

IV. Research Status and Preliminary Results

At the current stage, our prototype [11] detects deviations from the process sequence, e.g., if a step has started before the previous one is completed or its quality constraints are fulfilled. We applied the prototype to the change history [12] of 109 complete SubWP processes, involving multiple artifacts from our industry partner. A SubWP roughly corresponds to steps S2, S3, S6, and S7 in Figure 1, with each step involving up to four quality constraints that check properties and traces among requirements, design specs, and test cases.

Table I details in how many process instances a step was subject to a preceding step starting too early (due to still being

INCOMPLETE or due to having *QA_UNFULFILLED*). Row two, for example, reports that in 13 process instances, step S2 had quality constraints unfulfilled when S3 or S6 or S7 started. In 10 instances, S2's quality constraints were repaired after 1,075 hours on average.

Two aspects draw attention: first, the long average duration until the inconsistencies were fixed, mostly due to coordination needs (as determined by manual process inspection and interviews with QA engineers), but also because the kind of automated guidance this paper proposes was not available. Second, no *Sx_INCOMPLETE* inconsistencies were ever repaired. The inspection of these cases revealed that engineers simply forgot to fill out specific checkboxes in the JIRA ticket used for manual progress tracking. This is another excellent example of the potential inconsistency isolation and repair support envisioned: informing the engineer that the checkbox could be one of the reasons the process step is in an inconsistent state and provide a simple automated repair. At the same time, the prototype would also identify the artifacts involved in the previous step to inform the engineer what changes to expect if the previous step is indeed not completed yet.

	All	Fixed	Open	Avg.Dur. (hours)
S2_INCOMPLETE	8	0	8	–
S2_QA_UNFULFILLED	13	10	3	1,075
S3_INCOMPLETE	8	0	8	–
S3_QA_UNFULFILLED	38	34	4	1,124
S6_STEP_INCOMPLETE	9	0	9	–
S6_QA_UNFULFILLED	14	10	4	2,085
S7_INCOMPLETE	0	0	0	–
S7_QA_UNFULFILLED	0	0	0	–

TABLE I: Preliminary evaluation: amount and duration of two types of high-level process inconsistencies due to prematurely starting steps.

V. Related Work

Most related work focuses on formally verifying processes [13] rather than attempting to fix them. Fixing is limited to achieving sound process models but is not applicable to instances as we aim for. The few, recent approaches that address inconsistencies and their repair exhibit limited expressiveness for specifying constraints: LTL for expressing task and event constraints [14], [15] or Mixed-Integer Programming for determining runtime compliance of task and resource allocation [16]. However, software engineering processes are artifact intensive (e.g., requirements, models, and code) and collaboration intensive. Other related work includes Maggi et al. [14], which detects inconsistencies 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. Kumar et al. [16] calculate repair plans but do not address continued reasoning support in the presence of inconsistencies. VanBeest et al. [17] apply AI-centric planning for process repair. Their approach then generates a single repair plan for automatic execution immediately after detecting an inconsistency.

Research in the 90s resulted in a number of approaches. Step-centric modeling and active execution frameworks [18]–[22] determine which steps may be done at any given moment, automatically executing them where possible. While such research supports detailed guidance, deviations from the prescribed process are not well supported. In contrast, systems utilizing ECA rules or pre- and post-conditions [23], [24] provide significant freedom of action to the engineer but offer limited guidance.

Our approach relies on detecting inconsistencies so as to be able to propose repairs for violations in the execution of processes. Most recent work (e.g., [25]–[30]) allows checking consistency of models from arbitrary pre-defined meta-models, which enables us to adapt their techniques for our process consistency checking. However, they do not provide full repair strategies for resolving detected inconsistencies, i.e., the actions (and their order) available to an engineer to return to a consistent state when a process is violated.

Providing repairs for inconsistencies in models is an active field of research and is used for re-establishing process conformance in our approach. Taentzer et al. [31] proposed to repair inconsistent models w.r.t. their metamodels. They relied on the model change history, thus reducing the amount of possible repairs. Similarly, Ohrndorf et al. [32] use an initial change to propose repairs for arising inconsistencies. Puissant et al. [33] proposed a planning technique to generate repair plans for inconsistencies while aiming at a fast computation of repairs without assessing the relevance of the repair plans. Kretschmer et al. [34] provide a technique for obtaining repairs based on elements already present in the model and values obtained with the help of generator functions.

Work by Nohrer et al. [6], [35] is one of the few attempts to isolate inconsistencies to support correct reasoning on models. Their approach, however, does not ensure correct reasoning: there may be choices outside of the union of all MUSes affected by the inconsistency; if engineers make decisions about those choices, they will have to review them when the inconsistency is fixed in the future. Furthermore, MUSes union is computed by repeatedly calling a SAT-solver, which is computationally very expensive and does not scale for large models [5], [7].

This brief overview of related approaches shows how the needed models, techniques, and algorithms for inconsistency-tolerating process guidance are spread over –so far– rather disjoint research communities: software processes, model inconsistency checking, and formal methods.

VI. Conclusion

We have presented an approach for supporting engineers to deviate from processes and guide them back to a consistent state. Preliminary results have shown our prototype's ability to detect deviations in a process control flow and its involved artifacts. The evaluation has shown that, in practice, process deviations are rarely fixed immediately. This fact motivates inconsistency tolerating tools as well as guidance mechanisms.

Consequently, our next steps will focus on isolating inconsistencies to prevent engineers' rework when deviation fixing is postponed, and ultimately determining repair plans.

ACKNOWLEDGMENT

Part of this work was funded by the Austrian Science Fund (FWF) under the grant numbers P31989 and P29415-NBL, and by the state of Upper Austria LIT-2019-8-SEE-118.

VII. DATA AVAILABILITY

Data used in this paper are subject due to confidentiality agreements with out industry partner and therefore cannot be disclosed.

REFERENCES

[1] P. Rempel, P. Mäder, T. Kuschke, and J. Cleland-Huang, "Mind the gap: assessing the conformance of software traceability to relevant guidelines," in *International Conference on Software Engineering (ICSE)*. Hyderabad, India: ACM, 2014, pp. 943–954.

[2] P. Diebold and S. A. Scherr, "Software process models vs descriptions: What do practitioners use and need?" *Journal of Software: Evolution and Process*, vol. 29, no. 11, 2017.

[3] C. Mayr-Dorn, M. Vierhauser, S. Bichler, F. Keplinger, J. Cleland-Huang, A. Egyed, and T. Mehofer, "Supporting quality assurance with automated process-centric quality constraints checking," in *43rd International Conference on Software Engineering (ICSE 2021)*. IEEE / ACM, 2021, p. to appear.

[4] M. Kuhlmann and M. Gogolla, "From UML and OCL to relational logic and back," in *International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Innsbruck, Austria: Springer, 2012, pp. 415–431.

[5] M. Liffiton, A. Previti, A. Malik, and J. Marques-Silva, "Fast, flexible MUS enumeration," *Constraints*, vol. 21, no. 2, pp. 223–250, 2016.

[6] A. Nöhrer, A. Biere, and A. Egyed, "Managing sat inconsistencies with humus," in *International Workshop on Variability Modeling of Software-Intensive Systems (VaMoS)*. New York, NY, USA: ACM, 2012, pp. 83–91.

[7] J. Luo and S. Liu, "Accelerating MUS enumeration by inconsistency graph partitioning," *Science China Information Sciences*, vol. 62, no. 11, pp. 1–11, 2019.

[8] S. Zhou, J. Xiang, and W. E. Wong, "Reliability analysis of dynamic fault trees with spare gates using conditional binary decision diagrams," *Journal of Systems and Software*, vol. 170, pp. 1–23, 2020.

[9] S. Bova, "SDDs are exponentially more succinct than OBDDs," in *International Conference on Artificial Intelligence (AAAI)*. Phoenix, Arizona, USA: AAAI Press, 2016, pp. 929–935.

[10] R. Heradio, D. Fernández-Amorós, C. Mayr-Dorn, and A. Egyed, "Supporting the statistical analysis of variability models," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, J. M. Atlee, T. Bultan, and J. Whittle, Eds. IEEE / ACM, 2019, pp. 843–853. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00091

[11] C. Mayr-Dorn, S. Bichler, F. Keplinger, and A. Egyed, "Guiding engineers with the passive process engine environment," in *43rd International Conference on Software Engineering: Companion (ICSE-Companion)*. IEEE / ACM, 2021, p. to appear.

[12] C. Mayr-Dorn, M. Vierhauser, F. Keplinger, S. Bichler, and A. Egyed, "Timetracer: a tool for back in time traceability replaying," in *ICSE '20: 42nd International Conference on Software Engineering, Companion Volume, Seoul, South Korea, 27 June - 19 July, 2020*, G. Rothermel and D. Bae, Eds. ACM, 2020, pp. 33–36. [Online]. Available: https://doi.org/10.1145/3377812.3382141

[13] S. Morimoto, "A survey of formal verification for business process modeling," in *International Conference on Computational Science (ICCS)*. Kraków, Poland: Springer, 2008, pp. 514–522.

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

[15] F. M. Maggi, C. Di Francescomarino, M. Dumas, and C. Ghidini, "Predictive monitoring of business processes," in *International Conference on Advanced Information Systems Engineering (CAiSE)*. Thessaloniki, Greece: Springer, 2014, pp. 457–472.

[16] A. Kumar, W. Yao, and C.-H. Chu, "Flexible process compliance with semantic constraints using mixed-integer programming," *INFORMS Journal on Computing*, vol. 25, no. 3, pp. 543–559, 2013.

[17] N. Van Beest, E. Kaldeli, P. Bulanov, J. C. Wortmann, and A. Lazovik, "Automated runtime repair of business processes," *Information Systems*, vol. 39, pp. 45–79, 2014.

[18] C. Fernstrom, "Process Weaver: adding process support to UNIX," in *International Conference on the Software Process (ICSP)*. Berlin, Germany: IEEE, 1993, pp. 12–26.

[19] S. Bandinelli, E. D. Nitto, and A. Fuggetta, "Supporting cooperation in the SPADE-1 environment," *IEEE Transactions on Software Engineering*, vol. 22, no. 12, pp. 841–865, 1996.

[20] J. C. Grundy and J. G. Hosking, "Serendipity: Integrated environment support for process modelling, enactment and work coordination," *Automated Software Engineering*, vol. 5, no. 1, pp. 27–60, 1998.

[21] A. Geppert, D. Tombros, and K. R. Dittrich, "Defining the semantics of reactive components in event-driven workflow execution with event histories," *Information Systems*, vol. 23, no. 3–4, p. 235–252, 1998.

[22] K. Pohl, K. Weidenhaupt, R. Dömges, P. Haumer, M. Jarke, and R. Klamma, "PRIME – toward process-integrated modeling environments: 1," *ACM Transactions on Software Engineering and Methodology*, vol. 8, no. 4, p. 343–410, 1999.

[23] N. S. Barghouti, "Supporting cooperation in the Marvel process-centered SDE," *ACM SIGSOFT Software Engineering Notes*, vol. 17, no. 5, pp. 21–31, 1992.

[24] C. Montangero and V. Ambriola, *Software Process Modelling and Technology*. Research Studies Press Ltd., 1994, ch. OIKOS: Constructing Process-Centred SDEs, p. 131–151.

[25] A. Reder and A. Egyed, "Determining the cause of a design model inconsistency," *IEEE Transactions on Software Engineering*, vol. 39, no. 11, pp. 1531–1548, 2013.

[26] ——, "Incremental consistency checking for complex design rules and larger model changes," in *International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Innsbruck, Austria: Springer, 2012, pp. 202–218.

[27] J. Cabot, R. Clarisó, and D. Riera, "On the verification of uml/ocl class diagrams using constraint programming," *Journal of Systems and Software*, vol. 93, pp. 1–23, 2014.

[28] D.-E. Khelladi, R. Bendraou, S. Baarir, Y. Laurent, and M.-P. Gervais, "A framework to formally verify conformance of a software process to a software method," in *Annual ACM Symposium on Applied Computing (SAC)*. Salamanca, Spain: ACM, 2015, pp. 1518–1525.

[29] R. Jongeling, F. Ciccozzi, A. Cicchetti, and J. Carlson, "Lightweight consistency checking for agile model-based development in practice," *Journal of Object Technology*, vol. 18, no. 2, pp. 1–20, 2019.

[30] M. A. Tröls, A. Mashkoor, and A. Egyed, "Multifaceted consistency checking of collaborative engineering artifacts," in *International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. Munich, Germany: IEEE, 2019, pp. 278–287.

[31] G. Taentzer, M. Ohrndorf, Y. Lamo, and A. Rutle, "Change-preserving model repair," in *International Conference on Fundamental Approaches to Software Engineering (FASE)*. Uppsala, Sweden: Springer, 2017, pp. 283–299.

[32] M. Ohrndorf, C. Pietsch, U. Kelter, and T. Kehrer, "ReVision: a tool for history-based model repair recommendations," in *International Conference on Software Engineering (ICSE): Companion Volume*. Gothenburg, Sweden: IEEE, 2018, pp. 105–108.

[33] J. P. Puissant, R. Van Der Straeten, and T. Mens, "Resolving model inconsistencies using automated regression planning," *Software & Systems Modeling*, vol. 14, no. 1, pp. 461–481, 2015.

[34] R. Kretschmer, D. E. Khelladi, A. Demuth, R. E. Lopez-Herrejon, and A. Egyed, "From abstract to concrete repairs of model inconsistencies: An automated approach," in *Asia-Pacific Conference on Software Engineering (APSEC)*. Nanjing, China: IEEE, 2017, pp. 456–465.

[35] A. Nöhrer and A. Egyed, "C2O configurator: a tool for guided decision-making," *Automated Software Engineering*, vol. 20, no. 2, pp. 265–296, 2013.