FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Department Informatik

Technical Reports / ISSN 2191-5008

Klaus-Benedikt Schultis, Christoph Elsner, and Daniel Lohmann

Architecture-Violation Management for Internal Software Ecosystems: An Industry Case Study

Technical Report CS-2016-02

March 2016

Please cite as:

Klaus-Benedikt Schultis, Christoph Elsner, and Daniel Lohmann, “Architecture-Violation Management for Internal Software Ecosystems: An Industry Case Study,” Friedrich-Alexander-Universität Erlangen-Nürnberg, Dept. of Computer Science, Technical Reports, CS-2016-02, March 2016.

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik

Martensstr. 3 · 91058 Erlangen · Germany

www.cs.fau.de

Architecture-Violation Management for Internal Software Ecosystems: An Industry Case Study

Klaus-Benedikt Schultis, Christoph Elsner, and Daniel Lohmann
Distributed Systems and Operating Systems
Dept. of Computer Science, University of Erlangen, Germany
{klaus-benedikt.schultis,christoph.elsner}@siemens.com, lohmann@cs.fau.de

***Abstract*—Large-scale intra-organizational, yet decentralized software projects that involve various self-contained organizational units require architecture guidelines to coordinate development. Tool support allows for managing architecture-guideline violations to ensure software quality. However, the decentralized development across units results in significant violation-management hurdles that must be considered.**

Derived from our previous research, we have elaborated a set of capabilities required to manage guideline violations within two of these large-scale software projects at Siemens. Their main purpose is process support for resolving violations, aiming to reduce the architects' and developers' effort required to handle them. We developed a prototype that implements the capabilities and conducted a qualitative case study on their usefulness, involving 9 experts from our study systems. Our capabilities are considered as very important and reveal great potential to ease violation management for large-scale software engineering.

***Index Terms*—Software ecosystem, software product line, decentralized software engineering, architecture-violation management, technical-debt management, industry case study**

I. Introduction

Large-scale organizations, such as Siemens, develop a broad field of products for varying domains. Organizational-wide reuse of software across products, even across domains, gives these organizations a competitive advantage. This involves large-scale reuse approaches where (unlike traditional product-line approaches [1] that are commonly established within distinct organizational units) software is developed in a decentralized manner by several internal, yet self-contained organizational units. Those units are separate profit centers with own business objectives, organizational independent with own product management, and have to a wide extent autonomous processes and software-engineering life cycles. Thus, the view on the organizational structure moves from strict hierarchies, as it is common in product-line engineering [1], towards more decentralized topologies. We define those software systems as *internal software ecosystems (ISECOs)* [2].

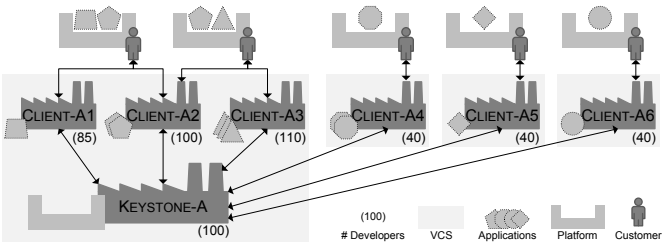

Fig. 1. A simplified illustration of ISECO-A.

We are investigating two of our largest ISECOs at Siemens. Both study systems, ISECO-A and ISECO-B, have been under development for more than 10 years and involve about 500 and 950 active developers, respectively. They comprise a keystone unit that provides a platform and multiple client units that build one to a handful large applications upon it (see Fig. 1). Units partially employ separated version control systems (VCSs).

Within such a decentralized environment, traditional process-centric coordination mechanisms become increasingly inefficient. Rather, architecture guidelines are required to coordinate cross-organizational development [2]. Besides defining architecture guidelines, it is equally important to control guideline compliance at an ecosystem-wide level. However, feature and schedule pressure regularly result in intentional and unintentional violations by several units [2].

Previously, we reported on collaboration practices among units and resulting architecture challenges for ISECO-A and ISECO-B [2]. We identified the management of architecture violations as the key challenge—both ISECOs have to deal with *thousands* of architecture violations, that have accumulated over time, which implies significantly increased development and maintenance

TABLE I
TAGGED PLATFORM APIS (LEFT), DOCUMENTED PRE-CONDITIONS (MIDDLE), AND RECOMMENDED FACTORY USAGE (RIGHT)

```
/// <summary>...</summary>
/// <apiflag>API:PUBLIC</apiflag>
public interface ILayoutRepository {...}


/// <summary>...</summary>
/// <apiflag>API:NON-PUBLIC</apiflag>
public interface INavigationSetupOperations {...}
```

```
/// <summary>...</summary>
/// <param name="howOften">
/// Specifies the number of repetitions.
/// Value must be greater than 0.
/// </param>
/// <remarks>The pipeline must be activated
/// first <see cref="Activate"/>.</remarks>
public void RunPipeline(int howOften) {...}
```

```
public class Options {
    internal Options() {...}
}
public static class OptionsFactory {
    static Options CreateOptions() {
        return new Options();
    }
}
```

costs [2]. In this paper, based on our previous research, we outline a set of *violation-management hurdles* (e.g., late feedback) for our ISECOs that result in high violation-handling effort. High effort, in turn, results in even more violations that are not handled—the architecture ages [2].

For that reason, we propose the TRAVIM approach: A set of *violation-management capabilities* (from integrated tool support to automated assignments of resolution tasks) that tackle the identified hurdles. Their main purpose is process support for resolving violations, aiming to reduce the architects' and developers' effort required to handle them. We present a prototype that realizes the capabilities for our ISECOs. Using the prototype, we conduct a case study on the capabilities' usefulness to overcome the hurdles, involving 9 experts from our ISECOs with an average professional experience of about 17 years. All of them consider that the capabilities reveal great potential to ease violation management. We are currently planning our prototype's industrial roll out for our ISECOs.

Our major contributions are (1) the conceptual elaboration of violation-management capabilities for ISECOs, (2) their concrete realization for our ISECOs, and (3) a thorough analysis of the capabilities' usefulness. As many of the identified hurdles refer to general software-engineering problems that are likely to appear also in other mid- to large-scale projects, we think that the elaborated capabilities can contribute to the development and improvement of violation-management tools in general, in particular for projects that face similar hurdles.

The paper is laid out as follows: In Section II, we outline example guidelines. In Section III, we present the hurdles, the elaborated capabilities to address them, and the capabilities' realization for our ISECOs. In Section IV, we outline our case-study research method and results. In Section V, we discuss the generality of our approach and limitations. In Section VI, we provide an overview on related work. In Section VII, we conclude the paper.

II. ARCHITECTURE GUIDELINES

ISECOs require architecture guidelines to coordinate cross-organizational development. Those guidelines relate to a variety of architecture concerns, including dependency management, interface stability, or guarantee of software qualities across the ecosystem [2]. Often, such guidelines are established problem-driven and they are specific to the project. Compliance is checked manually by reviews and fully automatically by static-analysis and testing tools. Since manual reviews are costly and time-consuming, both ISECOs spend considerable efforts to automate the detection of architecture violations, frequently by highly-customized or homegrown tools that partially rely on source-code annotations. Automated checks are executed decentralized per VCS with different frequencies (e.g., per commit, during the nightly build, once a week) at various phases of the development processes, analyzing only sources of units that work on the respective VCS (see Fig. 1). Below, we outline example guidelines of our ISECOs:

1) Do Not Use Platform-Internal APIs: The clients' schedule pressure required both keystones to open the platform in early stages and to expose a vast amount of APIs publicly accessible. This led to thousands of undesired dependencies between client applications and platform APIs [2]. Over time, the keystone of ISECO-A explicitly marked APIs either as `public` for client usage or as `non-public` for platform-internal usage only, using a flag in the APIs' documentation comments (see Table I (left)). Clients must only use APIs that are tagged as `public`. Compliance is checked fully automatically by a custom guideline for the static-analysis tool FxCop [3]. Additional dependencies are not allowed. Already existing ones are to be resolved incrementally.

2) API Design Guidelines: Platform APIs must be of high quality to reduce the client-side impact of platform changes. For ISECO-B, the keystone established guidelines for designing high-quality APIs that follow two design principles: First, make APIs easy to find, to understand, and to use. Second, make APIs evolution ready and backward compatible. An example for the former principle is the guideline *Do Document Pre-Conditions* to communicate the non-obvious (see Table I (middle)), an example for the latter one is the guideline

Do Prefer Factories Over Constructors to maximize information hiding (see Table I (right)). Platform developers are trained. Compliance is checked manually by reviews and fully automatically by custom guidelines for the static-analysis tools FxCop [3] and NDepend [4].

3) Behavioral Breaking Changes: The keystone does not always know how (e.g., with regard to assumptions on external qualities or execution orders) platform APIs are used. Hence, the client-side impact of behavioral changes cannot always be assessed [2]. For both ISECOs, clients support the keystone in testing APIs according their actual usage. They deliver test cases that are integrated into the keystone's test suite to detect behavioral changes. In this regard, guidelines are implemented by dedicated test cases. Siemens-internal test frameworks are used to fully automatically check for compliance. Detected changes are handled by the keystone and affected clients.

III. TRAVIM APPROACH

We propose the TRACEABLE ARCHITECTURE-VIOLATION MANAGEMENT (TRAVIM) approach: A collection of 6 capabilities for managing architecture violations within ISECOs [5]. The capabilities' main purpose is process support for resolving violations. They tackle 6 major violation-management hurdles for ISECOs. Those hurdles imply high violation-handling effort—high effort, in turn, results in an increasing number of violations over time [2]. Most hurdles (*Hurdle*$_3$ to *Hurdle*$_6$) were explicitly identified in our previous research [2]. *Hurdle*$_1$ and *Hurdle*$_2$ were not explicitly mentioned. However, in this work, we identified the resolution of both as prerequisites to address the other hurdles, and therefore included them as implicit hurdles when developing the TRAVIM approach.

The TRAVIM approach targets decentralized violation management per VCS (see Fig. 1). In a nutshell, we process results of employed compliance-checking measures to create and track issues for identified violations. Results of automated measures are processed[1] every night. We track only violations already published to the VCS. We use separate issue databases per VCS to allow ecosystem partners to protect confidential information. Developers and architects create, find, and handle violation issues using tailored dashboards.

Below, for each capability, we present (1) the identified hurdle, (2) the derived capability to tackle it, and (3) the concrete realization of the capability for our ISECOs.

[1]The report-files' locations of employed measures must be provided in a configuration. We process updated reports. We do not execute checks.

A. Integrated Approach

1) Hurdle$_1$*:* Within our ISECOs, guideline documents are spread across repositories. Moreover, compliance is checked by heterogeneous measures at various phases of the development processes. The heterogeneity of tools and methods increases violation-handling effort.

2) Capability$_1$*:* Integrate all employed compliance-checking measures into a single data model and provide explicit and consistent violation-handling processes. Integrate guideline information. Integrate the approach into employed development tools and processes.

3) Capability$_1$ *Realization:* Our ISECOs employ a range of *static-analysis tools*[2] and *test frameworks* to fully automatically check for guideline compliance, often homegrown or highly-customized ones. They also apply *manual reviews* as expert knowledge is frequently required. We process identified violations, convert violation information to a uniform data format, and store them in a database. Expected information is reduced to the essentials required to track a violation: (1) The *guideline* that is violated. (2) The *item* that violates the guideline. For manual reviews and static-analysis tools, an item is specified by a file and a line. For test frameworks, an item is specified by a set of files. The items' specification by concrete files and lines is intended as entry point for handling violations. For manual reviews, this information must be specified by the user. With regard to automated measures, required information is extracted from reports. The static-analysis tools employed by our ISECOs usually directly map violations to files and lines. If not, we apply heuristics to determine required information, such as using the first line of a method if reports provide only its signature. For test frameworks, guidelines are implemented by dedicated tests. Reports always include information on failed tests that represent the guidelines. In addition, we use a configuration that relates tests to items.

Moreover, we integrate into employed tools and processes. We provide dashboards that are seamlessly integrated into the development environment (see Fig. 2). They deliver a single user interface for architects and developers to handle violations. For each violation, we provide relevant information including the item, the guideline, and the developer the resolution task is assigned to. In addition to violations, we formalize and integrate corresponding guidelines and underlying requirements. For guidelines, we provide information such

[2]In the remaining paper, we use the terms static-analysis tool and test framework for fully automated tools.

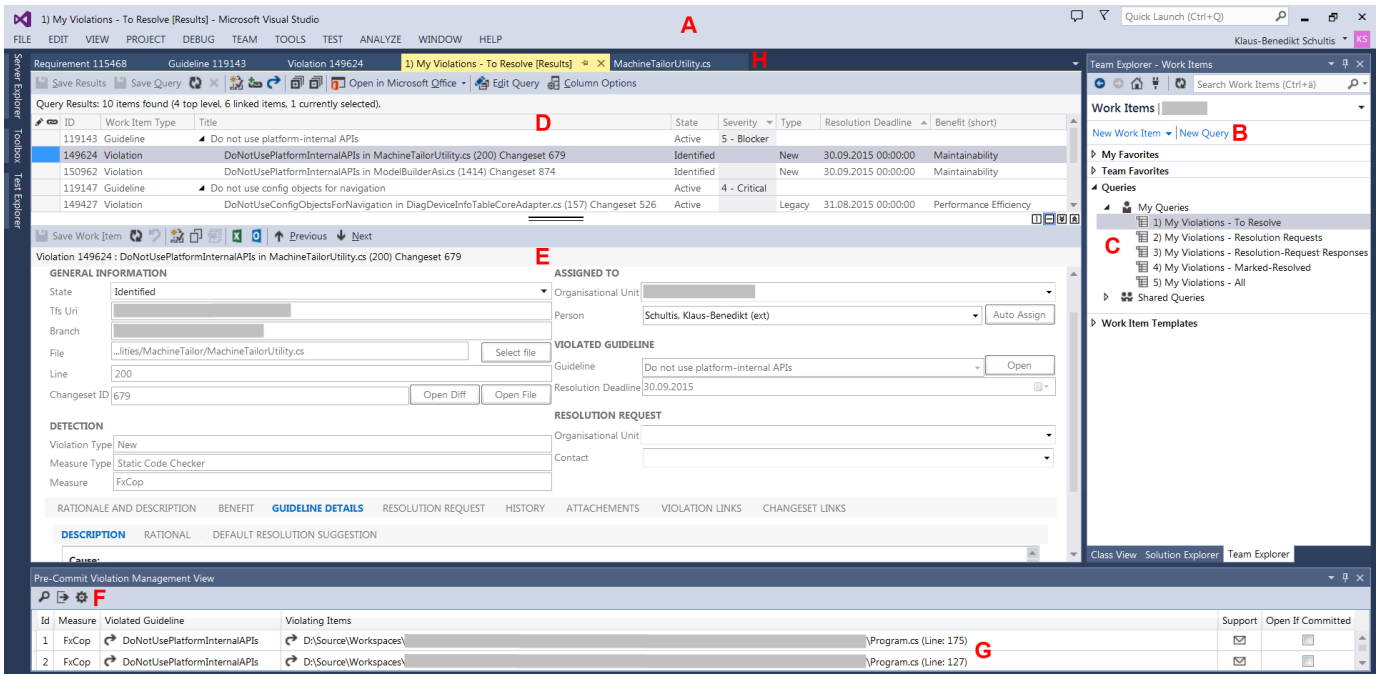

Fig. 2. Our dashboards are integrated into Visual Studio (A). They allow users to manually create violation issues (B), to find tracked ones by predefined and custom queries using a database query language (C), to display queried violations with guidelines as parent elements (D), and to review and update information of selected issues while providing violation-handling functionality (E). They also comprise a pre-commit view (F) to run checks on developer machines for handling violations in local changes (G). Just as source files, issues can be opened in separate tabs, too (H).

as descriptions, resolution strategies, and priorities. Items are linked to violations, violations are linked to guidelines, and guidelines are linked to requirements. By use of navigation buttons, users can quickly switch between these artifacts with only one user action and without exiting their development environment. For example, users may directly open a causing source file and line out of the violation representation to resolve the violation.

Violation-handling processes are modeled by means of workflows, comprising states and transitions between states. Once violation issues are created, they flow through the workflow taking one of the states. Transitions are triggered manually through the dashboards depending on user actions or automatically by the system. Fig. 3 shows the manual workflow for ISECO-A and ISECO-B. The automated workflow is slightly different: Our system enters the state *Resolved* if automated checks do not detect a violation anymore. Likewise, the state *Active* is reentered if a violation that was (marked as) resolved is detected by a subsequent automated check.

B. Traceable Resolution Processes

1) Hurdle$_2$: Automated checks deliver static and temporally-dependent snapshots of currently existing violations within a software system, which makes it difficult to continuously assist and track the resolution process of individual violations.

2) Capability$_2$: Continuously trace violations through software evolution in order to assist and track the resolution process of each individual violation throughout the life cycle.

3) Capability$_2$ Realization: We map current snapshots to previously identified violations already tracked in our database in order to create issues for additional violations and to update issues of already tracked ones. In order to map violations, we uniquely recognize violations over time.

For test frameworks, violations are uniquely identified by the guideline and the item. As discussed above, each guideline is implemented by a specific test case that either fails or succeeds. Hence, information required to map violations is available.

For static-analysis tools, violations are uniquely identified by the guideline, the commit where the violating item changed most recently before the previous analysis, and the committed item. For each violation the original commit and the committed item are identified as follows: Based on analysis reports for the current source-code version we know the violated guideline as well as the file

and line that violate the guideline. Querying the VCS, we trace the file's history *across branches* back to the commit where the given line has been added or edited. Along the descending history, we map the line given in the current report to the corresponding line of each version in order to check if it has been changed. In the process, we ignore white-space changes, ignore pure merges resulting in new versions, and consider moved and renamed files.

C. Temporal Differentiation

1) Hurdle$_3$: Usually, guidelines are not fully defined in initial project phases, but evolve progressively over time [2]. This requires to deal with *thousands* of already existing violations that overwhelm developers when guidelines are adopted, calling for pragmatism [2].

2) Capability$_3$: Differentiate between *Legacy Violations* and *New Violations*. *Legacy Violations* are violations that do already exist before adopting a guideline. Trigger their resolution incrementally on demand. *New Violations* are violations that are created afterwards. Trigger their resolution instantly.

3) Capability$_3$ Realization: All violations identified during the initial run of automated checks are marked as *Legacy Violation*. From this point in time, additionally identified violations are marked as *New Violation*, which is possible since we trace violations through software evolution (see *Capability$_2$*). For manually created issues, users define the violation type. We mark a tracked *Legacy Violation* as *New Violation* if the specified line changes. If desired, architects can manually change the type, for example to perform refactorings. All violations are tracked (see *Capability$_1$*), which is essential to provide a profound base for technical-debt removal decisions. Resolution processes are triggered depended on the type: *Legacy Violations* are initially hidden from developers in order to not overwhelm them by thousands of violations. Architects manually trigger their resolution incrementally on demand, either for collections of violations or for single ones. From this point in time, they appear on developer dashboards. For *New Violations*, resolution processes are triggered automatically instantly after their identification. Developers are immediately notified on their dashboards.

D. Individual Violation-Handling Plans

1) Hurdle$_4$: Organizational units must build consensus on guidelines to be established [2]. Guidelines may imply varying costs and benefits for involved units, which results in long negotiation phases where violations are not handled [2].

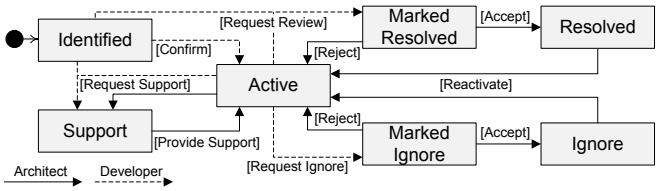

Fig. 3. The manual workflow for ISECO-A and ISECO-B. For simplification, not all possible transitions are depicted.

2) Capability$_4$: Allow definition of individual violation-handling strategies for each organizational unit according to individual capabilities and priorities. Trigger resolution processes accordingly.

3) Capability$_4$ Realization: We support the definition of individual violation-handling plans per guideline for each unit. They comprise *New-Violation Reaction Periods* and *Legacy-Violation Remediation Objectives*. In our ISECOs, such plans are decided on consensually in architecture boards involving the main architects of all units.

New-Violation Reaction Periods define the timeframe provided to an unit to resolve *New Violations*. For each issue, we automatically set resolution deadlines accordingly. We consider the following reaction periods: (1) Violations are not allowed. The resolution deadline is set to date of identification. (2) Violations are temporarily allowed but must be resolved within a defined timeframe (e.g., within the current release). The resolution deadline is set accordingly. (3) Violations are allowed. No resolution deadline is set, but violations are tracked to make them explicit.

Legacy-Violation Remediation Objectives define points along the project timeline along with selected *Legacy Violations* to be resolved or measurable objectives such as performance improvements to be achieved. Resolution deadlines are set manually by architects when triggering resolution tasks.

E. Automated Assignments & Pre-Commit Checks

1) Hurdle$_5$: Late feedback implies additional effort to get familiarized again with already completed tasks [2]. It proved difficult to resolve added violations that are not handled close to point in time where code is created [2].

2) Capability$_5$: Provide early feedback by automatically assigning resolution tasks to architects and developers for violations that are already published to the VCS. Assign tasks close to point in time where checks are executed and provide immediate notifications. In addition, allow for pre-commit checks on developer machines to avoid violations.

3) Capability$_5$ Realization: We process updated reports of automated checks during the nightly build. Violations are assigned to architects by assigning them to the architects' unit, using the item and a configuration that maps directories to units. The heuristic works well for our ISECOs where each unit owns a sub tree in the project's directory structure. For a violation detected by static-analysis tools, we determine the commit where the affected item changed most recently (see *Capability$_2$*). We assign the resolution task to the developer who performed the commit. Similarly, we allow users to initiate the assignment of tasks for manually detected violations. For test frameworks, suitable developers are determined by a configuration that relates test cases to items and developers. If desired, developers may reassign tasks or move them to a pool of tasks to be manually assigned by architects. We only assign tasks to developers for *New Violations* but not for *Legacy Violations*, which allows starting violation handling in a brown field. Based on the assignments, our tailored dashboards display only violations that are relevant for the respective developer or architect.

Moreover, we support pre-commit checks on developer machines to handle violations before publishing them to the VCS. Developers can configure target sources and measures to be checked. Using the dashboards, they can trigger the measures' execution and handle identified violations. Developers are informed about *New Violations* in their local changes only but not about *Legacy Violations*, having the following opportunities for action: (1) They may navigate between source files, violation details, and guideline details to resolve the violation. Its resolution can be confirmed by rerunning the checks. No violation issue is created. (2) They can request resolution support with predefined contacts. An e-mail is generated including the request message, the local changes, and optional attachments. No issue is created. (3) They can explicitly decide to commit affected files. A violation issue is created and optionally opened to enter additional information.

F. Collaborative Violation Handling

1) Hurdle$_6$: The handling of specific violations commonly lacks efficient processes, defined roles and responsibilities, and mechanisms to collaborate, in particular across organizational units [2].

2) Capability$_6$: Provide violation-handling processes that are tailored to the organizational and product context and support the collaborative resolution of violations with dedicated contacts accordingly. Track all collaborative actions and provide a full history for each violation.

3) Capability$_6$ Realization: In our ISECOs, developers may require resolution support, they need a permit for ignoring violations, and fixes of manually identified violations must be reviewed. Architects may require support from each other. We support the collaborative resolution of violations accordingly.

For each unit, we define a default contact responsible for requests. Contacts can be overwritten on guideline level. Users can initiate a request for a violation issue by selecting the corresponding workflow state (see Fig. 3). By default, the contact defined for the unit to which the violation is assigned to is preselected as receiver. Users can change the receiver to any person with a valid user account. After entering mandatory information (e.g., a rationale for ignoring a violation) users can submit the request. Receivers are immediately notified on their dashboards. Issues contain information relevant for providing support, performing reviews, and deciding on ignore-permit requests, including a full history of all actions already performed and a list of related commits.[3] Users respond by selecting the corresponding workflow state (see Fig. 3).

We target decentralized violation management per VCS. However, individual violations may require collaboration across VCSs, for example to resolve undesired dependencies between applications and platform APIs. We allow architects to request resolution support across VCSs. Violation issues are stored in separate databases for each VCS. By default, each database holds only issues assigned to units that work on the corresponding VCS. To enable collaboration by use of issues, we copy and continuously synchronize selected ones between databases. Exchange happens by clearly defined interfaces to control access to confidential data. Thus, we allow users to collaborate uniformly regardless of their distribution to VCSs.

G. Prototypical Implementation

Our prototype extends Microsoft's Team Foundation Server (TFS) [6], in particular its workflow-based issue tracker, which is already in use within our ISECOs. We developed custom issue types for violations and guidelines that define the violation-handling workflow and information to be tracked. We developed a set of parsers to process violation reports of automated measures employed by our ISECOs, such as FxCop [3], NDepend

[3]In our ISECOs, commits by convention must be linked to related tasks and issues. This allows providing review functionality out of the issue.

[4], and a compiler for a Siemens-internal domain-specific language. Our business logic uses the API of TFS to determine commits violations were created and to programmatically create, find, and update violation issues as part of the nightly build. Issues are stored in the data warehouse of the respective TFS. To collaborate across TFSs, we developed a trusted interface to be installed on each TFS and a trusted interface connector realized by a set of WCF services [7]. Our dashboards (see Fig. 2) integrate into Visual Studio [6] and extend the issue-tracker's user interface. The pre-commit view is realized as Visual Studio plug-in.

IV. Investigating The Capability's Usefulness

In the previous section, we elaborated capabilities to resolve the hurdles of violation management in ISECOs. By describing their realization and the implemented prototype for our ISECOs, we show that the hurdles in general may be tackled. In the following we give a more in-depth view on the capabilities' benefits from the perspective of practitioners to get an impression of the approach's feasibility. Therefore, we present a qualitative case study on the capabilities' usefulness. We investigate the following research question:

RQ: What is the capabilities' usefulness for managing architecture violations within ISECOs? How well do the capabilities tackle the identified hurdles?

We have investigated both ISECOs for a period of more than 3 years, starting with the case study on architecture challenges [2], continuing with the development of the TRAVIM approach to tackle selected challenges [5], and concluding with this investigation to evaluate the usefulness of the approach. We developed a case study protocol and structured our study based on the guidelines by Runeson and Höst [8].

A. Methodology

In this section, we depict the research setting we have investigated and the research method we have applied (see Fig. 4). The study involves 9 experts from our ISECOs. Their professional experience ranges from 6-30 years with an average of about 17 years. They are involved in our ISECOs for 1-13 years with an average of about 8 years. In a nutshell, they used the prototype to perform 8 violation-management tasks. We observed, surveyed, and interviewed them to understand the capabilities' usefulness. Below, we describe each study phase:

A) Case Study & Literature Review: We prepared a questionnaire and an interview guideline based on our previous study [2] and existing research on usefulness (e.g., [9], [10]).

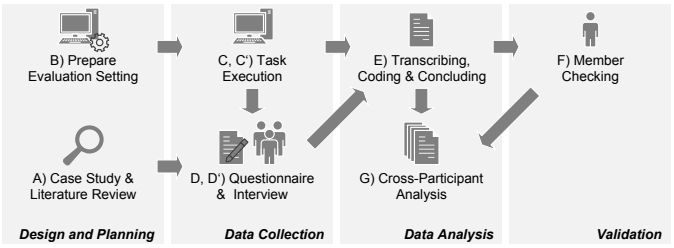

Fig. 4. Research method.

For the questionnaire, we phrased statements claiming the capabilities' usefulness to tackle the corresponding hurdle. For example: "Temporal differentiation is useful to deal with evolving guidelines that imply a range of already existing violations." Participants were asked to rate statements on a five point likert scale, from strongly agree to strongly disagree.

For the interview guideline, we prepared the following four open questions to access the capabilities' usefulness: What did you like? What did you dislike/should be improved? What are the opportunities when using this approach in daily business? What are the risks when using this approach in daily business?

B) Prepare Evaluation Setting: To minimize risks for our study systems, we reduced usage of their productive TFSs. We simulate a realistic evaluation environment for a randomly selected set of violations of ISECO-A: (1) We used all FxCop reports of ISECO-A as FxCop is its most employed compliance checking measure. In addition, we used the compliance-checking reports generated by a compiler for a Siemens-internal domain-specific language. We randomly selected 500 source files that contain at least one violation. (2) For each violation within the 500 files, we queried the TFS of ISECO-A to determine the commit the violating item changed most recently, the commit date, and the developer who made the commit. (3) We set up a TFS along with our extensions, committed the selected files, created the corresponding requirement, guideline and violation issues, and assigned resolution tasks to the determined units and developers. Violations where the item had changed within the previous 5 months were considered as *New Violations*. (3) For exemplification, we created issues for a set of violations identified by test frameworks and manual reviews, too. (4) Next, we committed a sub-project of ISECO-A that can be build separately to evaluate the pre-commit checks, employing FxCop to check for compliance to the already formalized

TABLE II
VIOLATION-HANDLING TASKS.

Developer Tasks	
Pre-Commit Checks	We instructed participants to execute pre-commit checks, to resolve* one detected violation, and to confirm its resolution by rerunning the checks. (In advance, we included violations to the local changes.)
View and Check Dashboard	We asked participants to view and check assigned violations, corresponding guidelines, and underlying requirements. This should be the initial use case for developers.
Resolve Violation	We asked participants to select one assigned violation, to view violation and guideline details, to resolve* the violation, to commit the fix, and to link the commit to the violation issue.
Request Support	Participants selected one assigned violation, formulated a request message, selected the predefined contact for their unit as receiver, and submitted the request. (The participant was predefined as contact.)
Architect Tasks	
Assign Legacy Violations	We asked participants to trigger the resolution of their unit's *Legacy Violations* with highest priority and to assign resolution tasks to the first researcher. They set a resolution deadline, too.
Provide Support	We instructed participants to open a received support request, to view the request message, to view the source files, and to provide support. (They answered the request they have previously sent to themselves.)
Review Violation Fix	Participants selected a received violation-fix–review request, viewed violation details, reviewed related commits, and accepted or rejected the fix. (In advance, we sent a request to the participant.)
Manual Check	We asked participants to review source files, to identify one violation to a given guideline, to manually create a violation issue, and to assign the issue to the developer who created the violation.

*Participants resolved easy-to-fix violations. Our evaluation targets process support.

guidelines. (5) We installed Visual Studio along with our extensions and FxCop on two laptops. (6) Each session targeted a particular unit. We cover the keystone and all core clients. Before each session, we randomly selected a set of developers of the respective unit and reassigned their violations to the first researcher.

C) Task Execution: We contacted 3 of the architects who participated in our previous study [2] and briefed them on the goal and process of our investigation. We clarified if they feel suitable to participate again and asked them for further participants. Thus, we got 6 architects for our main study: 1 keystone architect and 2 client architects of ISECO-A. 2 keystone architects and 1 client architect of ISECO-B. Half of them started as developer for our ISECOs, which was desired as we asked them to consider both perspectives.

We held an individual session with each participant, comprising the following steps: *(1) Briefing.* We started with a briefing on the objective of our study and our notion of violation management. *(2) Discuss Capabilities.* We extensively discussed the capabilities along the hurdles (about 45 minutes), in particular aspects not completely covered in subsequent tasks. *(3) Training.* We introduced our evaluation setting and trained the participant for about 30 minutes. *(4) Tasks.* Each participant performed 8 violation-handling tasks (see Table II). First, 4 developer tasks on the developer laptop using the first-researcher's developer account. Second, 4 architect tasks on the architect laptop using their own architect account. We used two laptops to support the role play. We instructed the participant to speak out loud performed actions—think aloud protocols are recommended to understand the process going on [11]. With the participant's consent, we used a screen cast tool to record the screen including all mouse movements and an audio track. The first and the second researcher observed the participant and took notes on additional observations not explicitly stated. The execution of all tasks lasted about 30 minutes on average.

D) Questionnaire & Interview: After completing all tasks, the participant completed the usefulness questionnaire. Finally, we performed the semi-structured interview [8] based on our interview guideline. The first researcher guided the interview, the second researcher took notes and participated in discussions. The interviews lasted about 30 minutes on average and were digitally recorded with the interviewee's consent. By employing observations, interviews, and questionnaires we achieve triangulation, which improves validity [8], [11].

C', D') Task Execution, Questionnaire & Interview: On advice of our participants, we were invited by the keystone of ISECO-A to present our work to the lead architect and to 2 integrators responsible for architecture compliance. Since the session's procedure was similar as described for *C)* and *D)* we dwell on differences only: We did not hold separate sessions but involved all 3 participants at once. Participants did not execute tasks, instead the first researcher presented the tasks by means of a live demo. Participants were not interviewed separately, instead we discussed the questions together.

TABLE III
USEFULNESS QUESTIONNAIRE RESULTS

Participant	Integrated Approach	Traceable Resolution Processes	Temporal Differentiation	Indi. Violation-Handling Plans	Automated Assignments	Pre-Commit Checks	Collaborative Violation Handling
ISECO-A_{A1}	(++)	(++)	(++)	(++)	(++)	(++)	(+)
ISECO-A_{A2}	(++)	(++)	(++)	(+)	(++)	(++)	(+)
ISECO-A_{A3}	(++)	(+)	(+)	(+)	(+)	(++)	(−)
ISECO-B_{A1}	(++)	(++)	(++)	(+)	(++)	(+)	(○)
ISECO-B_{A2}	(+)	(++)	(++)	(++)	(++)	(++)	(○)
ISECO-B_{A3}	(++)	(+)	(++)	(++)	(+)	(++)	(++)
ISECO-A_{A4}	(++)	(+)	(++)	(++)	(++)	(++)	(++)
ISECO-A_{I1}	(+)	(+)	(+)	(+)	(○)	(○)	(○)
ISECO-A_{I2}	(+)	(++)	(+)	(++)	(+)	(+)	(+)

(++) = Strongly Agree, (+) = Agree, (○) = Neutral, (−) = Disagree, (−−) = Strongly Disagree

E) Transcribing, Coding & Concluding: The first researcher transcribed the screen-cast and audio recordings, created codes for the capabilities, and coded the transcripts as described by Seaman [11]. This resulted in a total of 414 coded participant statements. Conclusions were drawn and summarized separately for each participant: This was an iterative process where the first researcher searched for patterns in the coded data, grouped the data accordingly, and analyzed related notes that were taken during the interview and observation. Results were carefully checked by the second researcher.

F) Member Checking: Member checking is a recommended method to validate conclusions [11]. We sent our conclusion summaries (about 1000 words on average) to the respective participant and sought for feedback. All participants agreed with the conclusions, only minor adaptations were suggested.

G) Cross-Participant Analysis: We performed a cross-participant analysis inspired by Eisenhardt [12]. Using the same technique as above [11], the first researcher created codes for the findings and coded the conclusion summaries. He arranged the coded data in a table where rows represent findings, columns represent participants, and cells are marked if a finding is confirmed by the respective participant. This technique supported us to identify cross-cutting findings that are confirmed by multiple data sources, which increases validity [8], [11]. Results were carefully reviewed by the other researchers.

B. Case-Study Results

Table III shows the result of the usefulness questionnaire. Our participants confirmed the validity of the identified hurdles and, overall, they rated the capabilities as very useful to tackle them. Below, we outline our major findings derived from the participants' comments and our analysis. We use the notation (x/y) to denote that a finding is confirmed by data of x out of y participants. For the quantification, we aggregate data of the 3 participants who were involved at once since, frequently, those participants agreed on topics that came up, and we do not want to double or even triple-count their voices. None of the findings is contradicted by any collected data.

1) Efficient Process Support: Feature and schedule pressure regularly result in violations to architecture guidelines—known as technical debt. Most participants highlighted the challenge to convince their managers to invest in its refactoring since refactoring costs are immediate whereas benefits are usually long-term and vague. This is particularly challenging for ISECOs where multiple managers of self-contained business focus on features their individual customers pay for. On the other hand, architecture compliance is even more important in such a decentralized environment to enable effective software engineering. Most participants confirmed that developers require efficient violation-handling support as soon as violations are created—it turned out to be very difficult to get the permission to refactor those architecture violations later on. They also confirmed that the effort required to handle violations is a decisive factor for developers to resolve them or not—if the effort is too high developers do not resolve violations. This is in line with our previous research on architecture challenges [2] and confirms the relevance of the TRAVIM approach.

2) Customized Tool Support: Most participants commented that important aspects regarding usefulness are

Finding 1: ISECOs require process support for handling violations (7/7) that reduces the developers' effort (7/7).

the full integration into the existing IDE, issue tracker, and VCS as well as the formalization and integration of violations and guidelines. They highlighted that the approach allows developers to efficiently resolve violations as developers can monitor violations and navigate to related guidelines and source files within their development environment. They consider integration as enabler for efficiently managing violations as developers can handle them while completing development tasks without significant interruptions and context switches.

Our ISECOs employ various compliance-checking measures, frequently customized or homegrown ones that are specific to the development context. Most participants said that the integration of all employed measures and the consistent handling of violations saves time and improves efficiency. Currently, developers have to deal with various tools, if there is tool support at all. Moreover, our participants emphasized the opportunity to track violations identified by reviews. Those are often of high severity and the approach provides a consistent medium, prioritization, and process for handling them.

Most participants stated that, for large-scale projects like ISECOs, it is worth the effort to develop and maintain a seamlessly-integrated custom tool for managing architecture violations—this significantly reduces violation-handling efforts.

Finding 2: ISECOs require customized violation-management tools (7/7) that integrate into the used IDE and tool chain (7/7) and that allow integration of a variety of project-specific compliance-checking measures (7/7).

3) Gated Check-Ins: Both ISECOs employ gated check-ins. Each commit triggers the execution of test suites and static-analysis tools on a server cluster to automatically check for compliance to selected high-priority guidelines. Commits are rejected in case of any violation—for including guidelines all *Legacy Violations* must be resolved, first.

Most participants highlighted the usefulness of the pre-commit checks, in particular in combination with the differentiation logic that informs developers only about added violations. They consider early feedback as essential to increase the developers' willingness for resolving violations. All of them recommended to integrate the checks into the commit process and to enforce their execution for each single commit. The opinion concerning the handling of violations was rather diverse: Half of them suggested to strictly reject each commit with any *New Violation* during the gated check-in—the temporal-differentiation logic allows immediate handling of *New Violations* to all guidelines (where checks are executable in an acceptable time) while *Legacy Violations* can be handled incrementally. Thus, *New Violations* can be completely avoided. The other participants preferred a voluntary handling of violations. They stated that by only warning developers, these may still decide whether to commit the violation. If not possible, developers are strictly forced to find their way around the compliance check, at worst by creating additional not detectable flaws. For our ISECOs, the degree of strictness would be decided separately by the lead architects of involved units.

Finding 3: Pre-commit checks must be combined with temporal-differentiation logic (6/7). They require commit-process integration (6/7) and architects should be able to choose between settings where developers must resolve *New Violations* or where they are allowed to commit them (6/7).

4) Automated Assignments: Violations cannot always be handled before committing them to the version control system. Schedule and business pressure may require violations, long-running checks cannot be executed during the gated check-in, and compliance cannot be enforced across the ecosystem. Hence, committed violations must be managed afterwards.

Most participants mentioned that the automated assignment of resolution tasks for *New Violations* during the nightly build facilitates and accelerates violation handling. They stated that, currently, resolution tasks need to be manually assigned and triggered. Violation reports are manually compared to determine added violations and suitable developers are manually identified and notified. Frequently, there is not enough time to consider all violations, which results in an increasing number of violations over time eroding the software. Our participants highlighted that the automated-assignment feature saves time. One of them even stated that the feature would save him up to 30 minutes each day which he spends for triggering resolution tasks. Furthermore, they emphasized that completeness is ensured since all detected violations

are handled. Finally, they pointed out that automated assignments are required to provide early developer feedback. All participants consider early feedback as basic requirement for managing violations—experience has shown that late feedback significantly hampers their resolution.

Finding 4: Early developer feedback is a basic requirement for managing violations within ISECOs $(7/7)$. Automated assignments save time $(5/7)$, ensure completeness $(4/7)$, and are required to provide early developer feedback (5/7).

5) Team Structures: In our ISECOs, developers are arranged in development teams with dedicated team architects. Our participants commented that most violations can be handled within teams and, usually, developers primary need to interact with their team members for resolving them. They recommended to allow modeling of roles, responsibilities, and team structures within the violation-management tool and to preselect contacts for developer requests accordingly. This would also increase reusability in other organizational contexts. Default contacts, who are actually defined for our ISECOs, are important for violations that cannot be handled within teams but should not be preselected for each request. The current lack of team structures is also notable in the usefulness assessment of the collaboration feature (see Table III).

We provide developer and architect dashboards based on the assignment of resolution tasks to developers and organizational units, respectively. Four participants recommended additional team dashboards by additionally assigning resolution tasks to development teams, in particular with regard to violations that require more costly refactorings.

Finding 5: Violation-management tools should allow representation of roles, responsibilities, and team structures $(4/7)$, contacts should be preselected accordingly $(3/7)$, and violations should be also assigned to teams $(4/7)$.

6) Structured Violation Management: Our ISECOs are under development for more than 10 years. Most guidelines were not known in initial project phases. They evolve progressively over time, frequently problem driven, such as the guideline *Do Not Use Platform-Internal APIs* (see Section II).

Most participants consider the temporal-differentiation between *New* and *Legacy Violations* as very important for the approach's usefulness. They stated that, usually, hundreds to thousands of violations do already exist when new guidelines are adopted, partially within already highly-matured components. The risk of changing those matured components frequently outweighs the benefit of guideline compliance—those violations will not be resolved. Existing violations that should be resolved cannot be handled instantly—they overwhelm developers. Our participants consider the huge amount of existing violations as a crucial hurdle for exercising governance. They perceive temporal differentiation as a prerequisite for managing violations in a structured and controlled manner—whereas *New Violations* are handled instantly *Legacy Violations* can be handled incrementally on demand.

Architecture guidelines may imply varying efforts and benefits for ecosystem partners, resulting in long negotiation phases where violations are not managed. Our participants consider the ability to define individual violation-handling plans as very useful for managing violations right from the beginning. They highlighted that ecosystem partners need to specify violation-handling strategies—strategies are explicit. And they pointed out that violation-handling processes are triggered accordingly—violations are managed. Three participants recommended to support the definition of violation-handling plans for development teams, derived from the plan of the corresponding organizational unit.

Finding 6: Evolving guidelines imply a huge amount of *Legacy Violations* $(7/7)$. Temporal differentiation $(6/7)$ and violation-handling plans $(5/7)$ are required to manage them in a structured and explicit manner.

7) Organizational Aspects: We track and manage violations to architecture guidelines—we make them explicit. All participants consider the explicit management of violations as necessary to avoid their accumulation over time, in particular for long-running projects like ISECOs.

However, several participants mentioned that an important step for exercising explicit violation management is the establishment of an organizational mindset that allows explicit violation management. They stated that, otherwise, the high degree of transparency on software-quality deficits may not be necessarily desired by all ecosystem partners. They also highlighted the risk of misusing the achieved transparency, in particular by the

management for over-controlling the project. Moreover, resources required to execute the process must be provided by the management. Otherwise, developers and architects might not be able to rigorously adhere to the process.

Finding 7: The explicit management of architecture violations requires the establishment of an organizational mindset towards explicit violation management (5/7).

V. Discussion

In the following, we discuss the generality and limitations of the defined capabilities, their realization for our ISECOs, and our case-study results.

A. Generality

We present 6 capabilities for managing architecture violations. Our list of capabilities is not intended to be definitely exhaustive. Rather, it intends to tackle those violation-management hurdles that turned out to be most crucial for two of the largest ISECOs at Siemens. We evaluate the capabilities' usefulness to tackle them.

Our results show that, for our ISECOs, the capabilities are very well suited to address the hurdles and reveal great potential to ease violation management. It seems likely that, because of the size and complexity of our ISECOs, the identified hurdles rather constitute a superset of hurdles expected to appear in other decentralized and other mid- to large-scale projects [2]. The capabilities' realization is tailored to our organizational and product context. However, we consider that the capabilities and our findings, which basically outline requirements for violation-management tools, can be of general use for other projects that face similar hurdles. They may serve as a starting point for practitioners to develop customized violation-management tools for their ISECOs, which is considered as very important by our findings. In addition, the identified capabilities and findings outline a set of real-world requirements that should be investigated by the research community.

The TRAVIM approach enables the management of architecture violations. Equally important (and maybe an important step for successfully fielding the approach) is the establishment of an organizational mindset towards architecture compliance, considering all roles from developers to managers. Our ISECOs apply, for example, regular trainings and conferences.

B. Limitations

We discuss potential limitations of our study along common validity threats: A threat to construct validity is the potential bias caused by the selected evaluation setting. However, our study does not focus on ISECO details but on the capabilities' usefulness utilizing the setting. According to our participants, the setting was representative of our ISECOs. Regarding conclusion validity, there is a threat that data analysis depends on our interpretation. We used recommended methods to improve conclusion validity, such as triangulation, member checking, and spending sufficient time with the cases (see Section IV-A). With regard to internal validity, participants might have behaved unnaturally and might have given answers that do not fully reflect reality since they were recorded. To address this, we guaranteed anonymity and assured that we will seek for feedback on conclusions to avoid misunderstandings. In addition, results might be biased as we did not involve developers. However, all participants were well experienced and in central positions. They worked closely together with developers and half of them even started as developer within our ISECOs. We asked them to consider all viewpoints. Also the number of participants may seem small. Each session lasted about 3 hours and after 9 participants hardly any new insights were gained—we neared "theoretical saturation" [8], [12].

C. Capability Realization

A potential weakness of our realization is its reliance on heuristics for several capabilities (e.g., assigning tasks to developers who committed items, setting the violation type to *New Violation* when the item changes). However, heuristics are required to automate violation management, in particular within large-scale industrial settings. Automation, in turn, is required to keep violation-handling effort manageable. By applying the prototype for its own development, first results indicate the validity of our heuristics. Our follow-on longitudinal study after industrial roll out will allow us to validate them in detail. We consider the development and validation of heuristics for automating violation management as an important research topic that should be further investigated by the community.

VI. Related Work

In this section, we outline related work considering violation-management approaches, architecture guidance, and case-study research on decentralized software engineering.

A. Violation Management

For violation-management approaches, we discuss related work along a use-case–driven categorization for quality-management tools proposed by Deissenboeck et al. [13], including sensors, system analysis workbenches, project intelligence platforms, and dashboard toolkits:

1) Sensors: Sensors comprise static-analysis tools (e.g., FxCop [3], NDepend [4]) and testing tools (e.g., NUnit [14], JUnit [15]) that fully automatically check for compliance to implementation and architecture guidelines [13]. We track and assist resolution processes of identified violations, with a focus on architecture.

2) System analysis workbenches: System analysis workbenches support experts to interactively analyze architectures considering a system snapshot [13], often by use of dependency structure matrices, source-code query languages, and reflexion models [16]. Several workbenches have been developed, such as Lattix Architect [17], Understand [18], Sonargraph [19], and Titan [20]. Experts of our ISECOs partially use them to check for compliance to guidelines that cannot be checked fully automatically. We track and assist resolution processes of identified violations.

3) Project intelligence platforms: Project intelligence platforms automatically measure and store product- and process-related metrics in ongoing projects to support quality analysis and project controlling [13]. Prominent examples are Hackystat [21], PROM [22], and Team Foundation Server [6]. For our purposes, intelligence platforms might support experts to identity and analyze quality issues for which concrete guidelines should be implemented for.

4) Dashboard toolkits: Dashboard toolkits are mostly related to our work. They allow users to configure analysis dashboards that automatically collect, aggregate, and visualize data of *sensors* [13]. Well known toolkits are SonarQube [23], ConQAT [13], and Teamscale [24]. All of them realize a subset of our proposed violation-management capabilities, at least to some extent. However, none of them realize all capabilities. Moreover, they target different use cases: They are separate tools for detecting and monitoring violations to best coding practices (e.g., clones, long methods, god classes), in particular by means of status reports and trend analysis. Guidelines are selected on a project level. In contrast, we target seamlessly integrated, tailored, and continuous process support for resolving violations to project-specific architecture guidelines according to individual capabilities of involved ecosystem partners. Furthermore, we are not aware about any study that investigates the usefulness of violation-management capabilities in real-world settings.

B. Architecture Guidance

Several knowledge-management approaches have been developed to support architecture-decision making by reusing codified knowledge. For example, Zimmermann et al. present a design method that combines pattern languages and reusable architecture decision models [25]. Moreover, Zimmermann proposes a decision-modeling framework for service-oriented architectures [26]. Tang et al. provide a comparative study of tools that support management and reuse of architecture knowledge [27]. In our context, such knowledge-management approaches may support experts to define guidelines.

C. Decentralized Software Engineering

Rommes et al. [28] discuss architecture, process and organization aspects of their medical imaging product line, which involves a set of independent product groups. Van Ommering et al. [29] coin the term product population for their decentralized software product line. Toft et al. [30] present a community-driven approach that allows to share components across their products without involving a central platform organizational unit. Dinkelacker et al. [31] depict the adoption of open-source software development practices within their organization. Riehle et al. [32] conduct an industry case study on challenges and opportunities of inner-source approaches in decentralized platform-based product engineering. Lettner et al. [33], [34] conduct an exploratory characteristic study on industrial software ecosystems. Based on their results, they propose feature feeds [35], a publish-subscribe approach to foster the awareness about feature implementations in industrial software ecosystems.

All of them conduct case-study research on intra-organizational decentralized software engineering. They analyze emerging challenges and approaches to counter them. However, none of them investigate the management of architecture violations.

VII. Conclusion

Large-scale decentralized software projects require architecture guidelines to coordinate development. Feature and schedule pressure regularly result in guideline violations. We investigate two of our largest decentralized projects at Siemens. Both projects have to deal with thousands of architecture violations, that have accumulated over time, which results in substantially increased development and maintenance costs. Architecture violations must

be explicitly managed, including both the management of existing violations and the prevention of future violations. However, the decentralized development context results in various formidable violation-management hurdles.

We propose a set of capabilities (from integrated tool support to automated assignments of resolution tasks) that tackle the identified violation-management hurdles. We spent considerable effort to present a practical realization of the capabilities for our projects and performed a thorough case study on their usefulness from the perspective of practitioners. The practitioners expressed that our capabilities are highly valuable and hold great potential to ease violation management. We are currently planning our prototype's industrial roll out for our projects. We think that the capabilities can contribute to the development and improvement of violation-management tools, in particular for projects that face similar hurdles.

REFERENCES

[1] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*. Addison Wesley Professional, 2002.

[2] K.-B. Schultis, C. Elsner, and D. Lohmann, "Architecture challenges for internal software ecosystems: A large-scale industry case study," in *22Nd ACM SIGSOFT Int. Sym. on Foundations of Softw. Eng.*, ser. FSE'14. ACM, 2014, pp. 542–552.

[3] FxCop, https://msdn.microsoft.com/, accessed on Jan. 20, 2016.

[4] NDepend, http://www.ndepend.com/, accessed on Jan. 20, 2016.

[5] K.-B. Schultis, C. Elsner, and D. Lohmann, "Architecture-violation management for internal software ecosystems," in *13th Working IEEE/IFIP Conf. on Softw. Architecture*, ser. WICSA'16. Washington, DC, USA: IEEE Comp. Society, 2016.

[6] VisualStudio, https://www.visualstudio.com/, accessed on Jan. 20, 2016.

[7] J. Lowy, *Programming WCF Services*. O'Reilly Media, 2007.

[8] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Softw. Eng.*, vol. 14, no. 2, pp. 131–164, Apr. 2009.

[9] F. D. Davis, "Perceived usefulness, perceived ease of use, and user acceptance of information technology," *MIS Q.*, vol. 13, no. 3, pp. 319–340, Sep. 1989.

[10] R. Rabiser, P. Grünbacher, and M. Lehofer, "A qualitative study on user guidance capabilities in product configuration tools," in *27th IEEE/ACM Int. Conf. on Automated Softw. Eng.*, ser. ASE'12. New York, NY, USA: ACM, 2012, pp. 110–119.

[11] C. B. Seaman, "Qualitative methods in empirical studies of software engineering," *IEEE Trans. Softw. Eng.*, vol. 25, no. 4, pp. 557–572, Jul. 1999.

[12] K. M. Eisenhardt, "Building theories from case study research," *Academy of Management Review*, vol. 14, no. 4, pp. 532–550, 1989.

[13] F. Deissenboeck, E. Juergens, B. Hummel, S. Wagner, B. M. y. Parareda, and M. Pizka, "Tool support for continuous quality control," *IEEE Softw.*, vol. 25, no. 5, pp. 60–67, Sep. 2008.

[14] NUnit, http://www.nunit.org/, accessed on Jan. 20, 2016.

[15] JUnit, http://www.junit.org/, accessed on Jan. 20, 2016.

[16] L. Passos, R. Terra, M. T. Valente, R. Diniz, and N. das Chagas Mendonca, "Static architecture-conformance checking: An illustrative overview," *IEEE Softw.*, vol. 27, no. 5, pp. 82–89, Sep. 2010.

[17] Lattix, http://lattix.com/, accessed on Jan. 20, 2016.

[18] Understand, https://scitools.com/, accessed on Jan. 20, 2016.

[19] Sonargraph, https://www.hello2morrow.com/, accessed on Jan. 20, 2016.

[20] L. Xiao, Y. Cai, and R. Kazman, "Titan: A toolset that connects software architecture with quality analysis," in *22Nd ACM SIGSOFT Int. Sym. on Foundations of Softw. Eng.*, ser. FSE'14. ACM, 2014, pp. 763–766.

[21] P. M. Johnson, "Requirement and design trade-offs in hackystat: An in-process software engineering measurement and analysis system," in *First Int. Sym. on Empirical Softw. Eng. and Measurement*, ser. ESEM'07. Washington, DC, USA: IEEE Comp. Society, 2007, pp. 81–90.

[22] I. D. Coman, A. Sillitti, and G. Succi, "A case-study on using an automated in-process softw. eng. measurement and analysis system in an industrial environment," in *31st Int. Conf. on Softw. Eng.*, ser. ICSE'09. Washington, DC, USA: IEEE Comp. Society, 2009, pp. 89–99.

[23] G. Campbell and P. Papapetrou, *SonarQube in Action*. Manning, 2013.

[24] L. Heinemann, B. Hummel, and D. Steidl, "Teamscale: Software quality control in real-time," in *36th Int. Conf. on Softw. Eng.*, ser. ICSE'14. ACM, 2014, pp. 592–595.

[25] O. Zimmermann, U. Zdun, T. Gschwind, and F. Leymann, "Combining pattern languages and reusable architectural decision models into a comprehensive and comprehensible design method," in *7th Work. IEEE/IFIP Conf. on Softw. Arch.*, ser. WICSA'08., Feb 2008, pp. 157–166.

[26] O. Zimmermann, "Architectural decisions as reusable design assets," *IEEE Softw.*, vol. 28, no. 1, pp. 64–69, Jan 2011.

[27] A. Tang, P. Avgeriou, A. Jansen, R. Capilla, and M. Ali Babar, "A comparative study of architecture knowledge management tools," *J. Syst. Softw.*, vol. 83, no. 3, pp. 352–370, Mar. 2010.

[28] E. Rommes and J. G. Wijnstra, "Implementing a reuse strategy: Architecture, process and organization aspects of a medical imaging product family," in *38th Annual Hawaii Int. Conf. on System Sciences - Volume 09*, ser. HICSS '05. Washington, DC, USA: IEEE Computer Society, 2005.

[29] R. C. v. Ommering and J. Bosch, "Widening the scope of software product lines - from variation to composition," in *2nd Int. Conf. on Softw. Product Lines*, ser. SPLC'02. London, UK, UK: Springer-Verlag, 2002, pp. 328–347.

[30] P. Toft, D. Coleman, and J. Ohta, "A cooperative model for cross-divisional product development for a software product line," in *1st Int. Conf. on Softw. Product Lines: Experience and Research Directions*, ser. SPLC'00. Norwell, MA, USA: Kluwer Academic Publishers, 2000, pp. 111–132.

[31] J. Dinkelacker, P. K. Garg, R. Miller, and D. Nelson, "Progressive open source," in *24th Int. Conf. on Softw. Eng.*, ser. ICSE'02. New York, NY, USA: ACM, 2002, pp. 177–184.

[32] R. Dirk, C. Maximilian, K. Detlef, and H. Lars, "Inner source in platform-based product engineering," Dep. of Comp. Sc., Friedrich-Alexander Univ. Erlangen-Nuremberg, Tech. Rep., 2015.

[33] D. Lettner, F. Angerer, H. Prähofer, and P. Grünbacher, "A case study on software ecosystem characteristics in industrial automation software," in *Int. Conf. on Softw. and Sys. Process*, ser. ICSSP'14. New York, NY, USA: ACM, 2014, pp. 40–49.

[34] D. Lettner, F. Angerer, P. Grünbacher, and H. Prähofer, "Software evolution in an industrial automation ecosystem: An exploratory study," in *40th EUROMICRO Conf. on Softw. Eng. and Advanced Applications*, ser. SEAA'14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 336–343.

[35] D. Lettner and P. Grünbacher, "Using feature feeds to improve developer awareness in software ecosystem evolution," in *9th Int. Works. on Variability Modelling of Softw.-intensive Sys.*, ser. VaMoS'15. New York, NY, USA: ACM, 2015, pp. 11–18.