Towards Collaboration-Centric Pattern-Based Software Development Support

Christoph Dorn
Distributed Systems Group
Vienna University of Technology, Austria
dorn@infosys.tuwien.ac.at

Alexander Egyed
Institute for Systems Engineering and Automation
Johannes Kepler University Linz, Austria
alexander.egyed@jku.at

***Abstract*—Software engineering activities tend to be loosely coupled to allow for flexibly reacting to unforeseen development complexity, requirements changes, and progress delays. This flexibility comes a the price of hidden dependencies among design and code artifacts that make it difficult or even impossible to assess change impact. Incorrect change propagation subsequently results in costly errors. This position paper proposes a novel approach based on monitoring engineering activities for subsequent high-level pattern detection. Patterns of (i) collaboration structures, (ii) temporal action sequences, and (iii) artifact consistency constraints serve as input to recommendation and automatic reconfiguration algorithms for ultimately avoiding and correcting artifact inconsistencies.**

***Index Terms*—monitoring, pattern detection, software engineering, recommendation, collaboration structures**

I. Introduction

Software development consists of design and development steps typically performed by multiple software engineers, using heterogeneous tools, models, and expertise to support requirements capture, design, implementation, testing, debugging, bug reporting and many other activities. Such activities tend to be loosely coupled for two main reasons: (i) artifacts such as requirements, pieces of code, or test scenarios are created and manipulated by separate engineers potentially even using different tools and (ii) dealing with changing requirements, unforeseen complexity, and progress delays during the development process requires a high degree of flexibility for developer coordination and communication.

This results in fragmentation where individual engineers are perhaps aware of their particular work focus but lack a more comprehensive overview. It is quite rare to explicitly describe how various development artifacts depend on one another (e.g., traceability). It is virtually never described how or why these artifacts came about and who created them using what other artifacts as input. Subsequently, dependencies among development artifacts remain implicit. This is particularly a problem when changes occur because the lack of explicit dependencies makes it nearly impossible to understand how changes affect the development artifacts. For example, a requirement change should not only trigger design/code changes but also changes to test scenarios, documentation, and other artifacts - artifacts which likely were created by different engineers/stakeholders using different tools at different times. A change then should trigger collaboration among software engineers to update these artifacts. Not understanding or perceiving the need of such collaborations is then the root cause for failure to correctly propagate changes and hence the root cause for inconsistencies among development artifacts - leading to costly rework and even project failure.

Our goal is supporting software engineers in perceiving required collaboration, dependencies among artifacts, and suitable actions by means of recommendations and automatic reconfigurations of the software development environment. To this end, we propose recording the actions of software engineers, deducing patterns from that information, and exploiting them for recommendations. We aim at identifying patterns that reflect three dependencies types: structural collaboration patterns among developers, tasks, and artifact (e.g., what artifacts do developers read/manipulate and how do they coordinate), temporal activity patterns (e.g., which sequences of artifact reading, manipulating, and communication exist), and artifact dependency patterns (e.g., what relationships exist among the artifacts - consistency, traceability, etc.). These patterns then form that basis for adaptation and recommendation techniques that are aware of the arising structural implications.

II. Motivating Example

Suppose a software development team utilizing dedicated tools for requirement elicitation, architecture design, code versioning, bugtracking, and an IDE for source code development. Additionally team members also apply instant messaging, email, and document sharing to coordinate their work.

At some stage, the team needs to handle a changing requirement demanding a functionality adjustment. Consequently, the architecture and certain components need analysis to assess the changed requirement's impact (i.e., identifying architectural changes that satisfy the changed requirement, if any). Subsequently identifying affected design artifacts and source code files, determining necessary tasks to maintain consistency with other related artifacts, and selecting a suitable set of developers with the necessary in-depth knowledge and skills is a tremendous, error prone task. It requires developers to remember how they implemented the now-changed requirement in the past (who previously designed, implemented, tested it) to appropriately change all affected artifacts.

This work is supported in part by the Austrian Science Fund (FWF) under grant number J3068-N23.

For this scenario, how can we support the system architect, when she receives no notification on requirement updates? How can we determine when architecture changes indirectly affect source code artifacts? How can the software architect detect that an affected code artifact has no longer a responsible developer and who might be a suitable replacement? In the next section we present our approach for a framework addressing such questions.

III. Approach

Mechanisms and techniques for supporting the evolution of software development artifacts and developer collaborations establish a tightly integrated feedback loop comprising steps for bottom-up activity monitoring, pattern matching and analysis, and subsequent recommendation and reconfiguration.

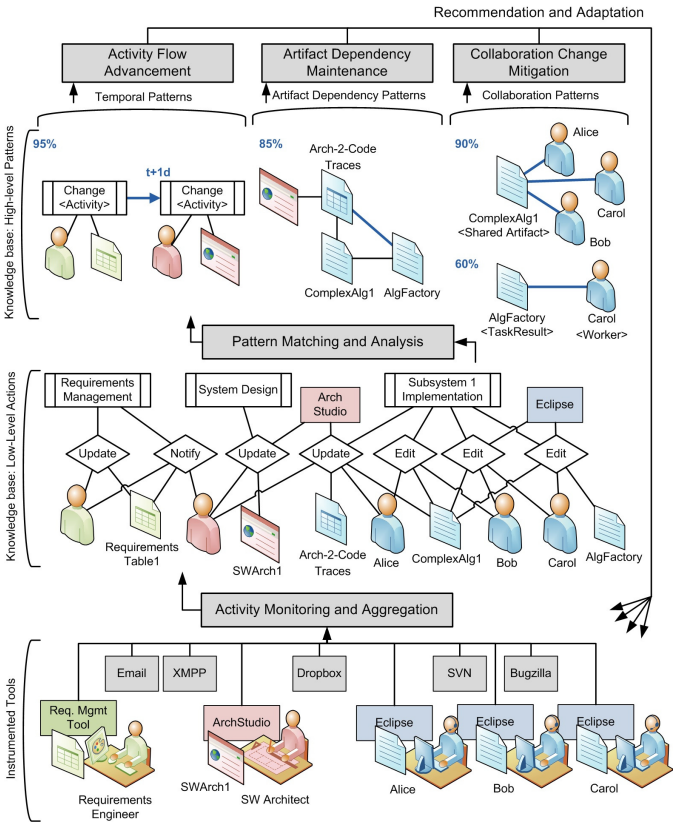

Fig. 1. Approach.

1) Activity Monitoring and Processing: Observation of developer actions is central to many previous approaches. Our approach foresees instrumenting tools (beyond IDEs) such as those from the scenario above. We essentially record which developer is manipulating what artifacts and, while doing so, communicates with which other developers.

Activity Monitoring and Aggregation takes the individual actions, filters out excessive actions, correlates actions, and resolves uncertainties. The *Low-Level Actions* are stored in the *Knowledge base* (see Fig. 1 center) as a composition of actions observed. The basic structure of an action is one that ties together an engineer with an artifact, an activity, and a tool. For example, Alice added a method to the ComplexAlg1 source file using Eclipse. These tiny fragments are merged together into a connected network of low-level actions.

2) Pattern Matching and Analysis: The low-level view serves as the input to three types of *Pattern Matching and Analysis* heuristics which extract high-level dependencies and structures among engineers and artifacts. A heuristic focusing on **temporal dependencies** detects multiple occurrences of following example event sequence: {an requirement engineer edits the requirement description, followed by an email to the software architect, and the architecture subsequently updating the architecture model} and concludes that a requirement change is followed by an architecture change. Multiple, interleaving edits on code file ComplexAlgorithm1 by developers Alice, Bob, and Carol lead a **collaboration pattern** heuristic to the conclusion that the file is worked upon as a shared artifact rather than a task specifically assigned to Alice (with Bob and Carol merely giving advice). Given a set of consistency constraints (e.g., each architecture component must map to at least one source code artifact) and a set of immediate architecture-to-code mappings, an **artifact dependency** heuristic may analyze software source code dependencies and tag *supporting* code as relevant to a particular architecture element. Subsequent analysis calculates pattern-specific properties needed by recommendation and adaptation algorithms (e.g., update rate for a shared artifact, average duration between changing code and running regression tests) before the high-level patterns are stored in the *Knowledge base* in form of likely relations, dependencies, and constraints among developers, tasks, and artifacts (Fig. 1 upper half).

3) Recommendation and Adaptation: Rather than having to deal with low-level individual actions, high-level patterns provide the basis for algorithms that ultimately deliver user recommendations and instrument reconfigurations autonomously when a change occurs (Fig. 1 top). Such changes describe high-level pattern events (e.g., new pattern instances, updated pattern properties) as well as low-level actions (e.g., a file has been changed). In our scenario, a combination of three mechanisms assists the engineering team in the following ways. *Activity Flow Advancement* exploits detected temporal patterns. In our scenario it will notify the software architect of the exact requirement change and the affected architecture elements if it observes neither communication between the requirements engineer and the architect nor read or write access to the affected architecture elements within a given time frame.

Once the system architect identifies (directly and indirectly) affected components and connectors, *Artifact Dependency Maintenance* identifies the location and cause for potential artifact inconsistencies. Here, it uses the artifact consistency patterns to determine the respective primary and secondary source code artifacts (here: ComplexAlg1, AlgFactory).

Collaboration Change Mitigation assists in determining the most suited developers based on their involvement in source code authoring for reflecting the architecture changes in code artifacts. This mechanism generally deals with undesirable

team-topology situations (e.g., code without responsible developer). Taking the underlying detected collaboration patterns into consideration, we may recommend that Carol should take over sole responsibility for ComplexAlg1 and AlgFactory and subsequently carry out the change propagation.

While recommendations certainly present a low-invasive technique to support a software developer team, our approach foresees also autonomous reconfigurations. Example adaptations include automatically annotating model and code artifacts with warnings and information of the ongoing change, creating a chat room linked to the relevant artifacts and developers, and even assigning change propagation tasks to particular engineers.

4) Discussion: We are of the opinion that near-realtime tool integration is key for supporting co-evolution of software development artifacts and collaboration structures. Thus, initially, there will be a trade-off between (a) achieving high precision and extend of the collaboration structures and artifact dependencies that are automatically observable, and (b) simultaneously keeping the burden onto the developer minimal. Yet, an increase in the number of integrated tools, the level of observable detail, and the number of participating developers will enable a more complete, meaningful view. Note, however, that there will always be an element of uncertainty in the artifacts dependencies and collaboration structures we observe due to:

• *Incomplete observations*: Ideally we would like to capture complete artifact dependencies and collaboration structures. Yet, it is likely infeasible to obtain complete developer actions. Some information communication may be non-accessible or incomprehensible (e.g., an oral communication) and must be disregarded. Even communication and manipulations involving a tool can be problematic. Tools by third party vendors typically offer only limited monitoring capabilities. Thus only parts of all development actions are available for reasoning and recommendation.

• *Context switching*: A core assumption is that artifacts investigated and manipulated concurrently are likely dependent on one another [1]. However, developers typically do switch between different strands of work dynamically and engage in frequent short-term side-tracking from their main activity/goal.

While it may thus be impossible to completely eliminate the possibility of falsely identified dependencies, following mechanisms address the inherent uncertainty:

• delayed uncertainty resolving: during live editing we understand temporal dependencies among artifacts and developers but a latter, fine-grained diffing of artifacts pre-post state may reveal in more level of detail what exactly was changed.

• continuous pattern refinement: incoming actions allow for better pattern detection thus enabling refinement or even revocation of pattern instances.

• probabilistic pattern instantiation: detection algorithms assign probabilities to pattern instantiations and pattern elements to express uncertainty and thus allowing recommendation algorithms to trigger only for specified reliability thresholds.

IV. Related Work

A. Collaboration Awareness

In Software Engineering environments, artifacts replace activities as the main, explicit element for collaboration and coordination (merely generic tasks for work assignment remain). Most development awareness tools address the code implementation and maintenance phase; occasionally also the testing phase. Their primary focus is typically on change management (e.g., ELVIN/Tickertape, SoftCHANGE), conflict avoidance (e.g., Palantir, Codebook, RaisAware), and understanding developer activities (e.g., FASTDash, Jazz, Ariadne, Tesseract, ProxiScientia) [1]. Figueiredo and de Souza [3] extend change impact analysis to design artifacts using traceability links. These tools have several aspects in common. All either lack near-real time information when relying on code and communication repositories as data source and/or provide near-real time awareness but remain restricted to a single (stand-alone or integrated) tool. Awareness information typically consists of direct relations among artifacts and developers at the source code level, thereby remaining unaware of consistency and traceability concerns with design artifacts. Relevance is derived from an aggregated set of low-level data without extraction of high-level patterns.

B. Coordination Requirements

Socio-technical congruence (STC) [4] measures the extent of developer coordination capabilities meeting the underlying work coordination requirements. While STC mostly focuses on alignment (or lack thereof) of task dependencies with developer communication links, Jiang et al. [5] introduce also knowledge-centric and resource-centric congruence. Doing so, existing approaches remain unaware of different dependency types and corresponding coordination mechanisms [6]. We propose high-level patterns for better addressing a team's various coordination needs. We thereby aim beyond mining task routing patterns (e.g., [7]) which remain artifact unaware.

C. Adaptation and Recommendation

Recommendation mechanisms in CWEs and software development projects typically determine a set of relevant people (Ensemble [8]), expertise, and artifacts (code, bug reports, documentation) for the situation at hand. Relevance arises from project repositories via topic searches and filters (Expertise Recommender [9]), task involvement and development activities (Hipikat/Mylar [10]), frequently or subsequently visited code (Team Tracks [1]), or developer activity similarity (Proximity [11]). The later works provide more sophisticated recommendations as they consider more than only immediate relations (see also collaboration awareness tools above). In general, recommendations remain limited to listing/ranking users or artifacts. To the best of knowledge, no tools exist that recommend steps or tasks for the underlying situation, much less autonomously carry out support activities and collaboration reconfigurations.

[1]Due to page limits find the respective references in following survey [2].

D. Flexible process systems

Software development-focused process support takes on various forms. On the micro-level, Zhao et al. apply Little-JIL for describing fine-grained steps involved in rework or refactoring [12]. While this approach captures which artifacts where changed by what rework activity, the involved users and internal dependencies remain implicit. Hebig et al. [13] describe how various software design and code artifacts dependencies emerge from MDE activities. Process support at the macro-level assumes pre-defined process models and rigorous tool integration. Kedji et al. provide a collaboration-centric development process model and corresponding DSL [14]; Moser et al. [15] build an engineering bus for integration of development activities for flexible industrial automation systems. These micro and macro-level approaches are software development artifact aware but lack the required flexibility.

E. Inconsistency Management and Change Impact

In context of design models, change impact analysis is in its infancy. Most progress has been on the detecting of inconsistencies (e.g., [16], [17], [18]) because an inconsistency is indicative of an incomplete or incorrect change propagation. There also have been attempts in generating fixes for inconsistencies [19], [20], [17] where the "fixes" could be seen analogous to propagated changes. Change propagation has been addressed more directly by Briand et al. [21] who identify specific change propagation rules. However, there is generally no guarantee of correctness associated with change propagation (which is indicative of the challenges we are addressing and the heuristical nature we propose). The problem of change propagation is analogous to the quite extensive works on model transformation [22] and code generators. However, our proposed work is meant to be useful for models (or model elements) that cannot be derived from other models (or model elements).

V. Conclusions

Extracting high-level patterns for software development recommendation and adaptation constitutes a great challenge due to inherent collaboration dynamics and uncertainty but also promised great benefit. Our upcoming research activities focus primarily on the technical framework and mechanisms ultimately aimed at significantly reducing inconsistent and/or incomplete change propagation. In the course of our research two main empirical research questions of high significance will then arise: (i) how precise and comprehensive do we have to document developer actions, artifacts, and their relations and (ii) how extensive and invasive may our techniques be when dynamically reconfiguring a developer team, for example, to ensure that the most suitable developer available is involved in a change task.

References

[1] R. DeLine, M. Czerwinski, and G. Robertson, "Easing program comprehension by sharing navigation data," in *Proc. of the IEEE Symposium on Visual Languages and Human-Centric Computing*, Washington, DC, USA: IEEE Computer Society, 2005, pp. 241–248.

[2] E. Trainer and D. Redmiles, "A survey of visualization tools that promote awareness of software development activities," Institute of Software Research, University of California, Irvine, http://www.isr.uci.edu/tech_reports/UCI-ISR-09-5.pdf, Tech. Rep., May 2009.

[3] M. Figueiredo and C. de Souza, "Wolf: Supporting impact analysis activities in distributed software development," in *Cooperative and Human Aspects of Software Engineering (CHASE), Int. Workshop on*, June 2012, pp. 40–46.

[4] M. Cataldo, P. A. Wagstrom, J. D. Herbsleb, and K. M. Carley, "Identification of coordination requirements: implications for the design of collaboration and awareness tools," in *Proc. of the ACM Conf. on Computer Supported Cooperative Work*, New York, NY, USA: ACM, 2006, pp. 353–362.

[5] L. Jiang, K. M. Carley, and A. Eberlein, "Assessing team performance from a socio-technical congruence perspective," in *ICSSP*, 2012, pp. 160–169.

[6] C. Dorn and R. N. Taylor, "Analyzing runtime adaptability of collaboration patterns," in *Int. Conf. on Collaboration Technologies and Systems (CTS)*. Los Alamitos, CA, USA: IEEE Computer Society, 2012.

[7] S. Dustdar and T. Hoffmann, "Interaction pattern detection in process oriented information systems," *Data Knowl. Eng.*, vol. 62, pp. 138–155, July 2007.

[8] P. F. Xiang, A. T. T. Ying, P. Cheng, Y. B. Dang, K. Ehrlich, M. E. Helander, P. M. Matchen, A. Empere, P. L. Tarr, C. Williams, and S. X. Yang, "Ensemble: a recommendation tool for promoting communication in software teams," in *Int. Workshop on Recommendation systems for software engineering*, New York, USA: ACM, 2008, pp. 1–2.

[9] D. W. McDonald and M. S. Ackerman, "Expertise recommender: a flexible recommendation system and architecture," in *Proc. of the ACM Conf. on Computer Supported Cooperative Work*, New York, NY, USA: ACM, 2000, pp. 231–240.

[10] D. Čubranić and G. C. Murphy, "Hipikat: recommending pertinent software development artifacts," in *Proc. of the Int. Conf. on Software Engineering*, Washington, DC, USA: IEEE Computer Society, 2003, pp. 408–418.

[11] K. Blincoe, G. Valetto, and S. Goggins, "Proximity: a measure to quantify the need for developers' coordination," in *Proc. of the ACM Conf. on Computer Supported Cooperative Work*, New York, NY, USA: ACM, 2012, pp. 1351–1360.

[12] X. Zhao and L. Osterweil, "An approach to modeling and supporting the rework process in refactoring," in *Proc. of the Int. Conf. on Software and System Process (ICSSP)*, june 2012, pp. 110 –119.

[13] R. Hebig, A. Seibel, and H. Giese, "Toward a comparable characterization for software development activities in context of mde," in *Proc. of the Int. Conf. on Software and Systems Process*, New York, NY, USA: ACM, 2011, pp. 33–42.

[14] K. A. Kedji, R. Lbath, B. Coulette, M. Nassar, L. Baresse, and F. Racaru, "Supporting collaborative development using process models: An integration-focused approach," in *ICSSP*, 2012, pp. 120–129.

[15] T. Moser, S. Biffl, W. D. Sunindyo, and D. Winkler, "Integrating production automation expert knowledge across engineering domains," *IJDST*, vol. 2, no. 3, pp. 88–103, 2011.

[16] A. Egyed, "Automatically detecting and tracking inconsistencies in software design models," *IEEE Transactions on Software Engineering*, vol. 37, no. 2, pp. 188–204, 2011.

[17] Y. Xiong, Z. Hu, H. Zhao, H. Song, M. Takeichi, and H. Mei, "Supporting automatic model inconsistency fixing," in *7th Joint Meeting of ESEC and FSE, Amsterdam, The Netherlands*, Aug 2009, pp. 315–324.

[18] M. Kamalrudin, J. C. Grundy, and J. G. Hosking, "Managing consistency between textual requirements, abstract interactions and essential use cases," in *34th Annual IEEE Int. Computer Software and Applications Conf. (COMPSAC), Seoul, Korea*, July 2010, pp. 327–336.

[19] C. Nentwich, W. Emmerich, and A. Finkelstein, "Consistency management with repair actions," in *Proc. of the Int. Conf. on Software Engineering*, Portland, Oregon, USA, May 2003, pp. 455–464.

[20] A. Egyed, "Fixing inconsistencies in uml design models," in *Int. Conf. on Software Engineering, Minneapolis, USA*, May 2007, pp. 292–301.

[21] L. C. Briand, Y. Labiche, and L. O'Sullivan, "Impact analysis and change management of uml models," in *19th Int. Conf. on Software Maintenance (ICSM) Amsterdam, The Netherlands*, Sep 2003, pp. 256–265.

[22] J. Cabot and E. Visser, Eds., *Proc. of the Int. Conf. on the Theory and Practice of Model Transformations (ICMT)*. Springer LNCS, 2011.