Feature-Oriented Software Evolution

Leonardo Passos[*]
University of Waterloo
lpassos@gsd.uwaterloo.ca

Krzysztof Czarnecki
University of Waterloo
kczarnec@gsd.uwaterloo.ca

Sven Apel[†]
University of Passau
apel@fim.uni-passau.de

Andrzej Wąsowski[‡]
IT University
wasowski@itu.dk

Christian Kästner
Carnegie Mellon University

Jianmei Guo
University of Waterloo
gjm@gsd.uwaterloo.ca

ABSTRACT

In this paper, we develop a vision of software evolution based on a feature-oriented perspective. From the fact that features provide a common ground to all stakeholders, we derive a hypothesis that changes can be effectively managed in a feature-oriented manner. Assuming that the hypothesis holds, we argue that feature-oriented software evolution relying on automatic traceability, analyses, and recommendations reduces existing challenges in understanding and managing evolution. We illustrate these ideas using an automotive example and raise research questions for the community.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*complexity measures*; D.2.9 [**Software Engineering**]: Management—*software configuration management*; K.6.3 [**Management of Computing and Information Systems**]: Software Management—*software maintenance*

General Terms

Management, Measurement

Keywords

Feature-oriented development, Software evolution, Traceability, Analysis, Recommendation

[*]Funded by CAPES, grant BEX 0459-10-0.

[†]Funded by DFG, grants AP 206/2, AP 206/4, AP 206/5.

[‡]Partially funded by ARTEMIS JU grant nº 295397 VARIES.

VaMoS '13 January 23 - 25 2013, Pisa, Italy

1. INTRODUCTION

Software systems evolve to meet changing requirements, platforms, and other environmental pressures [23]. For example, automotive embedded software undergoes continuous evolution due to new regulations (e.g., the European Union now requires all new cars to have an anti-lock brake system—ABS), market-differentiating enhancements (e.g., electronic stability control improves ABS by preventing skidding), and the availability of new technology (e.g., laser-based distance sensors for automatic braking are more precise than their radio-frequency counterparts).

Understanding evolution of large software systems is far from trivial. A change integrating the electronic stability control into an existing ABS can be scattered across multiple artifacts, requiring different kinds of expertise that are unlikely to be mastered by all stakeholders. Software engineers have a deep understanding of the changes at the code level, but much less so at the system level, including physical models. The contrary can be said of mechanical and control engineers: they understand the physical models, but the mathematical equations they wrote may look unrecognizable to them at the code level, due to optimizations and other low-level details. Similarly, product managers are likely to understand the change in terms of user and market impact rather than the technical details. The diversity of stakeholders and development artifacts makes understanding change drivers and the impact of change challenging, leading to ineffective communication, software flaws, architectural decay, and higher maintenance costs.

We hypothesize that managing change at the level of features can address the challenges described above.

Features represent cohesive bundles of requirements addressing important capabilities of the system [12]. We believe this predestines them as the ideal means of organization of discourse about change: (i) requirements are an agreement among all stakeholders on what the system should do. Since requirements are generally understood by all stakeholders, the tight relation between features and requirements make features likely to be understood by all stakeholders. Furthermore, features are more coarse-grained than individual requirements, thus facilitating understanding (requires less cognitive effort) and traceability of evolution. Even in situations in which individual requirements need to be traced and understood, a feature view provides an initial point to locate them; (ii) changes can be stated and understood relative to

features (e.g., introduction of new features, retirement of old ones, etc.), providing a simple and common ground to all stakeholders; (iii) features are the starting point to bootstrap software product lines (SPLs). Managing evolution at the feature level allows stakeholders to identify common and variable features in system families, and thus the variation points in the architecture required to support the necessary range of feature variation; (iv) features make dependencies and change-impact easier to be understood, providing a high-level requirements view of how different parts are related.

Feature-oriented development is already part of existing methodologies and systems [3]. In agile methods, such as *Feature Driven Development* (FDD) [32], management is centered around user stories (short descriptions documenting a feature), which then drive the architecture, coding scheduling, effort estimation and test cases. In SPLs, features allow managing software variants (valid feature combinations), with an extensive catalog of successful case studies.[1] Some open-source systems also follow a feature-oriented process. The feature-driven development of Linux kernel, for instance, enables its scaling to an enormous size, both in the number of features and contributors [28].

These examples, however, also highlight a poor support for effective feature-oriented evolution: although FDD is centered on a feature-based software management, it does not prescribe traceability, except when writing test cases [7]. In SPLs and systems like Linux, traceability is present across different artifacts, but querying it to extract any meaningful knowledge is complex, as there is no tool support. Moreover, temporal traceability is not recorded over time. The existing approaches and systems have little support for feature-interaction detection and impact analysis, while lacking any support for strategic decision making and change recommendations over the evolution history.

Vision: *Assuming the validity of our hypothesis, we believe that organizing software evolution around features, supported by tracing, analyses and recommendations will address many of the challenges in understanding and managing change.*

By *tracing*, we envision automatic recovery and management of traceability links among features and their associated implementation artifacts, all summarized along a timeline. Based on these links, *analyses* collect various measures over features, which are aggregated to aid strategic decision making. If put on a dashboard (see Fig. 1), the resulting data can be used to highlight good and bad indicators of evolution (e.g., there has been an increase in the number of feature smells). *Recommendations* further enhance analyses, providing suggestions for feature refactorings, test-selection criteria, feature retirement candidates, etc.

Since not all systems are built in a feature-oriented fashion, features may not be explicitly represented in implementation artifacts. In that case, existing techniques can leverage feature localization [42, 17, 21, 29, 37, 11, 19, 41] to relate (code) artifacts (or fragments of them) to the features of the system (feature set). To a certain extent, this makes our vision independent of any implementation technique.

In the remaining part of the paper, we present a motivating example, and use it to guide the discussion of each part of our vision of feature-oriented software evolution.

[1] sei.cmu.edu/productlines/casestudies/catalog/index.cfm

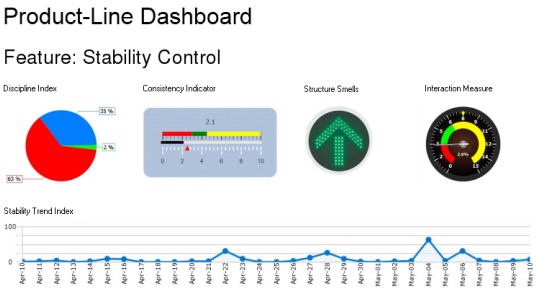

Figure 1: Mock-up dashboard interface

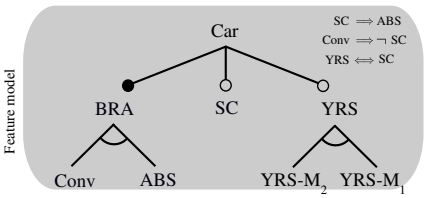

Figure 2: Feature model of the automotive example

2. MOTIVATING EXAMPLE

Consider an automotive product line supporting conventional (Conv) and anti-locking braking (ABS), with stability control (SC) as an optional feature. See the feature model in Fig. 2. Stability control is a safety feature that interacts with ABS to brake and help steering the vehicle according to the driver's intention. To detect mismatches between wheels and steering wheel turns, SC requires a yaw-rate sensor (YRS), which can be one of two possible kinds: YRS-M_1 and YRS-M_2. In our example, both are based on the same technology, but YRS-M_2 has better precision measure, whereas YRS-M_1 has yaw-rate prediction support.

As time progresses, the product line evolves: at a given point in time (t_1) stakeholders decide that it is increasingly expensive to keep two similar, but independent yaw-sensor models, and thus decide to merge YRS-M_1 into the existing YRS-M_2 sensor, cloning its yaw-rate prediction code. Later on (t_2), stakeholders decide to simplify the resulting feature model: since YRS-M_2 is now the only supported sensor, having both YRS-M_2 and YRS is redundant. To that end, stakeholders combine the two features into a single new feature, naming it YS (our stakeholders have a preference for two-letter feature names). This leads to a new instant of evolution (t_3).

3. TRACING

Suppose that after some time since the last change, stakeholders start facing issues on how different features are connected in time. For instance, after finding a bug in the new YS sensor, stakeholders set to check whether the same issue occurs in older sensors (still used in older car models) that eventually contributed to the code of YS. To this end, stakeholders verify all instants prior to the creation of YS, but struggle to understand whether and how YRS-M_1 (at t_1) and YRS-M_2 (at t_1 and t_2) relate to YS, as such information was not recorded during evolution. At best, stakeholders can only inspect commit patches (textual diffs) based on changes to

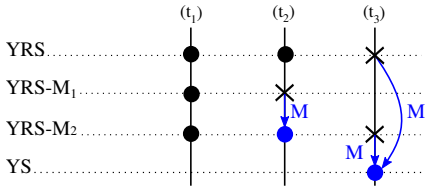

Figure 3: Traceability timeline

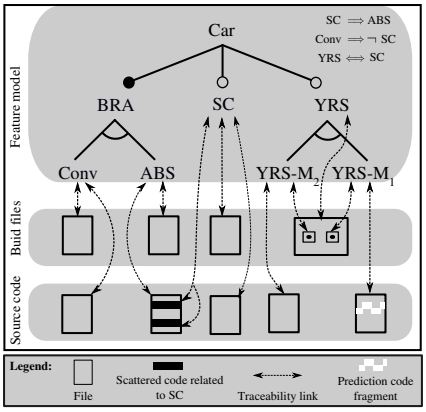

Figure 4: Recovered traceability links at instant t_1

different artifacts, which in turn, are not explicitly associated with the features they realize. Communication issues also appear along the way. A stakeholder, for instance, when analyzing the evolution of the feature model alone, may wrongly conclude that YRS-M$_2$ and YS have no yaw-rate prediction support, as changes in the feature model do not report the fact the yaw-rate prediction code was cloned from YRS-M$_1$ into YRS-M$_2$ (resulting in the snapshot at t_2), and that YS "inherits" it when merging with YRS-M$_2$ (resulting in the snapshot at t_3).

Tracing how the features of the system evolve over time mitigates these problems. If a timeline visualization was available (Fig. 3), stakeholders could easily assess whether a feature exists at a given time (marked as a bullet), when it was deleted (marked as an X) and how it relates to other features (e.g., by merging, marked as M). Queries could filter specific traces.

Traces should be maintained automatically, as manual maintenance is an expensive, error-prone and labour-intensive activity [6]. Traceability recovery overcomes the effort of manually keeping traces by automating their retrieval. Starting from an initial point in time, recovery retrieves the links among the different types of artifacts that are related to the realization of each concrete feature. Figure 4 shows the links, depicted as dashed arrows, created by the recovery process for the example product line at the first point of its evolution (t_1), assuming a feature model, build and source code files. These links relate build and source files to their corresponding feature, while also tracing specific build rules (e.g., YRS-M$_1$ and YRS-M$_2$) and code fragments (e.g., SC). These fine-grained links are retrieved by analysing the variability encoded inside each source and build file.

The difficulty of recovering links is directly related to how

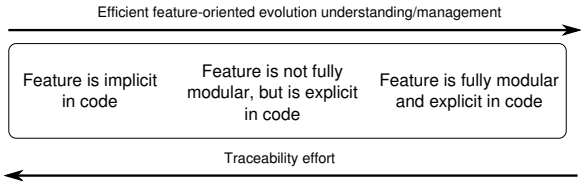

Figure 5: Spectrum of features and traceability

features are implemented. In particular, there are three main situations to consider when recovering a particular feature, as shown in the spectrum in Fig. 5.

Feature is fully modular and explicit in code. If a feature resides in a well defined module explicitly linked to a feature of the system, then there is nothing to be done in terms of its tracing, as the link is already given by a one-to-one mapping. Examples of such features include Conv, ABS, YRS-M$_1$ and YRS-M$_2$ in Fig. 4.

Feature is not fully-modular, but it is explicit in code. The midpoint of the spectrum is when a feature is explicit in code, but it is not fully modular, i.e., it has code fragments scattered across the code base (for example SC). Due to scattering, the feature must explicitly rely on variability encoding (for instance, pre-processing directives) to guard its related fragments. Recovering features in this part of the spectrum is more complex than when features are *fully-modular and explicit*, as it requires possibly analysing multiple files, whose implementation may contain complex and nested preprocessing directives. Since variability encoding can create arbitrary complex guard conditions in code, recovered links often lead to multiple features (many-to-many mappings).

Feature is implicit in code. The last and most challenging situation in the spectrum occurs when a feature is implemented in the source code, but is not related by any means to a feature in the software feature set (it is implicit in code). In this case, feature-location techniques are required to identify which parts of code realize a given feature, along with which compilation rules. Recovery then links each feature to its associated located elements.

As evolution progresses, recovery has to update its initial set of traces, while defining new ones resulting from newly added features. Figure 6 illustrates this in the context of the automotive product line: from t_1, recovery identifies the merge of the two sensors and the resulting deletion of the build rule and source file related to YRS-M$_1$, along with the copy of its prediction code (shown in white squares) to YRS-M$_2$. From that, recovery reaches a new point in time (t_2), for which a new set of traces is obtained. It then creates a link between YRS-M$_1$ (at t_1) and YRS-M$_2$ (at t_2) to capture their temporal evolution.

Recovery proceeds to analyze the next evolution step, in which YRS-M$_2$ is combined with YRS to simplify structure, while lifting all artifacts of YRS-M$_2$ to YRS. In the same evolution step (same commit), stakeholders also rename YRS to YS. Given the limitation of existing source code management (SCM) systems, recovery cannot identify which of these changes occurs first. Rather, it infers that the changed features converge to a common target, and thus take them to be a single merge towards YS. The newly obtained traces are then stored in a new time frame. As before, recovery creates temporal traces to map evolution across different points in

time, in this case from YRS and YRS-M$_2$ (at t_2) to the newly added feature YS (at t_3).

Traceability is a complex problem, as it has to be kept across different artifact types, with different abstraction levels. To our advantage, feature-centric traceability defines a cross-cutting perspective to associate and track all artifacts.

Future directions and research questions

Although traceability has many challenges on on its own [24], we leverage discussion to a feature-oriented perspective.

We argue that traceability research should focus on recovery techniques to efficiently extract links of non-modular-explicit features. Special attention should also be given to improve or devise new feature-location techniques to bootstrap traceability of implicit features in code (once a feature is located, recovering its links is as difficult as tracing explicit features, modular or not). In contrast, tracing explicit-modular features is easy, although such setting hardly occurs outside academia (e.g., FeatureHouse benchmark [4]).

Establishing links for non-modular-explicit features requires analysis over build rules and code fragments.

Build files are notoriously complex, as existing build languages allow dynamic creation of compilation rules during the build process [38], beside being comparable to general programming languages [18]. Tracing non-modular-explicit features in code is also challenging, as annotated code is often tangled with the presence of other features. Tracing it requires analysis of all features that trigger the inclusion of such a code in the corresponding set of variants. As there are no recovery techniques in industry nor in academia that retrieve links at the level of single build rules or individual lines of code, we pose the following research question:

How to effectively recover traceability links in build files and source code in variability-aware systems?

New techniques aiming to answer these questions can rely on existing approaches on variability-aware parsing [26] and build-file analysis [18].

Another research direction is how to leverage tracing to the temporal dimension imposed by evolution:

How to efficiently recover temporal traceability links?

One approach is to explore patterns of evolution [34], i.e., structural changes in specific artifacts that allow us to heuristically infer whether and how changes relate different features (e.g., as it happened in the automotive product line, the deletion of a feature and its implementation artifacts, together with the cloning of part of its code to another feature is likely to characterize a merge). This leads to three research questions:

Which evolution operations/changes occur at the feature level (e.g., add, rename, delete, merge, etc.) and how different artifacts change as a result?

Which change patterns arise from such a change?

Can they be used to devise specific recovery heuristics?

Although existing research attempts to answer the first question, its focus is on evolution that preserves behavior and the set of variants [35, 31], or on cases where only the feature model and its mapping to other assets are taken into account [36]. Such question is also investigated by Passos et al. [34], who present a preliminary set of evolution patterns extracted from the Linux kernel evolution history.

Recovery can also rely on external data sources to increase confidence of heuristic-based techniques, including documentation, bug-tracking systems and mailing lists, each with different levels of trust. That said, we pose two questions:

How to decide on conflicting information from different sources?

Which information sources provide a higher degree of trust?

Future research shall also focus on how to incorporate feature-based tracing in existing SCMs [43]. In this case, SCMs could record changes on a feature level, linking them to changes to the feature model (if existent). This guarantees tracing to be kept at each commit, in addition to detecting possible inconsistencies [39, 30] between the feature model and related artifacts at commit time (e.g., a concrete feature cannot exist in the feature model without any artifact realizing it), thus improving traceability quality.

4. ANALYSES

Let us return to our example. As the automotive product line evolves, stakeholders notice that maintenance is taking longer, development productivity has decreased, and bugs are starting to rise. These symptoms are well known in software evolution, and are related to the *aging* phenomenon. Software aging is the result of a bad evolution, leading systems to reduced performance, architecture decay, less reliability and higher maintenance costs [33].

To countermeasure these issues, stakeholders set out to understand what is causing them. They would like to improve the maintenance process to prevent future problems. In that sense, stakeholders ask questions like:

1) *Are the feature models, build system, and code consistent after changes?*
2) *Which features and which artifacts are affected by changing a given feature?*
3) *Does changing a feature cause unexpected behavior?*
4) *Is the software modularized (architecturally aligned) according to the features it implements?*
5) *How are features evolving and how is evolution taking place?*
6) *Are features becoming more complex? In which respect (coupling, size, scattering, etc.)?*

These questions address three specific concerns: software consistency (1), change-impact analysis (2, 3) and architectural analysis (4, 5, 6). We explain each of these from a feature-based perspective.

Consistency checking

Using features to organize code and feature models as the high-level description of the implemented system introduces a risk that the two get out of sync. Consistency monitoring is needed to prevent such situations. Fig. 7 illustrates the problem. In the first fragment, feature ABS contains a code block guarded by the presence of feature Conv. Implementers of ABS provided software support for switching to conventional braking in case ABS fails. According to the feature model of the automotive example, that code is dead [39, 30], as Conv is always absent when ABS is present (they are

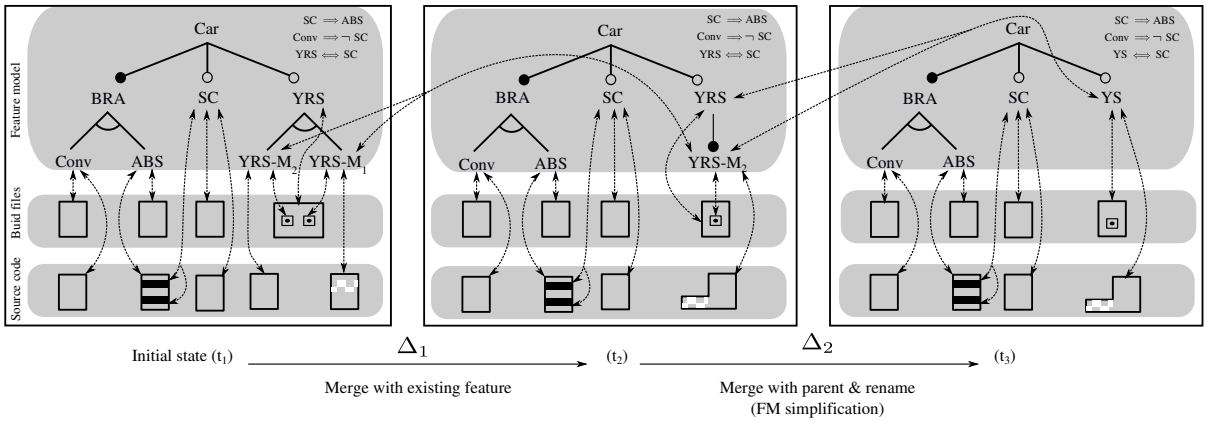

Figure 6: Traceability recovery in the automotive example

inside a mutex-group). Therefore, the feature model and the implemented code are inconsistent.

In the second fragment of Fig. 7, ABS control checks whether stability control (SC) is supported. If so, it requests some data to later check over-steering. Since this check is outside the ifdef blocking initializing the data variable, checking over-steering will cause a null pointer exception at runtime if SC is absent. Such issues can be addressed using variability-aware flow analyses, as in [10].

The third fragment illustrates a parsing error caused when SC and YRS-M$_1$ are both present, since the declaration statement inside the ifdef block does not contain a semicolon.

The fourth fragment shows a type error, as p will be taken as an integer primitive type if both SC and YRS-M$_1$ are absent, preventing any method call from it. Recent tools addresses syntax and type errors that arise under certain configurations of features, manifesting divergence between the model and implementation (e.g., [26, 22]). A similar problem is solved by Czarnecki and Pietroszek [16], who verify the consistency of configurations against structural constraints in OCL. Classen et al. [15] achieve the same for temporal constraints (which is also close to program analysis techniques mentioned above). Thüm et al. [40] show a manual proof technique, based on proof composition, using as an example proving consistency between Java code and feature models.

Although existing research and tools have managed to adapt existing analysis techniques to take variability into account, there is still open issues that need to be addressed.

Future directions and research questions

Although existing tools [39, 30] have managed to scale inconsistency detection to large programs (e.g., Linux), variability-aware type checking [27, 13, 25] and flow analyses have not yet been demonstrated in large and complex real-world systems. Similarly model checking techniques for product lines have been developed at the theoretical level [15, 14, 5], but hardly bridged to realistic systems.

Do existing variability-aware type checking, model checking and flow analyses scale to large systems?

Furthermore, existing flow analyses are intra-procedural. Inter-procedural analysis is yet a challenge, and existing research needs to address it [9]:

How to adapt existing inter-procedural analyses to handle variability?

Another important aspect is how to leverage existing variability-aware error detection techniques (inconsistencies, dead code, parsing, etc.) to take evolution into account. Even when faced with small changes, the proposed techniques require recomputing results from scratch, as such techniques are not incremental. That said, we ask:

How to make existing variability-aware error detection techniques incremental?

Change impact analysis

Before making changes, stakeholders must be able to assess or estimate the impact they will have on other features of the system. Change-impact analysis provides a sound basis to judge whether a change is worth the effort, or if inevitable, which features should be changed as a consequence.

A complete impact analysis is achieved by the traceability links retrieved during recovery, allowing us to calculate all feature dependencies among different artifacts and across time. For instance, if stakeholders decide to change the service interface of the stability control, ABS has to be adapted as a consequence. Since older versions of the system still need to be maintained (older cars still use them), stakeholders decide that the new API should also be used across different versions of the system, as this facilitates maintenance and communication within the development team. Impact analysis, in this case, completely depends on the recovered temporal links.

In addition to assessing change-impact analysis based on structural dependencies among artifacts and the elements they contain, one can lift analysis to the level of modular formal feature requirements specifications [14]. In this case, before committing to a change, stakeholders investigate which features will be affected, how, and in which variants, assessing whether the change breaks existing temporal properties; thus introducing undesired feature interactions. To illustrate this, consider that the stakeholders of the automotive product line formalize the specification of each individual feature attempting to detect possible bugs. At a given point, they decide to introduce support for the cruise control (CC) feature, and as such, write its related specification, stating:

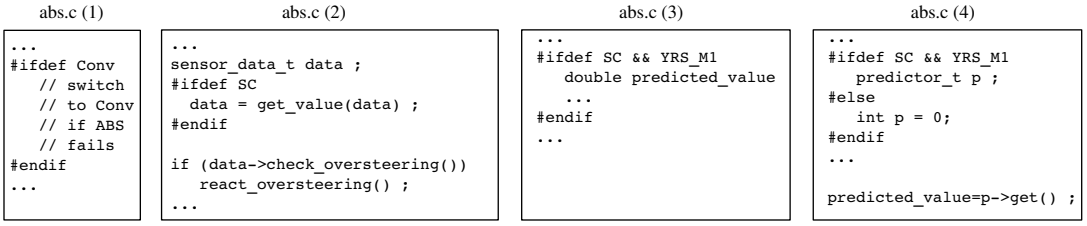

Figure 7: Example errors hidden by variability encoding

CC should increase the car's speed until it reaches the driver's set cruise speed.

In addition, the specification of the stability control declares:

When detecting loss of steering control, SC brakes the car to adjust steering to match the driver's intention. At this point, no subsystem should increase acceleration.

In this case, introducing CC causes an undesirable feature interaction in variants with CC and SC, as their composition cannot guarantee that no subsystem will increase acceleration when SC takes over control. The following sequence illustrates this: (1) the driver sets the cruise speed (thus engaging cruise control); (2) driver looses control of the car (SC is engaged); (3) cruise control continues to accelerate to achieve cruise speed. By eliciting the consequence of the change, stakeholders refrain from committing it. In turn, they set to improve it to prevent inconsistencies.

Similarly, impact analysis can verify whether changes in formal specifications will contradict the implemented software (or one of its variants), and vice-versa. Again, it preventively acts to avoid inconsistencies from being introduced.

Future directions and research questions

When reporting impact analysis, new tools could project slices of the feature model to provide a view containing only the impacted features. Each of these features would in turn, be connected to a second view made of program slices that correspond to the elements to be changed.

Although feature model slicing is not new [1], creating code slices as described is a goal for future research:

How to effectively create a code-slices view for each feature affected by a change?

Another major challenge is how to enable impact analysis to verify consistency between formal feature specifications and their realization in code, as currently this is mostly intractable and does not scale.

How to guarantee consistency between formal feature specification and their corresponding realization?

Architectural analysis

Architectural analysis aims to track the "health" of the features in the evolving system. Considering feature models to be one view of the software architecture, such tracking allows stakeholders to understand and manage the evolution of the architectural aspects at the feature level. In systems that do not rely on feature models, stakeholders can still follow whether the evolving software is aligned with the planned architecture (e.g., one class implementing ideally one feature, requiring scattering to be kept at a low pace, etc.).

Enabled by the traceability among all artifacts and across time, different metrics can be collected to provide indicators of the evolution in place and trend analysis based on the evolution history.

Metrics can be based on the aggregation of existing code (code size, cyclomatic complexity, etc.) and process metrics (number of bugs, number of developers, number of changes, etc.) on a feature basis, in addition to feature specific metrics like scattering, tangling, coupling and cohesion [20, 2]. Metrics can also be collected at the feature model level, on a feature basis (e.g., in/out degree dependencies), per subsystem/subtree (e.g., degree of orthogonality), or relative to the whole product line (variability factor, homogeneity, etc.) [8].

By quantifying and monitoring these feature-based architectural quality attributes, stakeholders can assess the evolution in place and have better decision making support in devising maintenance activities.

Future directions and research questions

Existing tools do not aggregate feature-related metrics, nor do they provide trend analysis over the evolution history.

In addition, it not clear how feature-based metrics relate to one another. For instance, Eaddy et al. [20] provide initial evidence that scattering is related to defects. Although promising, their results were collected in small to medium size programs, and analysis was focused on a single version of the subjects under analysis. Moreover, the authors manually traced the code to the features[2] of the investigated systems, but did not manage to trace it entirely. Keeping traceability among artifacts along the evolution timeline and monitoring how scattering evolves along the way would likely provide a better understanding on their hypothesis in a more realistic setting. We then set to ask:

Can we provide more evidence for the relationship between scattering and defects?

Considering that scattering is one effect of architecture misalignment, we set a more general question:

Is there a relation between architecture misalignment and defects?

5. RECOMMENDATIONS

Building on top of tracing and analyses, recommendations aim to assist stakeholders by concrete suggestions for consistency, impact and architectural analyses.

Future directions and research questions

When variability encoding causes inconsistencies, a recommender system can propose fixes. Different fixing strategies

[2]In their study, the authors refer to features as concerns.

must be used for different artifact types, and must not cause new inconsistencies to appear in any artifact (independent of its type and abstraction). That said, we pose the following research question:

How to devise a fixing recommender that integrates different types of artifacts, with different levels of abstraction, whose proposed fixes do not cause new inconsistencies in any of those artifacts?

Impact analysis recommendations can suggest which features are more likely to contain bugs after a certain change occurs, listing which artifacts and code elements deserve more attention.

Building on top of the collected metrics related to architectural analysis, the recommender system should also suggest to stakeholders which features are likely to have defects, or alternatively, provide a ranking of which features should be tested first (this is not connected to any specific change). That leads to the following question:

Which feature-based metrics are good defect predictors?

Another direction for future research is on proposing refactorings. Some concrete scenarios include: (a) features are too similar: when features become increasingly similar, the recommender system can propose their merge. This requires measuring similarity along the evolution timeline, and requires monitoring different artifacts with different levels of abstraction; (b) feature retirement: when a feature has been superseded by another feature in terms of its functionality, and has been reported to contain more bugs over time, the recommender system can suggest it to be retired; (c) feature modularization: when a feature is scattered, the recommender system can propose it to be put inside its own module. Scattering here, however, should not be blindly used, as not all features are worth modularizing (impact analysis can help on this)—if a feature is scattered, but the features containing its fragments never change, it possibly signals that modularization does not provide an immediate value.

Future research has to address which scenarios are likely to be required in practice and how to support them.

6. CONCLUSION

Controlling and executing change is a major challenge for most software projects. We have postulated that feature orientation of software design, and of the software development process is able to handle change more effectively than previous methods.

To this end, we have envisioned a feature-oriented project management and system development platform supporting traceability, feature-oriented analyses of implementation artifacts, and feature-oriented project-specific recommendation systems. For each of these areas, we have listed examples of existing work, which indicate that achieving our vision is feasible. We have also specified a number of existing research challenges within these fields.

In the future, we intend to work towards realizing this vision. We plan to execute empirical studies to verify our hypothesis, by studying change from the feature-oriented perspective. We also intend to build tools for handling traceability, analyses and recommendation for the suggested feature-oriented project management and system development platform.

7. ADDITIONAL AUTHORS

Additional authors: Claus Hunsen (University of Passau, email: `claus.hunsen@uni-passau.de`)

8. REFERENCES

[1] M. Acher, P. Collet, P. Lahire, and R. B. France. Decomposing feature models: language, environment, and applications. In *Proceedings of the International Conference on Automated Software Engineering*, pages 600–603. IEEE, 2011.

[2] S. Apel and D. Beyer. Feature cohesion in software product lines: an exploratory study. In *Proceedings of the International Conference on Software Engineering*, pages 421–430. ACM, 2011.

[3] S. Apel and C. Kästner. An overview of feature-oriented software development. *Journal of Object Technology*, 8(5):49–84, 2009.

[4] S. Apel, C. Kastner, and C. Lengauer. FeatureHouse: Language-independent, automated software composition. In *Proceedings of the International Conference on Software Engineering*, pages 221–231. IEEE, 2009.

[5] S. Apel, A. v. Rhein, P. Wendler, A. Größlinger, and D. Beyer. Strategies for product-line verification: Case studies and experiments. In *Proceedings of the International Conference on Software Engineering*. IEEE, 2013. To appear.

[6] H. U. Asuncion and R. N. Taylor. Automated techniques for capturing custom traceability links across heterogeneous artifacts. In *Software and Systems Traceability*, pages 129–146. Springer London, 2012.

[7] K. Beck and M. Fowler. *Planning Extreme Programming*. Addison-Wesley Longman Publishing Co., Inc., 1st edition, 2000.

[8] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.*, 35(6):615–636, 2010.

[9] E. Bodden. Inter-procedural data-flow analysis with ifds/ide and soot. In *Proceedings of the International Workshop on State of the Art in Java Program Analysis*, pages 3–8. ACM, 2012.

[10] C. Brabrand, M. Ribeiro, T. Tolêdo, and P. Borba. Intraprocedural dataflow analysis for software product lines. In *Proceedings of the International Conference on Aspect-oriented Software Development*, pages 13–24. ACM, 2012.

[11] K. Chen and V. Rajlich. Case study of feature location using dependence graph, after 10 years. In *Proceedings of the International Conference on Program Comprehension*, pages 1–3. IEEE, 2010.

[12] K. Chen, W. Zhang, H. Zhao, and H. Mei. An approach to constructing feature models based on requirements clustering. In *Proceedings of the International Conference on Requirements Engineering*, pages 31–40. IEEE, 2005.

[13] S. Chen, M. Erwig, and E. Walkingshaw. An error-tolerant type system for variational lambda calculus. In *Proceedings of the International Conference on Functional Programming*, pages 29–40. ACM, 2012.

[14] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. Symbolic model checking of software product

lines. In *Proceedings of the International Conference on Software Engineering*, pages 321–330. ACM, 2011.

[15] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. Model checking lots of systems: efficient verification of temporal properties in software product lines. In *Proceedings of the International Conference on Automated Software Engineering*, pages 335–344. ACM, 2010.

[16] K. Czarnecki and K. Pietroszek. Verifying feature-based model templates against well-formedness OCL constraints. In *Proceedings of the International Conference Generative Programming and Component Engineering*, pages 211–220. ACM, 2006.

[17] J.-C. Deprez and A. Lakhotia. A formalism to automate mapping from program features to code. In *Proceedings of the International Workshop on Program Comprehension*, pages 69–. IEEE, 2000.

[18] C. Dietrich, R. Tartler, W. Schröder-Preikschat, and D. Lohmann. A robust approach for variability extraction from the Linux build system. In *Proceedings of the International Software Product Line Conference*, pages 21–30. ACM, 2012.

[19] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software Maintenance and Evolution: Research and Practice*, 2011.

[20] M. Eaddy, T. Zimmermann, K. D. Sherwood, V. Garg, G. C. Murphy, N. Nagappan, and A. V. Aho. Do crosscutting concerns cause defects? *IEEE Trans. Softw. Eng.*, 34(4):497–515, 2008.

[21] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Trans. Softw. Eng.*, 29(3):210–224, 2003.

[22] P. Gazzillo and R. Grimm. SuperC: parsing all of C by taming the preprocessor. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 323–334. ACM, 2012.

[23] M. W. Godfrey and D. M. German. The past, present, and future of software evolution. In *Frontiers of Software Maintenance*, pages 129–138. IEEE, 2008.

[24] O. Gotel, J. Cleland-Huang, J. H. Hayes, A. Zisman, A. Egyed, P. Grünbacher, A. Dekhtyar, G. Antoniol, , and J. Maletic. *The Grand Challenge of Traceability (v1.0)*, pages 343–412. Springer-Verlag, 2012.

[25] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type checking annotation-based product lines. *ACM Trans. Softw. Eng. Methodol.*, 21(3):14:1–14:39, July 2012.

[26] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, pages 805–824. ACM, 2011.

[27] A. Kenner, C. Kästner, S. Haase, and T. Leich. Typechef: toward type checking #ifdef variability in C. In *Proceedings of the International Workshop on Feature-Oriented Software Development*, pages 25–32. ACM, 2010.

[28] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wasowski. Evolution of the Linux kernel variability model. In *Proceedings of the International Software Product Line Conference*, pages 136–150. Springer-Verlag, 2010.

[29] A. Marcus, V. Rajlich, J. Buchta, M. Petrenko, and A. Sergeyev. Static techniques for concept location in object-oriented code. In *Proceedings of the International Workshop on Program Comprehension*, pages 33–42. IEEE, 2005.

[30] S. Nadi and R. Holt. Mining kbuild to detect variability anomalies in Linux. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, pages 107–116. IEEE, 2012.

[31] L. Neves, L. Teixeira, D. Sena, V. Alves, U. Kulezsa, and P. Borba. Investigating the safe evolution of software product lines. In *Proceedings of the International Conference on Generative programming and Component Engineering*, pages 33–42. ACM, 2011.

[32] S. R. Palmer and M. Felsing. *A Practical Guide to Feature-Driven Development*. Pearson Education, 1st edition, 2001.

[33] D. L. Parnas. Software aging. In *Proceedings of the International Conference on Software engineering*, pages 279–287. IEEE, 1994.

[34] L. Passos, K. Czarnecki, and A. Wąsowski. Towards a catalog of variability evolution patterns: the Linux kernel case. In *Proceedings of the International Workshop on Feature-Oriented Software Development*, pages 62–69. ACM, 2012.

[35] S. Schulze, T. Thüm, M. Kuhlemann, and G. Saake. Variant-preserving refactoring in feature-oriented software product lines. In *Proceedings of the International Workshop on Variability Modeling of Software-Intensive Systems*, pages 73–81. ACM, 2012.

[36] C. Seidl, F. Heidenreich, and U. Aßmann. Co-evolution of models and feature mapping in software product lines. In *Proceedings of the International Software Product Line Conference*, pages 76–85. ACM, 2012.

[37] S. Simmons, D. Edwards, N. Wilde, J. Homan, and M. Groble. Industrial tools for the feature location problem: an exploratory study: Practice articles. *J. Softw. Maint. Evol.*, 18(6):457–474, 2006.

[38] R. Stallman, R. McGrath, and P. D. Smith. Gnu make manual, 2010.

[39] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat. Feature Consistency in Compile-Time Configurable System Software. In *Proceedings of the EuroSys Conference*, pages 47–60. ACM, 2011.

[40] T. Thüm, I. Schaefer, M. Kuhlemann, and S. Apel. Proof composition for deductive verification of software product lines. In *Proceedings of the International Workshop on Variability-intensive Systems Testing, Validation and Verification*, page 270–277. IEEE, 2011.

[41] M. T. Valente, V. Borges, and L. Passos. A semi-automatic approach for extracting software product lines. *IEEE Trans. Softw. Eng.*, 38(4):737–754, July 2012.

[42] N. Wilde and M. C. Scully. Software reconnaissance: mapping program features to code. *Journal of Software Maintenance*, 7(1):49–62, 1995.

[43] A. Zeller and G. Snelting. Unified versioning through feature logic. *ACM Trans. Softw. Eng. Methodol.*, 6(4):398–441, 1997.