2016 IEEE/ACM 38th IEEE International Conference on Software Engineering Companion

A Variability Aware Configuration Management and Revision Control Platform

Lukas Linsbauer
Advised by: Prof. Alexander Egyed and Dr. Roberto Erick Lopez-Herrejon
Institute for Software Systems Engineering
Johannes Kepler University (JKU) Linz, Austria
http://isse.jku.at/
lukas.linsbauer@jku.at

ABSTRACT

Modern systems need to run in many different contexts like hardware and software platforms or environmental conditions. Additionally different customers might have slightly different requirements towards systems. Therefore software systems need to be highly configurable and provide variable sets of features for different customers. There are various approaches to developing and managing such systems, like ad-hoc clone-and-own approaches or structured software product line approaches for each of which again several different techniques and tools exist to support them. While the different approaches come with advantages they also have several disadvantages and shortcomings. Some work only with specific implementation artifacts (e.g. source code but not models) and others exist only as plugins for specific IDEs which makes them intrusive or even unusable in some development environments. In our work we present a development process and tools for managing and engineering of highly configurable and variable systems that is generic, incremental, flexible and intuitive. We evaluated our approach on several case study systems from various different domains and origins like open source, academia or industry. The results so far showed promising results.

CCS Concepts

•Software and its engineering → Software configuration management and version control systems;

Keywords

Features, Variants, Configuration, Variability, Versioning

1. INTRO AND PROBLEM STATEMENT

Many real-world systems nowadays need to be highly configurable and provide a variable set of features so that they can be adapted to different contexts like hardware and software platforms, environmental conditions or to different customer requirements. A very prominent example is the Linux kernel that runs on many different architectures or hardware platforms like laptops, phones and embedded devices. The current techniques used to realize and maintain such highly configurable and variable systems have several important limitations.

Ad-hoc Approaches. These approaches are often also referred to as Clone-and-Own (C&O) approaches where existing system variants are cloned and adapted to fit a new context. The advantage is that C&O is very intuitive and flexible. A new variant can be created whenever needed without much preparation and there is no danger to affect existing variants in an undesired way. However, the disadvantage is in the maintenance and evolution of the system and its variants. Already with a fairly small number of variants propagating changes (e.g. bug fixes) to all affected variants is a time consuming and error prone task. Determining which variants are even affected is already not trivial. Additionally, when creating a new variant deciding which variant to clone and use as a base is also not easy once a larger number of variants to choose from have been created [2]. Also, often it would be ideal to take implementations from several existing system variants, and not just from a single one. In practice, since this is very difficult to do manually, developers often end up implementing the same functionality multiple times in different variants which makes further maintenance even more difficult.

Structured and Systematic Approaches. An example for a structured reuse approach is intended for systems with high variability is Software Product Line Engineering (SPLE). Functionality (i.e. features) is implemented in a common, integrated platform as opposed to separate variants, which makes applying fixes easy. However, applying a change also requires careful consideration of the effects it will have, e.g. a change impact analysis to determine whether system variants will be affected that should not be. Additionally, building the integrated platform requires a large upfront investment of time and money as all possible system variants and their features must be carefully planned and implemented at once. Evolving this platform at a later time (e.g. adding a new system variant or new features) is difficult as existing ones must not be broken in the process. Various mechanisms exist to implement SPLs. However, in practice often simple pre-processors in combination with annotated source code or simple runtime configuration files are used. Often companies even write their own custom platforms because existing ones do not satisfy their needs

ICSE '16 Companion, May 14-22, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-4205-6/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2889160.2889262

[6]. These mechanisms are often limited to a certain type of implementation artifacts, for example a pre-processor needs to operate on textual implementation artifacts like source code but cannot be used on models or diagrams. This is becoming increasingly important as especially complex systems very rarely only consist of a single type of artifacts.

2. PROPOSED APPROACH

We propose a development process and supporting tools and mechanisms that are generic with respect to the implementation artifacts and transparent to possible IDEs or build tools. Our approach leverages benefits of both adhoc approaches like clone-and-own and systematic reuse approaches like SPLE. It is flexible, robust and intuitive and allows for structured and efficient reuse, maintenance and evolution of systems.

We believe that the correct point of integration for such an approach is not a custom tool, not an IDE specific solution, [14, 9] and also not the implementation of the system under development itself (e.g. annotations in the source code). The most intuitive and most unobtrusive link in the development chain that pretty much all development processes share is the revision control system (for example Git or Subversion). However, current version control systems are not aware of variability at all. Therefore developers often have no other choice but to use additional tools and mechanisms (for example pre-processors) to deal with the variability in their systems, or they try to use mechanisms like branching to represent different variants of a system [10] which is less than ideal, as for example common code of variants is replicated in every branch and propagation of changes to those common parts to all affected branches must be performed diligently to avoid inconsistencies.

Our approach maintains a central repository similar to SPLE approaches on the one hand and revision control systems on the other hand (a very convenient commonality indeed). However, in contrast to SPLE the repository is not filled upfront but rather incrementally on demand like in every revision control system. New features and new system variants (as new combinations of (new) features) are added to the repository whenever needed with support for automated reuse from existing variants. Next we explain how we can achieve this.

2.1 Operations

Our approach is inspired by current revision control systems like Git or Subversion. This has the benefit that developers are already familiar with them. Also, the realization of our approach as a revision control system seems logical because almost every project, regardless of the type of implementation artifacts or choice of IDE, makes use of them which makes an even more seamless integration possible.

We define two basic operations that enable our approach:

- `checkout <configuration>`: Retrieves a given configuration from the repository.
- `commit <configuration>`: Adds a given configuration to the repository.

A `<configuration>` represents a variant of a system and is simply given as a list of features with optional versions, for example the configuration string `"F1.1, F2, F3'"` indicates feature F1 in version 1, feature F2 in its most current version and a new version of feature F3. A configuration containing a new version of a feature (or similarly a totally new feature) as feature F3 in this example simply indicates that a new (version of) a feature was added to the implementation.

Instead of storing revisions of the full system we store revisions of every feature individually. Consequently, instead of checking out a specific revision of a system, a specific configuration (i.e. variant of the system) is checked out.

The operations `commit` and `checkout` require complex logic in the background as we do not require a developer to manually trace features to their implementation or modularize the implementation in any way. In fact, the development of the actual system will be no different. The operations are based on algorithms presented in our previous work (see [8, 3]) that, given a set of system variants, trace features to their implementation artifacts at a fine-granular level (e.g. an implementation artifact can be a single statement in a source code file), which is necessary because variability is often realized below file level. For this current work we extend these algorithms to also be able to deal with evolving features (i.e. different versions of features). The `commit` operation compares the contents of the repository with the configuration and its implementation being committed and automatically computes traces from features to their implementation artifacts. The repository then does not store each variant separately but instead stores these traces. The `checkout` operation then needs to select the necessary implementation artifacts from the repository required for a given configuration and recompose them correctly. Note, that the `checkout` operation is also able to check out configurations (i.e. retrieve system variants) that have never been configured (i.e. committed) before. In such cases, as much as possible of the known implementation is reused from the repository (e.g. known features that are part of the configuration). Unknown features or new combinations of existing features might require implementation artifacts that are not yet in the repository. In such cases the checkout will, in addition to returning an as complete as possible partial implementation, also return a list of indicators for missing implementations to the developer.

As an example consider the scenario where we want to make a change to the implementation of a variant. Such change may have several causes. It may be a bug fix to a feature F that should affect every system variant that provides that feature. Clone-and-Own approaches would require this bug fix to be applied to every variant separately. In SPLE the fix only needs to be applied once to the integrated platform. However, it may also be a change to the behavior of a feature that was requested by a customer and should only affect variants intended for that customer. In this case Clone-and-Own can provide great flexibility while SPLE approaches run into a problem here as a change to a feature always affects all variants with that feature, so an option would be to clone the feature into a new one and change it. Our approach can deal with this simply by checking out any configuration containing feature F and specifying the correct configuration during the `checkout` and `commit` operation. First, check out any configuration suitable for applying the change. This may be any configuration containing the feature F that the developer feels comfortable with, for example `checkout BASE, F`. Then simply use `commit BASE, F'` to indicate that the changes are intended for a new version of feature F, or `commit BASE, F, CSF` to indicate that the feature F has not changed but rather there is a customer specific change – let us refer to it as customer specific feature (CSF) – that interacts with feature F in such a way that it changes its behavior. The traceability algorithms will take care of computing the correct traces in either scenario [8, 3].

2.2 Heterogeneous Implementation Artifacts

Systems are rarely implemented by just a single type of artifacts. Therefore we aim at supporting heterogeneous types of artifacts in systems, by making it extensible with artifact adapters, starting from simple text files to source code, UML models or even CAD diagrams. Every artifact type for which an adapter exists can automatically be traced to features during a `commit` operation and subsequently composed during a `checkout` operation.

2.3 Migration to our Approach

The migration from existing sets of system variants to our approach is simple and can easily be automated. All that is required is that every existing system variant be checked into the repository with the correct configuration.

For the migration of SPLs we see two possible scenarios. All possible variants can be derived from the SPL platform and then checked into our repository just as in the case of a set of separate system variants. However, often this set of possible variants is too large to derive all of them. As an alternative we can leverage the fact that traceability information must be present at least implicitly in the SPL platform, otherwise it could not derive its variants. Therefore a custom migration tool can be created (using the APIs provided for our approach) that fills our repository with the necessary traceability information directly. How easy or difficult it is to retrieve this information from the SPL platform however depends on the variability mechanisms it uses. For example, extracting traceability information from a SPL that uses pre-processor annotated source code will require a custom parser that is aware of the presence conditions in the annotations.

3. EVALUATION

We evaluate our proposed approach based on two aspects. First we assess the quality of the implementation of a variant after it has been checked out. We do this by comparing the implementation of the checked out variant to a reference implementation of the same variant (i.e. with the same configuration). The closer the checked out variant is to its reference implementation the higher its quality. The process is shown in Figure 1. From a given set of n variants of a case study system (1) we use subsets $v \subseteq V$ of different cardinalities to be *commited* (2) into our repository (3). Subsequently we *check out* (4) variants V' (5) with the same features as the given variants V and compare the variants in V to the corresponding ones in V'. Our results so far have shown that already subsets v of rather small cardinalities result in a high quality of checked out variants, even those that were not contained in v [8, 3].

Secondly, we assess the correctness of dependencies between the features in the repository. These dependencies are derived from the dependencies between implementations to which the features were traced. The assumption here is that wrong traces would lead to wrong dependencies between features and that correct dependencies are a good indication for

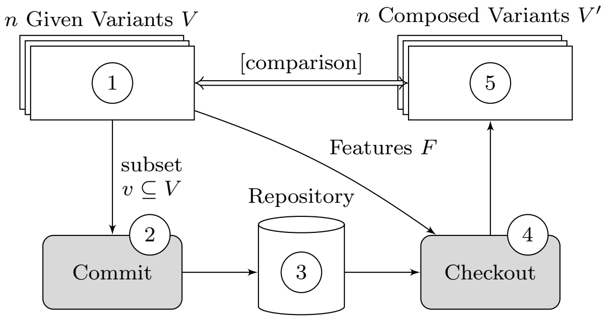

Figure 1: Traceability Evaluation

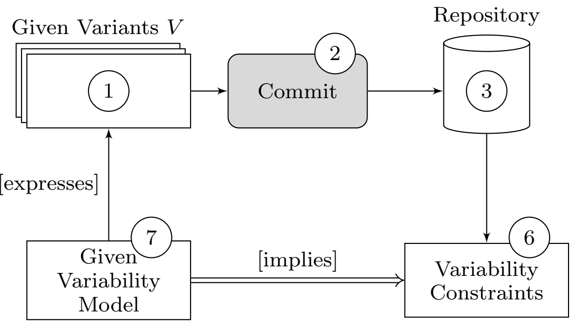

Figure 2: Variability Constraints Evaluation

the computed traces being correct. For this evaluation we assume a variability model to be available for the case study systems that describes exactly the dependencies and constraints between the features in the case study system. The process is shown in Figure 2. Again, a set of given variants V (1) is committed (2) into our repository (3). Then, the dependencies among features in the repository (6) (based on the dependencies between the implementation artifacts the features were traced to during the commit operation) are compared to the given variability model (7). The given variability model may not allow variants that violate any of the constraints in our repository, otherwise the variant would be malformed (i.e. not compile or be otherwise incomplete). However, the given model may impose more constraints than those in the implementation (e.g. variants that would be well-formed but make no sense semantically). Therefore, for a positive outcome, the given variability model must imply the variability constraints in our repository. Again our results so far have shown that the traces (and the resulting dependencies between implementations of features) are mostly consistent with given feature models [6]. In fact, in some cases where inconsistencies occurred we found that the fault was actually not in our computed traces but in the given variability model.

4. CONTRIBUTIONS

We briefly summarize the major contributions of our work:

- *Extraction of variability information from system variants.* This includes the computation of traces from features to fine-grained implementation artifacts with the ability to compute not only traces for single features, but also feature interactions and absence of features, non-unique traces and dependencies between traces (e.g. a feature requiring the presence of another feature). (`commit` operator)

- *Automatic composition of variants using the extracted variability information.* (`checkout` operator)
- A *generic, incremental, intuitive and flexible development process and tools* that leverage benefits of both structured and ad-hoc approaches.

5. RELATED AND PRIOR WORK

Martinez et al. present a generic and extensible approach for adopting software product lines from sets of product variants, called BUT4reuse [9]. Similarly to our work they also perform feature location and constraints discovery.

Montalvillo and Diaz propose to enhance existing version control systems with first class operations for supporting the two typical SPL development life cycles domain engineering and application engineering [10]. As a proof of concept they augment the GitHub website with such extra functionality. In contrast, since our goal is to support the most common scenario in industry, our operations are oriented more towards well known revision control systems and not so much towards SPLE paradigms.

Rubin et al. survey feature location techniques for mapping features to their implementing software artifacts [11]. The extraction process in our work can also be categorized as a feature location technique, only that we also consider additional problems like feature interactions instead of just single features and also the order of artifacts instead of just their presence or absence.

Rubin et al. propose a framework for managing product variants that are the result of clone-and-own practices [12]. They outline a series of operators and how they were applied in three industrial case studies. These operators serve to provide a more formal footing to describe the set of processes and activities that were carried out to manage the software variants in the different scenarios encountered in the case studies. We believe that our variability extraction techniques can provide the functionality of some of these operators and we therefore plan to apply our techniques to such scenarios.

6. PROGRESS AND PUBLICATIONS

We have already devised algorithms for computing fine-grained traces from features to implementation artifacts and vice versa that lay the foundation of our proposed development process and enabling tools [8, 3]. Our very first work on the subject computes traces from features and feature interactions to their implementation artifacts from a set of product variants [8]. Such traceability information is crucial when dealing with variable features as it is fundamental for many tasks to know where and how features are implemented. In our subsequent work we improved on this by allowing for non-unique traces and more fine-grained traceability information [3]. A proof of concept application to a real world system from industry showed promising results [6]. An implementation of the early version of the tool as an Eclipse plugin was implemented [4]. Lastly we broadened the vision of our approach to also cover the evolution of systems and their features [7]. The realization of that broader vision is currently ongoing work.

What remains to be done is the integration of these algorithms into a usable tool and to provide a proper API so that they can be utilized in custom tools like for example IDE specific plugins. An evaluation using other implementation artifacts like UML models or CAD diagrams and more case study systems in general is also a remaining goal. Lastly we plan a study in a realistic environment to evaluate the usefulness of the approach in practice.

7. ACKNOWLEDGMENTS

This work is supervised by Prof. Alexander Egyed (alexander.egyed@jku.at) and Dr. Roberto Erick Lopez-Herrejon (roberto.lopez@jku.at). This research is funded by the Austrian Science Fund (FWF) projects P25289-N15 and P25513-N15.

8. REFERENCES

[1] *ICSME 2014*. IEEE Computer Society, 2014.

[2] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki. An exploratory study of cloning in industrial software product lines. In *CSMR 2013*, pages 25–34, 2013.

[3] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed. Enhancing clone-and-own with systematic reuse for developing software variants. In *ICSME 2014* [1], pages 391–400.

[4] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed. The ECCO tool: Extraction and composition for clone-and-own. In *ICSE 2015*, pages 665–668. IEEE, 2015.

[5] T. Kishi, S. Jarzabek, and S. Gnesi, editors. *SPLC 2013*. ACM, 2013.

[6] L. Linsbauer, F. Angerer, P. Grünbacher, D. Lettner, H. Prähofer, R. E. Lopez-Herrejon, and A. Egyed. Recovering feature-to-code mappings in mixed-variability software systems. In *ICSME 2014* [1], pages 426–430.

[7] L. Linsbauer, S. Fischer, R. E. Lopez-Herrejon, and A. Egyed. Using traceability for incremental construction and evolution of software product portfolios. In *SST 2015*, pages 57–60, 2015.

[8] L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed. Recovering traceability between features and code in product variants. In Kishi et al. [5], pages 131–140.

[9] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. L. Traon. Bottom-up adoption of software product lines: a generic and extensible approach. In Schmidt [13], pages 101–110.

[10] L. Montalvillo and O. Díaz. Tuning github for SPL development: branching models & repository operations for product engineers. In Schmidt [13], pages 111–120.

[11] J. Rubin and M. Chechik. A survey of feature location techniques. In *Domain Engineering, Product Lines, Languages, and Conceptual Models*, pages 29–58. 2013.

[12] J. Rubin, K. Czarnecki, and M. Chechik. Managing cloned variants: a framework and experience. In Kishi et al. [5], pages 101–110.

[13] D. C. Schmidt, editor. *SPLC 2015*. ACM, 2015.

[14] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich. Featureide: An extensible framework for feature-oriented software development.