
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Department of Computer Science and Engineering
Software Engineering Master Thesis Proposal, 30 hec

An Empirical Assessment of an Intention Language for Software Merging

---

Completed courses relevant for thesis work:
Software Evolution Project
Model-driven Engineering
Empirical Software Engineering

1 Introduction

Maintaining variability in software systems can be achieved in two ways: forking and integrating. Variants of the same software can coexist in an ecosystem consisting of so-called forks, which are large-scale clones, with some small-scale alterations to change the desired behavior [1]. By integrating all variability in a single repository, maintainability can be increased, and duplication of effort reduced [2], [1]. In an integrated platform, variability is controlled at compile-time or at runtime.

When a fork variant is merged into an integrated platform through a version control system, the developer performing the merge must manually handle both the variability and potential conflicts on a source code level [3], [4]. Since feature integration is a complex task, performing it manually is time-consuming and error-prone. Understanding the patterns of such conflict resolution and variability management, and abstracting it away from the source code, considering the *intentions* of the developer allows for development of variability-aware merging tools. The author is currently involved in collaborating in the development of a language with the aim at capturing such intentions. The expected benefits of a variability-aware merging tool based on such a language is increased work speed and reduced software defects.

This study aims to analyze and refine the aforementioned intentions language describing the intentions of software merging and to empirically assess a merging process that employs the language and its implementation in a merge tool. The language is validated and refined based on data obtained from mining the open-source 3D-printer firmware Marlin[1], which in 2014 had 1588 forks [1]. Following this, the language is implemented in a merge tool, which is evaluated using a controlled experiment. This thesis will contribute a dataset of merges; an instantiation of the intentions language for all the merges; empirical data on the the use of the language; and a qualitative investigation into the differences between intention-based merging and manual merging.

[1] `https://github.com/MarlinFirmware/Marlin`

2 Statement of the problem

Manual, unstructured merging is time-consuming, and possibly concerned with the wrong abstraction level (i.e., source code) with respect to variants, which could be error-prone [3], [4].

Features and variants. Features represent some logical unit of a system, and are regarded as part of the domain or as required by the market [5], [6]. Variants of software are created to fulfill different requirements in similar products [7], [1]. One or more features can be grouped inside a variant. A common manifestation of variants is forks, which are repository-scale clones of an original codebase, called the *mainline*. This strategy is known as *clone-and-own*, where small-scale changes are made to a large-scale copy in order to create a new variant [1]. Features are spread though out the array of forks that comprise the system.

Orthogonal to variability through clones, stands integrated variability. By gathering all features in one common repository, variants are created by composing features. This is realized by source code-level constructs (such as the C preprocessor `#ifdef` statement) guarding the feature code, meaning that behavior can be enabled or disabled at compile-time or runtime.

There is a tradeoff between time effort and maintainability with respect to the two variability strategies. Clone-and-own is an easy and fast method, but does not scale due to the inherent impact on the maintainability of a large number of clones.

Reintegrating forks. Using the terminology of Buckley et al. [8], fork variants represent divergent changes being developed asynchronously in parallel [3], [1]. Merging variants back into an integrated mainline platform (from clone-and-own to integrated platform) has the advantages of increased maintainability; features and bug-fixes being consolidated; and reducing unintentional code duplications [2], [1]. Since forks are inherently decentralized with respect to both organization and actual code base, knowledge and effort can be lost if they are not circulated back into the ecosystem [1], [9]. Currently, the process of merging a fork into the mainline is based on manual unstructured merging, relying fully on the developer to create a semantically correct merge [3], [4]. In order to resolve a merge, the ambitions of the involved code must interpreted and leveraged. Since the ambition is not always explicit, there is significant mental overhead involved in the activity.

Motivation for a descriptive language. Today, a developer performing a feature merge must make ad-hoc decisions on how the final result should behave with respect to code blocks and presence conditions [10]. At the same time, the concern is at the source-code level dealing with fundamental (preprocessor) language constructs. We propose the following distinction between *intentions* and *operations*: intentions are abstract actions describing how the developer wishes to resolve the merge. Examples include "keep this block of code" or "keep this block of code but make it a feature". Operations are actual transformations on the source code such as those proposed in [10]. A mapping or parameterization from intentions to operations would enable tool support for intention-based merging. The semantics of the intentions language ensures the integrity of the code upon which operations are applied to. The language would be the basis for creating a tool to aid the merging process structurally [3], [4]. In the collaboration, we are working to define such an intentions language, but we do not yet know how applicable or complete it is.

Investigation into tool support. Gousios et al. point out that an important improvement for the merging process is tool support in terms of work prioritization and estimated time for merging [9]. A prerequisite for the capability of migrating *clone-and-own* to an integrated platform is having proper merge support [7]. An empirical evaluation of a prototype tool is required in order to establish whether tool support is actually beneficial for the merging process. Empirical assessment of such tooling is difficult and requires proper execution.

3 Purpose of the study

The purpose of the study is to explore and evaluate the merge process using intentions that capture the merge semantics as building blocks. The soundness of the intentions language is verified against actual merges. Following this, an evaluation of a novel merge tool prototype is performed to assess the improvement to the fork merging process. By contributing a dataset of merges; an instantiation of the language for each merge; empirical data and qualitative analysis on the use of the language within the tool, the long-term goal of providing an automated tool and workflow for fork integration is abridged. This is intended to benefit both researchers and practitioners.

4 Review of the literature

Merging and tool support. Merge conflicts arise when software revisions and variants diverge concurrently [4], [8]. Revision control systems can be divided into two classes: *unstructured* revision control systems that operate on plaintext; and *structured* revision control systems that rely on structure and semantics of the document being stored in order to resolve conflicts automatically [3], [4]. The former has reached popularity due to being language independent, examples of such revision control systems include Git, Subversion, and CVS, while structured revision control systems are of academic interest, since they are not language independent [4]. Apel et al. [4] introduce the concept of a *semistructured* revision control system, and in particular the *semistructured* merge, which combines the strengths of the two classes, while minimizing their inherent weaknesses.

Forked and integrated variants. For forked variants, the variability lies in an array of repository clones with minor changes in them. An integrated platform consolidates the features and leverages programming language constructs to manage multiple variants simultaneously [1]. The former has lower initial costs, but comes with maintainability issues, while the latter requires a significant commitment [7]. Antkiewicz et al. [7] propose a strategy for migration from *clone-and-own* with low-risk, step-by-step adaption of an integrated platform through what they call a *virtual platform*. It has also been shown that the two types are used in parallel in the Marlin project [1]. Stănciulescu et al. [10] demonstrate the benefit of a projection-based variation control system to alleviate the complexities of maintaining and evolving code with preprocessor annotations.

Practical studies on clone-and-own systems. Schmorleiz and Lämmel [2] realize a naïve virtual platform by introducing a tool that analyzes and annotates changes across cloned-and-owned variants. As a way to manage similarity, code fragments are annotated with invariants that propagate to across fragment clones in all variants. Adherence to the invariants is either performed manually or automatically by their tool. Santos and Kulesza [11] investigate conflicts that arise from integration of evolved clones. In their study of a web-based Java system, the most prominent conflict type is indirect – changes in the source system are in the call graph of other changes in the target system – which they find is in line with the motivation for semistructured merging [4].

Efficiency experiment. Berger et al. [12] have previously conducted a controlled experiment using students and industrial developers, to determine their editing efficiency in the projectional editor MPS, compared to a traditional, parser-based editor. The participants are given four simple programming tasks to complete, with their screens being recorded to measure completion time and editing efficiency in terms of operations and mistakes. Afterwards, participants respond to a questionnaire with open and closed questions, and take part of a debriefing

interview, to gather quantitative and qualitative experiences of the differences and benefits of the two editor types.

5 Research questions

The research questions are divided into two parts: language evaluation (**RQ1**), and an evaluation of the intention-based merge tool (**RQ2-3**).

RQ 1: *To what extent do the intentions properly model and reflect the intentions as evidenced in actual merges?* We perform this verification step in order to assert that the intentions language can be instantiated to capture actual witnessed merges from real scenarios. This seeks to establish the completeness and possibly correctness of the language.

RQ 2: *Is using the intention-based merge tool beneficial for merging variants? To what degree does the intention-based merge tool facilitate correct merges? How much faster is the merge process using the intention-based merge tool?* Our goal is that both code quality and time effort can be improved by a workflow incorporating the intention-based merge tool, which we summarize as the overall beneficence of the tool.
Hypothesis 1: Using the intention-based merge tool leads to fewer bugs than manual merging.
Hypothesis 2: Using the intention-based merge tool gives faster merging.

RQ 3: *How is the merge process different using the intention-based merge tool?* This is an investigation into the perceived and evident qualitative differences of the two approaches, synthesizing the results from the findings of the previous questions.

6 Methods and procedures

The methods and procedures are presented chronologically and linked to the research questions they correspond to. After this, a rough time schedule of activities is listed. Note that the development of the intention-based merge tool is a shared effort in the collaboration.

1. Identify 50 merge examples in the Marlin ecosystem that contain `#ifdefs`. These merges represent manual variability management, where fork variants have been integrated and a variability mechanism has been inserted or deleted. They are relevant both on their own as real-life examples, and as a way to verify the soundness of our proposed language. Such merge examples can either be located in the mainline, or in forks that integrate the remote mainline. This could for example be present in commits preceding a pull request.

2. Continuous analysis and refinement of the intentions language on the 50 identified merges by instantiating the language (**RQ1**). This means verifying that the language is able to capture real-life examples of merges and if necessary, refining it.

3. Internal evaluation with merge process examples being performed by the researchers. Correctness metrics together with preliminary findings regarding performance and efficiency (**RQ2-3**) are collected. This is meant as an internal pre-study in preparation of the controlled experiment, both as quality assurance of the intention-based merge tool and as an elicitation of valuable information for the controlled experiment, questionnaire and interviews.

4. Conduct a controlled experiment to assess usefulness by evaluating fork merging using the intention-based merge tool compared to traditional manual unstructured merging from the perspective of the developer in merge scenarios. For the experiment, students must be recruited. The students will be split randomly into two groups, one using an unstructured merging tool, and one using the intention-based merge tool. They will be presented with code and instructions for the merging tasks to perform. The instructions contain a quick domain background, describe the goal of the merge, possibly with code references. The resulting merges will be compared to the known merge outcome to measure the correctness (**RQ2**) of the two groups. There must be some leniency with respect to syntax – i.e., only the semantics of the merge comparison should be equal, since especially the manual unstructured merging can introduce arbitrary syntax changes.

   The screens of the participants will be recorded, and afterwards the recordings will be reviewed in order to measure the completion time and number of mistakes made (**RQ 2**).

5. Each participant from the **tool** group is asked to participate in a questionnaire after the experiment (**RQ3**). The questionnaire contains closed and open questions about their background (experience) and their impressions about the possible benefit of the intention-based merge tool. Some examples of possible Likert questions are: *Merging is faster. There are fewer mistakes. Intention-based merging is too complex. Tool is not mature enough.* Together with this, we also conduct recorded interviews with participants about perceived benefits, challenges, and improvements of the intention-based merge tool.

Time schedule. The available time comprises about 18 weeks total, from mid January to late May.

- Iterative fork mining, analysis and refinement of the intentions language. 6 weeks.
- Experiment design. 6 weeks.
- Experiment execution. 2 weeks.
- Synthesis of results. 4 weeks.

7 Limitations and delimitations

The development of the intentions language and the intention-based merge tool is a shared effort in the research collaboration. The thesis intended to be limited to the assessment of the two.

Threats to internal validity. When observing the merge examples to establish their intentions, mistakes can be made. This is mitigated by comparing the conflict and the resulting merge.

If our prototype tool is defective or otherwise buggy, this introduces bias. The early internal evaluation serves as quality assurance, to find potential bugs or issues in our intention-based merge tool an mitigate any such issues before the controlled experiment. Along with this, in order to reduce bias from non-essential factors (e.g. the menu structure), both groups of the controlled experiment will receive an introduction to their respective editor before performing the tasks.

Threats to external validity. The source of the real-life examples is a single ecosystem, that of Marlin. Due to the size and complexity of the codebase and ecosystem, the verification of the intentions language should be both representative and generalizable.

Since Marlin is developed in C++, the C preprocessor (i.e. conditional compilation) will be the variability mechanism considered throughout the study, although this should not impact the generalizability to other line-based mechanisms.

Using students for the controlled experiment could be called into question, since they are neither domain experts of Marlin, nor professional software developers. Programming-wise, students should be as proficient as professionals, but the questionnaire asks for the background and previous programming experience so that this can be taken into account. The tasks in the controlled experiment do not require previous domain knowledge as such, since the outcome is described in the task. Therefore, the findings should be applicable to practitioners.

8 Significance of the study

Overall the study aims to towards simplifying the process of merging variants to an integrated platform, which has relevance for both practitioners and researchers. The concrete contributions are **providing a dataset of merge examples** together with **instantiation of our intentions language** that can be used in future research and prototyping, together with the **empirical assessment of the intention-based merge tool prototype**. Tool support is key to the effective management of forks and their integration [7], [9], which in turn enables migration from clone-and-own to integrated platform with lower associated cost and risk. For example, this facilitates adoption of the virtual platform migration strategy.

References

[1] S. Stanciulescu, S. Schulze, and A. Wasowski, "Forked and integrated variants in an open-source firmware project," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sept 2015, pp. 151–160.

[2] T. Schmorleiz and R. Lämmel, "Similarity management of 'cloned and owned' variants," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, ser. SAC '16, 2016, pp. 1466–1471.

[3] T. Mens, "A state-of-the-art survey on software merging," *IEEE Transactions on Software Engineering*, vol. 28, no. 5, pp. 449–462, May 2002.

[4] S. Apel, J. Liebig, B. Brandl, C. Lengauer, and C. Kästner, "Semistructured merge: Rethinking merge in revision control systems," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11, 2011, pp. 190–200.

[5] T. Berger, D. Lettner, J. Rubin, P. Grünbacher, A. Silva, M. Becker, M. Chechik, and K. Czarnecki, "What is a feature?: A qualitative study of features in industrial software product lines," in *Proceedings of the 19th International Conference on Software Product Line*, ser. SPLC '15, 2015, pp. 16–25.

[6] S. Apel and C. Kästner, "An overview of feature-oriented software devlopment," *Journal of Object Technology*, vol. 8, no. 5, pp. 49–84, 2009.

[7] M. Antkiewicz, W. Ji, T. Berger, K. Czarnecki, T. Schmorleiz, R. Lämmel, S. Stănciulescu, A. Wasowski, and I. Schaefer, "Flexible product line engineering with a virtual platform," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion 2014, 2014, pp. 532–535.

[8] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel, "Towards a taxonomy of software change," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 17, no. 5, pp. 309–332, 2005.

[9] G. Gousios, A. Zaidman, M.-A. Storey, and A. van Deursen, "Work practices and challenges in pull-based development: The integrator's perspective," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15, 2015, pp. 358–368.

[10] S. Stanciulescu, T. Berger, E. Walkingshaw, and A. Wasowski, "Concepts, operations, and feasibility of a projection-based variation control system," in *32nd IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016.

[11] J. Santos and U. Kulesza, "Quantifying and assessing the merge of cloned web-based system: An exploratory study," in *The 28th International Conference on Software Engineering and Knowledge Engineering, SEKE 2016*, 2016, pp. 583–588.

[12] T. Berger, M. Völter, H. P. Jensen, T. Dangprasert, and J. Siegmund, "Efficiency of projectional editing: a controlled experiment," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, 2016, pp. 763–774.