CHALMERS | UNIVERSITY OF GOTHENBURG

Empirical Assessment of a Language for Variant Integration

Master's thesis in Information Technology

WILHELM HEDMAN

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2017

Empirical Assessment of a Language for Variant Integration

WILHELM HEDMAN

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2017

Empirical Assessment of a Language for Variant Integration

WILHELM HEDMAN

© WILHELM HEDMAN, 2017.

Supervisor: Thorsten Berger, Dept. of Computer Science and Engineering
Examiner: Jan-Philipp Steghöfer, Dept. of Computer Science and Engineering

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Gothenburg, Sweden 2017

Empirical Assessment of a Language for Variant Integration

WILHELM HEDMAN
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Developing new variants by cloning is fast and simple, but as the drawbacks outweigh the early benefits, the clones must be re-engineered into an integrated platform. The challenging re-engineering process is hindered by the lack of effective tool support, because variant integration is an architectural concern, and contemporary merge tools being used for the task operate on source code as plain text. In this thesis, the recently proposed process of *intention-based* variant integration is evaluated on the subject systems Marlin, BusyBox and Vim. We replay actual integration merges using the *intentions* language to verify that it can be used in a real setting, followed by a controlled experiment comparing the efficiency of integration in the prototype tool INCLINE and an unstructured two-way merge tool. The results show that the intentions can capture the changes in the 35 sampled integration merges. The controlled experiment shows that for intention-based integration, fewer edit operations are required, but more actual time, and no difference can be observed for the number of defects inserted. This lays the foundation for tool improvement and subsequent user studies of intention-based integration in software product lines.

Keywords: software product lines, variability, clone management, re-engineering, software merging, controlled experiment, empirical software engineering.

Contents

1 Introduction **1**

2 Background **3**

2.1 Revision Control Systems 3
2.1.1 Merging 3
2.1.2 Forks 4
2.2 Features and Variability 4
2.2.1 FOSD and SPLE 4
2.2.2 What is a Feature? 5
2.2.3 Software Variants and Variability 5
2.2.4 Platform Re-engineering 6
2.3 Projectional Editing 6
2.4 Intention-based Variant Integration 7

3 Methodology **9**

3.1 Data Collection 9
3.1.1 Collecting Ground Truth Merge Examples 9
3.1.2 Creating Authentic Integration Scenarios 10
3.2 Completeness Evaluation 11
3.3 Internal Evaluation 11
3.4 Controlled Experiment 12
3.4.1 Objective 12
3.4.2 Treatments 13
3.4.3 Subjects 13
3.4.4 Design 13
3.4.5 Execution 14
3.4.6 Analysis 14

4 A Dataset of Integration Examples **15**

4.1 Variability-related Merges from Marlin 15
4.2 Replaying Merges with Intentions 16
4.3 RQ1: Does the set of intentions suffice for variant integration? 16

5 Empirical Evaluation **19**

5.1 Internal Evaluation 19
5.2 Controlled Experiment 20
5.2.1 Editing Efficiency 20

5.2.2 Defects . . . 20
5.2.3 Post-experiment Opinions . . . 22
5.3 RQ2: Is there a benefit over manual integration with a diff tool? . . . 22
5.4 RQ3: How is the integration process different using the intention-based integration tool? . . . 23
5.4.1 Editing . . . 23
5.4.2 Intentions Semantics . . . 23
5.4.3 Integration Support . . . 24

6 Discussion **27**
6.1 Inferences . . . 27
6.2 Threats to Validity . . . 28
6.2.1 Internal Validity . . . 28
6.2.2 Conclusion Validity . . . 28
6.2.3 External Validity . . . 29

7 Conclusion **31**

Bibliography **33**

A Intention Language Examples **I**

B Controlled Experiment Tasks **III**

C Questionnaires **V**
C.1 Registration Questionnaire . . . V
C.2 Post-experiment Questionnaire . . . V

1 Introduction

Highly configurable software systems stem from the need of adapting software to fit different hardware or to meet specific customer demands. Such software systems can be developed in two orthogonal strategies: *clone-and-own*, and *integrated platform*. Clone-and-own is a quick and simple approach to create new variants of software, by cloning the entire source code of a project and making the required small-scale changes. The integrated platform approach instead offers systematic reuse, but requires significant engineering effort to adopt, which is costly. When the number of clones in a clone-and-own setting spirals out of control, the cloned variants must be re-engineered into a single integrated platform from which the variants can be derived. By adopting an integrated platform, maintainability can be increased and duplication of effort reduced [1], [2].

Legacy systems using clone-and-own are abundant. Since rewriting them is not a viable option in industrial settings, re-engineering them from clone-and-own to an integrated platform is required at some point. This re-engineering process of variant integration, is typically performed using revision control systems and diff tools, meaning that practitioners undertake the task similarly to the task of software merging. The developer performing the merge must manually handle both the variability and potential conflicts on a source code level [3], [4], while features are in fact an architectural concern. This mismatch of abstraction levels means that feature integration is a complex task, and performing it manually is time-consuming and error-prone [5]. 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 integration tools.

This thesis evaluates a prototype variability-aware variant integration tool, called INCLINE, based on a novel domain specific language for specifying the integration goal in terms of so-called intentions. The domain specific language is validated based on data obtained from mining the open-source 3D-printer firmware Marlin. Based on this, the tool prototype is extended and refined. Using a controlled experiment, the prototype tool is evaluated comparing it to an unstructured two-way diff tool, using variant integration tasks sampled from the UNIX utilities distribution BusyBox and the text editor Vim.

The following research questions are investigated:

Q1 **Completeness:** *Does the set of intentions suffice for variant integration?* 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 of the language.

Q2 **Efficiency:** *Is there a benefit over manual integration with a diff tool?* Our goal is that time effort, required edit operations, and code quality can be improved by a workflow incorporating the intention-based integration tool, which we summarize as the overall beneficence of the tool.
We conjecture that a workflow with INCLINE is more efficient than a manual workflow.

Q3 **Qualitative differences:** *How is the integration process different using the intention-based integration tool?* This is an investigation into the perceived and evident qualitative differences of the two integration approaches.

To answer the research questions, we first mine highly configurable open source systems for variant integration merges, and analyze them to show that the intentions are capable expressing the witnessed merges, meaning that they are complete (Q1). We then perform a controlled experiment using students, measuring their editing efficiency and defects introduced when integrating variants in Eclipse CDT and INCLINE (Q2). The qualitative differences between unstructured integration and intention-based integration are elicited from participants in the controlled experiment using questionnaires and screen recordings (Q3).

This thesis contributes:

- a dataset of variability-related merges from Marlin,
- data from a preliminary internal tool evaluation with three participants,
- empirical data on the variant integration using our prototype intention-based variant integration tool with 16 student participants, and
- a qualitative investigation into the differences between intention-based and manual variant integration

The thesis is structured as follows: Chapter 2 outlines the background and rationale of variant integration. The methodology for data collection and experiments is presented in Chapter 3. Chapter 4 contains the results from the data collection of variant integration examples. The results of the empirical evaluation is reported in Chapter 5. Chapter 6 contains a discussion on inferences and validity threats to the study. Chapter 7 concludes with an outlook on future work.

2
Background

This chapter introduces revision control systems, followed by feature-oriented software development and software product line engineering concepts and challenges. After that, an introduction to projectional editing is given. The chapter concludes with a description of the intention language for variant integration, and the prototype variant integration tool using the language.

2.1 Revision Control Systems

A revision control system is used to manage the revisions and variants of a software system as it evolves over time. Revisions represent the state of the source code at a particular point in time, and are used as the basis for subsequent revisions. As such, revisions can evolve or develop in isolation or in interaction with other revisions, and any revision might have several different subrevisions. The operation of combining functionality from diverging revisions, by combining their changesets into a new revision, is known as *merging*. A recurring problem is that whenever incompatible concurrent divergent changes have occurred, a merge *conflict* arises, meaning that the system cannot infer which changes to apply because both are equally valid [3], [4], [6]. In order to resolve this conflict, the task is delegated to the user.
Revision control systems can be divided into two classes: *unstructured* revision control systems that operate on plain text; and *structured* revision control systems that rely on structure and semantics of the document being stored in order to leverage this knowledge for merge conflict resolution [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 mainly 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.

2.1.1 Merging

Since separate development tasks are carried out in parallel by different developers concurrently, contemporary revision control systems are *optimistic*, meaning that each developer can work on their own personal copy of a particular artifact [3]. In practice, this means that two different developers can change the same file at the same time. Merging is the process that occurs whenever parallel changes to the

same file are reconciled to a new revision, combining the changes. In cases where the merging algorithm cannot infer how to apply the parallel changes because they conflict, the resolution task is delegated to the user. Contemporary revision control systems use three-way merging, which minimizes the number of conflicts that must be delegated, because evolution can be inferred [3]. The name three-way merge comes from its input of three revisions: the two revisions of the software artifact to be merged; and their common ancestor revision [3]. A two-way merge takes only the two revisions to be merged as input, meaning that all differences between the two artifacts must be reconciled manually by the user – none can be inferred.

2.1.2 Forks

Historically, the term *forking* had a negative connotation, signifying a community schism or split causing subsequent independent divergent development efforts [2]. Today, forking (creating a *fork*, a clone or duplicate of an entire project) is an integral part of open-source software communities [7], [2]. Forking is an explicit part of modern social revision control platforms, such as Github or Bitbucket [8]. Making forks a first-class citizen enables traceability and facilitates forking as a part of a pull-based development model [2]. In this model, changes are propagated across the ecosystem dynamically, using *pull requests* [7], [8], [2]. In projects that employ a pull-based development model, any individual can create a fork and apply their desired changes, locally and decentralized. To spread their changes, possibly back into the mainline, they submit a pull request, indicating to the mainline developers that they should consider pulling and integrating the fork, should the quality be sufficient. During the lifetime of the pull request, it is possible that evolution occurs on the mainline – it is then up to the requester to maintain their changes compatible with the evolved mainline.

2.2 Features and Variability

This section explains the complicated relationship between variants, features, and variability.

2.2.1 FOSD and SPLE

Feature-Oriented Software Development (FOSD) is a paradigm for constructing large-scale software systems [9]. In FOSD, features are first-class citizens of the implementation of a software system [9]. As such, a highly configurable system is composed of a set of enabled features, which constitute the resulting system. A Software Product Line (SPL) is the set of software systems that can be derived from a set of features. Both FOSD and Software Product Line Engineering (SPLE) enable systematic reuse of components [9], [10], meaning that they alleviate the negative aspects of clone-and-own by consolidating features.

2.2.2 What is a Feature?

The definition of what a feature is can be viewed from the problem space or the solution space, as defined by Czarnecki and Eisenecker [11], where the problem space contains domain-specific abstractions that describe the requirements and intended behavior of a software system, and the solution space contains implementation-oriented abstractions defining how those requirements are met and the behavior is implemented [9]. There is a mapping from the problem space onto the solution space. From the problem space viewpoint, a feature is defined as "*a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems*" [12], whereas from the solution space, a feature is defined as "*a structure that extends and modifies the structure of a given program in order to satisfy a stakeholder's requirement, to implement and encapsulate a design decision, and to offer a configuration option*" [13]. The common denominator of both types is that a feature is always a logical unit of a system, delineated from other parts of it. Numerous additional definitions of *feature* in the literature on the spectrum from problem space to solution space are reported in [9] and [14].
These theoretical definitions are complemented by contemporary industrial notions on what constitutes a feature in a recent empirical study by Berger et al. [14]. This study shows that the meaning of a feature also varies among companies, but underlines that features should be distinct and well-delineated units of the system. Additionally, in practitioner settings, the origins of features are closely related to the business of the company; features can be customer-specific changes to a product, or arise from a particular market demand [14].

2.2.3 Software Variants and Variability

Variants of a software are created to fulfill different requirements in similar products [10], [2]. Conceptually, the source code is altered in some way, to achieve different behavior. From a feature-oriented perspective, one or more features can be grouped inside a variant, and in FOSD and SPLE, any product is defined in terms of the features it is composed of [9]. The fastest way to create a variant of a software system is to copy the entire source code and make the required changes. 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 [2].
Orthogonal to variability through clones stands integrated variability. By centralizing all variant code in one common repository, distinct variants are instead derived by composing features. In the source code, this is realized by language-level constructs guarding the feature code, meaning that behavior can be enabled or disabled at compile-time (C preprocessor `#ifdef` statement) or runtime (regular if-statements, design patterns). The inclusion of a particular feature into a variant is dictated by an associated boolean *presence condition*, enabling or disabling code based on the selected feature configuration. This also enables the possibility to compose variants with arbitrary features, as opposed to clone-and-own, which requires one cloned project for each variant.
There is a tradeoff between time effort and maintainability with respect to the two variability strategies. Clone-and-own has low initial costs, but does not scale due

to the inherent impact on the maintainability of a large number of clones, since each new variant requires a new clone [10]. An integrated platform, on the other hand, requires significant up-front commitment and investment into architecture and infrastructure enabling systematic reuse [10].
In the context of software ecosystems, a common manifestation of variants is forks, which are repository-scale clones of an original codebase, called the *mainline*. For forked variants, the variability lies in an array of repository clones with minor changes in them, altering the desired behavior. Using pull requests, the features within forks can be propagated across the ecosystem, integrated or adapted by others or become part of the mainline [2], using a pull-based development model [7]. The open-source 3D-printer firmware project Marlin uses both forks (clone-and-own) and integrated variability (features using `#ifdefs`) in parallel [2], meaning that it can be used to study features originating in forks (cloned variants), being integrated into the mainline with variability.

2.2.4 Platform Re-engineering

A recent mapping study on the topic of re-engineering legacy applications into SPLs by Assunção et al. identifies 119 publications in the field [15]. Most of these focus on variant detection and analysis to facilitate feature identification and feature location within systems, which is a prerequisite for moving from clone-and-own to an integrated platform. Antkiewicz et al. [10] propose a strategy for migration from clone-and-own with low-risk, step-by-step adaption of an integrated platform through what they call *virtual platform*.
Using the terminology of Buckley et al. [6], cloned variants represent divergent changes being developed asynchronously in parallel [3], [2]. Integrating forked variants back into an integrated mainline platform (analogue to migrating from clone-and-own to integrated platform) has the advantages of increased maintainability; features and bug-fixes being consolidated; and reducing unintentional code duplications [1], [2]. 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 [2], [8]. Currently, the process of integrating variants is based on manual unstructured merging, relying fully on the developer to create a semantically correct merge [3], [4]. An open-source proxy for this is merging forks back into the mainline in a pull-based development model [7], since the pull requests come from large scale-clones with small-scale changes.

2.3 Projectional Editing

Projectional editors differ from traditional parser-based editors in what is edited, and how. A traditional editor operates on a text buffer, that is eventually parsed by a compiler, given that it contains well-formed syntax. A projectional editor, on the other hand, operates directly on an abstract syntax tree (AST), meaning that it does not have to be parsed [16]. The benefit is that only legal tokens can be entered into a projectional editor – it is impossible to achieve syntax errors, but

the disadvantage is that navigation and manipulation of the AST structure can be unintuitive.
When editing variational software, comprehension can be increased by viewing the source code of a subset of features, and filter out the source code of other features [17]. In source code with annotated features (e.g., C preprocessor), issues with alignment arise because the hidden code still impacts the indentation hierarchy [18]. In a projectional editor, the rows can be placed with correct alignment, because there, the indentation is semantic rather than syntactic.
However, the usability of projectional editors has long been questioned. Berger et al. [19] have previously conducted a controlled experiment using students and industrial developers, to determine their editing efficiency in the projectional editor Jetbrains MPS, compared to a traditional parser-based editor. The participants are given four simple programming tasks to complete by editing a provided program, transforming it into a solution. They find that with training, projectional text editing is in fact more efficient than parser-based text editing.

2.4 Intention-based Variant Integration

Today, a developer performing a feature-related merge in a revision control system must make ad hoc decisions on the final result, with respect to presence conditions over source code [17]. This means that the concern of the developer is at the source-code level, dealing with fundamental language constructs defining the presence conditions of particular code blocks. We note that feature integration is a concern on the architectural level, while the merge is performed line-by-line on the source-code level. The mismatch between the architectural concern and the abstraction level on which it is carried out ensures that the integration (or merge) process is time consuming and has a high risk of introducing defects. Because of the mental overhead involved in considering multiple variants and their control flow, variability-related code is more defect-prone. Previous studies show that variability is detrimental for program comprehension [5], [20], [21], [22], [23], [24].
We propose to address the mismatch of abstraction levels by introducing a domain specific language (DSL) capturing the semantics of the feature integration. In particular, the language describes the outcome that the developer wants to achieve. Examples include “keep this functionality” or “remove this functionality”. We subsequently refer to these concepts simply as the *intentions* of the developer. Each intention carries a mapping to actual operations that are performed on the source code. The semantics of the intentions language ensures the integrity of the underlying code upon which operations are applied to. This intentions language is the basis for creating a tool to aid the integration process structurally [3], [4]. With proper tool support, the integration process can become more efficient from time consumption, risk, and quality perspectives. Indeed, tool support has shown to reduce variability errors in previous studies [25].
The intentions and their underlying transformations can be described using choice calculus, a formal notation for variational software, independent of programming language [26], [27]. In the implementation, this can be realized as conditional compilation (C preprocessor) or design patterns that support variability (object-oriented

```
Variant: Mainline
#ifdef ULTIPANEL
  uint8_t lastEncoderBits;
  uint32_t encoderPosition;
  #if PIN_EXISTS(SD_DETECT)
    uint8_t lcd_sd_status;
  #endif
#endif

menu_t cM = lcd_status_scrn;
bool ignore_click = false;
```

```
Variant: Fork
#ifdef ULTIPANEL
 uint8_t lastEncoderBits;
 int8_t encoderDiff;
 uint32_t encoderPosition;
 #if (SDCARDDETECT > 0)
  bool lcd_oldcardstatus;
 #endif
#endif

menu_t cM = lcd_status_scrn;
```

```
Variations inlined:
#ifdef ULTIPANEL
  uint8_t lastEncoderBits;
  #ifdef FORK
    int8_t encoderDiff;
  #endif
  uint32_t encoderPosition;
  #ifdef FORK
    #if SDCARDDETECT > 0
      bool lcd_oldcardstatus;
    #endif
  #else
    #if PIN_EXISTS(SD_DETECT)
      uint8_t lcd_sd_status;
    #endif
  #endif
#endif

menu_t cM = lcd_status_scrn;
#ifndef FORK
 bool ignore_click = false;
#endif
```

Figure 2.1: Upper row: variant sources with differences highlighted. Lower row: inlined variational AST of the same variants in INCLINE, with presence condition `FORK`. Example adapted from [28].

languages). Intentions could also be mapped to the source-code operations identified by Stanciulescu et al. in [17].

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 [8]. Tool support for proper variant-aware integration is also a prerequisite for the capability of migrating cloned product lines to an integrated platform through the virtual platform approach [10].

The work on the intentions language and the prototype tool INCLINE, built on top of Jetbrains MPS, has been carried out jointly between Max Lillack, Ștefan Stănciulescu, Wilhelm Hedman, Thorsten Berger, and Andrzej Wąsowski, and is described in [28]. The intentions language is designed as a DSL for variant integration, the key concept of which is intentions. Intentions are inplace transformations on a variational AST. The complete formal definition of the intentions language is given in [28]. A recapitulation with visual examples of the intentions and the effect of their associated AST transformation is given in Appendix A. In INCLINE, the two variants being integrated are inlined into a single variational AST, with the differences between them wrapped in the presence condition `FORK`. An example is shown in Figure 2.1. Subsequently, this inlined notation is used.

3 Methodology

This chapter describes the data collection of merge samples and the empirical evaluation of the intentions language and intention-based variant integration tool INCLINE.

3.1 Data Collection

We choose Marlin as our initial source of merge examples, because previous work has identified important concepts in the Marlin ecosystem that we build upon, and because of access to the authors of [2], [17]. Two methods for collecting data from open-source ecosystems have been used: one for gathering a ground truth for how variant integration is achieved through merging, and one for procuring smaller scale authentic integration candidate scenarios. We denote the output of the former method *merge examples* and the output of the latter method *integration scenarios*. The merge examples are used for understanding the context and execution of the variant-integrating merge. They can also be replayed for validation, since they represent the ground truth outcome of the two merge parents. As such, the merge examples serve as an important validation tool. However, since the merge examples are extracted from a three-way merging tool, they are disadvantageous for replaying in INCLINE, since three-way information is not available in that environment. To this end, we use the integration scenarios, which are free from three-way contamination, but are instead taken out of their context and are scaled down. They are now described in detail.

3.1.1 Collecting Ground Truth Merge Examples

To identify and validate the integration intentions we analyze variant-related merges and extract common patterns. We begin by retrieving all merges and discard those that merged without conflicts. Merge instances with conflicts signify a syntactic conflict which is an indicator of potential feature integration. In order to investigate further, for each of these merges, we save the state of the entire source tree at the following revisions: a) after the merge, i.e. the *result*, b) in the first parent, i.e. the *head*, and c) in the second parent, the *merge head*. We call this triple of source trees the *related artifacts* of the merge, the relationship between its parts is shown in Figure 3.1. Retaining and inspecting the source trees of the parents allows for insights to be gathered, and in particular the ability to replay the merge in order to reach the known result. Note that the search space, the entire set of merges encompasses *all*

merges, that is, merges that occurred in a fork but were subsequently merged into the mainline are also included. We do *not* use the `--first-parent` option to Git, which would discard such further merges residing in recursive parents, and instead use only the first parent when finding merges. The merges that are included by this broader search scope originate either from ordinary feature merging or from pull requests where the branch of the requester is out of date.

When the related artifacts from each merge have been collected, we inspect them and discard merges with any of the following properties: a) all conflicts occurred in non-source-code artifacts, b) at least one artifact cannot be compiled, c) there are only whitespace changes between the merge parents, or d) contain some project-specific uninteresting changes (see Section 4.1 for examples of what *uninteresting* entails in practice).

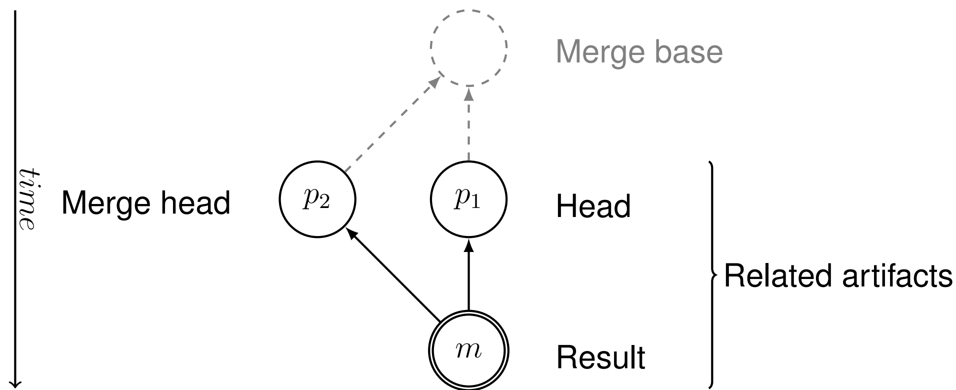


Figure 3.1: Merge commit context. The merge commit m has two parents, p_1 and p_2. The entire state of the source tree is gathered, in these three commits. These source sets are denoted *result*, *head*, and *merge head*. This triple of source tree sets constitute the *related artifacts* of the commit m.

3.1.2 Creating Authentic Integration Scenarios

Since the merge examples discussed in the previous section would be replayed without three-way information, and are potentially very large, it is desirable to have a neutral and concentrated basis for task creation, referred to as *integration scenarios*. The contents of integration scenarios are based on variant integration examples from the ecosystem (mainline and forks), with the diff chunks consolidated so that there is not an abundance of code with no changes. Additionally, changes may originate from several different source files, and can have syntax and semantics changed. The structure of the source chunks, and the conflict resolution ground truth is left intact. Since these scenarios are to be used in experiments, the goal is that the result should encompass ca. 50 LOC, as to minimize navigation overhead in the file for the experiment participants.

3.2 Completeness Evaluation

To verify the completeness of the intentions language, we replay all the identified integration commits, and apply intentions to resolve them. The criterion for a successful replay is that the result of the created intention-based resolution is semantically identical (with respect to the C preprocessor) to the ground truth merge resolution. As input to the completeness evaluation, we use the variability-related merge examples from Marlin, which are guaranteed to have well-formed syntax. The completeness evaluation is limited to examples from Marlin, but achieves generality because of the sampling across source-files and forks. This theoretical sampling of merges is employed, as it is impossible to ensure that the evaluated set is representative of the set of all variant integrating merges. The objective is to answer the first research question:

Q1 Does the set of intentions suffice for variant integration?

3.3 Internal Evaluation

As a prestudy for the controlled experiment, we perform an internal evaluation with the three tool developers, exploring how tooling and tasks should be set up for the experiment. There are two types of tasks: to replay *merge examples*, and to integrate actual forks with the mainline. The characteristics of these tasks are shown in Table 3.1. All metrics relate to the inlined model of the programs [28] (cf. Figure 2.1). *Integrated LOC* signifies the LOC of the source code with inlined variants. *Chunks* is the number of variation blocks from the variants. *Variable LOC* (VLOC) is LOC count of lines under a presence condition containing the literal `FORK` – i.e. the LOC count inside the chunks in the inlined model – in particular, these are the lines that must be manipulated by the participants. The merge tasks are chosen from the set of merge examples in Marlin, and the forks are chosen based on the findings of Stănciulescu et al. [2]. It does not matter whether they are still actively maintained or not, because we set the task at the `HEAD` of the fork, and then compare with the latest common ancestor commit `#ancestor` between fork and mainline, and devise a resolution strategy for the task.

Each developer performs 3-4 tasks, at least one of each kind. Each task consists of a number of input files from the two variants being integrated, together with a textual description of what the integration goal is. The integration task is performed once in the unstructured two-way merge tool of Eclipse CDT and once in INCLINE. The number of edit operations required per task in each editor is saved for analysis, as is the resulting output file. In INCLINE, the edit operations are counted once for applying an intention, and for undoing an intention, regardless of the number of nodes selected. In Eclipse, the edit operations are counted once per completed action of insertion, cloning, deletion, and undoing (cf. Berger et al. [19]). The output files are compared to the ground truth, and checked for defects introduced. Syntactic differences are allowed, so long as the content is similar semantically with respect to the C preprocessor.

Table 3.1: Internal evaluation task characteristics.

Name	Type	Files	Integ. LOC	#chunks	VLOC
08856d9	Merge	1	1029	16	312
17de96a	Merge	1	929	33	392
2daa859	Merge	2	4019	152	1107
3116271	Merge	1	146	6	51
373f3ec	Merge	1	2663	106	749
46f80e8	Merge	1	130	2	2
47c1ea7	Merge	1	3400	242	2391
esenapaj	Fork	3	14909	196	754
jcrocholl	Fork	1	4803	40	364
STM32	Fork	3	7670	159	966

3.4 Controlled Experiment

This sections describes the experimental design and setup, together with the *integration scenarios* from BusyBox and Vim, that are the subject programs in the experiment. Before the experiment, we elicit the background of the participants through a questionnaire containing Likert-scale questions about how familiar they are with certain programming languages and merging techniques and tools. The questions are listed in Appendix C.

3.4.1 Objective

To establish the beneficence of an intention-based integration strategy in general, and INCLINE in particular over two-way merging in Eclipse CDT, we evaluate the tool using subject programs from other sources than Marlin, since Marlin was used to elicit the set of intentions. The purpose of the controlled experiment is to answer the following two research questions. Additionally, we formulate hypotheses related to Q2:

- Q2 Is there a benefit over manual integration with a diff tool?
  - H1 Integration in INCLINE is faster than in Eclipse CDT.
  - H2 Integration in INCLINE requires fewer edit operations than in Eclipse CDT.
  - H3 Integration in INCLINE does not lead to more defects than in Eclipse CDT.
- Q3 How is the integration process different using the intention-based integration tool?

To answer these questions, we let participants integrate variants using two different tools, while controlling the confounding factors of developer competence and learning. We measure completion time, required edit operations, and defects inserted, and aggregate this as the benefit of a tool. The exploratory question of the differences between processes of intention-based integration over two-way merging has no hypotheses.

3.4.2 Treatments

In order to study the beneficence of intention-based integration, we let each participant solve one task using the ordinary unstructured two-way merge tool inside Eclipse CDT, and one task using INCLINE. The two treatment levels are thus *Eclipse CDT*, denoted `Ecl` and INCLINE, denoted `INC`.

3.4.3 Subjects

Participants. The experiment is performed with $N = 16$ participants: 15 graduate students, and one Ph.D. student. All participants had programming experience, and more than half had more than one year of industrial experience. Regarding integration, all participants were familiar with the Git version control system, and were familiar with ordinary software merging.
Programs. Using the process for collecting authentic variant *integration scenarios*, we prepare two sample programs to be used in the experiment: `P1`, based on BusyBox, and `P2`, based on Vim. These two programs are created as condensed versions of the changeset of a particular fork variant, for brevity and comprehension during the experiment. This should be understood as selecting the most suited changes across the files, and placing them in a single file, in order to not waste valuable time for the participants on long blocks without any changes, as to keep the programs brief. Table 3.2 lists the characteristics of the subject programs, using the same metrics as in the prestudy described in Section 3.3. *Integrated LOC* is the line count after the variants have been inlined, *Chunks* is the number of variability blocks, and *Variable LOC (VLOC)* is the LOC count within the chunks.
We choose BusyBox and Vim as representative subjects because they are highly-configurable open source systems, and have been subjects in previous studies [29], [30], [31], [32].

Table 3.2: Characteristics of the programs `P1` and `P2`.

Prg	Origin	Integ. LOC	#chunks	VLOC
`P1`	BusyBox	74	8	37
`P2`	Vim	50	5	32

3.4.4 Design

The experiment is a 2×2 within-subjects counterbalanced Latin square design [33]. Each developer performs two tasks with the treatment order and program order randomized in order to reduce learning effects. Table 3.3 shows the base Latin square. Since developers are assigned at random, no further permutation of the treatments in the 2×2 Latin square is required, since the two other possible combinations would be redundant with respect to program order and treatment. The benefit of a within-subject design is the reduced number of required subjects participating in the experiment and the fact that every subject is exposed to every treatment and program. This however comes at the cost of carryover effects, in this

case particularly learning effects. We therefore employ a counterbalanced design, where the order of the tasks is randomized.

Table 3.3: Latin square instance (2×2).

developers	*programs* P1	P2
D1	Ecl	INC
D2	INC	Ecl

3.4.5 Execution

Before the participants are given their tasks, we present a simple tutorial on integration and variability, and demonstrate how to solve a sample task in each editor. All of tutorials were prerecorded to ensure that all participants receive the same introduction, regardless of which session they attend. Before solving the INCLINE task, the participants are given a warmup task to understand navigation in a projectional editor, and the application of intentions inside the editor. A cheat sheet with examples of intentions is also provided. An instance of a task sheet is shown in Appendix B.

During the tasks, the screens of the participants are recorded, which allows measurement of the task completion time. Using the same measurements as in the prestudy, described in Section 3.3, both treatment tools are instrumented to output logs with information about keystrokes and menu usage, thereby measuring the required edit operations. Last, the resulting integrated source files created by the participants are collected, to measure the number of defects.

After the experiment, participants fill in a questionnaire with closed and open questions about their perceptions of INCLINE and intention-based integration compared to two-way unstructured integration in Eclipse CDT. The qualitative answers are categorized using open coding [34]. The questions are listed in Appendix C.

3.4.6 Analysis

The Mann-Whitney U-test is used to test the significance of differences between the treatment groups. Cliff's delta is used to calculate the effect size, particularly because it does not require normality.

4

A Dataset of Integration Examples

This chapter reports on the data collection from Marlin, demonstrates replayed integrations using intentions, and reports the completeness of the intentions language.

4.1 Variability-related Merges from Marlin

In Marlin, we retrieve all 2065 merges, and extract those that were merged with conflicts, yielding 49 merges. We discard 2 merges that had conflicts only in non-source-code files (documentation), 2 that conflicted due to whitespace changes, 3 that conflicted due to configuration changes[1]. Another 3 merges are discarded because some related artifact had syntax errors and could not be compiled. Additionally, 4 merges are discarded because they simply accept the mainline changes as evolution, i.e. empty changeset, and are therefore uninteresting. This is summarized in Table 4.1. The remaining 35 merge commits are used for subsequent analysis of integration scenarios and a subset are used for replaying in our internal tool evaluation.

Table 4.1: Marlin merge commit statistics

Commit range: `99653ff..2ed1331`	
Nbr. of commits	7,254
↪ Nbr. of merge commits	2,065
↪ Nbr. of conflict merge commits	49
Documentation conflicts	–2
Whitespace changes only	–2
Configuration changes	–3
Syntax errors	–3
Evolution – no merge	–4
↪ Nbr. of useful/relevant merge commits	35

[1]While configuration management is indeed an important part of FOSD, in these cases, the developers have mistakenly committed their personal 3D-printer configurations into a configuration template file while committing other relevant changes, cf. Stanciulescu et al. [2].

4.2 Replaying Merges with Intentions

This section demonstrates the replaying of selected chunks from Marlin. We present examples of differences from both evolution and features being reconciled. Refer to Appendix A for the recapitulation on the intentions DSL.
Figure 4.1 shows how a change from the mainline is accepted as evolution. Figure 4.2 shows how a change from the mainline is discarded as evolution. The snippet in Figure 4.3 accepts a change from the fork as a feature. In Figure 4.4, two mutually exclusive changes from the mainline and fork are integrated as a feature. Last, we show the composition of intentions in Figure 4.5.

```
#include "watchdog.h"
#ifndef FORK
   #include "language.h"
#endif
#include "Sd2PinMap.h"
```

```
#include "watchdog.h"
#include "language.h"
#include "Sd2PinMap.h"
```

Figure 4.1: Keep intention applied (left) to replay a change from `47c1ea7::temperature.cpp`, ground truth and outcome (right).

```
#ifndef FORK
   unsigned long ms = millis();
#endif
if (temp_meas_ready == true) {
  [...]
}
```

```
if (temp_meas_ready == true) {
  [...]
}
```

Figure 4.2: Remove intention applied (left) to replay a change from `47c1ea7::temperature.cpp`, ground truth and outcome (right).

```
#ifdef FORK
   static float delta[3] = 0.0, 0.0, 0.0;
#endif
static float offset[3] = {0.0, 0.0, 0.0};
```

```
#ifdef DELTA
  static float delta[3] = {0.0, 0.0, 0.0};
#endif
static float offset[3] = {0.0, 0.0, 0.0};
```

Figure 4.3: KeepAsFeature(`DELTA`) intention applied (left) to replay a change from `373f3ec::Marlin_main.cpp`, ground truth and outcome (right).

4.3 RQ1: Does the set of intentions suffice for variant integration?

The intentions language can be used to replay all 35 variability-related merges in the Marlin mainline. From these, certain chunks were listed above to illustrate the use of intentions in real scenarios.

RQ1. The proposed set of intentions indeed suffice for variant integration.

```
#ifndef FORK
  if([...]) {
    plan_buffer_line([...]);
  }
#else
  float difference[NUM_AXIS];
  for (int8_t i=0; i < NUM_AXIS; i++) {
    difference[i] = destination[i] -
      current_position[i];
  }
#endif
```

```
#ifndef DELTA
  if([...]) {
    plan_buffer_line([...]);
  }
#else
  float difference[NUM_AXIS];
  for (int8_t i=0; i < NUM_AXIS; i++) {
    difference[i] = destination[i] -
      current_position[i];
  }
#endif
```

Figure 4.4: AssignFeature(`DELTA`) intention applied (left) to replay a change from `373f3ec::Marlin_main.cpp`, ground truth and outcome (right).

```
#ifndef FORK
  SERIAL_ECHOLN(MSG_PID_AUTOTUNE_START);
#else
  SERIAL_ECHOLN("PID Autotune start");
#endif
```

```
SERIAL_ECHOLN(MSG_PID_AUTOTUNE_START);
```

Figure 4.5: Keep and Remove intentions applied (left) to replay a change from `47c1ea7::temperature.cpp`, ground truth and outcome (right).

5

Empirical Evaluation

This chapter contains the results from the internal evaluation prestudy of INCLINE, and the results of the controlled experiment, along with the quantitative data from the experiment and post-experiment questionnaire. The results are tied to the the research questions about beneficience of INCLINE and differences between the intention-based and manual integration processes.

5.1 Internal Evaluation

The results from the internal evaluation are shown in Table 5.1. We report the number of defects introduced by the developer in the integration in both tools, and the number of required editing operations for both tools.

Table 5.1: Internal evaluation results.

Name	Defects `Ecl`	Defects `INC`	Operations `Ecl`	Operations `INC`
08856d9	0	0	4	1
17de96a	1	2	13	5
2daa859	2	5	14	5
3116271	1	1	15	5
373f3ec	4	3	35	7
46f80e8	1	0	0	1
47c1ea7	1	1	5	4
esenapaj	5	2	271	185
jcrocholl	0	2	116	44
STM32	1	5	93	122

Note that substantially fewer operations are required when using intentions, compared to the operations required when using an ordinary merge tool, and that there is no particular difference in number of defects between editors. However, the larger tasks consume up to an hour of time to finish, because of the large number of changes, and because the integration goal has to be interpreted. We also note that without being onboarded in projectional editing and INCLINE, the participants will not be able to perform even the most rudimentary operations.
Based on these experiences, we draw three conclusions that are incorporated into the experiment design for the controlled experiment: the integration goal should be provided verbatim, rather than being described in abstract terms, cf. [19]; the tasks

must be smaller so that they can be performed in a reasonable time for participants; and, subjects will require training in INCLINE.

5.2 Controlled Experiment

This section first reports the quantitative data from the metrics and statistical tests from the controlled experiment, followed by quantitative data from the post-experiment questionnaire. We recorded more than 8 hours of screen recordings, and collected questionnaire responses from all 16 participants.

5.2.1 Editing Efficiency

Figure 5.1 shows the distributions of the completion times, with Table 5.2 showing the average completion times, the significance tests, and effect sizes. In analogue to this, the distributions of the edit operations is shown in Figure 5.2, and the average edit operations, significance tests, and effect sizes shown in Table 5.3.
For completion times, the Mann-Whitney U-test shows a significant difference across both subject programs, with large effect sizes. We thus find no support for H1: *Integration in* INCLINE *is faster than in Eclipse CDT*, in fact observing the opposite. For edit operations, INCLINE requries fewer median operations, but only the BusyBox task shows a significant difference, while no significant difference can be observed for Vim. There is thus support for H2: *Integration in* INCLINE *requires fewer edit operations than in Eclipse CDT*.

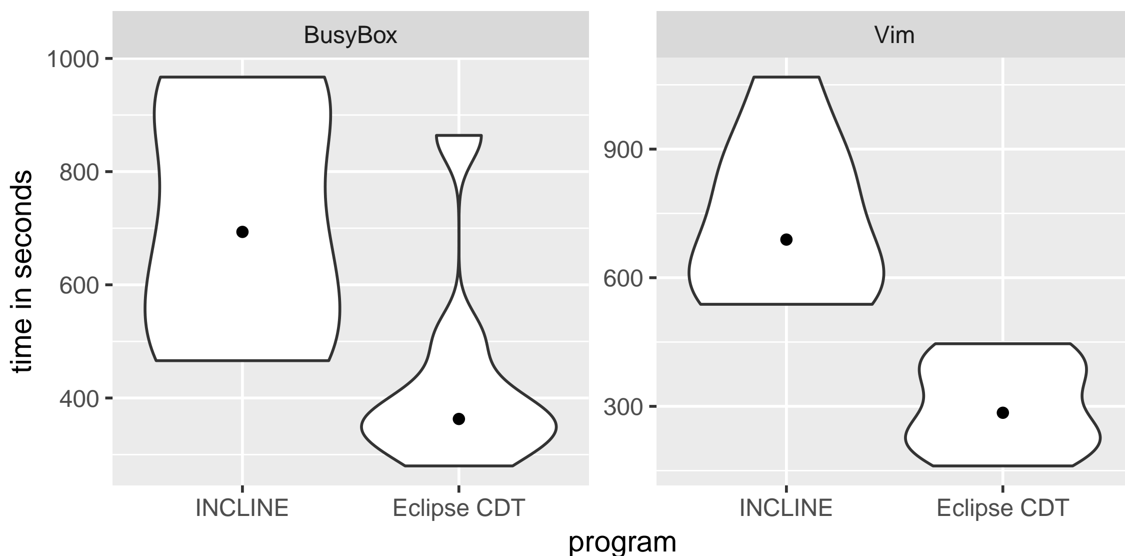


Figure 5.1: Task completion times in seconds. (Violin plot, dot denoting median.)

5.2.2 Defects

The number of completetely defect-free integrations is reported per program and editor in Figure 5.3, corresponding to the zero bin in the histogram of number

Table 5.2: Average completion times, significance tests, and effect sizes.

Prg	`INC`	`Ecl`	Significance	Effect size
BusyBox	711	433	Mann-Whitney: $W = 7, p = .007$	Cliff's delta: $d = 0.78$
Vim	733	302	Mann-Whitney: $W = 7, p = .001$	Cliff's delta: $d = 1$

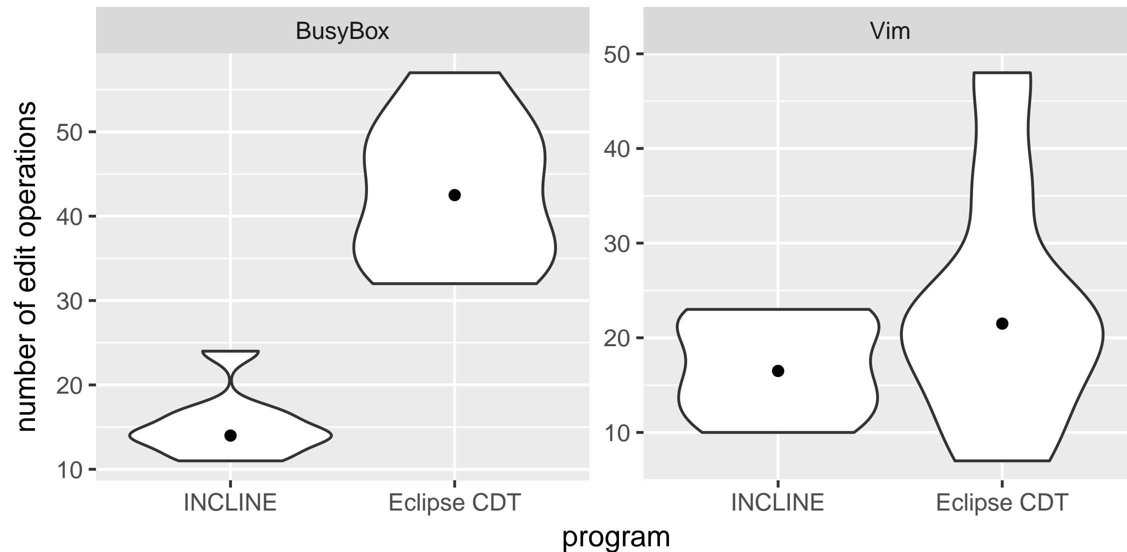


Figure 5.2: Required edit operations.

of defects of Figure 5.4. For defects, we do not provide any statistical tests for significant difference between the two treatments, because of the many zero samples. Instead, we provide a graph of errors per participant in Figure 5.5. In it, we note that participants E, F, and L have committed multiple mistakes, with both editors. The defects are categorized in Table 5.4. In Eclipse, most defects are related to presence conditions, while in INCLINE, overdeletion and overlooking variability are the most common sources of defects. From the Figures 5.4 and 5.5, we infer that INCLINE does not perform worse than Eclipse CDT with respect to inserted defects, supporting H3: *Integration in* INCLINE *does not lead to more defects than in Eclipse CDT*.

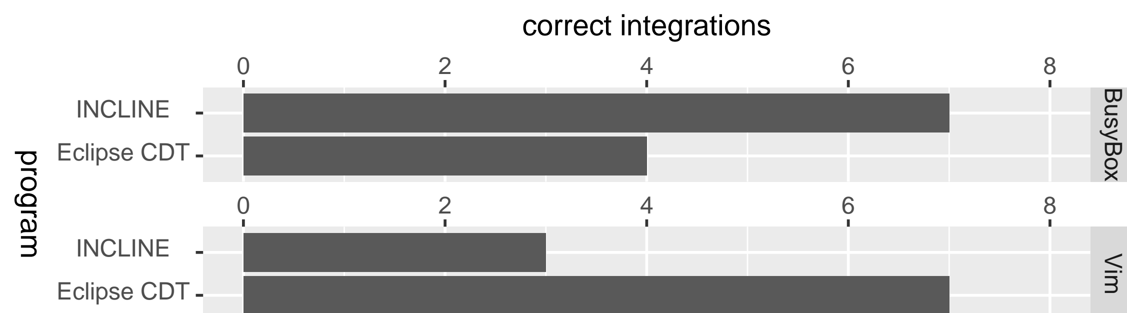


Figure 5.3: Number of completely correct integrations.

Table 5.3: Average edit operations, significance tests, and effect sizes.

Prg	INC	Ecl	Significance	Effect size
BusyBox	15	43	Mann-Whitney: $W = 0, p = .001$	Cliff's delta: $d = -1$
Vim	17	23	Mann-Whitney: $W = 23, p = .37$	Cliff's delta: $d = -0.28$

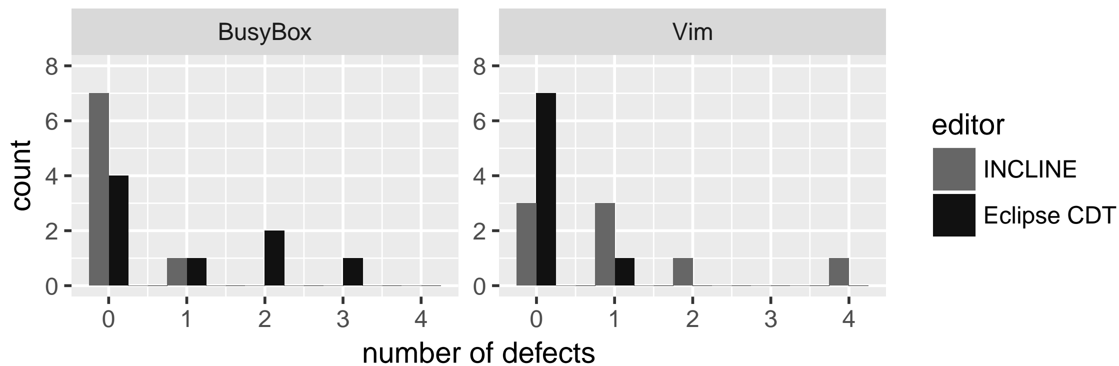


Figure 5.4: Histogram of count of defects introduced.

5.2.3 Post-experiment Opinions

The quantitative results from post-experiment questionnaire with opinions of the benefit of INCLINE is shown in Figure 5.6. INCLINE is perceived as faster and as facilitating the detection and correction of defects introduced in the integration. Intention-based integration is not viewed as complex, and *all* intentions are perceived as intuitive.

5.3 RQ2: Is there a benefit over manual integration with a diff tool?

Recall that the overall benefit is the aggregate of completion time, the edit operations required, and the defects inserted. For both tasks, there is a significant difference with respect to actual completion time, with high effect sizes. There is however a significant difference in the required number of edit operations for one task, with a large effect size, while for the other task it is lower, but not significantly so. There is no comparable difference in the number of defects inserted between the two editors, and INCLINE does not perform worse in that respect.

RQ2. Intention-based variant integration is beneficial with respect to edit operations, comparable in number of defects, but worse in total time, compared to manual integration with a two-way diff tool.

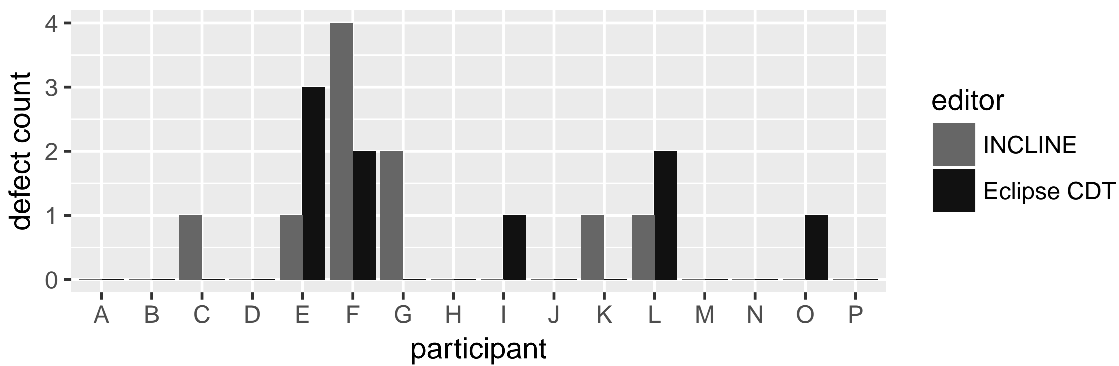


Figure 5.5: Defects per participant.

Table 5.4: Defects by category.

INCLINE		**Eclipse CDT**	
Overdeleted chunk	5	Incorrect presence condition	4
Variability not handled	3	Inverted presence condition	2
Incorrect variant accepted	1	Overdeleted chunk	1
Non-removed chunk	1	Non-removed chunk	1
		Illegal syntax	1

5.4 RQ3: How is the integration process different using the intention-based integration tool?

This section identifies differences among INCLINE and Eclipse CDT, by analyzing the open questions of the post-experiment questionnaire, screen recordings, and quantitative data from the experiment. From these, three major themes emerged: editing, intentions semantics, and integration support.

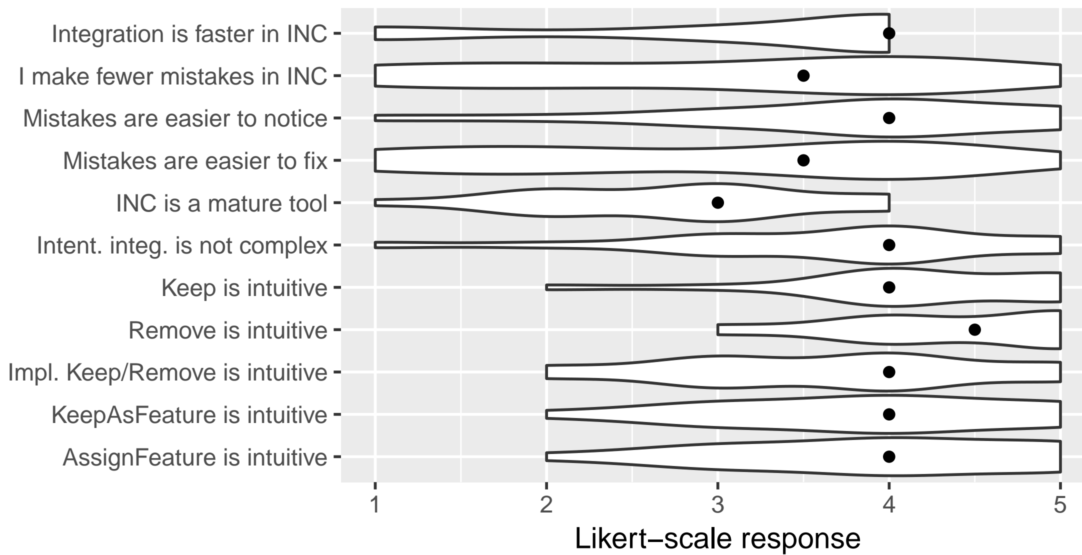
5.4.1 Editing

INCLINE does not allow manual text insertion, instead relying fully on the intentions to transform the AST. Participants do not see this as a limitation, instead commending it as making the integration less-error prone (eg., *"easier [to] avoid [...] bugs [...] and subtle differences"* [r3], and *"harder to make syntatic mistakes"* [r11]).
A repeated criticism (or encouragement) is to create keyboard shortcuts for intentions in INCLINE, to increase the editing speed.

RQ3. The lack of free-text editing in INCLINE is not seen as a drawback.

5.4.2 Intentions Semantics

There is a noticeable tendency among participants to overselect, selecting an entire `#ifdef`-structure, rather than the nodes inside them when applying intentions. Note that this is related to the semantics of the intentions, as opposed to editing.



1: strongly disagree, 2: disagree, 3: neutral, 4: agree, 5: strongly agree

Figure 5.6: Post-experiment questionnaire opinions. Refer to Appendix C for the full questions.

Since the intentions are applied in a particular order, certain combinations of intentions will cancel out the effect completely, which leads to confusion: (eg., *"hard to grasp how multiple conflicting intentions are prioritized"* [r3]). A recurring pattern is that developers apply both the Keep and KeepAsFeature intentions on nodes, without any result, as the Keep intention is an identity of the KeepAsFeature intention.

Overall, the perceived drawback of using INCLINE is the learning curve of the intentions semantics over the well-known paradigm of free-text editing (eg., *"Involves a learning curve that copy and paste does not."* [r14]). Compare also proficiency in free-text editing contra structured projectional editing to Berger et al. [19].

RQ3. Integration with intentions requires knowledge of the semantics. No similar knowledge is required in an unstructured editor.

5.4.3 Integration Support

Variant integration is given explicit concern in INCLINE. The perceived benefits noted are: a) that the developer has a concrete list of all variation points that need to be integrated, and can be sure that all have been handled (eg., *"you wont forget part of the integration"* [r16], and *"It offers a nice way to work through the variabilities while making it hard to make any stupid mistakes."* [r8]), b) that the views and preview help to understand the integration (eg., *"It gives a better overview and it is easier to know where the differences are."* [r7], and *"the preview [...] and the projections [are] hugely helpful"* [r11]).

Participants conjecture that INCLINE is more beneficial than two-way merging in files larger than those in the tasks (eg., *"Incline for longer periods of time and on larger projects. Manual for shorter statements."* [r9], and *"If it was a large*

scale project I would feel more comfortable with INCLINE." [r6]). Indeed, for future variant integration tasks, 12 respondents would prefer to use INCLINE.

RQ3. A systematic integration approach is enabled by the fact that all variability is explicit in INCLINE, aided by the preview and projection views. None of this is available in an unstructured editor.

6

Discussion

This chapter provides selected inferences from the results, and reports on the validity threats to the completness evaluation and controlled experiment.

6.1 Inferences

Editing efficiency. Interestingly, no participant identifies the projectional editing node selection as a disadvantage of INCLINE, and no participant struggles with it. This is in line with the findings of Berger et al. [19], showing that users are proficient inside a projectional editor after a short training session.
Completion times for tasks are categorically in favor of Eclipse CDT. Since a research prototype cannot compete with the user interface design capacities of a large project such as Eclipse, we argue that the required edit operations is a more interesting metric for comparison. However, all variability is made explicit in INCLINE, meaning that actions such removing a chunk requires an explicit editing operation, whereas in an unstructured editor, it incurs no additional edit operation – since no action must be undertaken on the particular chunk at all. The two approaches are therefore not analogous in causes of editing operations. However, with larger samples, these differences should even out. To be sure, with better interface design, the completion times in INCLINE should decrease compared to Eclipse CDT.
Defects. Even though the experiment is inconclusive with respect to statistics regarding defects, integration in INCLINE is viewed as less error-prone among the participants (cf. Figure 5.6). It is interesting that the participants are prefer support and find the structured approach favorable, rather than going in blind into an unstructured tool. Tools aside, there is also the aspect of the subject, and as seen in Figure 5.5, three subjects introduce defects in both tools. For INCLINE, the defects related to overlooked variability (cf. Table 5.4) can easily be reduced to zero by forcing the developer to handle each variation point. Overall, the defects produced in the controlled experiment might not be representative of real integration defects, since the participants have a clear integration goal.
Benefit of intention-based approach. Recall the internal evaluation performed by the three developers of INCLINE, who must be said to be experts on the tool and its usage. For those, substantially larger tasks, fewer edit operations were required in INCLINE than in Eclipse CDT. Additionally, the opinions of the experiment participants, who were not experts, are pointing towards a perceived benefit in INCLINE, with respondents conjecturing that the intention-based approach is more effective than the unstructured approach on larger files in realistic settings. To move

away from conjectures, and further investigate the benefit of intention-based variant integration, the next step should be a user study replicating the controlled experiment, using subject developers that are familiar with the tool and SPL engineering. In the post-experiment questionnaire, the learning curve of INCLINE is a recurrently listed disadvantage, but in the same questionnaire the intentions are being judged as intuitive (Figure 5.6). It therefore seems that the learning curve is quite short. None the less, participants in subsequent studies should be properly trained on the intentions formalization and the usage of INCLINE.

6.2 Threats to Validity

This section reports on the threats to validity in the study. No validity threats are reported for the internal evaluation, as it has only been used to guide the experiment design, and its results have not been used to answer the research questions. In general, we recommend that replication studies should be carried out at a larger scale, with practitioners.

6.2.1 Internal Validity

Required knowledge. Since no domain knowledge was required for conducting the integration tasks, having no previous knowledge of the projects or domains is not a disadvantage. No previous knowledge of the editors was required, as all participants were shown the same introductory videos, explaining how to complete an example task in the editor they would use.

Selection bias. The participants are randomly assigned to programs and treatments in the Latin square, minimizing selection bias. Additionally, each participant is exposed to both programs, and both treatments.

Carryover effects. Since two tasks are performed consecutively, it is possible that carryover effects influence the outcome of the second task, in particular the positive carryover effect of learning. To mitigate any carryover effects, we used a counterbalanced design, where the order of the two tasks are randomized.

Social desirability bias. Students participating in the experiment may have been overly positive in their post-experiment questionnaire responses, since they want to encourage and commend the author. The possibility has however been decreased by making the questionnaire completely anonymous.

6.2.2 Conclusion Validity

Carryover effects. A counterbalanced design is used to diminish carryover effects – but, there could still be effects of asymmetric skill transfer occurring in such a design – meaning that a particular order of tasks or treatments yield carryover effects, while the other does not. This has not been taken into account in the statistical analysis.

6.2.3 External Validity

Choice of subject programs. We chose Marlin, BusyBox, and Vim for the subject programs because they are well-known, highly configurable systems, and representative of industrial counterparts [32]. The completeness evaluation is made using all $N = 35$ examples from Marlin, but could be expanded to other ecosystems that have the same level of traceability for integration commits. There is no indication that the Marlin project employs specific merge strategies for integration commits, as such we argue that the intentions language is complete based on the evaluation. In the controlled experiment, we choose one integration scenario from BusyBox and Vim each, to expand from the Marlin-centric view.
Larger programs. In a real setting, files would naturally be larger than ca. 50 LOC, but since larger programs would take longer time to integrate, it would be harder to attract participants to the experiment. As such, the controlled experiment experiment is tailored towards internal validity, because of the tradeoff in experiment design between internal and external validity [35]. For integrating files beyond 50 LOC, there could be a different relationship between the effects observed in the controlled experiment.
Students as subjects. The participants in the controlled experiment were M.Sc. and Ph.D. students from Chalmers University of Technology, more than half with industrial experience. Previous studies have shown that graduate students can be used as proxies for professional developers [36]. In addition, all participants were familiar with integration techniques and tools.

7
Conclusion

We presented an empirical evaluation of the novel variant integration language and tool INCLINE, investigating the completeness of the language, and the beneficence of the tool, measuring editing efficiency and error frequency in variant integration. The completeness analysis is based on variant integration commits mined from the open source 3D-printer software Marlin. The editing efficiency and error frequency are evaluated using a controlled experiment with 16 students that perform variant integration tasks using the unstructured two-way merging tool in Eclipse CDT and INCLINE on subject programs derived from BusyBox and Vim. The controlled experiment is designed to reduce learning effects and optimize for internal validity. For completeness, all variant integration commits in Marlincan be replayed using the intentions language. Our results show that INCLINE does not perform worse than two-way merging in the number of defects introduced, but that fewer editing operations are required in INCLINE. However, it should receive an overhaul with respect to usability, since the task completion time using it was much longer compared to that of Eclipse CDT.
We are optimistic that this is a stepping stone towards proper variant integration tool support that can be used in the re-engineering of software product lines (eg., virtual plaform [10]). More in-depth studies of variant integration should be conducted, in two directions: open-source, using the developers of Marlinor projects similar to it, to elicit their insights in a pull-based setting; and professional developers working with re-engineering clone-based product lines. With an improved user interface and with developers comfortable with intention-based integration, INCLINE should prove beneficial to the software product line re-engineering process with respect to time, editing operations, and defects.

Bibliography

[1] 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.

[2] S. Stanciulescu, S. Schulze, and A. Wąsowski, "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.

[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] J. Melo, C. Brabrand, and A. Wąsowski, "How does the degree of variability affect bug finding?" in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16, 2016, pp. 679–690.

[6] 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.

[7] G. Gousios, M. Pinzger, and A. v. Deursen, "An exploratory study of the pull-based software development model," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014, 2014, pp. 345–355.

[8] 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.

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

[10] M. Antkiewicz, W. Ji, T. Berger, K. Czarnecki, T. Schmorleiz, R. Lämmel, S. Stănciulescu, A. Wąsowski, 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.

[11] K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications.* New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000.

[12] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," Software Engineering Institute, Carnegie Mellon University, Tech. Rep., 1990.

[13] S. Apel, C. Lengauer, B. Möller, and C. Kästner, "An algebra for features and feature composition," in *Algebraic Methodology and Software Technology: 12th International Conference, AMAST 2008 Urbana, IL, USA, July 28-31, 2008 Proceedings*, 2008, pp. 36–50.

[14] 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.

[15] W. K. G. Assunção, R. E. Lopez-Herrejon, L. Linsbauer, S. R. Vergilio, and A. Egyed, "Reengineering legacy applications into software product lines: a systematic mapping," *Empirical Software Engineering*, pp. 1–45, 2017.

[16] M. Voelter, J. Siegmund, T. Berger, and B. Kolb, "Towards user-friendly projectional editors," in *Software Language Engineering: 7th International Conference, SLE 2014, Västerås, Sweden, September 15-16, 2014. Proceedings*, 2014, pp. 41–61.

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

[18] B. Behringer, J. Palz, and T. Berger, "PEoPL: Projectional editing of product lines," in *Proceedings of the 39th International Conference on Software Engineering (ICSE)*, 2017, to appear.

[19] 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.

[20] J. M. Favre, "Understanding-in-the-large," in *Program Comprehension, 1997. IWPC '97. Proceedings., Fifth Iternational Workshop on*, Mar 1997, pp. 29–38.

[21] M. D. Ernst, G. J. Badros, and D. Notkin, "An empirical analysis of c preprocessor use," *IEEE Transactions on Software Engineering*, vol. 28, no. 12, pp. 1146–1170, Dec 2002.

[22] I. Abal, C. Brabrand, and A. Wąsowski, "42 variability bugs in the linux kernel: A qualitative analysis," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14, 2014, pp. 421–432.

[23] F. Medeiros, C. Kästner, M. Ribeiro, S. Nadi, and R. Gheyi, "The love/hate relationship with the C preprocessor: An interview study," in *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, 2015, pp. 495–518.

[24] F. Medeiros, M. Ribeiro, and R. Gheyi, "Investigating preprocessor-based syntax errors," in *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, ser. GPCE '13, 2013, pp. 75–84.

[25] M. Ribeiro, P. Borba, and C. Kästner, "Feature maintenance with emergent interfaces," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014, 2014, pp. 989–1000.

[26] M. Erwig and E. Walkingshaw, "The choice calculus: A representation for software variation," *ACM Trans. Softw. Eng. Methodol.*, vol. 21, no. 1, pp. 6:1–6:27, Dec. 2011.

[27] E. Walkingshaw and M. Erwig, "A calculus for modeling and implementing variation," *SIGPLAN Not.*, vol. 48, no. 3, pp. 132–140, Sep. 2012.

[28] M. Lillack, S. Stanciulescu, W. Hedman, T. Berger, and A. Wąsowski, "Intention-based integration of variants," 2017, Under submission.

[29] T. Berger, S. She, R. Lotufo, A. Wąsowski, and K. Czarnecki, "A study of variability models and languages in the systems software domain," *IEEE Transactions on Software Engineering*, vol. 39, no. 12, pp. 1611–1640, 2013.

[30] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, "An analysis of the variability in forty preprocessor-based software product lines," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10, 2010, pp. 105–114.

[31] J. Liebig, C. Kästner, and S. Apel, "Analyzing the discipline of preprocessor annotations in 30 million lines of c code," in *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*, ser. AOSD '11, 2011, pp. 191–202.

[32] C. Hunsen, B. Zhang, J. Siegmund, C. Kästner, O. Leßenich, M. Becker, and S. Apel, "Preprocessor-based variability in open-source and industrial software systems: An empirical study," *Empirical Software Engineering*, vol. 21, no. 2, pp. 449–482, 2016.

[33] G. E. P. Box, J. S. Hunter, and W. G. Hunter, *Statistics for Experimenters: Design, Innovation, and Discovery.* Wiley-Interscience, 2005.

[34] A. Strauss and J. Corbin, *Basics of Qualitative Research : Techniques and Procedures for Developing Grounded Theory.* Sage Publishing, 1998.

[35] J. Siegmund, N. Siegmund, and S. Apel, "Views on internal and external validity in empirical software engineering," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15, 2015, pp. 9–19.

[36] R. P. Buse, C. Sadowski, and W. Weimer, "Benefits and barriers of user evaluation in software engineering research," *SIGPLAN Not.*, vol. 46, no. 10, pp. 643–656, Oct. 2011.

A

Intention Language Examples

Adapted from the examples in [28]. The left hand side of the figures shows AST projection with nodes with applied intentions highlighted. The right hand side of the figures shows AST projection after application and transformation.

```
#ifndef FORK
  int servo_e1[] = SE
  int servo_e2[] = SEA
#else
 int16_t servo_e1 = SE
 int16_t servo_e2 = SEA
#endif
```

```
int servo_e1[] = SE
int servo_e2[] = SEA
#ifdef FORK
 int16_t servo_e1 = SE
 int16_t servo_e2 = SEA
#endif
```

Figure A.1: Keep intention (left) and result (right). Note condition rewriting.

```
#ifndef FORK
  int servo_e1[] = SE
  int servo_e2[] = SEA
#else
 int16_t servo_e1 = SE
 int16_t servo_e2 = SEA
#endif
```

```
#ifdef FORK
 int16_t servo_e1 = SE
 int16_t servo_e2 = SEA
#endif
```

Figure A.2: Remove intention (left) and result (right). Note condition rewriting.

```
#ifndef FORK
  int servo_e1[] = SE
  int servo_e2[] = SEA
#else
  int16_t servo_e1 = SE
  int16_t servo_e2[] = SEA
#endif
```

```
 int servo_e1[] = SE
 int servo_e2[] = SEA
```

Figure A.3: Keep intention with *implicit* Remove intention (left) and result (right). Applying both intentions at once is user interface sugar offered in INCLINE, and is not a part of the original intentions DSL.

```
#ifndef FORK
  int servo_e1[] = SE
  int servo_e2[] = SEA
#else
  int16_t servo_e1 = SE
  int16_t servo_e2[] = SEA
#endif
```

```
int servo_e1[] = SE
int servo_e2[] = SEA
```

Figure A.4: Remove intention with *implicit* Keep intention (left) and result (right). Applying both intentions at once is user interface sugar offered in INCLINE, and is not a part of the original intentions DSL.

```
#ifdef FORK
 card.pauseSDPrint();
 serial.writeln("Aborted");
#endif
```

```
#ifdef SDSUPPORT
 card.pauseSDPrint();
#endif
#ifdef FORK
 serial.writeln("Aborted");
#endif
```

Figure A.5: KeepAsFeature intention applied with argument `SDSUPPORT` (left) and result (right).

```
#ifdef FORK
 card.pauseSDprint();
#endif
```

```
#ifdef SDSUPPORT
 card.pauseSDprint();
#endif
```

Figure A.6: AssignFeature intention applied with argument `SDSUPPORT` (left) and result (right).

B

Controlled Experiment Tasks

A sample task sheet is given below, for participants in the group with VIM in `INC` followed by BUSYBOX in `Ecl`, cf. Latin square of Figure 3.3.

Task: Startup

1. Start recording the screen of the VM using the menu option View -> Video Capture.
2. Log into the account you've been assigned with the same password as the username.

Task INCLINE: Warmup

Integration goal:

```
#include "Marlin.h"
#define VERSION_STRING "1.0.0"

float current_position[NUM_AXIS] = { 0.0, 0.0, 0.0, 0.0 };
#if defined(BARICUDA)
int ValvePressure=0;
int EtoPPressure=0;
#endif

static float destination[NUM_AXIS] = { 0.0, 0.0, 0.0, 0.0 };
#if defined(DELTA)
static float delta[3] = {0.0, 0.0, 0.0};
#endif

void process_commands() {
#if !defined(DELTA)
plan_set_position(current_position[X_AXIS], current_position[Y_AXIS],
current_position[Z_AXIS], current_position[E_AXIS]);
#else
calculate_delta(current_position);
plan_set_position(delta[X_AXIS], delta[Y_AXIS], delta[Z_AXIS], current_position[E_AXIS]);
#endif
}

#if defined(DELTA)
void calculate_delta(float cartesian[3]) {
   delta[X_AXIS] = sqrt(sq(DELTA_DIAGONAL_ROD)
       - sq(DELTA_TOWER1_X-cartesian[X_AXIS])
       - sq(DELTA_TOWER1_Y-cartesian[Y_AXIS])
       ) + cartesian[Z_AXIS];
   delta[Y_AXIS] = sqrt(sq(DELTA_DIAGONAL_ROD)
       - sq(DELTA_TOWER2_X-cartesian[X_AXIS])
       - sq(DELTA_TOWER2_Y-cartesian[Y_AXIS])
       ) + cartesian[Z_AXIS];
   delta[Z_AXIS] = sqrt(sq(DELTA_DIAGONAL_ROD)
       - sq(DELTA_TOWER3_X-cartesian[X_AXIS])
       - sq(DELTA_TOWER3_Y-cartesian[Y_AXIS])
       ) + cartesian[Z_AXIS];
}
#endif
```

Task: INCLINE

You will now perform an integration in the INCLINE tool. Your task is to transform the provided code into the integration goal, provided below, using intentions.

1. Launch MPS from the desktop.
2. From the project structure on the left, find the Examples solution, open the `integrated-*.c` file.
3. Arrange the views (**Ctrl+Alt+Shift+V**) and proceed with applying intentions to reach the integration goal (next page).
4. When you are satisfied, save the file in `~/results/incline.c` using the menu "Tools -> Export C++ File".

Task INCLINE: Vim

Integration goal:

```
static char *(p_bg_values[]) = {"light", "dark", NULL};
static char *(p_nf_values[]) = {"bin", "octal", "hex", "alpha", NULL};
#if defined(FEAT_CMDL_COMPL)
static char *(p_clcot_values[]) = {"menu", "menuone", "longest", "noinsert", "noselect",
NULL};
#endif /* defined(FEAT_CMDL_COMPL) */
static char *(p_ff_values[]) = {FF_UNIX, FF_DOS, FF_MAC, NULL};

options[] =
#if defined(OS2)
(char_u *)"/c",
#else
(char_u *)"-c",
#endif /* defined(OS2) */

#if defined(FEAT_CMDL_COMPL)
/* Don't allow recursive cmdline mode when busy with completion. */
if (clpum_compl_started || clpum_compl_busy || clpum_visible())
{
EMSG(_(e_secure));
return NULL;
}
clpum_compl_clear();    /* clear stuff for clpum */
#endif /* defined(FEAT_CMDL_COMPL) */

switch (c) {
case K_UP:
#if defined(FEAT_CMDL_COMPL)
if (clpum_visible())
showmode();
#endif /* defined(FEAT_CMDL_COMPL) */
#if defined(FEAT_CMDHIST)
i = hiscnt;
#endif /* defined(FEAT_CMDHIST) */
beep_flush();
}
```

Task: Eclipse

You will now perform an integration task in Eclipse CDT. Your task is to merge the two variant files you are provided with into the integration goal, provided below. We will use RCPTT to record editing actions.

1. Launch RCPTT from the desktop.
2. The file called `recording` should already be open.
3. On the bottom panel labeled Application, right click the item "`org.eclipse`..." and press Run.
4. In the RCPTT instance, press the "Record" button. The RCPTT instance will minimize.
5. Open the workspace folder `task`, select the mainline and fork files in it.
6. Right click and select "Compare With -> Each other".
7. A diff view is opened. The files can be swapped from left to right if they are on the opposite side of what you would prefer.
8. The diff view is editable, so you may make any manual edits you like, as well as using the tools available in the diff view. Proceed with applying edits until you reach the integration goal (next page).
9. When you are satisfied, save the resulting file in `~/results/eclipse.c`. (It is not enough to just leave it in the workspace.)
10. Stop recording your actions in the RCPTT window by pressing the red button.

Task Eclipse: BusyBox

Integration goal:

```
#if defined(ANDROID) || defined(__ANDROID__)
#define endgrent() ((void)0)
struct timex;
pid_t getsid(pid_t pid);
int stime(const time_t *t);
int sethostname(const char *name, size_t len);
#ifdef READAHEAD
int adjtimex(struct timex *buf);
int pivot_root(const char *new_root, const char *put_old);
ssize_t readahead(int fd, off64_t offset, size_t count);
#endif
#endif

#if ENABLE_SELINUX
#include <selinux/selinux.h>
#include <selinux/context.h>
#ifdef FLASK
#include <selinux/flask.h>
#include <selinux/av_permissions.h>
#endif
#endif

extern loff_t bb_copyfd_eof(int fd1, int fd2) FAST_FUNC;
extern loff_t bb_copyfd_size(int fd1, int fd2, loff_t size) FAST_FUNC;
extern void bb_copyfd_exact_size(int fd1, int fd2, loff_t size) FAST_FUNC;
extern void complain_copyfd_and_die(loff_t sz) NORETURN FAST_FUNC;

#define OFF_T_MAX ( ( off_t ) ~ ( ( off_t ) 1 << ( sizeof ( off_t ) * 8 - 1 ) ) )

struct BUG_off_t_size_is_misdetected {
char BUG_off_t_size_is_misdetected[sizeof(off_t) == sizeof(uoff_t) ? 1 : -1];
};

#ifdef BIN
#define LIBBB_DEFAULT_LOGIN_SHELL "-/bin/sh"
#else
#define LIBBB_DEFAULT_LOGIN_SHELL "-/sbin/sh"
#endif

#ifndef _LARGEFILE64_SOURCE
/* For lseek64 */
#define _LARGEFILE64_SOURCE
#endif

#ifdef BIN
putenv((char *) "SHELL=/bin/sh");
#else
putenv((char *) "SHELL=/sbin/sh");
#endif
```

C

Questionnaires

Question types annotated with prefix and color: Likert-scale, Open, Choice.

C.1 Registration Questionnaire

1. C How many years of programming experience do you have?
2. C How many years of professional programming experience do you have?
3. L How familiar are you with C and the C preprocessor?
4. L How familiar are you with Java?
5. L How familiar are you with merging?
6. L How familiar are you with diffing?
7. L How familiar are you with the UNIX patch tool?
8. L How familiar are you with pull requests?
9. C Which version control systems do you have experience with?
10. C Do you have previous experience with projectional editing?

C.2 Post-experiment Questionnaire

1. L Integration with INCLINE is faster than in Eclipse.
2. L I make fewer mistakes in INCLINE than in Eclipse.
3. L Mistakes are easier to notice in INCLINE than in Eclipse.
4. L Mistakes are easier to fix in INCLINE than in Eclipse.
5. L INCLINE is a mature tool.
6. L Intention-based integration is not complex.
7. L The Keep intention is intuitive.
8. L The Remove intention is intuitive.
9. L The Implicit Keep/Remove intentions are intuitive.
10. L The Keep as Feature intention is intuitive.
11. L The Assign Feature intention is intuitive.
12. O Which intentions did you find the most useful?
13. O What are the advantages of using intentions for variant integration compared to manual integration with Eclipse?
14. O What are your perceived disadvantages of using intentions for variant integration compared to manual integration with Eclipse?
15. O Are there any possible improvements to intention-based integration or in particular INCLINE?

16. O How would you prefer to perform a variability-related integration?