Intention-Based Integration of Software Variants

Max Lillack*, Ștefan Stănciulescu†§, Wilhelm Hedman‡, Thorsten Berger‡, Andrzej Wąsowski§
*Leipzig University, Germany
†ABB Corporate Research Center, Switzerland
‡Chalmers | University of Gothenburg, Sweden
§IT University of Copenhagen, Denmark

***Abstract*—Cloning is a simple way to create new variants of a system. While cheap at first, it unfortunately creates a maintenance cost in the long term. Eventually, the cloned variants need to be integrated into a configurable platform. Such an integration is challenging: it involves merging the usual code improvements between the variants, and also integrating the variable code (features) into the platform. Thus, variant integration differs from traditional software merging, which does not produce or organize configurable code, but creates a single system that cannot be configured into variants. In practice, variant integration requires fine-grained code edits, performed in an exploratory manner, in multiple iterations. Unfortunately, little tool support exists for integrating cloned variants.**

In this work, we show that fine-grained code edits needed for integration can be alleviated by a small set of integration *intentions*—domain-specific actions declared over code snippets controlling the integration. Developers can interactively explore the integration space by declaring (or revoking) intentions on code elements. We contribute the intentions (e.g., 'keep functionality' or 'keep as a configurable feature') and the IDE tool INCLINE, which implements the intentions and five editable views that visualize the integration process and allow declaring intentions producing a configurable integrated platform. In a series of experiments, we evaluated the completeness of the proposed intentions, the correctness and performance of INCLINE, and the benefits of using intentions for variant integration. The experiments show that INCLINE can handle complex integration tasks, that views help to navigate the code, and that it consistently reduces mistakes made by developers during variant integration.

I. Introduction

Software variants emerge when architects experiment with new ideas or customize systems towards new market segments, new hardware platforms, runtime environments, or non-functional properties. Variants are often created by *cloning*—copying existing code and adapting it to new requirements by implementing new or modifying existing features [1]–[6]. Cloning is easy and cheap [7]. It is encouraged by version control tools through branching or forking. However, the long-term effort of maintaining and evolving cloned variants outweighs the benefits as soon as a few variants exist. When the challenges of maintenance and further evolution accumulate, architects often choose to re-integrate forked variants to benefit from code reuse techniques. One popular way is to build a configurable platform (or a software product line [2], [8]) that shares the variants' common code and allows to configure the variable features. The latter are typically represented by configuration options (features) that control variation points (e.g., preprocessor directives such as `#if` or `#ifdef`).

Even though re-engineering variants is the most common way to create integrated platforms in software industry [2], concerningly few methods and tools exist for the problem of transforming existing variants into a configurable platform; much less than for creating product lines from scratch. Prior work has mainly focused on understanding the commonalities and differences among variants [6], [8], largely sidestepping the actual integration. Integration research appears necessary to bring software product line results to a broader practice.

Not only is integrating cloned variants challenging, but it also consumes the most valuable resources in software teams. One of our industrial partners states: *"Developers tend to be specialized in one variant. [...] They have difficulties to switch from one variant to another. Due to these difficulties, merges are done by the most experienced developers, who we would want to use on more useful tasks."* An engineer performing the task, needs to understand the richness of the variants and their differences. She also needs to consistently distinguish and integrate evolution changes, common code, and variable code, injecting variation points. Our contribution is to help this second challenge.

The two code excerpts on the left in Fig. 1 are simplified variants of a file adapted from the mainline and a fork of a 3D printer firmware project. To integrate them (the rightmost excerpt) developers need to comprehend the code, understand the differences (and how they are aligned), make design decisions (e.g., what to keep, what to remove), and low-level edits. This can be done in many ways. Developers apparently need to explore different possibilities and undo frequently. Specifically, they need to obtain a single consistent `#if` hierarchy guarded by correct `#if` expressions, especially cumbersome and error-prone when multiple variations in forks overlap. We show in Sec. II that it is difficult for integrators to align changes and obtain an overview on the variants, even when using a modern diff tool. Developers sometimes even give up to integrate larger and conflicting variants [9], [10], especially when variants have intricate and undisciplined `#if` structures typical in configurable C projects, including our evaluation subjects [11]–[13].

We show how to partially automate the process, and how to make it more flexible by centering it around recording (and revoking) mostly independent, higher-level decisions, *integration intentions*—programming-language-independent actions declared over code. They define how variants should be integrated, whether to keep functionality, declare variants as exclusive, or extract a configurable feature. While intentions do not replace domain knowledge, they support exploring different

integrations much more flexibly than an undo system. One can immediately observe results and revoke some of the recorded intentions (with no undo stack discipline). Intentions can be nested. Even though they are intuitively simple, their resolutions on code can be complex (e.g., when intentions interact), automatically creating variation points (i.e., `#if` structure) and the created `#if` structures are correct by construction.

We propose and define the integration intentions, and implement a prototype IDE tool INCLINE (intention-based clone integration). INCLINE works with C preprocessor, but it is otherwise language-independent. It offers five editable views on the variant code. Unlike the views of diff tools, designed for code merging, our views take integration of variants with configuration options as a first class concept. We evaluate INCLINE using five popular open-source systems with forks: Marlin, a 3D printer firmware; Vim, a UNIX text editor; BusyBox, a suite of shell tools; libuv, an IO library; and PHP, an interpreter. We perform a set of realistic variant-integration simulations with file variants up to 4K lines, and a controlled experiment with 12 participants. We find that intentions are sufficiently rich to integrate real variants, their resolutions are correct. Most importantly, developers make less mistakes when integrating with INCLINE, and find declaring intentions simple and intuitive compared to manual integration.

We hope that INCLINE can further advance the integration practice, and can be used as a device to obtain more data from industrial partners and open-source developers about this process with a further improvement in view.

II. Motivation and Background

We now discuss variant integration challenges. These originate from our running example, our own (action-research) experiences with industrial partners and open-source variant-rich systems, as well as a think-aloud exercise. In the latter, we let three developers execute two integration tasks, respectively from Marlin and Busybox, using Eclipse's diff tool. The participants received three files: one with the mainline code, one with the related fork code, and the target solution. The Marlin files had over 2,000 lines of code and over 100 `#ifdef` blocks; the Busybox files 25 lines of code and only two `#ifdef` blocks. During the integration, the participants were asked to speak aloud about what they are doing. We recorded this process and reviewed the spoken comments, used to illustrate and confirm challenges in this section. In the following, we refer to our running example (Fig. 1) and the think-aloud participants.

Challenge 1: Variant Integration is Not Code Merging. Variant integration differs from traditional code merging [14], which combines changes performed in isolation into a *single system*. Merging does not directly support realizing variants or building a configurable platform. Traditional merge algorithms combine as much code as possible and delegate conflicts they cannot resolve to the developer. In contrast, our focus is on efficiently transforming system variants—that were developed in parallel and that can realize conflicting (i.e., mutually exclusive) functionality—into a platform where they can co-exist. Deciding whether a change should be shared by all variants or be specific to some variants, has to be done regardless whether a merge conflict occurs or not. Even smoothly merging changes might need to become optional.

Research on variant integration is found under the broader area of re-engineering legacy products into a software product line [3], [8], [15], [16]. However, works in this area almost solely focus on discovering commonalities and variabilities between codebases to gain an understanding of how similar or far apart they are, together with research on identifying and locating features or synthesizing so-called feature models [17].

Challenge 2: Domain Knowledge. Integration requires domain knowledge of the developer, which we cannot alleviate. Yet, even with such knowledge, the comprehension of what changed in the variants is a challenge. Consider viewing the excerpts from our running example in a traditional diff tool, as shown in Fig. 3, which highlights the differences, but does not help with the integration. Working with a diff tool, the developer needs to comprehend such diffs, while editing the text to create an appropriate `#if` structure. In fact, all our think-aloud participants confirmed this challenge and demanded better views to explore the variants and to observe how changes influence the result.

Challenge 3: Code Alignment. A closely related challenge is code alignment, confirmed by all think-aloud participants. Diff tries to align text and changes, but often fails, leading to mismatching. Furthermore, using a single view, where variants are explored and also modified, is problematic, since edits change the diff. Using a new file where changes are copied into increases effort (also cognitive effort), unfortunately. One think-aloud participant stated the need to create a new file where all the changes should be stored, such that the two variant views can be used to understand the differences. Another one was overwhelmed by the diff tool highlighting every line.

Challenge 4: Create a Valid Variation Point Structure. Developers need to create a target `#if` structure, edit the code to include respective changes, and create *presence conditions* (Boolean expressions over features, determining when the respective code is included in a variant derived from the platform) [18]. In our running example (Fig. 1) the fork's line 3 (added variable `encoderDiff`) could be either made mandatory or optional, the latter by adding an `#ifdef` NEWPANEL. Furthermore, the fork's lines 5–7 could also belong to this feature or another one (depending on domain knowledge)

The most trivial solution would be to create a new feature that represents the variant (we will call such a feature `FORK` in the remainder) and wrap the complete files in an `#ifdef-#else` block. This fails to recognize any commonalities and creates much redundancy, not providing any benefit. Another strategy could be wrapping all differences in `#ifdef` blocks. GNU's *diff* tool using `diff -w -D FORK file1 file2` actually supports that, where FORK would be the feature name. However, this can easily lead to an invalid `#if` structure, since the added directives interfere with existing ones in the variants. This happens for our example, as seen in Fig. 4. Figure 5 shows a correct structure.

Challenge 5: Low-Level Editing. Traditional integration amounts to doing many low-level editing tasks, many of which

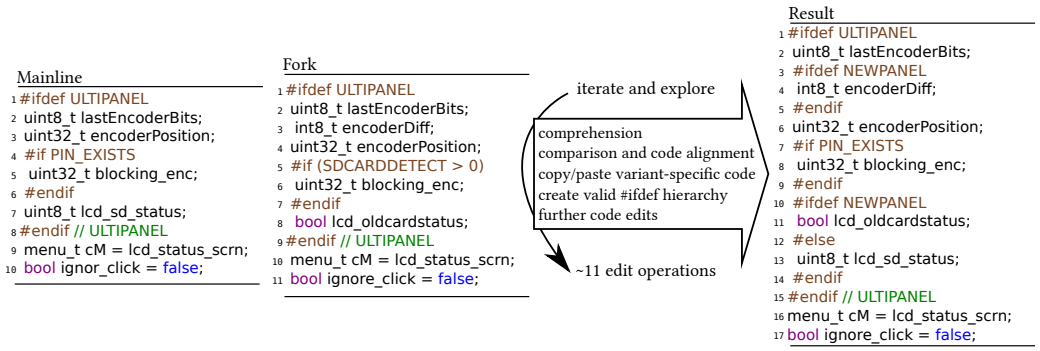


Fig. 1: Running example. Left: mainline and fork variants to be integrated. Right: integration goal. Middle: necessary activities.

need to be explored by developers.

For our running example, a developer *could* perform the following edit operations (bottom right), based on her domain knowledge that the fork realizes a feature called `NEWPANEL`: (1) Take mainline as a base, copy fork's line 3 and add `#ifdef NEWPANEL` to make it optional. (2) Copy line 8 from fork above line 7 in mainline, wrap by a new `#ifdef NEWPANEL`, move mainline's previous line 7 to the respective `#else` branch, to preserve mainline's functionality. (3) Fix typo on mainline's line 10 as already done in the fork (line 11). Essentially, this amounts to, 11 line-level-based editing operations (e.g., highlight a line, copy it, write an `#ifdef` NEWPANEL).

Challenge 6: Introduce Features. The strategy of pairwise diffing and wrapping changes in `#ifdefs` is relatively common. For instance, the company Danfoss used it to integrate forks [19], [20], creating "features" representing individual variants. While this integration was quick (months) for a system with 1.5M lines of code, it took four years to achieve the desired platform. The process was mostly manual, with minimal tool support; during it, the platform was iteratively verified. It was especially challenging to refactor the variant-based "features" into around 1000 variant cross-cutting, intuitive features (e.g., a feature representing a specific motor instead of PRODUCT_A).

Challenge 7: Iterative Exploration of the Integration. Recall that many different edit sequences can lead to the desired integration. Developers need to explore their edits,

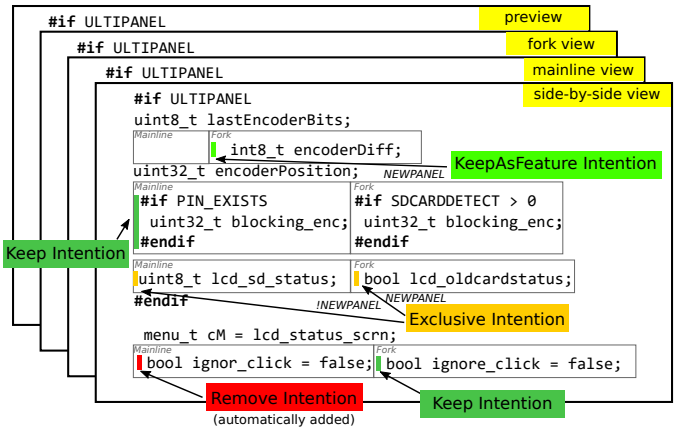


Fig. 2: The integration of our example (Fig. 1) with INCLINE. Developers add intention on the code within different views

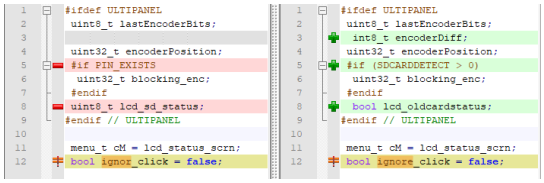


Fig. 3: Our running example (Fig. 1) in a traditional diff tool

```
1  #ifdef ULTIPANEL
2   uint8_t lastEncoderBits;
3  #ifndef FORK
4   uint32_t encoderPosition;
5   #if PIN_EXISTS
6  #else /* FORK */
7    int8_t encoderDiff;
8   uint32_t encoderPosition;
9   #if (SDCARDDETECT > 0)
10 #endif /* FORK */
```

Fig. 4: An invalid `#if` structure created by `diff -D` for our example (Fig. 1). Grey lines show the added structure, red lines violations due to existing variant `#ifs`.

including refactoring `#if` structures and backtracking (undo changes), which can be cumbersome and error-prone for low-level editing. For instance, consider lines 4–6 in the mainline and lines 5–7 in the fork (Fig. 1). In our example, the order of these blocks when integrated does not matter, since they have no side effects. But, if these were statements, it might be necessary to move them to the right order—an insight developers could just obtain after doing a first integration. All our think-aloud participants confirmed the need for an iterative exploration, with the ability to easily undo changes.

```
1  #ifdef ULTIPANEL
2   uint8_t lastEncoderBits;
3  #ifdef FORK
4    int8_t encoderDiff;
5  #endif /* defined(FORK) */
6    uint32_t encoderPosition;
7  #if (defined(FORK) || PIN_EXISTS) && (!defined(FORK) || SDCARDDETECT > 0)
8  uint32_t blocking_enc;
9  #endif
10 #ifdef FORK
11   bool lcd_oldcardstatus;
12 #else
13   uint8_t lcd_sd_status;
14 #endif /* defined(FORK) */
15 #endif /* defined(ULTIPANEL) */
16 menu_t cM = lcd_status_scrn;
17 #ifdef FORK
18   bool ignore_click = false;
19 #else
20   bool ignor_click = false;
21 #endif /* defined(FORK) */
```

Fig. 5: The default integration for our running example (Fig. 1)

Challenge 8: Cognitive Load of Variability. There is also the cognitive load of the C preprocessor, whose `#if` directives clutter source code and challenge comprehension [12], [21]. This can easily lead to code ending up in the wrong variant (the wrong `#if` block) [22]. Graphical representation of the preprocessor using dedicated tools have been proposed in the literature [23], [24], but not for variant integration.

III. Integration with INCLINE

INCLINE addresses the challenges described above as follows.
Challenge 1: Variant Integration is not Code Merging. Instead of merging, INCLINE creates an integrated platform with a valid variation point (`#if`) hierarchy. INCLINE's input are pairs of file variants (mainline and fork), The variants, written in any programming language, may already contain features and variation points, which INCLINE reads and manipulates. INCLINE shows differences in multiple editable views (explained shortly), including a view providing a default integration. For the latter, INCLINE adds the feature `FORK` and wraps variant-specific code with the presence condition `!FORK` (mainline) or `FORK` (fork), conjoined with presence conditions of already existing variation points in the variants. Figure 5 shows the default integration for our example (Fig. 1).
Challenge 2: Domain Knowledge. To foster comprehension, the developer can navigate and comprehend the default integrated platform, the variants, and their differences using five *views*. Figure 6 displays four of the views. The *mainline view* and the *fork view* (top left and top right of Fig. 6) show the previous variants (internally realized as a partial configuration of the default integrated platform). The green bar represents a *Keep* intention (explained shortly). The *integrated side-by-side view* (bottom left of Fig. 6) shows the integrated platform, but with the differences between mainline and fork arranged next to each other, without `#ifdef` directives. The fifth view (illustrated in Fig. 5), called *integrated view*, would display the integrated platform like the *integrated side-by-side view*, but using `#if` annotations. The *result view* (bottom right of Fig. 6) previews the final result with all intentions resolved.

On a side note, we implemented INCLINE using the language workbench MPS [25]. It relies on *projectional editing* (a.k.a., syntax-directed or structural editing [26], [27]), where a user's editing gestures directly change the underlying AST, without using any parser. The AST is still rendered into concrete syntax (program code). Projectional editing is well-suited for creating editable views. The variant views rely on a partial configuration of the variational AST, for which we use Z3 [28] to reason about presence conditions and calculate the projections.
Challenge 3: Code Alignment. The *integrated side-by-side view* arranges the differences between mainline and fork differently than an ordinary diff tool would do. Within the source code, the view aligns chunks of code that are common and shows chunks of code that differ in horizontal boxes, which can be nested (just like preprocessor directives can be nested). Showing the boxes and even nesting them is possible through the technology projectional editing. This helps to represent the code that differs together, side-by-side.

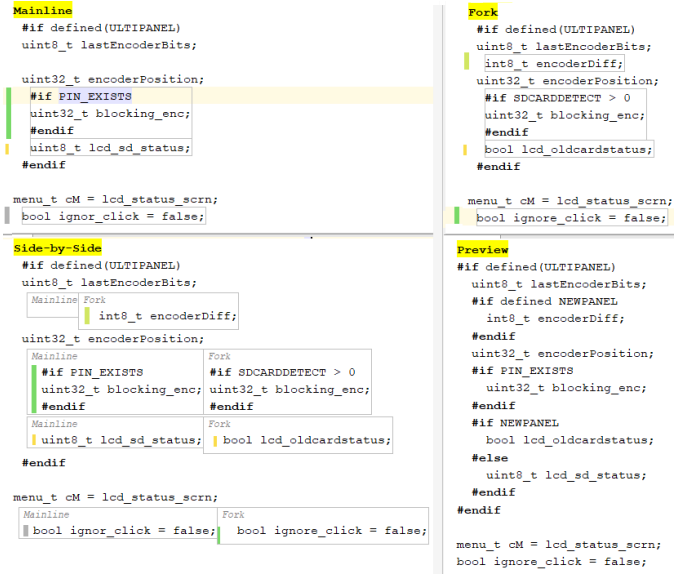


Fig. 6: Screenshot of INCLINE views: *mainline view* (top left), *fork view* (top right), *integrated side-by-side view* (bottom left), and *result view* (bottom right)

Challenge 4: Create a Valid Variation Point Structure. INCLINE implements a simplified version of the C preprocessor language (`#ifdef`, `#else`, `#if`, `#elif`, `#endif` directives). The actual program code is stored as an unstructured uninterpreted text (string of characters). So, INCLINE is independent of the host language and would work on, say, C++ programs the same as on C programs, without modification. To enhance editing the target language (e.g., code completion, syntax highlighting), our preprocessor language could also be composed with a language available in MPS, such as C99 [29]. To import existing source files into INCLINE, we built a C preprocessor parser upon *clang*. To create the integrated variational AST, we use the input files' parse results to create XML-based representations of their ASTs, and then use JNDiff [30] to obtain a valid variation point hierarchy, which is then transformed into our variational AST (explained shortly in Sec. V).
Challenge 5: Low-Level Editing. As the main contribution, INCLINE aims at alleviating low-level editing operations to improve the efficiency of developers and to reduce mistakes (e.g., writing incorrect presence conditions or creating a wrong `#if` structure where code ends up in the wrong variants).

Intentions specify the goal of integrating a change, that is, how the integrated AST should be customized with respect to the input variants. For instance, a developer could ask: should the change be made common to all variants? Or only to some of them? Or should it remain variant-specific? Is the change standalone or is it intended to belong to a feature implementation? In the views, the developer defines intentions and if necessary can also manually edit the code.
Challenge 6: Introduce Features. INCLINE provides intentions (*KeepAsFeature*, *AssignFeature*) that are parameterized to allow specifying features. When adding such an intention on the code in INCLINE, the user is asked about the feature.

Note that an effective management of an integrated platform

also needs mechanisms for modeling and managing features and their dependencies. These are, however, orthogonal concerns to variant integration and therefore beyond the scope of this work. Yet, an integration would be valuable future work.

Challenge 7: Iterative Exploration. When declaring intentions in the views, the developer explores their effects in the views, especially the *result view*. Since all views rely on one variational AST, they are updated (synchronized) immediately. Intentions can be easily removed the same way they are added—with just one keystroke. Furthermore, since the integrated variational AST is always syntactically correct, developers can at any time derive individual variants (e.g., to run a test suite).

IV. Research Methodology

Definition of Intentions. To conceive our primary contribution, the integration intentions, we relied on three sources. First, we reflected the variability mechanisms that appear in integrated platforms, since there code can be mandatory or optional, controlled by presence conditions. Second, we relied on experiences from the think-aloud exercise. Third, we inspected diffs of the Marlin ecosystem. Marlin (>40 KLOC of C++ code) has over 4000 forks, many of which evolve separately and independently add new functionality. Given this richness and the existing re-integration efforts of the community, Marlin is an ideal subject for conceiving intentions. our evaluation. Since most forks contain no or just minimal changes, we specifically looked at forks that implement new features and how they were either already integrated into the mainline or what challenges an integration would entail. We also inspected merge commits that involved preprocessor directives, pull requests, and conflicting commits—all examples of developers performing integrations while dealing with variability. In the remainder, we formally present the intentions and their deterministic resolutions. We will later show that these intentions are sufficient for integrations in five open-source systems, including Marlin.

Tool Design. Conceiving an end-user tool is intrinsically difficult, especially given the resources of researchers. To obtain a usable tool, the design followed an iterative process, with a prototyping and evaluation feedback loop. In our first prototype of INCLINE, we implemented the basic functionality: source file import, user interface to declare intentions, and back-end to resolve intentions. After that, we have executed a pilot user study, to detect and resolve inefficiencies in the tool.

To check the usability of the first prototype, we recruited 16 MSc students to execute two integration tasks. We compared the performance of INCLINE against Eclipse's diff/merge tools. We extracted two tasks from the histories of BusyBox and Vim. Each task concerned the integration of code from a fork (for Vim one that adds support for command-line completion; for BusyBox one that is tailored for Android) to the mainline. The participants were given a correct target integration and a brief description of the integration goal. Showing a target solution was a pragmatic way to reduce the influence of (lack of) domain knowledge for this first usability check. The subjects solved the task in controlled time. We observed how they worked using screen recordings. The tasks used are available in the online appendix [31, Pre-Study].

Improvement Cycle. Based on the results, we improved the prototype and customized the default configuration. Specifically, we: (i) added keyboard shortcuts for intention declaration actions as we could see in the screencasts that mouse input and menus were inefficient, (ii) by default, arrange views as in Fig. 6 and use *integrated side-by-side view* instead of *integrated view* by default, (iii) for convenience, added heuristic proposals for further intentions to the user (e.g., when *Keep* declared for all nodes in an `#if` branch, INCLINE proposes a corresponding *Remove* intention for the nodes in the `#else` branch), and (iv) improved the highlighting of the applied intentions, so it is easier to see for which nodes intentions have been declared, and reduced the observed trial-and-error.

Evaluation. We replayed real edits mined from the history of five variant-rich open-source systems and conducted a controlled experiment. See Sec. VI for details.

V. The Integration Intentions

We propose the following intentions to be used for high-level control of the integration process: *Keep*, *KeepAsFeature*, *Exclusive*, *Remove*, *AssignFeature*, and *Order*.

A. Variational AST and Views

A variational *AST* is a syntax tree with embedded `#if`s. To simplify the discussion we often see it just as a set of nodes (*Nodes*). For each node n, we identify a sequence of conditions, $c_1^n, c_2^n, \cdots, c_k^n$, used in the `#if`s on the path from the AST's root (c_1^n) to n (c_k^n). A node that is not wrapped by any `#ifdef` (non-variable node) has an empty sequence of conditions.

We define the *presence condition* $pc(n)$ to be the conjunction of all conditions used by the `#if`s the node $n \in$ *Nodes* is contained in. For a non-variable node it is *true*.

$$pc(n) = \bigwedge_{i \in 1..k} c_i^n$$

We define a $block$ as a set of nodes in the AST: $block \in \mathcal{P}(Nodes)$. We also introduce an *order* of nodes, which describes the syntactic order of the C/C++ program. If a node n_1 exists before node n_2 in the syntactic order, we write $n_1 < n_2$.

A *view* is a projection of the AST showing only a specific set of variants specified by a *view constraint* ρ over features. For the views *mainline* and *fork*, $\rho = \neg FORK$ and $\rho = FORK$, respectively. For the *integrated views* $\rho = true$. Of course, ρ can be a more complex expression, not just a literal, to filter out more variability not relevant for the integration.In the remainder, we limit ourselves to simple view constraints, though. Note that the view through which an intention is declared forms the context for the intention and as such influences its resolution.

To determine how conditions are shown in a view, we substitute every occurrence of the view constraint in the conditions with *true*:

$$\forall n \in Nodes \quad c_\rho^n = c^n[\rho \leftarrow true],$$

where c_ρ^n denotes the conditions shown in the view.

```
#ifndef FORK // block_not_fork
int servo_e1[] = SE
int servo_e2[] = SEA
 #else // block_fork
 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
```

Fig. 7: *Keep* intention (left) and result (right)

In the variant views, we render the AST as follows:

- We hide nodes where $pc(n)[\rho \leftarrow true] \equiv false$
- We show nodes without the surrounding `#if` if $c_k^n \equiv true$
- For the remaining nodes, we simplify presence conditions [18] with respect to the view constraint using an SMT solver

Special care needs to be taken for blocks containing nodes with *complex* presence conditions. Complex conditions contain both the view constraint ρ and some unrelated terms. When using views, a complex presence condition is shown as a non-complex condition, since the view constraint is simplified. Still, the hidden view constraint is part of the context that the intentions are declared in, just as if the node was implicitly wrapped in an `#if` with the view constraint as its condition. Formally, we say a condition is *complex* iff $c^n \not\equiv c^n[\rho \leftarrow true]$. For an `#if` node n with a *complex* condition, we rewrite the sequence of conditions so that it ends with the view constraint. We use the notation $pc(n, \rho)$ to denote the presence condition of node n in a view with constraint ρ:

$$pc(n, \rho) = \bigwedge_{i \in 1..k} c_i^n \wedge \rho$$

B. Semantics of Intentions

We now define the individual intentions and illustrate them with examples. Intentions are partial functions transforming ASTs. We formalize their semantics as effects they have on the presence conditions and ordering of nodes. In Sec. V-C, we show how the intentions are resolved (implemented) on the AST. The figures (for instance, Fig. 7) show the integrated AST on the left, selected nodes on which an intention is declared in gray, and the desired result on the right. The examples use the *integrated view*, which shows all variants at once. The verbosity of this view makes it most suitable to explain how intentions work. For each intention we use the notation $pc'(n)$ to illustrate the resulting presence condition of the node n, and $pc(n)$ for the presence condition before the intention resolution.

Keep. The *Keep* intention includes a block as it appears in mainline or fork in an unconditional manner, without guarding it with any additional feature. Consider the example in Fig. 7 where we define *block_not_fork* to represent the set of nodes in the $\neg$*FORK* branch highlighted with gray, and the *block_fork* represents the set of nodes in the *FORK* branch. The fork changes the type of the *servo* variables to be 16-bit signed integers, because different hardware and compiler are used for this variant. During the integration process, it is decided that the hardware used in the fork should no longer be supported, and only the code from mainline is kept. We apply the *Keep* intention on the *block_not_fork* set of nodes. The right side shows

```
#ifdef FORK // block_fork
 card.pauseSDPrint();
#endif
```

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

Fig. 8: *KeepAsFeature* intention (left) and result (right)

the result of applying *Keep* on the selected *block_not_fork*. The nodes from the fork should be removed (which can be done with the dual of *Keep*, *Remove*, described below). Note that the integration is not completed, as there is still a block from the forked variant, which should be resolved later.

The effect of *Keep(block)* on the presence conditions is:

$$pc'(n) = \begin{cases} c_1^n \wedge ... \wedge c_{k-1}^n, & \text{if } n \in \textit{block} \\ pc(n), & \textit{otherwise} \end{cases}$$

The nodes for which *Keep* was declared should no longer be under the constraint created by the `#ifdef` that directly wraps those nodes (in the example we drop `!FORK`). Their new presence condition is the conjunction of all but the last constraint that directly wrapped the nodes. All nodes that are not part of the intention are unchanged.

KeepAsFeature. The *KeepAsFeature* intention preserves a block from one of the variants, but makes it conditionally present, only linked to a certain feature or combination of features. It wraps the block with a new presence condition given with the intention. In the example of Fig. 8, a fork developer added functionality to pause a 3D print from an SD card. Not concerned with other devices than the one for which the fork was developed, she included the new behavior unconditionally. However, in the integration process, it became clear that this functionality only makes sense in variants supporting SD cards, thus, it needs to be included conditionally. The desired result is shown on the right side of the figure.

KeepAsFeature(block, F) replaces the last constraint from the sequence of constraints with the new presence condition:

$$pc'(n) = \begin{cases} c_1^n \wedge ... \wedge c_{k-1}^n \wedge F, & \text{if } n \in \textit{block} \\ pc(n), & \textit{otherwise} \end{cases}$$

Exclusive. This intention declares that two code blocks should be mutually exclusive (enforcing the separation of conflicting functionality), controlled by a choice condition. In Fig. 9 the fork introduces a new optional feature `FIL_DISPLAY` and keeps the line that prints a message on the LCD display under a specific condition. The integration requires keeping the optional feature and ensuring that when this feature is not selected a message is shown on the LCD (to not break the

```
#ifndef FORK //block_2
 lcd.print(msg);
#else
 #ifdef FIL_DISPLAY //block_1
 if(condition){
  lcd.print(msg);
 }else{
  lcd.print(trnsf(data));
 }
 #endif
#endif
```

```
#ifndef FIL_DISPLAY
 lcd.print(msg);
#else
 if(condition){
  lcd.print(msg);
 }else{
  lcd.print(trnsf(data));
 }
#endif
```

Fig. 9: *Exclusive* intention with the three parameters $block_1$, $block_2$, *FIL_DISPLAY* (left) and result (right)

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

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

Fig. 10: *AssignFeature* intention (left) and result (right)

mainline variant). Therefore, we keep both blocks as a mutually exclusive implementation using the *Exclusive* intention.

We introduce the helper function *common(block)* which returns the longest common subsequence of conditions of nodes in the block (semantically akin to the prime implicate of the set of presence conditions of the block's nodes):

$common(block) = c_1 \wedge \ldots \wedge c_s$ such that

$$\forall_{n \in block} \bigwedge_{i \in 1..k} c_i^n \rightarrow \bigwedge_{i \in 1..s} c_i \text{ and } s \text{ is maximal such.}$$

Then the *Exclusive(block1, block2, F)* has the effect as follows:

$$pc'(n) = \begin{cases} common(block_1 \cup block_2) \wedge F, & \text{if } n \in block_1 \\ common(block_1 \cup block_2) \wedge \neg F, & \text{if } n \in block_2 \\ pc(n), & otherwise \end{cases}$$

We use the common conditions of the nodes in $block_1$ and $block_2$ as the basis and then include the feature condition F (or its negation) to control the selection of the variant.

Remove. This intention deletes the selected nodes from the AST. By definition, it ensures that the selected nodes do not exist in the updated AST': $\forall n \in block \quad n \notin AST'$

AssignFeature. This intention is used when code was already integrated, but its presence condition should be changed (e.g., simplified, weakened, or strengthened). This intention can only be declared for complete #if-#else-#endif blocks. Fig. 10 shows the renaming of feature *SD* (left) to *SDSUPPORT* (right). The effect of *AssignFeature(n, F)* is that the last constraint of nodes from both branches (`#if` and `#else` of the `#ifdef` block) is replaced with the given feature, and respectively the negated feature:

$$pc'(n) = \begin{cases} c_1^n \wedge \ldots \wedge c_{k-1}^n \wedge F, & \text{if } n \in \textit{if branch of } n \\ c_1^n \wedge \ldots \wedge c_{k-1}^n \wedge \neg F, & \text{if } n \in \textit{else branch of } n \\ pc(n), & \textit{otherwise}. \end{cases}$$

Order. This intention prescribes an order of blocks from the variants for the integrated AST (with respect to the concrete syntax). As a notation, we resort to the operators $>$ and $<$, which declare that the first block be put before the second block, and vice versa. This intention re-orders blocks or ensures their correct order during integration, especially when further intentions are applied. For example, we apply the intention *Keep* on a mainline and a fork block, but we want the mainline code to be executed first, we declare $order(block\_fork, block\_main, <)$, which then yields the correct order.

C. Intention Resolution

We conceived a deterministic AST in-place transformation to resolve intentions, which needs to consider *all* declared intentions at once, since intentions can interact. Although intentions are declared on blocks of nodes, their resolution will affect other nodes. Algorithm 1 outlines the intention resolution.

Algorithm 1 Intention Resolution Algorithm

```
Require: AST
 1: for all type in {Keep, KeepAsFeature, Exclusive, ...} do
 2:     for all node in breadth-first(AST) do
 3:         intention = getIntention(AST, node, type)
 4:         if intention != null then
 5:             AST = EXECUTEINTENTION(AST, intention)
 6:         end if
 7:     end for
 8: end for
```

We resolve intentions in a specific *order*: based on our own defined intention priority, *and* on the AST structure (top down, from outermost to innermost node). The intention priorities are (descending, from resolved first to resolved last): *Keep*, *KeepAsFeature*, *Exclusive*, *Order*, *AssignFeature*, *Remove*. The priority aims to minimize unexpected interactions of intentions. For example, when resolving a *Keep*, a node might be moved out of its parent for which a *Remove* intention could be declared. The defined priority ensures the node is actually *kept* in the result and not overruled by the *Remove* intention.

Each intention resolution will use the result from the previous intention resolution as its base, therefore, the resolution of one intention will likely influence the resolution of intentions that are declared for descendant nodes. The idea is that intentions for descendant nodes do not influence parent nodes.

The method EXECUTEINTENTION (line 5 in Algorithm 1) refers to the following specific resolution for each intention:

Keep. Fig. 11 illustrates the resolution with a simple example. Observe how the `#if` structure of sibling nodes is affected. In general, resolving $Keep(block)$ amounts to first moving all following sibling nodes of *block* to a new `#if` node placed as a following sibling of the parent `#if` node of *block*. The new `#if` node has the same condition as the parent `#if` of *block*. Second, the nodes in *block* itself will be moved as new following sibling of their parent node, so that they are a preceding sibling of the newly created `#if` node. Recall that the view determines the context for intentions (cf. Sec. V-A), and that in the variant views the `#if` conditions shown are simplified (feature FORK is enabled or disabled). If a *Keep* intention is declared on such an `#if` with a simplified condition in the view, then the simplified condition is propagated to the AST.

KeepAsFeature. $KeepAsFeature(block, feature)$ is resolved like *Keep*, except that the nodes unwrapped during the *Keep* step are now wrapped in a new `#if` with the condition *feature*.

Exclusive. We resolve $Exclusive(block_1, block_2, feature)$ using *Keep* on $block_1$ and $block_2$. The results are wrapped in a new `#if` where the nodes of $block_1$ are moved to the *true* branch and the nodes from $block_2$ are used in the *else* branch. The condition of the new `#if` is given by the parameter *feature*.

Order. For $order(block_1, block_2, <)$ we assume the nodes of $block_1$ and $block_2$ have the same parent. If all nodes in $block_2$ are, based on their position in the parent's list of child nodes, be-

```
#if A
 a()
b()
 c()
#endif
```

```
#if A
 a()
#endif
b()
#if A
 c()
#endif
```

Fig. 11: Resolution of *Keep* (`b()`) splits `#if` block (right)

fore the nodes of $block_1$, we switch their position. We perform this action similarly with reversed parameters for operator $>$.

Remove. For $Remove(block)$ we remove all nodes in *block* from the AST. Nothing happens when a node was already removed (e.g., *Remove* was declared for an ancestor of the node).

A generic cleanup step after intention resolution removes empty `#if` nodes, which do no longer contain any nodes.

Example. In our running example (Fig. 2), we declare five intentions. First, a *KeepAsFeature* intention is used for the added variable `encoderDiff` on the fork side. Second, a *Keep* intention covers the `#if PIN_EXISTS` of the mainline side. Technically, `#if PIN_EXISTS` of the mainline side and `#if SDCARDDETECT>0` of the fork side are both representations of the same `#if` AST node. The AST node has a complex condition, based on our definition in Sec. V-A, from which simpler but view-specific conditions are derived for the representation. The *Exclusive* intention comprises two nodes (`lcd_sd_status` in mainline and `lcd_oldcardstatus` in fork view). The last intentions are a pair of *Keep* and *Remove* intentions.

The intentions are resolved first in order of their priority, i.e., INCLINE first resolves the two *Keep* intentions. Within the same priority, a breadth-first traversal is used so that first the intention for the last line of the fork view is resolved, then the one for `#if PIN_EXISTS` of the mainline, because the corresponding node is nested in the AST node for `#ifdef ULTIPANEL`. After that, the remaining intention types are resolved. Each intention is resolved by modifying the underlying AST, e.g., to resolve the *KeepAsFeature* intention a new `#ifdef` node is created which wraps the node with the variable `encoderDiff`. In a cleanup step, the now empty `#ifdef FORK` (c.f., line 3 in Fig. 5) is removed from the model.

VI. Evaluation

To establish how well INCLINE supports developers during variant integration, we formulated the following questions. Their evaluation methods are detailed in the following subsections.

- RQ1 *What is the coverage of the proposed set of intentions with respect to real-world integration needs?*
- RQ2 *Is the implementation of intention resolution correct with respect to their semantics, even for large files?*
- RQ3 *What is the benefit of INCLINE over manual variant integration from users' perspective?*

Subject Systems. We chose open source systems that use preprocessor directives to implement variability, and for which real forks are available. We used active projects from different domains to minimize the bias towards a particular usage of the preprocessor. Links to the projects are in our online appendix [31]. It is key to identify files for which alternative or new functionality has been added in a fork, so a realistic case for integration with variability can be made (as opposed to simple merging without variability, used for development branches).

Marlin (>40 KLOC of C++ code) is a 3d printer firmware as described above. *BusyBox* (>160 KLOC of C code) is a collection of common shell commands (e.g., grep, cut) for embedded Linux applications. *Vim* (>300 KLOC of C code) is a popular terminal-based text editor extending the Unix vi standard. *libuv* (>50 KLOC of C code) is a popular IO library. *PHP* (>1,000 KLOC of C code) is the interpreter for the PHP scripting language.

A. RQ1: Coverage of Intentions

Method. To show that the defined set of intentions is complete enough to handle real-world integration tasks, we replay non-trivial merges from the history of Marlin, libuv, and PHP. For Marlin, we retrieved all 2,065 merge commits of the mainline repositories. To identify complex merge tasks, we extracted those that had conflicts, yielding 49 merges. We discarded two merges that conflict only in documentation files, two conflicted in whitespace, and three conflicted in user configuration files. Another three merges were discarded because the pertinent artifacts had syntax errors and could not be compiled. Additionally, four merges were discarded because they simply accepted the mainline changes as evolution (empty changeset). In addition to the resulting 35 merges, we selected commits from two more projects, libuv and PHP, in the same way. From over 100 merge commit with conflicts, we sampled 10 commits from each project. We used the remaining 55 merge commits as tasks. We manually integrated these 55 cases, selecting intentions and performing the integration, so that the result was identical to the original historical merge.

Results. Using intentions for integration is successful in all 55 cases. In two of these cases additional code was added during the merge, which we needed to add in addition to the intentions as well. We observed neither obviously missing intentions nor cases requiring awkward sequences of intention applications.

Conclusion: The proposed set of intentions suffices for real-world variant integration.

B. RQ2: Correctness & Scalability of INCLINE

Method. We validated that INCLINE produces correct results when correct intentions are assigned, and that we can use it on large files without scalability problems. We simulated ten Marlin integrations in total. The first group of seven commits was selected randomly from the 35 merge commits of the previous experiment. The second group are three integration tasks selected from Marlin forks (see appendix [31])—we to avoid focusing on merge conflict commits only (in this group we merge parallel lines of development that have not been merged before). The selected forks contain significant changes to the mainline, covering both evolution and new features.

Most tasks comprised only a single file, some two or three, but each file can be very large (up to nearly 4,000 LOC and 40 `#ifdef` blocks to be handled during the task). Three authors served as evaluators, among whom we randomly distributed the ten tasks to execute with INCLINE. We then manually peer-reviewed the integration results to detect any discrepancies from the specifications of semantics in Sec. V-B.

Results. We only found minor errors in the implementation of INCLINE. These errors have been fixed, before proceeding to the next experiment. We also analyzed mistakes done by the

authors to improve the usability of INCLINE. Furthermore, the broad range of file sizes (from tens up to thousands of lines of code) witnesses the scalability of the tool. The intention resolution is instantaneous for smaller files, and for the largest file (∼4,000 LOC) it took about 2s on a regular office laptop.

Conclusion: INCLINE produces semantically correct output and it scales to files up to 4K lines of code without difficulties.

C. RQ3: Benefits of INCLINE

Method. We conducted a controlled experiment with 12 developers (experienced PhD students) using INCLINE and Eclipse to solve realistic tasks from Vim and BusyBox—those from the improvement cycle (cf. Sec. IV). In reality, a developer has some domain knowledge about the variants to be integrated. We captured this in the experiment setup by providing a detailed explanation of the purpose of the variants' parts and how they should be integrated (but unlike in the improvement cycle, we did not present the expected final result). We used a 2x2 within-subjects counterbalanced Latin square design:

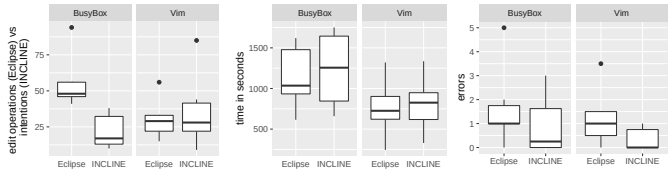
INCLINE/vim, Eclipse/busybox	INCLINE/busybox, Eclipse/vim
Eclipse/vim, INCLINE/busybox	Eclipse/busybox, INCLINE/vim

Each participant performed two tasks, using two treatments: Eclipse, and INCLINE, in a random order to reduce learning effects. Using a within-subjects design, we lower the number of participants, while every subject participates in each task. Furthermore, we mitigated learning effects by randomizing the order of the tasks (counterbalanced part of the design).

Participants were trained through a video tutorial on how to use both tools, as well as being instructed on preprocessor usage (they only needed to use `#ifdef`, `#else`, and `#endif`). Each participant first solved a training task extracted from Marlin, to get familiar with the tools. The training task performance is not included in the experiment results. During the actual experiment, we recorded the screen and log keystrokes (in Eclipse) and intentions (in INCLINE). We measured the performance of subjects using proxies: number of mistakes per task, the time to complete each task, and the number of edit operations (and number of intentions) applied per task.

We counted mistakes made by the subjects as follows. For Eclipse, a mistake can be a missing preprocessor annotation, missing code or extra code—this is because Eclipse merge tools are text-editing oriented. For INCLINE, we check for wrong intentions or no intentions applied by the participant that leads to errors in the resulting file. For both tools, errors concerning comments were counted as a half mistake, errors in formatting of code are ignored.

Results. The subjects using INCLINE made less mistakes than those using Eclipse (7 mistakes vs 17.5). Only four (33%) participants made integration errors working with INCLINE, compared to eleven (91%) with Eclipse. This is no surprise: INCLINE has better support for keeping or removing code without using clipboard, and the syntax of `#ifdef` structures created by INCLINE is correct-by-construction. The mistakes with INCLINE included missing relevant nodes in the declared intentions (3 mistakes), declared incorrect intentions (1), or declared different intentions for the same node with an unexpected result for the user (3). Common mistakes with Eclipse included lines guarded by incorrect presence conditions (8), leaving code that should be removed (4), removing too much (2) and a broken `#ifdef` structure (1). In four cases, subjects integrated code in wrong *order* which we account with half error each.



(a) Intentions/edit operations (b) Task completion times (c) Errors

Fig. 12: Effort, completion times, and errors

INCLINE required less decisions to be executed during integration (Fig. 12a). The BusyBox task involved 15 intentions on average, while almost 50 edit operations were used in Eclipse on average to achieve the same goal. This effect is not so pronounced for the Vim task, where the performance is similar (28 for INCLINE, 25.5 for Eclipse). The most commonly used intentions were *Keep* (48%), *KeepAsFeature* (21%) and *Remove* (18%). *Exclusive* was not used often (7 %), although ten (83%) users used *Exclusive* at least once. *AssignFeature* and *Order* were not used at all. *AssignFeature* is an escape intention, that allows to handle unanticipated changes, and *Order* is rather intricate–these two not being used speaks well for the expressiveness of the core intentions.

Subjects performing INCLINE integrations were almost as fast as the Eclipse users (Fig. 12b). It appears that the tasks are relatively small, so substantial influence comes from understanding the variants. Second, participants spend a lot of time (which we count in the result) reading back and forth through the descriptions to understand the integration goal. Third, some participants always verify the preview after applying an intention. This likely happens as the users are not very familiar with the tool and intentions, and thus either do not trust the tool or are not sure if they applied the right intention. The efficiency of INCLINE would likely improve greatly, should the tool be used professionally. For new users though, INCLINE shines already, as it facilitates learning and exploration with quick undo, and multiple views for manipulating the different variants and the integration result. One of the subjects mentions that *"It was really useful to declare all the intentions while still having the original files in sight and previewing the result."*. Note that the times reported in Fig. 12b *do not* include the cost of correcting the mistakes introduced by subjects. Since Eclipse integrations contained more than twice the number of mistakes, the actual cost of completing the integration with Eclipse is likely much higher.

Conclusion: INCLINE supports integration with lower error rate than Eclipse merge tools, with less edit operations, at no significant increase in cost (time).

We complemented the controlled experiment with a survey to gain qualitative insights on the benefits of INCLINE.

Method. The survey was designed as a mix of closed and open questions. The closed questions, using the Likert scale (1 strongly disagree, 5 strongly agree), target the intuitiveness of intentions, and gauging how participants feel about their mistakes and integration difficulties with INCLINE. The open questions aim at gathering concrete evidence of the advantages and challenges of using INCLINE, suggestions for improvements, and tool preference for integration tasks. We sent the 12 participants the survey via email after performing the integration tasks and received 10 answers. The questionnaire design and results are available in our online appendix [31].

Results. Participants strongly agree (mean 4.8, standard deviation 0.42) that the *Keep* and *Remove* intentions are intuitive. However, the *Exclusive* intention seems more confusing, because the user cannot directly select it. 40% of the subjects agree and 20% strongly agree that integration with INCLINE is faster than with Eclipse, but none of the participants disagrees (despite the actual time measurements saying the opposite, even if not very strongly, see Fig. 12b). One potential reason is that by not doing many clipboard operations or editing text, INCLINE appears faster through the usage of intentions. Similarly, 60% of the participants agree and 30% strongly agree that intention based integration is not complex, suggesting that there is potential for intention-based integrations. Finally, when asked what are the advantages of using intention-based over manual integration, some participants mentioned that *''Less effort and reduction in human error due to either typing or copying and pasting''* or *''Intentions are much more intuitive and user friendly. Saves you some typing and copy-pasting. The synchronized views were also very useful, because you can choose either the mainline, the combined view, or the fork to make updates, whatever is more convenient for the task at hand.''* There is a consensus that INCLINE is *''much more intuitive and less error-prone.''*

D. Threats to Validity

External Validity. We mitigated selection bias, as the main threat in our evaluation, by using multiple open source projects that have been actively developed and many variants have emerged. We used both mainline source files as well as forks to create realistic integration tasks. Results from these projects are also valid for other systems, since the C preprocessor is used similarly in open-source and industrial systems [32]. For the controlled experiment, we recruited experienced PhD students. Only basic program understanding was required, and we recapped the preprocessor use, to mitigate any potential difference in programming experience. Finally, recall that graduate students perform similarly to professional developers in software experiments like this one [33], [34].

Internal Validity. Simple bugs in the tool chain can hide or distract from evaluating the intention concept. We used an iterative design methodology, even conducting a pre-study as an experiment with 16 MSc students just to validate the prototype and to find usability issues to be fixed. The experiment participants using INCLINE have disadvantages compared to plain merge tools, mostly due to the lacking experience and the UI of a research prototype. It is very likely that this negatively biased the performance, so our results regarding the benefits of INCLINE can be seen as lower bounds. Still, we mitigated this threat by training users through a tutorial and a warmup task for INCLINE. We also randomly assigned the tasks to the participants, minimizing the learning effects.

VII. Related Work

Many works focus on re-engineering a *single system* into a product line [35]–[38], typically proposing refactoring techniques for creating configurable platforms. The main difference is that we focus on integrating *multiple system variants* into a product line, systematically guiding the process with intentions and views. Others provide support for *evolving existing product lines*. For instance, Liebig et al. [39] provide three refactorings (rename identifier, extract function, inline function) that preserve the variants. Instead, we support *obtaining product lines*. Our intentions are explicitly not variant-preserving.

A recent mapping study on re-engineering variants into product lines shows that the majority of papers on this topic focuses on detecting and analyzing commonalities and variabilities of the variant systems [8]. Only the following few support the actual variant integration. Rubin et al. [16] present a conceptual framework with seven operators to re-engineer cloned variants into product lines. The operators are abstract and some are related to our intentions, but none is implemented. We provide full tool support instead. Fischer et al. [40] propose a method to detect reusable features among variants, allowing to compose them to derive a new system. Martinez et al. present a framework for re-engineering a set of assets into a product line [41]. All these works lack support for handling variability using preprocessor directives as the most common technique for variation points.

Case studies of manual re-engineering also exist. Hetrick et al. re-engineer cloned variants into a product line, creating variation points, and switching to product line engineering [42]. Jepsen et al. [19] compute pairwise differences of two products, and wrap differences using `#ifdef` to create the initial integrated platform. The platform was iteratively refined, deciding to keep, remove or introduce a new feature [20].

Recall that variant integration is different from traditional merging. Still, we are inspired by techniques known from it. Our technique for creating the initial integrated platform works on ASTs and as such is related to structural merge [14], [43]–[45] and diff/merge for models beyond ASTs [30], [46].

VIII. Conclusion

We presented a method and a tool to integrate forked variants into a configurable integrated platform. The core idea is to offer a set of intuitive *integration intentions* resembling domain-specific actions to execute integration tasks. Instead of focusing on low-level `#if` directives, developers can express the integration goal using intentions declared on code blocks of the original variants, make edits to the code, and immediately observe the result. The experimental evaluation shows that

the method reduces the number of required editing operations and the number of integration errors against a baseline of a merge tool. We also showed that it can handle complex integration tasks and merges with conflicts. Declaring intentions was easy, and only rarely direct editing was required. Although understanding the integration goal is sometimes difficult, the different views help to explore and navigate the code. Applying intentions and revoking them is particularly useful to explore the result, before committing the changes.

ACKNOWLEDGMENTS

Supported by Vinnova Sweden (2016-02804) and the Swedish Research Council Vetenskapsrådet (257822902). We thank Klaus Schmid for comments on earlier versions of this paper.

REFERENCES

[1] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki, "An exploratory study of cloning in industrial software product lines," in *17th European Conference on Software Maintenance and Reengineering, CSMR 2013, Genova, Italy, March 5-8, 2013*, 2013, pp. 25–34. [Online]. Available: http://dx.doi.org/10.1109/CSMR.2013.13

[2] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wąsowski, "A survey of variability modeling in industrial practice," in *VaMoS*, 2013.

[3] J. Rubin, K. Czarnecki, and M. Chechik, "Managing cloned variants: a framework and experience," in *17th International Software Product Line Conference, SPLC 2013, Tokyo, Japan - August 26 - 30, 2013*, 2013, pp. 101–110. [Online]. Available: http://doi.acm.org/10.1145/2491627.2491644

[4] A. N. Duc, A. Mockus, R. L. Hackbarth, and J. D. Palframan, "Forking and coordination in multi-platform development: a case study," in *2014 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '14, Torino, Italy, September 18-19, 2014*, 2014, pp. 59:1–59:10. [Online]. Available: http://doi.acm.org/10.1145/2652524.2652546

[5] T. Berger, D. Nair, R. Rublack, J. M. Atlee, K. Czarnecki, and A. Wasowski, "Three cases of feature-based variability modeling in industry," in *Model-Driven Engineering Languages and Systems - 17th International Conference, MODELS 2014, Valencia, Spain, September 28 - October 3, 2014. Proceedings*, 2014, pp. 302–319. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-11653-2_19

[6] Ş. Stănciulescu, S. Schulze, and A. Wąsowski, "Forked and Integrated Variants in an Open-Source Firmware Project," in *31st International Conference on Software Maintenance and Evolution (ICSME)*, 2015.

[7] C. Kapser and M. W. Godfrey, ""cloning considered harmful" considered harmful: patterns of cloning in software," *Empirical Software Engineering*, vol. 13, no. 6, pp. 645–692, 2008.

[8] 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.

[9] S. McKee, N. Nelson, A. Sarma, and D. Dig, "Software Practitioner Perspectives on Merge Conflicts and Resolutions," in *Proceedings of the 33rd International Conference on Software Maintenance and Evolution*, ser. ICSME'17, 2017.

[10] M. L. Guimarães and A. R. Silva, "Improving early detection of software merge conflicts," in *ICSE*. Piscataway, NJ, USA: IEEE Press, 2012, pp. 342–352.

[11] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, "An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines," 2010.

[12] E. Walkingshaw and K. Ostermann, "Projectional editing of variational software," in *Generative Programming: Concepts and Experiences (GPCE)*, 2014. [Online]. Available: http://doi.acm.org/10.1145/2658761.2658766

[13] F. Medeiros, C. Kästner, M. Ribeiro, S. Nadi, and R. Gheyi, "The love/hate relationship with the c preprocessor: An interview study," in *ECOOP*, 2015.

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

[15] M. Antkiewicz, W. Ji, T. Berger, K. Czarnecki, T. Schmorleiz, R. Laemmel, S. Stanciulescu, A. Wasowski, and I. Schäfer, "Flexible product line engineering with a virtual platform," in *Proc. ICSE/NIER*, 2014.

[16] J. Rubin, K. Czarnecki, and M. Chechik, "Cloned product variants: from ad-hoc to managed software product lines," *STTT*, vol. 17, no. 5, pp. 627–646, 2015. [Online]. Available: http://dx.doi.org/10.1007/s10009-014-0347-9

[17] S. She, R. Lotufo, T. Berger, A. Wąsowski, and K. Czarnecki, "Reverse engineering feature models," in *ICSE'11*, 2011.

[18] A. von Rhein, A. Grebhahn, S. Apel, N. Siegmund, D. Beyer, and T. Berger, "Presence-condition simplification in highly configurable systems," in *ICSE*, 2015.

[19] H. P. Jepsen, J. G. Dall, and D. Beuche, "Minimally invasive migration to software product lines," in *Software Product Lines, 11th International Conference, SPLC 2007, Kyoto, Japan, September 10-14, 2007, Proceedings*, 2007, pp. 203–211. [Online]. Available: http://dx.doi.org/10.1109/SPLINE.2007.30

[20] H. P. Jepsen and D. Beuche, "Running a Software Product Line: Standing Still is Going Backwards," in *SPLC*, 2009.

[21] H. Spencer and C. Geoff, "#ifdef Considered Harmful, or Portability Experience With C News," in *USENIX Summer Technical Conference*, 1992, pp. 185–198.

[22] J. Melo, C. Brabrand, and A. Wąsowski, "How Does the Degree of Variability Affect Bug-Finding?" in *International Conference on Software Engineering (ICSE)*, 2016.

[23] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in Software Product Lines," 2008, pp. 311–320.

[24] B. Behringer, J. Palz, and T. Berger, "PEoPL: Projectional Editing of Software Product Lines," in *30th International Conference on Software Engineering (ICSE)*, 2017.

[25] "Jetbrains MPS," https://www.jetbrains.com/mps.

[26] M. Völter, J. Siegmund, T. Berger, and B. Kolb, "Towards User-Friendly Projectional Editors," in *SLE*, 2014.

[27] T. Berger, M. Völter, H. P. Jensen, T. Dangprasert, and J. Siegmund, "Efficiency of projectional editing: A controlled experiment," in *24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, 2016.

[28] L. D. Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proc. Int'l Conf. on Tools and algorithms for the construction and analysis of systems (TACAS/ETAPS)*. Berlin, Heidelberg: Springer, 2008, pp. 337–340.

[29] M. Voelter, D. Ratiu, B. Schaetz, and B. Kolb, "mbeddr: an extensible c-based programming language and ide for embedded systems," in *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*. ACM, 2012, pp. 121–140.

[30] A. Di Iorio, M. Schirinzi, F. Vitali, and C. Marchetti, "A natural and multi-layered approach to detect changes in tree-based textual documents," in *Proceedings of the 11th International Conference on Enterprise Information Systems*. Berlin, Heidelberg: Springer, 2009, pp. 90–101.

[31] "Online Appendix," https://sites.google.com/view/incline-online/.

[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*, 2015.

[33] M. Höst, B. Regnell, and C. Wohlin, "Using Students As Subjects—A Comparative Study of Students and Professionals in Lead-Time Impact Assessment," *Empirical Softw. Engg.*, vol. 5, no. 3, pp. 201–214, Nov. 2000.

[34] P. Runeson, "Using Students as Experiment Subjects—An Analysis on Graduate and Freshmen Student Data," in *Proc. EASE*, 2003.

[35] S. Schulze, T. Thüm, M. Kuhlemann, and G. Saake, "Variant-preserving refactoring in feature-oriented software product lines," in *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, ser. VaMoS '12, 2012.

[36] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi, "Refactoring a legacy component for reuse in a software product line: a case study," *Journal of Software Maintenance*, vol. 18, no. 2, pp. 109–132, 2006. [Online]. Available: http://dx.doi.org/10.1002/smr.329

[37] C. Kästner, S. Apel, and D. S. Batory, "A case study implementing features using aspectj," in *Software Product Lines, 11th International Conference, SPLC 2007, Kyoto, Japan, September 10-14, 2007,*

Proceedings, 2007, pp. 223–232. [Online]. Available: http://dx.doi.org/10.1109/SPLINE.2007.12

[38] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, *Aspect-oriented programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 220–242. [Online]. Available: http://dx.doi.org/10.1007/BFb0053381

[39] J. Liebig, A. Janker, F. Garbe, S. Apel, and C. Lengauer, "Morpheus: Variability-aware refactoring in the wild," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15, 2015.

[40] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed, "Enhancing clone-and-own with systematic reuse for developing software variants," in *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, 2014, pp. 391–400.

[41] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. L. Traon, "Bottom-up adoption of software product lines: a generic and extensible approach," in *Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015*, 2015, pp. 101–110. [Online]. Available: http://doi.acm.org/10.1145/2791060.2791086

[42] W. A. Hetrick, C. W. Krueger, and J. G. Moore, "Incremental return on incremental investment: Engenio's transition to software product line practice," in *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, 2006, pp. 798–804. [Online]. Available: http://doi.acm.org/10.1145/1176617.1176726

[43] 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*, 2011.

[44] S. Apel, O. Leßenich, and C. Lengauer, "Structured merge with auto-tuning: Balancing precision and performance," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2012. New York, NY, USA: ACM, 2012, pp. 120–129. [Online]. Available: http://doi.acm.org/10.1145/2351676.2351694

[45] O. Leßenich, S. Apel, and C. Lengauer, "Balancing precision and performance in structured merge," *Automated Software Engineering*, vol. 22, no. 3, pp. 367–397, 2015. [Online]. Available: http://dx.doi.org/10.1007/s10515-014-0151-5

[46] D. S. Kolovos, D. D. Ruscio, A. Pierantonio, and R. F. Paige, "Different models for model matching: An analysis of approaches to support model differencing," in *2009 ICSE Workshop on Comparison and Versioning of Software Models*, May 2009, pp. 1–6.