Projectional Editing of Variational Software

Eric Walkingshaw   Klaus Ostermann
University of Marburg, Germany
{walkiner,kos}@informatik.uni-marburg.de

Abstract

Editing the source code of variational software is complicated by the presence of variation annotations, such as `#ifdef` statements, and by code that is only included in some configurations. When editing some configurations and not others, it would be easier to edit a simplified version of the source code that includes only the configurations we currently care about. In this paper, we present a projectional editing model for variational software. Using our approach, a programmer can partially configure a variational program, edit this simplified view of the code, and then automatically update the original, fully variational source code. The model is based on an isolation principle where edits affect only the variants that are visible in the view. We show that this principle has several nice properties that are suggested by related work on bidirectional transformations.

Categories and Subject Descriptors D.2.3 [*Software Engineering*]: Coding Tools and Techniques—Program editors

General Terms Design, Languages, Theory

Keywords variation, projectional editing, software product lines, bidirectional transformations, view–update problem

1. Introduction

Editing variational software is complicated by the presence of variation annotations and code that is only conditionally included. As an extreme example, consider the code in Figure 1, which is from the BusyBox project.[1] This code is *variational*: it represents several different variants of a C program. Each variant can be obtained by running the C preprocessor with different configuration options enabled. For example, if the `SWAPON_DISCARD` option is enabled, but `SWAPON_PRI` is disabled, then the generated variant is shown below (note that adjacent string literals are concatenated in C).

```
ret = getopt32(argv, (applet_name[5] == 'n') ?
    "d::"
    "a" : "a"
    , &discard
    );
```

[1] http://git.busybox.net/busybox/tree/util-linux/swaponoff.c?id=faa9e94db619d1110061687278cde93a651e69de#n173

```
#if !ENABLE_FEATURE_SWAPON_DISCARD && \
        !ENABLE_FEATURE_SWAPON_PRI
    ret = getopt32(argv, "a");
#else
#if ENABLE_FEATURE_SWAPON_PRI
    if (applet_name[5] == 'n')
        opt_complementary = "p+";
#endif
    ret = getopt32(argv, (applet_name[5] == 'n') ?
#if ENABLE_FEATURE_SWAPON_DISCARD
        "d::"
#endif
#if ENABLE_FEATURE_SWAPON_PRI
        "p:"
#endif
        "a" : "a"
#if ENABLE_FEATURE_SWAPON_DISCARD
        , &discard
#endif
#if ENABLE_FEATURE_SWAPON_PRI
        , &prio
#endif
        );
#endif
```

Figure 1. Complex variational code from BusyBox.

Suppose we are only interested in editing this variant. Clearly, editing the variant directly is easier than editing the code in Figure 1. In the variational code, the `#if` annotations and irrelevant alternatives make the program more difficult to understand and make it difficult to determine which variants an edit will impact and how.

In this paper, we describe principles and methods to support *projectional editing* of variational programs. Projectional editing supports the following workflow: (1) partially configure the variational program to obtain a simpler version that contains only the variability that we currently care about; (2) edit the simpler variational program; (3) automatically apply the edit to the original fully variational source code. This workflow supports a *virtual separation of concerns*, as proposed by Kästner and Apel (2009). That is, it provides a way for programmers to work on some features of an annotation-based variational program, without having to consider many other irrelevant features.

The motivation for projectional editing and the challenges it entails are closely related to research on *lenses* in the functional programming community (Foster et al. 2007; Bohannon et al. 2008; Voigtländer 2009), the classic *view–update problem* from databases (Bancilhon and Spyratos 1981; Dayal and Bernstein 1982), and similar research in many other fields that can be collected under the label of *bidirectional transformations* (Czarnecki et al. 2009). Research in these areas strives to support the same basic workflow described above. Starting with a large data structure or database called the *source*: (1) obtain a simplified *view* of the source; (2) edit

GPCE'14, September 15–16, 2014, Västerås, Sweden.
Copyright © 2014 ACM 978-1-4503-3161-6/14/09…$15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

```
#if ENABLE_FEATURE_SWAPON_DISCARD
        "d::"
#if !ENABLE_FEATURE_SWAPON_PRI
        "d::"
#endif
#endif
#if ENABLE_FEATURE_SWAPON_PRI
        "p:"
#endif
        "a" : "a"
#if ENABLE_FEATURE_SWAPON_DISCARD
        , &discard
#if !ENABLE_FEATURE_SWAPON_PRI
        , &discard2
#endif
#endif
```

Figure 2. Relevant part of code from Figure 1 after a projectional edit. Added or changed code is restricted to the selected variants.

the view; (3) use the edited view to automatically update the source. Supporting this workflow requires the definition of two operations: *view* or *get* for obtaining the view in step (1), and *update* or *put* for propagating the edits in the view to the source in step (3).

The main challenge in all of this work is specifying and implementing the *update* operation. Specifically, one must determine how changes in the view should propagate back to changes in the source. Many different techniques and heuristics have been developed to address this problem, but one reason the problem is interesting is that there is no single best solution. However, when working within a specific domain, such as variational software, we can use knowledge of the domain to inform the specification of *update*.

To see why the intended behavior of *update* is not obvious, consider again our example from Figure 1. Suppose we generate the same view as before by enabling `SWAPON_DISCARD` and disabling `SWAPON_PRI`, then edit it to the following.

```
ret = getopt32(argv, "d::d::a", &discard, &discard2);
```

Observe that the new second argument replaces code that was spread across multiple `#if` blocks. How should this change be mapped back to the original source code? Also, should the new argument `&discard2` be added to the same `#if` block as `&discard`, or should it be added to the non-variational code after the corresponding `#endif`?

One contribution of this paper is a simple principle, framed in terms of variational software, for specifying the behavior of *update*: when updating the source program with an edited view, *only change the variants in the source that can be generated from the view*. In the example above, since the view contained only the variant where `SWAPON_DISCARD` is enabled and `SWAPON_PRI` is disabled, this is the only variant that should be affected by the update. Therefore, when pushed back into the source, code that was changed or added by edits to the view should be qualified by corresponding `#if` blocks. The relevant part of the updated source program is shown in Figure 2.

This principle leads to a kind of isolation of edits, not unlike a transactional context in databases—variants that are not in the view are never affected by edits. This isolation property is not desirable in all situations, for instance, when the developer does some code improvements within the view that should have an effect on a wider range of variants. However, we believe that there are some attractive scenarios that make our approach worthwhile:

- Consider a product line with customer-specific customizations. The producer of the product line may want to give a customer's programmers access to the source code. The edits of these programmers should never affect the other customers' variants. Furthermore, the customers should never see code that is specific to other customers. After careful consideration, the main developers of the product line may promote some of the customer-specific edits to a broader set of configurations. It is not obvious how to support this scenario with current technology, but it is directly supported by the projectional editing model we propose.
- Consider that a bug is known to occur in a particular set of variants, for instance, because of a variability-aware testing tool like Varex (Nguyen et al. 2014). In this case, all code that pertains to other variants is irrelevant to fixing the bug, and all edits to fix the bug should apply only to the selected variants in order to prevent breaking working code. The isolation of edits provided by our projectional editing model guarantees this.
- Consider the introduction of a new, cross-cutting feature to a product line, in which code must be added in many different places. Manually maintaining the required configuration context (for example, adding the same `#if` directive consistently in many places) is elaborate and error-prone. Using our projectional editing model, the programmer can generate a view corresponding to this configuration context, and all code specific to the feature will be qualified accordingly.

The edit isolation principle and its corresponding update model also have some formal advantages. For example, an update is never ambiguous. A classic challenge in the view–update problem is *alignment*: determining which parts of the edited view correspond to which parts of the source (Barbosa et al. 2010). We sidestep this challenge by simply introducing new variability everywhere that alignment issues would traditionally appear. The edit isolation principle also ensures the satisfaction of some desirable consistency laws taken from related research on lenses (Foster et al. 2007).

The main contribution of this paper is a *projectional editing model* for variational software. This model is introduced by way of several examples in Section 2, and formalized in terms of the *choice calculus* (Erwig and Walkingshaw 2011) in Section 3. Additionally, this paper makes the following complementary contributions:

- The *edit isolation* principle, introduced in Section 2.3 and formalized in Section 3.2, that describes how edits on a partially configured view of a variational program should be translated into edits on the source variational program.
- Reference implementations of the *view* function, in Section 3.2, and the *update* function, in Section 3.3, for choice calculus expressions that satisfy the edit isolation principle by construction.
- Formal evidence that edit isolation has the desirable properties described above. Namely, in Section 3.2, that it is unambiguous up to semantic equivalence, and, in Section 4, that it ensures the satisfaction of the lens consistency laws.
- An extension of the projectional editing model, in Section 5, to support *formula choices*, as used internally by the TypeChef tool (Kenner et al. 2010; Kästner et al. 2011), which can parse arbitrary `#if` variation in C. This illustrates how our approach can be applied to real variational software.

In Section 6, we motivate projectional editing from the perspective of supporting a virtual separation of concerns and compare our projectional editing model to related work on bidirectional transformations and task-focused editing. Finally, in Section 7 we conclude and offer directions for future work.

2. Projectional Editing Model

In this section, we describe our projectional editing model for variational software. We begin with an overview in Section 2.1, which describes the editing workflow, its basic operations, and the relationships between its various states. In Section 2.2 we present

$$\begin{tikzcd}
s \arrow[r, dotted, "\text{derived edit}"] \arrow[d, bend right, "view(p,s)"'] & s' \arrow[d, dashed, bend right, "view(p,s')"'] \\
v \arrow[u, dashed, bend right, "update(p,s,v)"'] \arrow[r, "\text{actual edit}"'] & v' \arrow[u, bend right, "update(p,s,v')"']
\end{tikzcd}$$

Figure 3. Summary of projectional editing relationships.

several tiny examples that illustrate some important aspects of the model. This presentation is continued in Section 2.3, where we describe the *edit isolation* principle that is the basis of our approach, and present a few more examples that illustrate its implications.

2.1 Overview of the Projectional Editing Model

Recall the projectional editing workflow described in Section 1: (1) the user partially configures a variational *source* program s to obtain a simplified *view* program v, (2) the user edits v to produce a new view program v', and (3) the original source program s is updated with v' to produce a new source program s'. To support this workflow, we need two functions: *view*, for implementing step (1), and *update*, for implementing step (3).

The diagram in Figure 3 summarizes the relationships between the programs s, v, v', and s', and how *view* and *update* move between them. The three steps in the workflow are captured by the three solid edges in the diagram. We use p to represent the partial configuration used to generate the view; we also call p a *projection* on s. Starting from s, the *view* function generates the view, $view(p,s) = v$; the user edits v into v'; and the *update* function generates the updated source, $update(p,s,v') = s'$. The *update* function effectively translates the user's actual edits from v to v' into derived edits from s to s', represented by the dotted edge in the diagram.

The two dashed edges in the diagram describe basic consistency principles that *view* and *update* should satisfy. These correspond to the "lens laws" from research on lenses (Foster et al. 2007). Per the dashed edge on the left, if we update s with the unedited view v, then we get back the original source s, unchanged. Per the dashed edge on the right, if we update s with v' and immediately obtain a new view using the same projection p, then we get back the same view v' that was just used in the update. Together, these properties ensure that *view* and *update* are mathematical inverses.

In most projectional editing scenarios, the correct behavior of *view* is obvious. For example, in the case of `#if` annotations, a projection p may consist of a set of explicitly enabled or disabled configuration options; then *view* corresponds to running a modified C preprocessor that resolves all `#if` blocks whose configuration options are mentioned in p, but leaves other `#if` blocks in the code. However, the correct behavior of *update* is not so obvious, and indeed specifying and implementing *update* is the overarching challenge in the classic *view–update* problem. Although the consistency properties described above constrain the specification of *update*, there remains a lot of open design space. In the next subsection, we motivate our own specification of *update* through several examples.

2.2 Projectional Editing, By Example

In this section, and throughout the rest of the paper, we use the *choice calculus* (Erwig and Walkingshaw 2011) as a concise notation for variational programs. The choice calculus is formally defined in Section 3.1. For now, it is sufficient to understand the correspondence between a choice calculus expression and its more verbose rendering with `#if` annotations, illustrated in Figure 4. A *choice* $D\langle e_1, e_2\rangle$ consists of a *dimension* of variation D and two alternatives e_1 and e_2. This corresponds to an `#if`–`#else`–`#endif`

```
#if A
  1
#else
  2
#endif
```

$A\langle 1,2\rangle$

```
#if A
  1
#else
  2
#endif
  +
#if A
  3
#else
  4
#endif
```

$A\langle 1,2\rangle + A\langle 3,4\rangle$

```
#if A
  #if B
    1
  #else
    2
  #endif
#else
  #if B
    3
  #else
    4
  #endif
#endif
```

$A\langle B\langle 1,2\rangle, B\langle 3,4\rangle\rangle$

Figure 4. Examples illustrating correspondence of choice calculus expressions and potentially nested `#if`–`#else`–`#endif` blocks.

block with D as the configuration option in the condition. A partial configuration of a choice calculus expression is given by a set of *selectors*. The selector $D.l$ means to replace each choice in dimension D with its left alternative, while the selector $D.r$ means to replace each choice in D with its right alternative. The examples in Figure 4 are used in the following discussion.

Let us begin by considering some straightforward projectional editing scenarios, where the correct behavior of *update* seems obvious. We present the scenarios by listing the values of s, p, v, and v'. The programs s, v, and v' are given as choice calculus expressions, while p is given by a set of selectors. Note that v is determined by the values of p and s since $v = view(p,s)$. The intended value of the updated source $s' = update(p,s,v')$ is the subject of discussion, since it will be determined by the behavior of *update*.

First, consider the scenario where we project away a single choice and edit the remaining alternative.

$$s = A\langle 1,2\rangle \quad p = \{A.l\} \quad v = 1 \quad v' = 3$$

That is, we use the first example in Figure 4 as the initial source and focus on its left alternative, which we edit from `1` to `3`. After applying *update*, we expect the corresponding alternative to update in the source, so that $s' = A\langle 3,2\rangle$.

If we have multiple choices in the same dimension, such as in the second example in Figure 4, they will always be synchronized, as would the corresponding `#if`–`#else`–`#endif` blocks. In the following scenario, we again select $A.l$ and edit the resulting view.

$$s = A\langle 1,2\rangle + A\langle 3,4\rangle \quad p = \{A.l\} \quad v = 1+3 \quad v' = 5+6$$

After editing the view `1 + 3` to `5 + 6`, we apply *update* and expect both of the corresponding alternatives to update in the source, so that $s' = A\langle 5,2\rangle + A\langle 6,4\rangle$.

Now consider a slightly more complicated case involving choices in multiple dimensions. In the following scenario, we start with the third example in Figure 4, which contains two dimensions of variation. We project away just one of these dimensions, leaving the other dimension still present in the view.

$$s = A\langle B\langle 1,2\rangle, B\langle 3,4\rangle\rangle \quad p = \{B.l\} \quad v = A\langle 1,3\rangle \quad v' = A\langle 5,6\rangle$$

Here, we have focused on the left alternatives of the choices in dimension B, leaving a simpler choice in A. After editing both of the alternatives in this choice, we once again expect the corresponding alternatives to update in the source, so that $s' = A\langle B\langle 5,2\rangle, B\langle 6,4\rangle\rangle$.

In all three of the previous scenarios, the intended result of *update* is clear since the edits made in the view are confined to subexpressions of the original source. Therefore, referring to Figure 3, there is a one-to-one correspondence between the actual edits from v to v' and the derived edits from s to s'. When this subexpression relationship is not preserved, the intended result of *update* is less obvious. For example, consider the following variant of the previous scenario, where instead of editing each alternative, we instead replace the whole choice in dimension A by 5.

$$s = A\langle B\langle 1,2\rangle, B\langle 3,4\rangle\rangle \quad p = \{B.l\} \quad v = A\langle 1,3\rangle \quad v' = 5$$

In this case, the edits are not confined to a subexpression of s since the choice $A\langle 1,3\rangle$, which was replaced in the view, does not appear as a subexpression in s. So what should *update* do?

One idea is that we can transform s in a semantics-preserving way to recover the subexpression relationship between edits. In our previous work, we enumerated a set of equivalence laws such that if two choice calculus expressions are equivalent, then they will always produce the same variants if they are configured in the same way (Erwig and Walkingshaw 2011).[2] From these laws, we can derive the following equivalence rule, which is useful for this example.

$$D\langle D'\langle e_1,e_2\rangle, D'\langle e_3,e_4\rangle\rangle \equiv D'\langle D\langle e_1,e_3\rangle, D\langle e_2,e_4\rangle\rangle$$

It is easy to verify, by transforming the left- and right-hand sides into the corresponding `#if`–`#else`–`#endif` notation, that applying this rule in either direction does not affect the meaning of an expression. Instantiating this rule with s yields the following.

$$A\langle B\langle 1,2\rangle, B\langle 3,4\rangle\rangle \equiv B\langle A\langle 1,3\rangle, A\langle 2,4\rangle\rangle$$

Now $A\langle 1,3\rangle$ appears as a subexpression on the right-hand side and the intended result of *update* is clear. We replace the choice with the edited view 5 in order to obtain $s' = B\langle 5, A\langle 2,4\rangle\rangle$. If we want the nesting of choices in s' to match the original nesting in s, we can apply another equivalence law $e \equiv D\langle e,e\rangle$ to split 5 into a choice $A\langle 5,5\rangle$, then apply the law above to obtain $s' = A\langle B\langle 5,2\rangle, B\langle 5,4\rangle\rangle$.

Unfortunately, not all edit ambiguities can be resolved by applying equivalence laws until the edits correspond to a subexpression of the source. Consider the following example, which illustrates the problem of ambiguous edit *alignment* (Barbosa et al. 2010).

$$s = A\langle 1,2\rangle \quad p = \{A.l\} \quad v = 1 \quad v' = 1 + 3$$

Here, the edit extended the view by adding + 3, but it is unclear whether this extension should be propagated back to the alternative containing 1, in which case the updated source is $s'_1 = A\langle 1 + 3, 2\rangle$, or whether it should be shared between both alternatives, in which case the updated source is $s'_2 = A\langle 1,2\rangle + 3$. Determining edit alignment is one of the main challenges of the view–update problem. Our strategy is to avoid this ambiguity by a simple principle that completely specifies the behavior of *update*. This principle, which is discussed in the next subsection, leads us to choose s'_1 for this example.

2.3 Edit Isolation Principle

Our projectional editing model is founded on a principle that states that when applying an update, *the only variants that change in the source are those that can be reached from the view*. That is, edits to the view are *isolated* from variants in the source that were hidden when the view was generated. Although edit isolation leads to just one of many possible specifications of *update*, it has some nice properties. First, it avoids the problem of alignment by specifying an unambiguous behavior for *update*. Second, in Section 4, we show that edit isolation ensures that several consistency properties on *view* and *update* are satisfied, including those discussed in Section 2.1. We defer the formal definition of the edit isolation principle to Section 3.2. In this section we illustrate its implications by example.

A potentially surprising implication of edit isolation is that an update may produce new choices in s' that are present in neither the original source s nor in the edited view v'. For example, suppose the user projects on the left alternative of dimension A in an expression that does not contain a choice in A, then makes an edit.

[2] See the denotational semantics of the choice calculus in Section 3.1.

```
  struct globals {
  ...
  #if ENABLE_FEATURE_WGET_TIMEOUT
    unsigned timeout_seconds;
+   bool connecting;
  #endif
  ...
  }
  ...
+ #if ENABLE_FEATURE_WGET_TIMEOUT
+ static void alarm_handler(int sig UNUSED_PARAM) {
+   /* This is theoretically unsafe (uses stdio
+      and malloc in signal handler) */
+   if (G.connecting)
+     bb_error_msg_and_die("download␣timed␣out");
+ }
+ #endif
```

Figure 5. Part of a BusyBox edit illustrating consistency.

$$s = 1 + 2 \quad p = \{A.l\} \quad v = 1 + 2 \quad v' = 1 + 3$$

By edit isolation, *update* should yield the new source $s' = 1 + A\langle 2,3\rangle$. It would be incorrect to simply replace 2 with 3 since then we would have changed the variants for configurations containing $A.r$, even though those variants are not reachable from the view.

Similarly, an update may eliminate an existing choice in s if an edit makes its alternatives the same. Consider the following scenario.

$$s = B\langle 1,2\rangle + 3 \quad p = \{B.l\} \quad v = 1 + 3 \quad v' = 2 + 3$$

The edit changes the left alternative of the choice in B into 2, which is the same as the right alternative that we projected away when generating the view. So, *update* should yield a new source $s' = 2 + 3$, in which the choice in B has been removed.

Note that although an update may introduce or eliminate choices, it does not modify the set of dimensions that choices may refer to. Therefore, an update will not fundamentally alter the variability in a program, although it may distinguish variants that were previously identical or merge variants that were previously different.

As a more realistic example, Figure 5 shows part of an actual edit to the BusyBox project that satisfies the edit isolation principle.[3] In the figure, lines that start with a + symbol indicate newly added code and the ellipses indicate omitted unchanged code. Observe that the edit contains changes both inside and outside existing `#if` blocks, but all of the newly added code is conditionally included only if the `WGET_TIMEOUT` feature is enabled. The full edit contains several more lines of conditional code, but also one small refactoring that is not variational. This change does not affect the semantics of the program when `WGET_TIMEOUT` is disabled, but such a change could be risky since the programmer must reason about both alternatives at once, even though he intended to change only the variants where `WGET_TIMEOUT` is enabled. Using our projectional editing model, the programmer could have partially configured the program with the `WGET_TIMEOUT` feature enabled, made his edits to a program without any variation annotations, and been guaranteed that his edits would not affect variants where `WGET_TIMEOUT` is disabled.

The edit isolation principle ensures that edits are isolated to a deterministic and easily identifiable set of variants—those variants that the user can see while they are editing. In addition to the example in Figure 5, we described several more general scenarios in Section 1 where we believe that this behavior is desirable. Of course, other scenarios may require other specifications of *update*, and this is a potential area for future work.

[3] The complete edit is here: http://git.busybox.net/busybox/commit?id=d074b416f8d3ca6b6ae7b44d17e204ea8d81e7a0

Plain programs:

$$t \quad ::= \quad a\prec t,\ldots,t\succ \quad \textit{Program AST}$$

Choice calculus syntax:

$$\begin{array}{rcll} e,s,v & ::= & a\prec e,\ldots,e\succ & \textit{Program AST} \\ & | & D\langle e,e\rangle & \textit{Choice} \end{array}$$

Selection: $e \times x \to e$

$$\lfloor a\prec e_1,\ldots,e_n\succ\rfloor_x = a\prec\lfloor e_1\rfloor_x,\ldots,\lfloor e_n\rfloor_x\succ$$

$$\lfloor D\langle e_l,e_r\rangle\rfloor_x = \begin{cases} \lfloor e_l\rfloor_x & \text{if } x = D.l \\ \lfloor e_r\rfloor_x & \text{if } x = D.r \\ D\langle\lfloor e_l\rfloor_x,\lfloor e_r\rfloor_x\rangle & \text{otherwise} \end{cases}$$

Partial configuration: $e \times p \to e$

$$\lfloor e\rfloor_p = \lfloor\ldots\lfloor e\rfloor_{x_1}\ldots\rfloor_{x_n} \quad \text{where } p = \{x_1,\ldots,x_n\}$$

(Complete) configuration: $e \times c \to t$

$$\lfloor a\prec e_1,\ldots,e_n\succ\rfloor_c = a\prec\lfloor e_1\rfloor_c,\ldots,\lfloor e_n\rfloor_c\succ$$

$$\lfloor D\langle e_l,e_r\rangle\rfloor_c = \begin{cases} \lfloor e_l\rfloor_c & \text{if } c(D) = l \\ \lfloor e_r\rfloor_c & \text{otherwise } (c(D) = r) \end{cases}$$

Denotational semantics: $e \to c \to t$

$$[\![e]\!] = \lambda c.\lfloor e\rfloor_c$$

Semantic equivalence:

$$e \equiv e' \overset{\textit{def}}{\Longleftrightarrow} [\![e]\!] = [\![e']\!] \iff \forall c \in \mathscr{C}.\,\lfloor e\rfloor_c = \lfloor e'\rfloor_c$$

Figure 6. Choice calculus language definition.

3. Formalization of Projectional Editing

In this section we formalize the projectional editing model in terms of the choice calculus. As necessary background, in Section 3.1, we briefly define the syntax and semantics of the choice calculus. In Section 3.2, we give the types of the *view* and *update* operations, identify an implementation of *view*, and formally define the edit isolation principle, which specifies the behavior of *update*. An implementation of *update* is presented in Section 3.3.

3.1 Background: Choice Calculus

The choice calculus is a minimalistic formal language for representing variational programs. Its syntax and semantics are given in Figure 6, along with operations for partially configuring choice calculus expressions. These definitions are based on our previous work (Erwig and Walkingshaw 2011).

From the perspective of the choice calculus, a program is represented by a generic abstract syntax tree (AST), encoded as a rose tree. A node $a\prec t_1,\ldots,t_n\succ$ contains a label a and a potentially empty list of children $t_1,\ldots,t_n$. For example, the expression `3 + 4` would be encoded in the choice calculus as $+\prec 3\prec\succ, 4\prec\succ\succ$. However, we prefer concrete syntax wherever there is no ambiguity. A choice calculus expression is simply an AST that may contain choices embedded arbitrarily within it. We use the metavariable t to range over plain programs, containing no variation, and the metavariables e, s, and v to range over choice calculus expressions.

Eliminating all choices in one dimension is called *selection*. We write $\lfloor e\rfloor_{D.l}$ to select within e the left alternative of every choice in dimension D, and $\lfloor e\rfloor_{D.r}$ to select the right alternative of every choice in D. For example, let $e = A\langle 3,4\rangle + A\langle 5,6\rangle$, then selecting $\lfloor e\rfloor_{A.l}$ yields `3 + 5`, while selecting $\lfloor e\rfloor_{A.r}$ yields `4 + 6`. Since the two choices are synchronized, it is impossible to select `3 + 6` or `4 + 5`. We range over the *selectors* $D.l$ and $D.r$ with the metavariable x.

Observe from the definition of selection that outer choices *dominate* inner choices in the same dimension. For example, given a choice $e = B\langle 3, B\langle 4,5\rangle\rangle$, it is not possible to select `4` from e; selecting $\lfloor e\rfloor_{B.l}$ yields `3`, while selecting $\lfloor e\rfloor_{B.r}$ replaces both choices by their right alternative, yielding `5`. Dominated choices can always be replaced by the relevant alternative without changing the meaning of the expression. In this case, we can simplify e to $B\langle 3,5\rangle$.

A partial configuration p is a set of selectors such that if $D.l \in p$ then $D.r \notin p$ and vice versa. A partial configuration defines a projection on choice calculus expressions by eliminating some dimensions of variation, but not necessarily all. We write $\lfloor e\rfloor_p$ to select $\lfloor e\rfloor_x$ for every $x \in p$. Since each selector in p eliminates a different dimension of variation, and since selectors do not affect choices in other dimensions, the order of selection does not matter.

A (complete) configuration $c : D \to (l + r)$ is a total function that maps each dimension name to either l or r. Conceptually, a configuration describes how to select *every* dimension that may appear in e. We write $\lfloor e\rfloor_c$ to configure expression e with configuration c. The definition of configuration is similar to selection. Since configuring will eliminate all choices from e, the result will consist only of AST nodes and so will be a *plain* program t. We define $\mathscr{C}$ to be the set of all possible configurations.

The denotational semantics of an expression, written $[\![e]\!]$, is a function from a complete configuration c to the plain variant produced by configuring e with c. Finally, two expressions e and e' are considered semantically equivalent, written $e \equiv e'$, if they have the same denotational semantics.

3.2 Specification of the Projectional Editing Operations

In Section 2 we presented a projectional editing model that relies on two functions, *view* and *update*. In this section, we formally specify the behavior of these functions. In terms of the choice calculus, we can assign *view* and *update* the following types.

$$\begin{array}{r l} \textit{view} : p \times e & \to e \\ \textit{update} : p \times e \times e & \to e \end{array}$$

The specification of *view*, as described in Section 2.2, is trivial in terms of the partial configuration operation defined in Figure 6.

Definition 3.1 (Projection)**.** *Given source program s and partial configuration p. Let $v = view(p,s)$. Then, $v = \lfloor s\rfloor_p$.*

We can implement this specification directly as shown below.

$$\textit{view}(p,s) = \lfloor s\rfloor_p$$

The specification and implementation of *update* are somewhat more involved. As described in Section 2.3, the behavior of *update* is specified by the edit isolation principle, which is defined below, and one possible implementation is offered in the next subsection.

The edit isolation principle states that when applying an update, the only variants that change in the source are those that can be reached from the view. To define this principle in terms of the semantics of the choice calculus, we need a way to relate the partial configuration p used to generate the view, and a total configuration c in the domain of the semantics. Recall that p is a finite set of non-conflicting selectors and c is a total function from dimension names to either l or r. Equivalently, we can view c as an infinite set of selectors, where for every dimension D, either $D.l$ or $D.r$ is a member of c, but not both. Therefore, we can say that p is a subset of c, that is $p \subset c$, if c contains all of the selectors in p.

In the following definition, we restrict v' from containing variation in dimensions listed in p to prevent edits from reintroducing variants that were explicitly projected away.

Definition 3.2 (Edit Isolation)**.** *Given source program s, partial configuration p, and edited view $v' = \lfloor v' \rfloor_p$. Let $s' = update(p,s,v')$. Then,*

$$\forall c \in \mathscr{C}.\ \lfloor s' \rfloor_c = \begin{cases} \lfloor v' \rfloor_c & \textit{if } p \subset c \\ \lfloor s \rfloor_c & \textit{otherwise} \end{cases}$$

Since the original view was generated by $\lfloor s \rfloor_p$ it follows that if $p \subset c$, then c describes a variant that can be generated from the view, otherwise c describes a variant that cannot be generated from the view. Since the edit should affect exactly those variants that can be reached from the view, the semantics of the updated source s' should draw from the semantics of the edited view v' whenever $p \subset c$, and it should draw from the semantics of the original source s otherwise.

By comparing this definition with the definition of $[\![\cdot]\!]$ in Figure 6, it should be clear that edit isolation completely specifies the denotational semantics of the updated source s' in terms of the semantics of the edited view v' and the original source s.

3.3 Implementation of Update

The edit isolation principle specifies the behavior of *update* in terms of the denotational semantics of its output. In this subsection we construct an implementation of this specification.

The implementation of $update(p,s,v')$ consists of two steps: First, we build up a choice calculus expression using a helper function *diff* that ensures by construction that the edit isolation principle is satisfied. Second, we minimize this expression by applying a function *minimize*, that transforms the expression into a semantically equivalent expression with minimal redundancy.

$$update(p,s,v') = minimize(diff(p,s,v'))$$

In the rest of this section we describe the implementation of the two helper functions *diff* and *minimize*.

The *diff* function uses the selectors contained in p to build up a tree of choices, where every leaf of the tree is s except for the leaf identified by p, which is v'.

$$diff(p,s,v') = \begin{cases} D\langle diff(p',s,v'),s\rangle & \text{if } p = \{D.l\} \cup p' \\ D\langle s, diff(p',s,v')\rangle & \text{if } p = \{D.r\} \cup p' \\ v' & \text{otherwise } (p = \varnothing) \end{cases}$$

For example, given $p = \{A.r, B.r, C.l\}$, the *diff* function will construct the following choice from p, s, and v'.

$$A\langle s, B\langle s, C\langle v', s\rangle\rangle\rangle$$

The *diff* function ensures that the edit isolation principle is satisfied by building up a tree of choices that implements the principle directly. For technical reasons, we assume that *diff* draws selectors from p according to a fixed dimension ordering (e.g. lexicographic). Note that s may itself contain choices in dimensions A, B, and C. These will be eliminated as part of the next step.

The *minimize* function reduces redundancy in the expression produced by *diff*. It is implemented indirectly by the $\rightsquigarrow$ rewriting relation, defined in Figure 7. These rules are derived from the equivalence laws defined in our previous work (Erwig and Walkingshaw 2011). This means that applying a rule may change the syntactic structure of an expression, but it will not change its semantics.

The AST-FACTORING rule supports factoring an AST node out of a choice if the label of the AST node is shared amongst all alternatives. For example, consider the choice $A\langle 2+3, 2+4\rangle$. The root of the AST in each alternative is the + symbol, so we can factor this out to get $A\langle 2,2\rangle + A\langle 3,4\rangle$. Or, to make the tree structure of the addition expressions more explicit:

$$A\langle +\prec 2,3\succ, +\prec 2,4\succ\rangle \rightsquigarrow +\prec A\langle 2,2\rangle, A\langle 3,4\rangle\succ$$

AST-FACTORING

$$D\langle a\prec e_1,\ldots,e_n\succ, a\prec e'_1,\ldots,e'_n\succ\rangle \rightsquigarrow a\prec D\langle e_1,e'_1\rangle,\ldots,D\langle e_n,e'_n\rangle\succ$$

CHOICE-IDEMPOTENCY

$$D\langle e,e\rangle \rightsquigarrow e$$

CHOICE-DOMINATION

$$\frac{\lfloor e_l\rfloor_{D.l} = e'_l \qquad \lfloor e_r\rfloor_{D.r} = e'_r}{D\langle e_l,e_r\rangle \rightsquigarrow D\langle e'_l,e'_r\rangle}$$

AST-CONGRUENCE

$$\frac{e_i \rightsquigarrow e'_i}{a\prec e_1,\ldots,e_n\succ \rightsquigarrow a\prec e_1,\ldots,e'_i,\ldots,e_n\succ}$$

CHOICE-L-CONGRUENCE

$$\frac{e_l \rightsquigarrow e'_l}{D\langle e_l,e_r\rangle \rightsquigarrow D\langle e'_l,e_r\rangle}$$

CHOICE-R-CONGRUENCE

$$\frac{e_r \rightsquigarrow e'_r}{D\langle e_l,e_r\rangle \rightsquigarrow D\langle e_l,e'_r\rangle}$$

Figure 7. Choice calculus minimization rules.

Now the left alternative of the factored expression contains a choice $A\langle 2,2\rangle$, where it doesn't matter which alternative we pick. We can eliminate such choices by applying the CHOICE-IDEMPOTENCY rule, for example, $A\langle 2,2\rangle \rightsquigarrow 2$. So the minimized expression is $2 + A\langle 3,4\rangle$. The CHOICE-DOMINATION rule eliminates dominated choices, as described in Section 3.1.

The remaining three congruence rules allow the first three rules to be applied to any subexpression of a choice calculus expression. For example, the AST-CONGRUENCE rule is needed in order to apply the CHOICE-IDEMPOTENCY rule to the expression $A\langle 2,2\rangle$ in the example above since this choice is a subexpression of an AST node.

Finally, we define $\rightsquigarrow^*$ to be the reflexive, transitive closure of the rewriting relation $\rightsquigarrow$, and define that $minimize(e) = e'$ if and only if $e \rightsquigarrow^* e'$. That is, *minimize* effectively applies the minimization rules repeatedly until the expression cannot be minimized anymore. Assuming a fixed dimension ordering, the $\rightsquigarrow^*$ rewriting relation is *normalizing*, meaning that every choice calculus expression can be translated into a unique minimal form. This property can be proved by induction over the minimization rules, showing that the rewriting relation is terminating and confluent. In our previous work, we have shown that a structurally equivalent rewriting system for variational types is normalizing (Chen et al. 2014).

4. Properties of Edit Isolation

In this section, we prove that our projectional editing model has some desirable properties identified in previous work on bidirectional transformations. The first of these are the so-called "lens laws", which define some basic consistency principles that *view* and *update* should satisfy (Foster et al. 2007).

$$update(p,s,view(p,s)) \equiv s \qquad (\text{GETPUT})$$
$$view(p,update(p,s,v')) \equiv v' \qquad (\text{PUTGET})$$

The GETPUT law states that updating s with an unedited view (that is, $v = v'$), yields the original source s. The PUTGET law states that updating s with a (potentially edited) view v', then immediately producing a new view with the same projection p, yields the same v'. Together, the lens laws describe the important property that *view* and *update* should be mathematical inverses. Recall that these properties correspond to the dashed lines in the diagram in Figure 3.

Any implementations of *view* and *update* that satisfy the specifications defined in Section 3.2 also satisfy the GETPUT and PUTGET laws. Recall that *view* and *update* are specified by Definition 3.1 (projection) and Definition 3.2 (edit isolation), respectively. To prove this, we first introduce a couple of lemmas describing relationships between partial and complete configurations.

Lemma 4.1 states that if $p \subset c$, then c *completes* the configuration of p, so configuring with p followed by c is the same as just configuring with c.

Lemma 4.1 (Completion)**.** *Given partial configuration p and complete configuration c. Then,* $p \subset c \implies \lfloor \lfloor e \rfloor_p \rfloor_c = \lfloor e \rfloor_c$.

This follows from the isomorphic definitions of $\lfloor \cdot \rfloor_p$ and $\lfloor \cdot \rfloor_c$, and the definition of the $\subset$ relationship in Section 3.2.

Lemma 4.2 states that a partial configuration p has *priority* over a complete configuration c if we apply p first. Therefore, we can replace p and c by a new complete configuration c' that draws from p if applicable, and otherwise defaults to c.

Lemma 4.2 (Priority)**.** *Given partial configuration p and complete configuration c. There exists a complete configuration c', such that $p \subset c'$, $\lfloor \lfloor e \rfloor_p \rfloor_c = \lfloor e \rfloor_{c'}$, and c and c' differ only in dimensions in p.*

Proof. Construct c' as follows.

$$c' = \lambda D. \begin{cases} l & \text{if } D.l \in p \\ r & \text{if } D.r \in p \\ c(D) & \text{otherwise} \end{cases}$$ □

Using these lemmas, we can prove that the GETPUT and PUTGET laws follow from the definitions in Section 3.2.

Theorem 4.3. GETPUT *follows from projection and edit isolation.*

Proof. Assume that *view* and *update* satisfy the projection and edit isolation principles, respectively. By expanding the definitions of *view* and equivalence, the GETPUT law can be expressed as follows.

$$\forall c \in \mathscr{C}.\, \lfloor update(p, s, \lfloor s \rfloor_p) \rfloor_c = \lfloor s \rfloor_c$$

Now we can expand *update* using the definition of edit isolation. There are two cases to consider.

1. If $p \subset c$, then $\lfloor \lfloor s \rfloor_p \rfloor_c = \lfloor s \rfloor_c$, which follows from Lemma 4.1.
2. Otherwise, $\lfloor s \rfloor_c = \lfloor s \rfloor_c$, which is trivially true. □

Theorem 4.4. PUTGET *follows from projection and edit isolation.*

Proof. Assume that *view* and *update* satisfy the projection and edit isolation principles, respectively. By expanding the definitions of *view* and equivalence, the PUTGET law can be expressed as follows.

$$\forall c \in \mathscr{C}.\, \lfloor \lfloor update(p, s, v') \rfloor_p \rfloor_c = \lfloor v' \rfloor_c$$

By Lemma 4.2, the inner configuration with p ensures that, regardless of c, *update* will expand to the first case given in Definition 3.2, yielding the following.

$$\forall c \in \mathscr{C}.\, \lfloor v' \rfloor_{c'} = \lfloor v' \rfloor_c$$

Also by Lemma 4.2, c and c' may differ only in the dimensions contained in p, and by edit isolation, v' may not contain choices in those dimensions. Therefore, $\lfloor v' \rfloor_{c'}$ and $\lfloor v' \rfloor_c$ are equivalent. □

It is assumed that all "well-behaved" bidirectional transformations should satisfy the basic GETPUT and PUTGET laws. Therefore, our projectional editing model is well-behaved.

A bidirectional transformation is considered "very well-behaved", if it satisfies two additional properties: (1) *view* and *update* are total functions and (2) *update* satisfies the following PUTPUT law (Foster et al. 2007; Johnson and Rosebrugh 2008).

$$update(p, update(p, s, v'), v) \equiv update(p, s, v) \qquad \text{(PUTPUT)}$$

Totality ensures that *view* and *update* will never fail. The PUTPUT law states that if we update a source program s with v' under projection p, then update it again with v under the same projection p, the result should be the same as if we had just updated s with v. In other words, the second update with v completely overwrites the first update with v'. This is desirable since it means that the effects of an update are always *reversible* (Johnson and Rosebrugh 2008).

Theorem 4.5. PUTPUT *follows from projection and edit isolation.*

Proof. Assume that *update* satisfies the edit isolation principle. By the definition of equivalence, the PUTPUT law can be expressed as follows.

$$\forall c \in \mathscr{C}.\, \lfloor update(p, update(p, s, v'), v) \rfloor_c = \lfloor update(p, s, v) \rfloor_c$$

Now we can expand the two outermost uses of *update* using the definition of edit isolation. There are two cases to consider.

1. If $p \subset c$, then $\lfloor v \rfloor_c = \lfloor v \rfloor_c$.
2. Otherwise, $\lfloor update(p, s, v') \rfloor_c = \lfloor s \rfloor_c$, and expanding the remaining use of *update* yields $\lfloor s \rfloor_c = \lfloor s \rfloor_c$. □

Therefore any implementations of *view* and *update* that are total functions and satisfy the projection and edit isolation principles yield a bidirectional transformation on choice calculus expressions that is very well-behaved. Observe that our reference implementations of *view* and *update* are both total functions, so the projectional editing model described here has this property.

5. Extension to Formula Choices

So far, we have presented our projectional editing model in terms of the choice calculus. In Section 2.2 we showed how the choice calculus is equivalent to a restricted use of `#if`–`#else`–`#endif` blocks where the condition of each `#if` is a single configuration option. A drawback of this representation is that expressing some variational programs requires redundancy. For example, suppose a program is `2` if either configuration options A or B are enabled, but `3` otherwise. In the choice calculus, we must represent this as $A\langle 2, B\langle 2, 3\rangle\rangle$, where `2` is repeated. If `2` were instead a large chunk of code, this redundancy would be very bad from a maintenance perspective.

As a solution, we can replace atomic dimension names by boolean formulas of configuration options, as described in our previous work (Walkingshaw et al. 2014). We call the resulting language the *formula choice calculus*. We can represent the example above by the formula choice $(A \vee B)\langle 2, 3\rangle$, which contains no redundancy. If the formula is satisfied by the chosen configuration, the first alternative `2` is chosen, otherwise the second alternative `3` is chosen. Using the formula choice calculus, we can encode `#if`–`#else`–`#endif` blocks with arbitrary boolean conditions.

The syntax and semantics of the formula choice calculus are defined in Figure 8. The syntax just replaces the dimensions D by boolean formulas F. The metavariable O ranges over atomic configuration options, such as A and B in the above example.

The denotational semantics $[\![\cdot]\!]$ is defined as a function from a complete configuration c to the plain program produced by $\lfloor e \rfloor_c$. A complete configuration is a total function from configuration options to boolean values, indicating whether each option is enabled or disabled. The helper function *eval* evaluates the truth value of a formula given a configuration, and $\lfloor e \rfloor_c$ configures an expression by evaluating the formula of each choice and replacing it by its left or right alternative, depending on whether the formula is true or false.

To support projectional editing of formula choice calculus expressions, we need some way to partially configure them. We could adapt the approach used for the choice calculus by representing partial configurations as a set of pairs of type $O \times B$, indicating configuration options that are enabled or disabled. However, a more flexible approach is to represent partial configurations by formulas, as defined in Figure 8; that is, $\lfloor e \rfloor_F$ yields an expression that contains all of the variants of e that are *consistent* with F. For example, given an expression $e = A\langle 1, B\langle 2, 3\rangle\rangle$ and a formula $F = A \vee B$, then $\lfloor e \rfloor_F = A\langle 1, 2\rangle$. The variant `3` is inconsistent with F and eliminated since selecting it would require both A and B to be false. Consistency can be efficiently checked using a SAT solver.

Formula choice calculus syntax:

$$\begin{array}{rcll} B & ::= & \mathit{true} \mid \mathit{false} & \\ F & ::= & B \mid O \mid \neg F \mid F \vee F \mid F \wedge F & \\ e,s,v & ::= & a\prec e,\ldots,e\succ & \mathit{Program\ AST} \\ & \mid & F\langle e,e\rangle & \mathit{Choice} \end{array}$$

(Complete) configuration: $e \times c \to t$

$$c : O \to B \qquad \mathit{eval} : c \times F \to B$$

$$\lfloor a\prec e_1,\ldots,e_n\succ \rfloor_c = a\prec \lfloor e_1\rfloor_c,\ldots,\lfloor e_n\rfloor_c \succ$$

$$\lfloor F\langle e_l,e_r\rangle\rfloor_c = \begin{cases} \lfloor e_l\rfloor_c & \text{if } \mathit{eval}(c,F) = \mathit{true} \\ \lfloor e_r\rfloor_c & \text{otherwise } (\mathit{false}) \end{cases}$$

Denotational semantics: $e \to c \to t$

$$[\![e]\!] = \lambda c.\lfloor e\rfloor_c$$

Partial configuration: $e \times F \to e$

$$\lfloor a\prec e_1,\ldots,e_n\succ \rfloor_F = a\prec \lfloor e_1\rfloor_F,\ldots,\lfloor e_n\rfloor_F \succ$$

$$\lfloor F'\langle e_l,e_r\rangle\rfloor_F = \begin{cases} \lfloor e_l\rfloor_{F_l} & \text{if } \neg \mathit{SAT}(F_r) \\ \lfloor e_r\rfloor_{F_r} & \text{if } \neg \mathit{SAT}(F_l) \\ \mathit{min}(F_l)\langle \lfloor e_l\rfloor_{F_l}, \lfloor e_r\rfloor_{F_r}\rangle & \text{otherwise} \end{cases}$$
$$\text{where } \; F_l = F \wedge F' \qquad F_r = F \wedge \neg F'$$

Figure 8. Formula choice calculus language definition.

The implementation of $\lfloor e\rfloor_F$ is best understood as eliminating alternatives that are inconsistent with F. Given a formula choice $F'\langle e_l,e_r\rangle$, the alternative e_l is selected when F' is true, while e_r is selected when F' is false. During partial configuration, we assume that the projection formula F is true. So, we replace the choice by e_r if it is not possible that, given F, F' is true, that is, if $F \wedge F'$ is not satisfiable. We replace the choice by e_r if it is not possible that, given F, F' is false, that is, if $F \wedge \neg F'$ is not satisfiable. In the case where both alternatives are consistent with F, we replace F' by the conjunction $F \wedge F'$. The function *min* indicates that this expression should be minimized, if possible, for usability.

Using the definitions of $\lfloor e\rfloor_F$ and $\lfloor e\rfloor_c$, it is straightforward to adapt the *projection* and *edit isolation* principles from Section 3.2 to specify the behavior of *view* and *update* for the formula choice calculus. These principles lead to the following implementations of *view* and *update*, which are quite similar to the implementations developed in Sections 3.2 and 3.3.

$$\mathit{view}(F,s) = \lfloor s\rfloor_F$$
$$\mathit{update}(F,s,v') = \mathit{minimize}(F\langle v',s\rangle)$$

We can also reuse much of the definition of *minimize* from Section 3.3. By substituting F for D, we can immediately reuse the AST-FACTORING, CHOICE-IDEMPOTENCY, and all three congruence rules. The CHOICE-DOMINATION rule can be adapted as follows.

$$\textsc{Choice-Domination}\quad \frac{\lfloor e_l\rfloor_F = e'_l \qquad \lfloor e_r\rfloor_{\neg F} = e'_r}{F\langle e_l,e_r\rangle \rightsquigarrow F\langle e'_l,e'_r\rangle}$$

However, there are more opportunities for minimizing formula choices than are captured by these rewriting rules. In particular, it is often possible to "join" redundant alternatives in nested formula choices, as shown by the additional minimization rules in Figure 9. For example, consider again the expression $A\langle 2,B\langle 2,3\rangle\rangle$. By applying the JOIN-OR rewriting rule, we can rewrite this expression without redundancy as $(A \vee B)\langle 2,3\rangle$. Apel et al. (2013) argue that identifying and exploiting such join opportunities early is a key to scaling variational analyses. We believe it is also important for projectional editing of variational software since it minimizes redundancy in the code base, easing maintenance.

$$\textsc{Join-Or}\quad F\langle e_l,F'\langle e_l,e_r\rangle\rangle \rightsquigarrow (F \vee F')\langle e_l,e_r\rangle$$

$$\textsc{Join-And}\quad F\langle F'\langle e_l,e_r\rangle,e_r\rangle \rightsquigarrow (F \wedge F')\langle e_l,e_r\rangle$$

$$\textsc{Join-Or-Not}\quad F\langle e_l,F'\langle e_r,e_l\rangle\rangle \rightsquigarrow (F \vee \neg F')\langle e_l,e_r\rangle$$

$$\textsc{Join-And-Not}\quad F\langle F'\langle e_r,e_l\rangle,e_r\rangle \rightsquigarrow (F \wedge \neg F')\langle e_l,e_r\rangle$$

Figure 9. Additional minimization rules for formula choices.

Unfortunately, the rules in Figure 9 do not represent all opportunities for joining redundant alternatives. For example, the following two expressions are semantically equivalent, but there is no way to transform the left expression into the right using the rewriting rules.

$$A\langle 2,B\langle C\langle 2,3\rangle,C\langle 2,4\rangle\rangle\rangle \equiv (A \vee C)\langle 2,B\langle 3,4\rangle\rangle$$

One possibility is to introduce additional rewriting rules to swap the nesting of choices (Erwig and Walkingshaw 2011), which can be used to setup the join rules in Figure 9. However, this complicates the implementation of *minimize* since we must search for an optimal sequence of swaps. It also unclear whether $\rightsquigarrow^*$ is still normalizing. For now, we assume that redundancy in non-immediate siblings is not automatically eliminated.

Although there are still some open challenges, this section illustrates how our projectional editing model can be extended to support arbitrary boolean conditions on variational code. The representation presented in this section is similar to the representation used by TypeChef to represent variability in `#if`-annotated C programs (Kenner et al. 2010; Kästner et al. 2011).

6. Related Work

6.1 Virtual Separation of Concerns

This paper has focused on variability embedded directly into software through the use of variation annotations (for example, `#if` statements or choices). An alternative approach is to separate a program into several composable modules that can be included or not in a particular program variant (Batory et al. 2004; Mezini and Ostermann 2004). There are many tradeoffs between annotative variation and composition-based variation (Apel and Kästner 2009; Walkingshaw and Erwig 2012). The primary benefit and motivation for composition-based variation is that it supports a *separation of concerns* (Tarr et al. 1999)—that is, it is possible to work on one feature without having to consider many other irrelevant features.

Kästner and Apel (2009) have proposed the idea of a *virtual separation of concerns* as a way to bring similar benefits to annotation-based product lines via tool support (see also Kim et al. 2008; Kästner et al. 2009). This idea is a major motivation for our work.

The CIDE tool (Kästner et al. 2008) supports a virtual separation of concerns by allowing the user to select which features they are currently interested in, then filtering out of the editor view the annotated code blocks related to uninteresting features. CIDE's representation of variation is more restricted than either the choice calculus or `#if` annotations since it supports only additive variability. That is, a feature is associated with some code that is added to the program if the feature is included and excluded otherwise. But including a feature cannot remove or replace existing code.

Projectional editing of variational programs is also motivated by previous work that has demonstrated that less obtrusive variation annotations (e.g. background colors) and the filtering of irrelevant code can help programmers understand variational programs (Le et al. 2011; Feigenspan et al. 2011; Stengel et al. 2011).

6.2 View–Update and Bidirectional Transformations

A huge amount of research in many different disciplines of computer science can be classified under the umbrella category of *bidirectional transformations*. The common thread in all of this work is to be able to translate changes in one artifact to changes in a related artifact. Czarnecki et al. (2009) provide a good overview. In this section we focus on foundational research in databases and recent work in functional programming, which most influenced our editing model.

The *view–update problem* is a classic problem in relational databases (Bancilhon and Spyratos 1981; Dayal and Bernstein 1982). The main challenge is that it is not always possible to translate view changes unambiguously into source changes. The technique used to avoid or resolve these ambiguities is called an *update policy*. One policy is to identify the properties of views that yield unambiguous updates, then only allow views that satisfy these properties to be updated (Dayal and Bernstein 1982; Gottlob et al. 1988). Another policy is to restrict the query language to yield only unambiguously updatable views (Dayal and Bernstein 1982), or to otherwise build the update policy into the projection itself (Medeiros and Tompa 1986). Most relevant to us is the work of Bancilhon and Spyratos (1981), who define update policies relative to a *constant complement* of the view that should not change when the update is performed. This is similar to our edit isolation principle, where hidden variants can be considered the constant complement of the visible variants in the view. Cosmadakis and Papadimitriou (1984) show that identifying the minimum constant complement of a relational view is a hard problem. However, this is not the case in our projectional editing model, where the edit isolation principle is easy to enforce, as demonstrated in Section 3.3.

In the functional programming community, work on bidirectional transformations has focused on libraries and languages for building reversible projections called "lenses" (Foster et al. 2007; Bohannon et al. 2008; Voigtländer 2009). A lens defines both a projection and an update policy. Section 4 presented several consistency laws from work on lenses, and showed that our editing model satisfies these laws. The GETPUT and PUTGET are also similar to invertibility laws described in the database literature (Fagin 2007).

6.3 Projectional and Task-Focused Editing

The idea of presenting a task-specific view of source code is not new. A ubiquitous example is the debugging view offered by most IDEs. Another example is the Mylar plugin (Kersten and Murphy 2005) for the Eclipse IDE. Mylar monitors a programmer's activity and automatically filters out files and code that it guesses to be irrelevant. The focus on filtering is similar to our approach; however, we require the programmer to intentionally project on a subset of variants, rather than the system inferring this. Mylar was shown to improve productivity in a field study (Kersten and Murphy 2006), suggesting that hiding irrelevant information is useful for programmers.

Janzen and De Volder (2004) describe a programming language and projectional editing model for editing the same program under different modularity schemes. In their system, a projection *restructures* the source code, that is, it presents the same information organized in a different way to support a particular kind of task. In contrast, our projections *filter* the source code, temporarily hiding variability that the user doesn't want to (or shouldn't be able to) see.

Most similar to our work is the Version Editor (VE), a projectional editing tool that supports within-file version control via `#if`-like `#version` annotations (Atkins 1998). The editing model is similar to ours: the user obtains a view of a source file corresponding to a particular version, then any edited lines are automatically wrapped in corresponding `#version` annotations. Variation in VE is more limited than `#if` annotations or the choice calculus. A `#version` annotation refers to a single configuration (version) of the software, whereas a choice refers to a dimension or formula that represents a set of related configurations, for example, all configurations containing a particular feature. Similarly, an edit in VE always affects exactly one version—a more extreme form of edit isolation. An analysis of over ten years of use at Bell Labs revealed that developers were 40% more productive using VE than editing `#version` annotations manually (Atkins et al. 2002), suggesting that projectional editing and the edit isolation principle are useful in practice.

The Leviathan file system (Hofer et al. 2010) supports projectional editing of annotation-based variation at the file-system level, rather than via an editor. The user specifies a (partial) configuration of a variational code base, then mounts a file system containing the projected code base. Any file edits on the projected code are translated into file edits on the original source. The main advantage of this approach is that users can use their usual editors and tool chains. Leviathan does not enforce edit isolation, and instead relies on heuristics to resolve alignment issues. This may be better for some use cases, but when updates are performed transparently by the file system, a more predictable editing model may be desirable. Since our editing model is not tied to any specific editor, it could easily be used as a drop-in replacement.

7. Conclusion and Future Work

In this paper we presented a *projectional editing model* for variational software that supports the following workflow: (1) partially configure a complex variational program, (2) edit the simplified view, (3) use the edited view to automatically update the original source program. The distinguishing feature of our approach is that it maintains an *edit isolation* principle that ensures that only variants that are visible in the view are updated in the source. We have formalized the specifications of the *view* and *update* functions needed to support the projectional editing workflow, and provided reference implementations of these functions. We have also proved that our specifications ensure the satisfaction of desirable theoretical properties from related work. Our projectional editing model is formulated in terms of the choice calculus, however, we have also shown that the model can be extended to choices with arbitrary boolean conditions, demonstrating that it can be applied to the kinds of variation that occur in real software.

As future work, we plan to implement the *update* operation in terms of an off-the-shelf tree-diff algorithm (Bille 2005). A tree-diff algorithm takes two trees as input and efficiently produces an edit script capturing the differences between the two. A simple patch function can then apply the script to the first tree in order to obtain the second. Our idea is to implement a variational patch function that takes an edit script describing the differences between v and v', and, taking into account the projection p, applies the script to s to produce s'. The strategy is similar to the work of Boneva et al. (2011) on efficiently computing updates across schema translations in XML. We expect this approach to be more efficient than the reference implementation of *update* described in Section 3.3, and so make our projectional editing model more practically useful.

Finally, two kinds of empirical evaluation would help to validate our approach: (1) Determine what percentage of existing edits are consistent (or nearly consistent) with the edit isolation principle, for example, by analyzing the revision history of a variational software project, such as BusyBox. (2) Determine whether programmers can create, modify, and debug variational code more effectively using our projectional editing model, for example, by performing a user study. Together, these would validate that our projectional editing model is useful and applicable to real variational software.

Acknowledgments

Thank you to Christian Kästner for pointers to related work. This work is supported by the European Research Council, grant 203099.

References

S. Apel and C. Kästner. An Overview of Feature-Oriented Software Development. *Journal of Object Technology*, 8(5):49–84, 2009.

S. Apel, A. von Rhein, P. Wendler, A. Größlinger, and D. Beyer. Strategies for Product-Line Verification: Case Studies and Experiments. In *IEEE Int. Conf. on Software Engineering*, pages 482–491, 2013.

D. L. Atkins. Version Sensitive Editing: Change History as a Programming Tool. In *European Conf. on Object-Oriented Programming*, volume 1439 of *LNCS*, pages 146–157. Springer, 1998.

D. L. Atkins, T. Ball, T. L. Graves, and A. Mockus. Using Version Control Data to Evaluate the Impact of Software Tools: A Case Study of the Version Editor. *IEEE Trans. on Software Engineering*, 28(7):625–637, 2002.

F. Bancilhon and N. Spyratos. Update Semantics of Relational Views. *ACM Trans. on Database Systems*, 6(4):557–575, 1981.

D. M. Barbosa, J. Cretin, N. Foster, M. Greenberg, and B. C. Pierce. Matching Lenses: Alignment and View Update. In *ACM SIGPLAN Int. Conf. on Functional Programming*, pages 193–204, 2010.

D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. on Software Engineering*, 30(6):355–371, 2004.

P. Bille. A Survey on Tree Edit Distance and Related Problems. *Theoretical Computer Science*, 337(1–3):217–239, 2005.

A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, and A. Schmitt. Boomerang: Resourceful Lenses for String Data. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 407–419, 2008.

I. Boneva, A.-C. Caron, B. Groz, Y. Roos, S. Tison, and S. Staworko. View Update Translation for XML. In *Int. Conf. on Database Theory*, pages 42–53, 2011.

S. Chen, M. Erwig, and E. Walkingshaw. Extending Type Inference to Variational Programs. *ACM Trans. on Programming Languages and Systems*, 36(1):1:1–1:54, 2014.

S. S. Cosmadakis and C. H. Papadimitriou. Updates of Relational Views. *Journal of the ACM*, 31(4):742–760, 1984.

K. Czarnecki, J. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. Terwilliger. Bidirectional Transformations: A Cross-Discipline Perspective. In R. F. Paige, editor, *Theory and Practice of Model Transformations*, volume 5563 of *LNCS*, pages 260–283. Springer, 2009.

U. Dayal and P. A. Bernstein. On the Correct Translation of Update Operations on Relational Views. *ACM Trans. on Database Systems*, 7(3):381–416, 1982.

M. Erwig and E. Walkingshaw. The Choice Calculus: A Representation for Software Variation. *ACM Trans. on Software Engineering and Methodology*, 21(1):6:1–6:27, 2011.

R. Fagin. Inverting Schema Mappings. *ACM Trans. on Database Systems*, 32(4):25:1–25:53, 2007.

J. Feigenspan, M. Schulze, M. Papendieck, C. Kästner, R. Dachselt, V. Köppen, and M. Frisch. Using Background Colors to Support Program Comprehension in Software Product Lines. In *Int. Conf. on Evaluation and Assessment in Software Engineering*, pages 66–75, 2011.

J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-update Problem. *ACM Trans. on Programming Languages and Systems*, 29(3), 2007.

G. Gottlob, P. Paolini, and R. Zicari. Properties and Update Semantics of Consistent Views. *ACM Trans. on Database Systems*, 13(4):486–524, 1988.

W. Hofer, C. Elsner, F. Blendinger, W. Schröder-Preikschat, and D. Lohmann. Toolchain-Independent Variant Management with the Leviathan Filesystem. In *Int. Workshop on Feature-Oriented Software Development*, pages 18–24, 2010.

D. Janzen and K. De Volder. Programming with Crosscutting Effective Views. In M. Odersky, editor, *European Conf. on Object-Oriented Programming*, volume 3086 of *LNCS*, pages 197–220. Springer, 2004.

M. Johnson and R. Rosebrugh. Constant Complements, Reversibility and Universal View Updates. In J. Meseguer and G. Roşu, editors, *Algebraic Methodology and Software Technology*, volume 5140 of *LNCS*, pages 238–252. Springer, 2008.

C. Kästner and S. Apel. Virtual Separation of Concerns—A Second Chance for Preprocessors. *Journal of Object Technology*, 8(6):59–78, 2009.

C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *IEEE Int. Conf. on Software Engineering*, pages 311–320, 2008.

C. Kästner, S. Apel, S. Trujillo, M. Kuhlemann, and D. Batory. Guaranteeing Syntactic Correctness for All Product Line Variants: A Language-Independent Approach. In *Int. Conf. on Objects, Components, Models and Patterns*, volume 33 of *LNBIP*, pages 175–194. Springer, 2009.

C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *ACM SIGPLAN Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 805–824, 2011.

A. Kenner, C. Kästner, S. Haase, and T. Leich. TypeChef: Toward Type Checking #ifdef Variability in C. In *Int. Workshop on Feature-Oriented Software Development*, pages 25–32, 2010.

M. Kersten and G. C. Murphy. Mylar: A Degree-of-Interest Model for IDEs. In *Int. Conf. on Aspect-Oriented Software Development*, pages 159–168, 2005.

M. Kersten and G. C. Murphy. Using Task Context to Improve Programmer Productivity. In *ACM SIGSOFT Int. Symp. on the Foundations of Software Engineering*, pages 1–11, 2006.

C. H. P. Kim, C. Kästner, and D. Batory. On the Modularity of Feature Interactions. In *ACM SIGPLAN Int. Conf. on Generative Programming and Component Engineering*, pages 19–23, 2008.

D. Le, E. Walkingshaw, and M. Erwig. #ifdef Confirmed Harmful: Promoting Understandable Software Variation. In *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, pages 143–150, 2011.

C. B. Medeiros and F. W. Tompa. Understanding the Implications of View Update Policies. *Algorithmica*, 1(1-4):337–360, 1986.

M. Mezini and K. Ostermann. Variability Management with Feature-Oriented Programming and Aspects. *ACM SIGSOFT Software Engineering Notes*, 29(6):127–136, 2004.

H. V. Nguyen, C. Kästner, and T. N. Nguyen. Exploring Variability-aware Execution for Testing Plugin-based Web Applications. In *IEEE Int. Conf. on Software Engineering*, pages 907–918, 2014.

M. Stengel, M. Frisch, S. Apel, J. Feigenspan, C. Kästner, and R. Dachselt. View Infinity: A Zoomable Interface for Feature-Oriented Software Development. In *IEEE Int. Conf. on Software Engineering*, pages 1031–1033, 2011.

P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *IEEE Int. Conf. on Software Engineering*, pages 107–119, 1999.

J. Voigtländer. Bidirectionalization for Free! (Pearl). In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 165–176, 2009.

E. Walkingshaw and M. Erwig. A Calculus for Modeling and Implementing Variation. In *ACM SIGPLAN Int. Conf. on Generative Programming and Component Engineering*, pages 132–140, 2012.

E. Walkingshaw, C. Kästner, M. Erwig, S. Apel, and E. Bodden. Variational Data Structures: Exploring Trade-Offs in Computing with Variability. In *ACM SIGPLAN Symp. on New Ideas in Programming and Reflections on Software (Onward!)*, 2014. To appear.