A Language for Software Variation Research

Martin Erwig

School of EECS
Oregon State University
erwig@eecs.oregonstate.edu

Abstract

Managing variation is an important problem in software engineering that takes different forms, ranging from version control and configuration management to software product lines. In this paper, I present our recent work on the choice calculus, a fundamental representation for software variation that can serve as a common language of discourse for variation research, filling a role similar to lambda calculus in programming language research. After motivating the design of the choice calculus and sketching its semantics, I will discuss several potential application areas.

Categories and Subject Descriptors D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement—Extensibility, Version Control; D.2.9 [*Software Engineering*]: Management—Software Configuration Management; D.3.1 [*Programming Languages*]: Formal Definitions and Theory

General Terms Languages, Theory

1. Introduction

Software variation can take quite different forms. Software can have different *versions*, which refer to variations of the software over time. Software can also be built to have different sets of *features*, which provides different functionalities. Moreover, we can consider alternative *implementation* choices in software. Finally, there is variation in terms of the context in which software is used; that is, software can exist in different *configurations*. Whole subfields have emerged for each of these kinds of variations, for example, feature modeling [4, 14] and software product lines [19, 20], feature-oriented programming [3, 18], revision control systems [21], and software configuration management [11]. What this fact demonstrates first and foremost is that software variation is an important topic that draws much attention and addresses important concerns of software development.

In addition, however, this specialization also raises the question of whether all these forms of variation are completely different or whether they have something in common. In the latter case, wouldn't it be a good idea to investigate the core ideas they share and identify general properties of variations that are valid in all these different areas? The potential benefits of studying variations from a general perspective include (1) a more profound understanding of the issues involved in variation management, (2) putting general results to use in several specialized areas, and (3) transferring results developed in one area to other areas. In addition, one particular benefit will be the opportunity for the integration of different kinds of variation.

In this paper I will present a language to represent and reason about variation. After motivating the design of the language in Section 2, I will sketch its semantics and discuss some of its properties in Section 3. The main part of the paper will be a description of possible applications in Section 4. After a discussion of related work in Section 5, conclusions follow in Section 6.

GPCE'10, October 10–13, 2010, Eindhoven, The Netherlands.

2. The Elements of Variations

In this section I will introduce the concepts of the choice calculus [10] through a small Haskell program for manipulating graphs. Suppose our goal is to implement a module for supporting computations with directed graphs. As a simple example we consider the implementation of the function `suc` that computes the successors for a given node in a graph. The first decision we are faced with is how to represent graphs. A very simple approach is to use a list of edges where each edge is given by a pair of integers.

```
type Node  = Int
type Graph = [(Node,Node)]
suc g v = [w | (u,w) <- g, u==v]
```

The implementation of `suc` uses a list comprehension to find all edges in `g` that originate in the node `v` and projects on their second component.

An alternative is to employ an adjacency list representation for graphs, that is, a list of nodes, each paired with the list of its successor nodes.

```
type Node  = Int
type Graph = [(Node,[Node])]
suc g v = head [ws | (u,ws) <- g, u==v]
```

The implementation of `suc` is very similar: Again, find the node `v`, which is in this representation directly paired with all of its successors. Since the list comprehension returns a (single-element) list of a successor lists, we have to extract it using the `head` function. We assume that in the adjacency list representation each node occurs only once with all of its successors. Therefore, the list comprehension will result in a list of one element and the `head` function yields the sought successor list. Note also that if `v` is not present in the graph, the `suc` function will produce a runtime error since the list comprehension will result in an empty list.

Which representation should we choose? In general, this decision depends on what further plans we have for the program. Some operations are better supported by the first representation, other operations are easier to implement or are more efficient when using the second representation. For example, we have already seen that

the two implementations of `suc` differ in their behavior for non-existing nodes. Another difference is that we cannot represent isolated nodes with the first approach.

Whenever we start a programming project, it is not always clear which operations we will need, and therefore the choice between the two representations is not obvious. It would therefore be helpful if we could delay this decision until later. Maybe we even want to maintain both versions forever. In any case, it would be nice if we could keep both options open and assume that both representations are available.

2.1 Choices

In practical terms, we would like to write just one Haskell program and express a choice between the two alternatives directly within the code. Alternatively, we could consider writing two separate programs, but this approach quickly leads to code redundancy, which entails the danger of update anomalies, a problem I will discuss in more detail in Section 2.3.

In a combined program, we have to make a choice at two places, the type definition for `Graph` and the definition for the function `suc`. What do we need to represent each choice? Of course, in each case we need the two alternative code fragments and an annotation that marks them as such. In addition, we need some aspect in the representation that allows us to systematically select alternatives later. One idea that quickly comes to mind is to assign tags to the alternatives. With this idea we can show choices as mappings from tags to alternatives.

We use angle brackets to enclose mappings and a colon to separate a tag from the alternative it labels. Using this notation, a combined graph program looks as follows.

```
type Node  = Int
type Graph = [(Node,⟨rel: Node,adj: [Node]⟩)]
suc g v = ⟨rel: [w | (u,w) <- g, u==v],
           adj: head [ws | (u,ws) <- g, u==v]⟩
```

We can recover each of the original two programs by traversing this annotated program and consistently selecting alternatives with either one of the tags *rel* or *adj* from each choice.

In this example we can notice the two implementations for `suc` share much of their code, which raises the question of whether the common part can be factored out of the choice. If we don't care about using different variable names, `w` and `ws`, to bind single and lists of nodes, respectively, it seems we can simply factor out the application of the `head` function in a choice. However, it is not immediately clear what we should put into the alternative for the *rel* implementation. In the original program no function was applied, so we really want to leave that alternative empty in the choice, which we indicate by using the symbol ε. We can therefore represent the variation in the implementation as follows.

```
suc g v = ⟨rel: ε,adj: head⟩ [w | (u,w)<-g, u==v]
```

In general, we must be careful with such a representation since variation annotations do not operate on strings, but on abstract syntax trees; that is, the first alternative of the above choice expression represents the application of ε to the list comprehension, but there is no such thing as an empty function in Haskell. We could instead use the identity function in the alternative for *rel*.

```
suc g v = ⟨rel: id,adj: head⟩ [w | (u,w)<-g, u==v]
```

However, that will cause the generation of a strange-looking variation when we select the alternative *rel*, because we obtain a spurious `id` function preceding the list comprehension. Therefore, we shall adopt the convention that any occurrences of ε in applications, such as $\varepsilon\ e$ or $e\ \varepsilon$, that occur after a selection will be replaced by e.

In general, factored representations have the advantage of avoiding update anomalies. On the other hand, a fully factored representation is sometimes more difficult to edit. Compare, for example, the first definition of `suc` with the following one in which use the choice ⟨*rel*: `w`,*adj*: `ws`⟩ to retain the different variable names in the alternatives.

```
suc g v = ⟨rel: ε,adj: head⟩
    [⟨rel: w,adj: ws⟩ | (u,⟨rel: w,adj: ws⟩)<-g, u==v]
```

The above code is much harder to read, but less redundant. It should not be too surprising that different representations are more or less suited for different purposes. If we can map unambiguously between different representations, we can exploit this fact. In Section 4.1 I will show a simple transformation rule that allows the factoring of common contexts into or out of choices.

It is clear that in this example the two choices are synchronized in the sense that if we select, say *adj*, for the type definition choice, we also need to select *adj* for the choice defining `suc`, because otherwise we would introduce a type error into the selected program. This synchronization can be easily achieved by defining the selection operation to take one tag, such as *adj*, and select it from *all* choices in the program. However, this solution is problematic since it requires that all tags be globally visible. This can be a problem when we want to merge two annotated files that have some tags in common that are used to represent independent choices that should not be synchronized.

A further problem is caused by the fact that synchronized choices should have the same number of alternatives and the same tags. Consider, for example, what happens if we extend the type definition for `Graph` by an incidence matrix implementation, tagged with *mat*, but we forget to extend the definition of `suc` accordingly. If we select *mat*, we end up with a partially annotated program: The type definition has been selected, but the definition for `suc` has not. In order to obtain a plain program, we have to perform a further selection, which can be only *rel* or *adj*, and that can lead to a program with a type error, or worse, to a program that performs nonsense computations that might remain undetected for a long time.

The problem is that the shown representation does not enforce the necessary constraints among synchronized choices. A solution to this problem is to organize all synchronized choices into a larger structure. This is what I will describe next.

2.2 Dimensions

A dimension defines the tags and scope for a set of synchronized choices. Since a program can, in general, contain choices in different dimensions, each dimension is identified by a name. Moreover, the choices that belong to that dimension are labeled by the dimension name to express this connection. A dimension definition has the following general form.

$$\textbf{dim}\ D\langle t_1,\ldots,t_n\rangle\ \textbf{in}\ e$$

Here D is the name of the dimension, $t_1,\ldots,t_n$ are its tags (which must be pairwise different), and expression e is the scope of the dimension, that is, all choices in e that are labeled by D are bound by this dimension declaration (as long as they are not preceded by another definition for D).

Using a dimension declaration, our graph example can be expressed as follows.

```
type Node = Int
dim Rep⟨rel,adj⟩ in
type Graph = [(Node,Rep⟨Node,[Node]⟩)]
suc g v = Rep⟨ε,head⟩ [w | (u,w) <- g, u==v]
```

We can observe that choices do not contain tags anymore. Instead, the correspondence between tags and alternatives is established by position; that is, the ith alternative in a choice is selected by tag t_i of its binding dimension. A choice has therefore the following general form.

$$D\langle e_1, \ldots, e_n \rangle$$

Coming back to the example, one might wonder about the position of the dimension declaration. Does it have to appear in the second line or can it be moved around? Since the scope of a dimension declaration is the expression following the **in** keyword, the **dim** declaration could just as well be moved to the first line, but it could not be moved into the type definition since this would leave the second choice unbound. Note that introducing a copy of the **dim** declaration in the last line would *not* fix this problem since that change would lead to two independent dimensions, which would force us to select from the two choices independently—as we have seen, we need them to be synchronized.

The purpose of dimensions is to structure choices and group them into meaningful units. Since dimensions are binding constructs that can appear anywhere in a program, one question is what it means when one dimension is nested within another one or within a choice. Does the nesting matter, or is it always possible to transform nested dimensions into some "flat" normal form?

Consider as an example the different behavior of the two `suc` implementations for nodes that are not contained in a graph. We could envision a variation of the code that allows us to choose between an error-producing or error-ignoring behavior for the `suc` implementation within the *adj* implementation. To that end we can use a function `safe` that extracts the first element only from a non-empty list and returns an empty list otherwise.

```
safe (s:_) = s
safe []    = []
```

Now we can add a dimension to the second alternative of the `suc` implementation that provides a choice between `head` and `safe`.

```
dim Rep⟨rel,adj⟩ in ...
suc g v = Rep⟨ε,dim Err⟨yes,no⟩ in Err⟨head,safe⟩⟩
          [w | (u,w) <- g, u==v]
```

The nested dimension expression looks quite complicated, and it might make the fact that there is a decision to make in addition to the selection of the graph representation less obvious. Can we move the *Err* dimension out of the choice and rewrite the program in the following way?

```
dim Rep⟨rel,adj⟩ in ...
dim Err⟨yes,no⟩ in
suc g v = Rep⟨ε,Err⟨head,safe⟩⟩
          [w | (u,w) <- g, u==v]
```

The grouping of all the dimension declarations at the top of the program certainly looks more tidy and is a more attractive way of presenting the available decisions. However, this second representation is *not* equivalent to the first one because the represented decisions change. The difference is that in the nested case, the selection in the *Err* dimension is only relevant for the adjacency-list graph representation and has to be made only if *adj* was selected for the *Rep* dimension. In contrast, the selection for the *Err* dimension must always be made in the second representation, no matter what is chosen for the *Rep* dimension, even though the choice has no effect in the case of *rel*.

This does not mean that the second representation is incorrect (although it is less economic since it requires in some cases a decision that has no effect). It is only that the two representations are not equivalent with respect to what decisions are represented.[1]

We could extend this example to also distinguish between the error and non-error case for the *rel* representation. For example, we could apply a function `check v g` to the list comprehension that first determines whether `v` is contained in `g`. If so, it evaluates the list comprehension, otherwise it raises an error. Representing this approach with nested dimension would result in the following code.

```
dim Rep⟨rel,adj⟩ in ...
suc g v = Rep⟨dim Err⟨yes,no⟩ in Err⟨check v g,ε⟩,
              dim Err⟨yes,no⟩ in Err⟨head,safe⟩⟩
          [w | (u,w) <- g, u==v]
```

Since in this code the *Err* dimension is contained in all alternatives of the *Rep* choice, the transformation that lifts the declaration of the *Err* dimension out of the choice preserves the decision semantics, and we can rewrite the program in the following form without changing the semantics.

```
dim Rep⟨rel,adj⟩ in ...
dim Err⟨yes,no⟩ in
suc g v = Rep⟨Err⟨check v g,ε⟩,Err⟨head,safe⟩⟩
          [w | (u,w) <- g, u==v]
```

The fact that two decisions are merged into one does not change the semantics here since the two decisions occur in different alternatives of the choice, which means that only one decision is only ever relevant.

Much more is to be said about further transformations of the code (for example, we can swap the *Rep* and *Err* choices, but we cannot swap the corresponding dimension declarations). We will encounter some more transformations later in Section 4.

2.3 Sharing

Let us consider as another variation for our program the possibility to choose different names for `Node`. The way to achieve this is to introduce another dimension declaration for the type name and replace all occurrences of the name `Node` by a corresponding choice.

```
dim Name⟨long,short⟩ in
dim Rep⟨rel,adj⟩ in
type Name⟨Node,N⟩ = Int
type Graph = [(Name⟨Node,N⟩,Rep⟨Name⟨Node,N⟩,
                                [Name⟨Node,N⟩])]⟩
...
```

Apart from the fact that the repeated use of the choice $Name\langle \texttt{Node},\texttt{N} \rangle$ is verbose and makes the program difficult to read, this representation reveals a serious technical challenge.

Suppose we extend the *Name* dimension by another alternative. In that case, we have to make sure that we change every choice in the same way. Should we forget to extend a choice, this might be detected by a static analysis tool that enforces that all choices have the same number of alternatives as *Name* has tags, but if we swap two entries or mistype a name, the consequences might be subtle semantics errors that are generally quite hard to detect.

Such update anomalies can be avoided if we represent the choice proper only once and reuse its decision whereever needed. This behavior can be achieved by employing a **let** binding that assigns a name to an expression and allows the reference to the expression's result through this name. In our example program, this would look as follows.

[1] Note that the two versions do represent the same variations, so they are in this weaker sense equivalent.

```
dim Name⟨long,short⟩ in
dim Rep⟨rel,adj⟩ in
let T=Name⟨Node,N⟩ in
type T = Int
type Graph = [(T,Rep⟨T,[T]⟩]⟩
...
```

With the **let** binding we need only one choice and can reuse it many times. In particular, if we want to extend the choice of names, we can do this easily and safely at one place and have the effect systematically replicated through the variable reference.

We can encapsulate the name choice even more by localizing the *Name* dimension right after the **let** binding.

```
dim Rep⟨rel,adj⟩ in
let T=(dim Name⟨long,short⟩ in Name⟨Node,N⟩) in
type T = Int
type Graph = [(T,Rep⟨T,[T]⟩]⟩
...
```

Here we can observe a pattern that occurs quite frequently, namely a dimension declaration whose scope is just a choice bound by that dimensions.

$$\textbf{dim}\ D\langle t_1,\ldots,t_n\rangle\ \textbf{in}\ D\langle e_1,\ldots,e_n\rangle$$

For such a pattern, we offer the following abbreviated notation.

$$D\langle t_1\colon e_1,\ldots,t_n\colon e_n\rangle$$

Note that we keep the name for the choice to be able to refer to that choice when making selections. Using this abbreviation in our example program leads to the following code.

```
dim Rep⟨rel,adj⟩ in
let T=Name⟨long: Node,short: N⟩ in
type T = Int
type Graph = [(T,Rep⟨T,[T]⟩]⟩
...
```

Such an abbreviated dimension plays a similar role as an anonymous function (lambda abstraction) since it is defined at the place where it is used, and it is used only at that place.

2.4 Summary

The choice calculus provides a succinct notation for describing program variations. Similar to XML, the notation is not intended to be used directly, but rather as an underlying representation for tools to view, edit, query, and transform variations. We will see several examples of such potential tools and applications in Section 4.

The chosen constructs and their notation are not arbitrary. The language design is intended to reflect the nature of variations with a few expressive and highly composable constructs. Getting this design right is crucial for the applicability of the choice calculus to problems in variation research. I believe we have achieved this goal, but only future research can tell for sure. The choice calculus is certainly open to all kinds of variations and extensions that emerge in the different application areas.

3. Variation Semantics

Once we have written a program that includes variation, we will eventually want to select specific variants from it. In the following we call a program that contains choice calculus annotations (that is, dimensions, choices, and sharing) a *variation program* in contrast to a *plain program* that does not contain any such annotations. We also use the term *expression* for a variation program and let the metavariable e range over all choice calculus expressions. The plain programs represented by a variation program are called its *variants*.

The selection of variants uses tags of the form $D.t$, which we call *qualified tags* since they include the name of the dimension. This operation is called *tag selection*, and it has the effect of *dimension* and *choice elimination* on the variation program. More precisely, tag selection with $D.t_i$ from the expression $\textbf{dim}\ D\langle t_1,\ldots,t_n\rangle\ \textbf{in}\ e$ yields the expression e in which all bounds choices are replaced by their ith alternative and where nested choices for D are recursively replaced as well. This idea is formalized by the following two rules that define dimension and choice elimination in two phases. Note carefully how the selection from a dimension with a qualified *tag* ($D.t$) name leads to the selection process with a qualified *index* ($D.i$) to find each choice and select the alternatives based on their position.

$$\textsc{Dim Elim}\quad \frac{\lfloor e\rfloor_{D.i}=e'}{\lfloor \textbf{dim}\ D\langle t_1,\ldots,t_n\rangle\ \textbf{in}\ e\rfloor_{D.t}=e'} \qquad \textsc{Choice Elim}\quad \frac{\lfloor e_i\rfloor_{D.i}=e'}{\lfloor D\langle e_1,\ldots,e_n\rangle\rfloor_{D.i}=e'}$$

For completeness we need several additional congruence rules that explain how tag selection moves recursively into subexpressions and how it stops at nested dimension definitions for D. For more details, see [10].

In general, several tag selections are required to obtain a plain, non-annotated program from a variation program, which means the meaning of an annotated program can be described as a mapping from sets or sequences of qualified tags to plain programs. As explained below, we choose a semantics based on ordered tags. For example, the semantics of the expression

$$\textbf{dim}\ A\langle a,b\rangle\ \textbf{in}\ A\langle \textbf{dim}\ B\langle c,d\rangle\ \textbf{in}\ B\langle 1,2\rangle,3\rangle$$

is the function

$$\{([A.a,B.c],1),([A.b,B.d],2),([A.b],3)\}.$$

Each pair in this function describes a decision as a sequence of tags and the result of making the decision, which is a plain program/expression. The essence of the semantics is captured in the following equation.[2]

$$[\![\textbf{dim}\ D\langle t_1,\ldots,t_n\rangle\ \textbf{in}\ e]\!] = \{((D.t_i,\bar{q}),e') \mid 1\le i\le n\wedge(\bar{q},e')\in[\![\lfloor e\rfloor_{D.i}]\!]\}$$

Again the equation shows only the core part of the definition, and we need additional equations to explain how to determine the semantics recursively over subexpressions. Finally, we also have to expand potential **let** expressions in the range of the function to remove the sharing annotations.

Why is the order of tags relevant? Couldn't we just as well define the semantics based on sets of tags? The use of sequences has the advantage that we can give semantics to expressions that contain several occurrences of one dimension. As an example consider the case in which we might want to decide on the parameter names of several functions independently of one another. This can be easily achieved by having a copy of a naming dimension with all the required information at each function. To make a selection in any one of those dimensions we have to say which one we are targeting, and the easiest solution is to make those selections in a specific order, such as given by a pre-order traversal.

The goal of the semantics is to describe the decisions represented by a variation program, that is, each possible sequence of dimensions and tags together with the plain program that results from that decision. However, in general it is not guaranteed that the range of the function produced by the semantics contains only plain programs. As the above equation illustrates, all tag selections

[2] Note that a nested tuple $(D.t_1,(D.t_2,(D.t_3\ldots)))$ represents the list of tags $[D.t_1,D.t_2,D.t_3,\ldots]$ and $\bar{q}$ ranges over lists of qualified tags.

are initiated by dimension declarations, which means that unbound choices will not be eliminated by the semantics. If, however, the variation program does not contain unbound choices or **let** variables, which can be easily checked by a static analysis, the semantics can be shown to only contain plain programs in the function range.

4. Applications

The choice calculus is an attempt to capture the essential aspects of variation in a small set of core constructs. Its design aims deliberately at simplicity and generality. The motivation behind the choice calculus is to have a fundamental language to support variation research in several different areas.

In this section I want to describe several potential applications for the choice calculus. These applications are in my view important areas of variation research, and the purpose in this section is not to deliver fully worked out results, but rather to illustrate how the choice calculus can help to identify the issues involved and present avenues to possible solutions. Moreover, the presented list is by no means exhaustive; the presented ideas are intended to demonstrate the generality of the choice calculus and its relevance to variation research.

4.1 Variation Design

Database theory provides sound guidance on how to design database schemas in order to avoid update (and other) anomalies with the stored data [5]. Similar considerations apply to variation representations. In a sense, a variation program can be viewed as a database of programs that can be queried to produce specific program instances and updated to add, change, or remove variations.

As demonstrated in [10], the variation representation offered by the choice calculus enjoys a rich set of laws that provides great flexibility in how variations can be represented. Therefore, the question arises of which kinds of representations should be favored and which ones should better be avoided.

In general, an ω-anomaly (where $\omega \in \{insert, delete, update\}$) occurs when the application of the operation ω to a variation program results in a representation of programs that are inconsistent or contain errors. The question of update anomalies is, of course, closely tied to the question of how to maintain and edit variations, and so the insights gained into variation design will directly support the applications discussed in Section 4.5.

In Section 2.3 I have discussed an example that demonstrated how forgetting to extend a choice would result in an inconsistent variation program, while changing choices inconsistently would lead to type errors in a selected program. The introduction of a **let** binding allowed us to factor a common subexpression, which helped prevent these update anomalies.

The transformation employed in the example is an instance of a more general strategy to avoid redundant code as much as possible—at least as far as it concerns variation annotations. While it might also often be a good idea to avoid redundant code in programs, this is not the concern of variation design; the focus here is how redundancy can be avoided in choices and dimensions.

The following guidelines can help minimize redundancy in variation representations.

(1) Factor common parts of choices.
(2) Factor identical choices through the use of **let** bindings.
(3) Eliminate redundant tags/alternatives, choices, and dimensions.

The first simplification is intended to make choices smaller and can be achieved by applying a rule that extracts a common context out of all alternatives of a choice. One might think that the context C used in the rule FACTOR must not contain any dimension definitions since the factoring would change the semantics by fusing several decisions into one and thus removing decisions. Therefore, one might think the rule needs a premise $BD(C) = \varnothing$, which uses the auxiliary function BD to determine bound dimensions in expressions. However, factoring a dimension definition out of a choice does, in fact, *not* change the semantics since different alternatives do not occur together in the semantics; that is, never more than one of the dimensions will be relevant. The factoring rule is therefore unconditionally valid.

$$\text{FACTOR}\quad D\langle C[e_1],\ldots,C[e_n]\rangle \equiv C[D\langle e_1,\ldots,e_n\rangle]$$

An example application of this rule was shown in Section 2.1 where we factored the list comprehension in the implementation of the function `suc`. It is easy to see that editing the non-factored version can easily lead to update anomalies and that the factored representation prevents much of that.

The second simplification is characterized by the following rule SHARE, which gathers a group of identical choices and combines them with a single **let** binding. The premise prevents multiple dimension definitions from being collapsed into one. Such a dimension merge generally changes the semantics since it replaces several independent decisions by one. The substitution of v for $D\langle e_1,\ldots,e_n\rangle$ implies substituting only for free occurrences of choices, which ensures that we don't capture choices bound by nested dimension definitions for D inside of e .

$$\text{SHARE}\quad \frac{\cup_{i=1}^{n} BD(e_i) = \varnothing}{e \equiv \textbf{let } v{=}D\langle e_1,\ldots,e_n\rangle \textbf{ in } [v/D\langle e_1,\ldots,e_n\rangle]e}$$

An application of this rule was shown in Section 2.3 where the choice representing the naming decision for the node type was factored using a **let** expression.

These two rules have to be supplanted by many other rules that allow the moving of **let** expressions, dimensions, etc.; see [10] for details.

The final simplification requires first a set of criteria to identify redundancy in alternatives, choices, and dimension definitions. Once a redundancy has been spotted, the transformation to get rid of it is rather straightforward.

It should be noted that this third level of simplification is generally *not* semantics preserving. For example, if we remove a redundant tag from a dimension, we eliminate a possible decision. However, the change in the semantics is not arbitrary and it might be just the right thing to do, because after all, the decision offered by a redundant tag doesn't really contribute any variation that isn't already available without it. Still, the removal of a redundant tag changes the semantics.

To make this idea of "benign" semantics changes more precise, we define that two variation expressions e and e' are *variant equivalent* if $rng([\![e]\!]) = rng([\![e']\!])$, and a transformation is called *variant preserving* if it maps an expression into one that is variant equivalent.

Two tags t and t' are equivalent if exchanging their positions in their dimension declaration does not change the semantics of the variation expression. In such a case, selecting t produces the same result as selecting t', which means that the tags are redundant and one of them can be removed, together with all corresponding alternatives in choices bound by the dimension definition. The simplified expression is variant equivalent to the original one.

A *pseudo-choice* is a choice in which all of whose alternatives are equivalent. Note that to be equivalent, the alternatives do not have to be identical. Consider, for example, the choice $e = A\langle A\langle 1,1\rangle,1\rangle$. Even though $A\langle 1,1\rangle$ is not the same as 1, all that can ever be produced from e is 1. Clearly, any pseudo-choice can be replaced by any of its alternatives without changing the semantics.

Finally, a *pseudo-dimension* is a dimension in which all tags are pairwise equivalent. A pseudo-dimension D can be safely removed, but we have to be careful to replace any choice $D\langle e_1,\ldots,e_n\rangle$ that is bound by it with one of the alternatives e_i.

In summary, we can see that the choice calculus provides a precise way to talk about redundancies in variations and their removal.

4.2 Support for Feature Modeling

Feature-oriented software development has become an important paradigm for realizing variability in software [1]. One line of research has been to develop languages that provide direct support for the implementation of features. On the other hand, domain analysis is sometimes considered as a separate activity with feature models [2, 14] as a popular way of describing variability in the problem space [4]. Software developers then have to ensure that the structure and constraints among features is reflected in an actual implementation. The implementation itself has to either employ language constructs or a separate annotation language, such as CPP [13], to implement the variability specified by the feature model. Whenever the implementation is separated from domain modeling, there will be a gap between the specification and implementation of software product lines, which opens the flood gates for all kinds of errors.

This is where the choice calculus comes into play since it offers an integrated representation for features and their implementation. Roughly speaking, a tag corresponds to a feature, and an alternative in a choice that is selected through a tag corresponds to the code that implements that feature. Moreover, a dimension corresponds to an abstract feature whose children, given by the dimension's tags, stand in an XOR relationship. Feature hierarchies can be represented by nested dimensions. Even though OR and AND groups, optional and mandatory features, as well as additional constraints do not have a direct correspondence in the choice calculus, it can be shown that any feature model can be translated without loss of information into the choice calculus [8].

It remains to be seen how well an integrated feature modeling/code development environment can be built on the basis of the choice calculus representation. Such an integration is not what I want to advocate at this point; this will be a question of future research. Nevertheless, the fact that the choice calculus offers an integrated representation for feature models and their implementation promises several other possible ways to support feature modeling.

One potential application is to check whether a program P is a faithful implementation of a feature model M. This can work as follows. First, we need a way to identify code fragments in P with features. For example, for a CPP-annotated program, some subset of the macros corresponds to features. For simplicity, we assume here that such a correspondence is given.

Checking the feature model can now proceed in two main stages. First, the CPP-annotated code is parsed and translated into a choice calculus expression. This works by interpreting boolean macros as binary dimensions and mapping conditional code into choices. Consider, for example, the following code fragment.

```
#ifdef A
  e
#else
  e'
#endif
```

Assuming that the macro `A` corresponds to feature A, the code would be translated into the choice $A\langle e,e'\rangle$ where e and e' are the choice expressions obtained from the translation of `e` and `e'`, respectively. The resulting choice calculus expression encapsulates the relationship between features, represented as tags, and their implementation, represented as alternatives in choices. As explained above, the dimensions and tags of the constructed choice calculus expression represent a feature model, which we call M_P.

The next step is to determine whether or not M_P and M are equivalent, which is not an easy problem. The equivalence of two feature models is generally difficult to check, because equivalence is defined based on the set of denoted products and the number of products for a given feature model is generally exponential in the number of features. This basically rules out as a method to simply compare the sets of products entailed by the feature models. A direct structural comparison is complicated by the fact that, due to the simple parsing process, the feature model M_P consists only of binary features and requires some extensive refactoring to become similar to M. It is unclear whether using such a refactoring of M_P, guided by the structure of M, is a practical method to decide the equivalence of M and M_P. Therefore, we have to find an approach to compare M and M_P indirectly. The goal is to compute some kind of measure μ that represents a feature model more abstractly. One such measure is the information about included and excluded features. For each feature f in a feature model, the function $I(f)$ yields the features that are included in every product that contains f, and similarly, $E(f)$ yields the features that cannot appear in any product that contains f. These functions have at most quadratic size, and even though their computation might still require exponential time in the worst case, the average case is expected to be much faster [8].

Given a measure μ for a feature model, such as the pair of functions I and E, we can now formulate equivalence criteria. For example, $\mu(M_P)\neq\mu(M)$ implies that the feature models are not equivalent. In contrast, we *cannot* conclude their equivalence of M and M_P if $\mu(M_P)=\mu(M)$.

A second application of the choice calculus in the context of feature modeling is to support the post-hoc reconstruction of feature models for existing code. We have already indicated how a CPP-annotated program can be translated into a choice calculus expression. We have also noted that, due to the boolean nature of macros, the choices and corresponding dimensions of the resulting expression are binary. While this often does not reflect the way in which a feature model for the implementation would be structured, it still provides useful information about the relationships among features.

We can try to identify patterns in the choice calculus expression that are indicative of specific arrangements in feature models. Consider, for example, the following nested choice.

$$\begin{array}{l}\textbf{dim}\ A\langle yes,no\rangle\ \textbf{in}\\ \textbf{dim}\ B\langle yes,no\rangle\ \textbf{in}\\ \textbf{dim}\ C\langle yes,no\rangle\ \textbf{in}\ A\langle e_1,B\langle e_2,C\langle e_3,e_4\rangle\rangle\rangle\end{array}$$

We can observe that the choice for B becomes relevant only in the case when *no* is selected for A. Likewise, the choice for C becomes relevant only when *no* is selected for B (and A). In other words, we can select a *yes* alternative for only one of the involved dimensions, which means that only one of the represented features, A, B, or C can be selected. Therefore, we can replace the cascaded nesting of the features by just one abstract feature with four children. In terms of the choice calculus this means to replace the expression by just one dimension with four tags.

$$\textbf{dim}\ D\langle a,b,c,d\rangle\ \textbf{in}\ D\langle e_1,e_2,e_3,e_4\rangle$$

Cases like this can be used to create suggestions about parts of the feature hierarchy of a reverse engineered feature model. Of course, sometimes features might appear in multiple chains, maybe inconsistently, which requires then more analysis before a suggestion is made, but the example demonstrates that the choice calculus representation and its selection semantics provides a basis for identifying transformations that hint at higher-level variation structures, which can be exploited for deriving feature models.

4.3 Processing Program Changes

The difference between two programs is an important source of information about the evolution a program has undergone. There have

been many proposals to compute program differences; a nice survey and classification can be found in Miryung Kim's dissertation [17].

On the abstract syntax level, program changes can be expressed as follows in the choice calculus. Consider two programs P and Q, represented as abstract syntax trees, that differ at n nodes in the following way: The subexpression e_i in P is replaced in Q by the expression e_i'. It is obvious that we can represent the changes by the following choice calculus expression: **dim** $\delta\langle old, new\rangle$ **in** P' where P' results from P by replacing each subexpression e_i by the choice $\delta\langle e_i, e_i'\rangle$.

At this point it seems not much has been gained. We have only obtained a very specific representation of the differences. However, we can now observe that specific kinds of changes are reflected in specific patterns of choice calculus expressions. For example, insertion, removal, and replacement have the following obvious choice calculus patterns.

Change	*Choice Calculus Pattern*
insert `e`	$\delta\langle\varepsilon, \mathtt{e}\rangle$
remove `e`	$\delta\langle\mathtt{e}, \varepsilon\rangle$
replace `e` by `e'`	$\delta\langle\mathtt{e}, \mathtt{e'}\rangle$

These simple changes are identified by most `diff` tools. In addition, however, there are more complex kinds of changes that can be expressed through choice calculus patterns.

For the following discussion note that we employ three different kinds of metavariable. First, we have variables that range over code fragments. For simplicity we use here one-letter symbols set in typewriter font, such as `f`, `x`, or `e`. Second, we have v that ranges over **let**-bound variables of the choice calculus. Finally, we also need context variables, such as C, that range also over code elements with (multiple) holes.

An example of a complex change is the swapping or exchanging of two program elements, which means to identify two expressions, say `e` and `e'` within some context C, and to simultaneously replace `e` by `e'` and `e'` by `e`. Swapping can come in many different forms, for example, swapping two lines in a program or swapping two parameters. Therefore, we need in addition to the expressions that are swapped a context parameter that defines the positions of the swapped elements in the program. A similar example of a complex code change is the moving of an expression from one place to another. Both complex changes can be expressed by choice calculus patterns as follows.

Change	*Choice Calculus Pattern*
swap `e` and `e'` in context C	$C[\delta\langle\mathtt{e}, \mathtt{e'}\rangle, \delta\langle\mathtt{e'}, \mathtt{e}\rangle]$
move `e` in context C	$C[\delta\langle\mathtt{e}, \varepsilon\rangle, \delta\langle\varepsilon, \mathtt{e}\rangle]$

The point of the choice calculus representation is to be able to give a succinct description of complex changes. The patterns illustrate rather nicely that, for example, swapping is essentially given by two related replacements and that moving is the same as an insert and remove.[3]

However, swapping and moving are examples of only "mildly complex" changes since they contain only two simple changes. In general, a complex change may involve a group with an unspecified number of changes. For obtaining descriptions of such cases we must be able to systematically relate a group of changes. To achieve this goal, we first have to analyze the set of all raw changes and find groups of related changes. At this point the dimension construct and its ability to assign a different name to each group comes in handy.

Consider the following program, which contains a definition of the function `twice` and a test value.

```
twice x = x+x
test = 5
```

Suppose that in an edited version of the program the parameter `x` was renamed to `y` and the value was changed from `5` to `7`. A `diff` tool could produce the following raw differences, represented as variation annotations.

dim $\delta\langle old, new\rangle$ **in**
`twice` $\delta\langle \mathtt{x}, \mathtt{y}\rangle$ `=` $\delta\langle \mathtt{x}, \mathtt{y}\rangle$`+`$\delta\langle \mathtt{x}, \mathtt{y}\rangle$
`test =` $\delta\langle 5, 7\rangle$

We can see that all changes are lumped together in the global δ dimension. A simple program analysis could detect that the three variable renamings belong together, and this could be exploited to generate a distinct dimension to group them together.

dim $\delta\langle old, new\rangle$ **in**
dim $\rho\langle x, y\rangle$ **in**
`twice` $\rho\langle \mathtt{x}, \mathtt{y}\rangle$ `=` $\rho\langle \mathtt{x}, \mathtt{y}\rangle$`+`$\rho\langle \mathtt{x}, \mathtt{y}\rangle$
`test =` $\delta\langle 5, 7\rangle$

In this representation all the changes pertaining to the renaming are grouped together into one dimension while all other changes are left unaffected.

To capture the renaming of a function parameter by a choice calculus pattern we have to denote all individual renamings in the function body. This can be achieved by employing a special kind of context that contains a varying number of holes, which are all filled by the same expression. We write $C^*[e]$ for such a "forall context", and we can use $C^*[\rho\langle \mathtt{x}, \mathtt{y}\rangle]$ for the pattern of the function body to capture all the renamings in the body. The choice calculus pattern for characterizing a renaming could then be given as follows.

Change	*Choice Calculus Pattern*
renaming	**dim** $\rho\langle x, y\rangle$ **in** `f` $\rho\langle \mathtt{x}, \mathtt{y}\rangle$ `=` $C^*[\rho\langle \mathtt{x}, \mathtt{y}\rangle]$

The identification of complex changes in the form of choice calculus patterns can be exploited on the user-interface level, and we can envision a tool that instead of showing all individual changes simply reports the renaming. A GUI could, for example, mark the definition of the variable (here the parameter) and show on a mouse-over event a tooltip message saying that this variable has been renamed to `y` (or, dually, when we show the new name, that it previously had the name `x`). The choice calculus patterns can also serve as high-level update scripts or programs, as discussed in [9].

We can thus see that the variation representation can be used as glue between all kinds of `diff` algorithms and tools on one end and change reasoning components and user interfaces on the other end. The algorithms can produce a collection of raw changes in the form of δ choices. A reasoning component can then identify patterns, such as swapping or renaming, and group changes into different dimensions. Finally, a GUI or some other tools can present or report changes, based on their form, in the most intuitive way to programmers.

The variation representation can also be used for higher-order change descriptions. Consider, for example, the case when a program P was edited in parallel by two programmers who produced the two changed programs Q and R. As before we can represent the changes between P and each of the new versions by two variation programs $\overline{PQ}$ and $\overline{PR}$, respectively. If somebody wants to continue to work on the program, they have to decide which changes of either programmer to adopt or to ignore. Representing the differences

[3] Note that we can apply the FACTOR rule (from right to left) and obtain the equivalent patterns $\delta\langle C[\mathtt{e}, \mathtt{e'}], C[\mathtt{e'}, \mathtt{e}]\rangle$ and $\delta\langle C[\mathtt{e}, \varepsilon], C[\varepsilon, \mathtt{e}]\rangle$, respectively, which provide an alternative, more transactional perspective on those changes.

between $\overline{PQ}$ and $\overline{PR}$ as a second-order variation program $\overline{\overline{PQ}\overline{PR}}$ can easily reveal which changes are in conflict, which are complementary, or identical. Consider the following example programs and the changes between them.

$$\begin{array}{ll} P = \texttt{a b c} & \overline{PQ} = \delta\langle \texttt{a},\texttt{x}\rangle\ \delta\langle \texttt{b},\texttt{y}\rangle\ \texttt{c} \\ Q = \texttt{x y c} & \overline{PR} = \delta\langle \texttt{a},\texttt{x}\rangle\ \delta\langle \texttt{b},\texttt{z}\rangle\ \texttt{c} \\ R = \texttt{x z c} & \overline{QR} = \texttt{x}\ \delta\langle \texttt{y},\texttt{z}\rangle\ \texttt{c} \end{array}$$

Note that $\overline{\overline{PQ}\overline{PR}}$ is different from $\overline{QR}$, which simply shows the differences between the plain programs Q and R and not the differences between the changes.

$$\overline{\overline{PQ}\overline{PR}} = \delta\langle \texttt{a},\texttt{x}\rangle\ \delta^2\langle \delta\langle \texttt{b},\texttt{y}\rangle, \delta\langle \texttt{b},\texttt{z}\rangle\rangle\ \texttt{c}$$

The choice calculus provides a rule that allows the regrouping of nested choices [10]. We can apply it here to the second choice of $\overline{\overline{PQ}\overline{PR}}$, which then yields the expression:

$$\overline{\overline{PQ}\overline{PR}} = \delta\langle \texttt{a},\texttt{x}\rangle\ \delta\langle \texttt{b}, \delta^2\langle \texttt{y},\texttt{z}\rangle\rangle\ \texttt{c}$$

This second-order change pattern gives a very detailed and precise account of the changes that happened: It shows that `c` was not changed in either program, that `a` was changed to `x` in both, and that both programs differ in how `b` changed, namely in Q it was changed to `y`, and R it was changed to `z`. Note that the difference between Q and R can be obtained from $\overline{\overline{PQ}\overline{PR}}$ by tag selection, that is, $\overline{QR} = \lfloor \overline{\overline{PQ}\overline{PR}} \rfloor_{new}$.

Summarizing, the choice calculus provides a structured way to represent program changes as variations. In particular, dimensions can be employed to group sets of small changes into complex ones, and applying the variation notation to itself leads to second-order change patterns that can support the reasoning about change conflicts to support the merging of changes.

4.4 Property Preservation

A variation program represents a potentially huge number of plain programs. Since the annotation schema provided by the choice calculus operates on the level of abstract syntax trees, the syntactic correctness of all programs can be automatically maintained across variations.[4] However, the question of type correctness for all programs is quite different. The approach of constantly re-running the type checker on all program variants is not scalable since the number of different programs that can be represented in a variation program can be enormous. For example, a variation program that contains n independent dimensions of k tags each represents k^n plain programs.

This problem has been investigated before for Java [15], and we consider it here in the context of the variation representation offered by the choice calculus. The interesting challenge is to exploit the structured sharing that is inherent in the choice calculus representation to obtain a more efficient approach to type checking.

To illustrate this problem, let us consider a typical typing judgment $\Gamma \vdash e : T$ that says that expression e is of type T within the context of the type assumptions Γ. The first thing to do is to extend the context of the typing judgment by a new parameter Δ that provides information about the context of the dimension declarations. Specifically, we associate each defined dimension with its size (the number of its tags).

Since we want the extended type system to be a conservative extension of existing ones, we next start to reformulate typing rules for the object language using the extended judgment. In many cases this will require only the use of the extended typing judgment.

[4] The situation is quite different for CPP, which is based on the linear textual representation of programs and which cannot guarantee any syntactical correctness.

The interesting questions arise when we consider what the typing rules should be for choices or dimensions. Consider, for example, the choice $e = A\langle 3, \texttt{True}\rangle$. Since there is no one single type T that we could assign to this choice, it seems we have two options for defining the type of choices like e. First, we could consider e to be type incorrect. However, that would be overly restrictive since, as we have seen in Section 2, variations in types and expressions that have this type seems to be an essential element of exploratory variations. The second option is to extend the set of types to include variations. With this concept, we can assign the type $A\langle \texttt{Int}, \texttt{Bool}\rangle$ to e. We can capture this idea in the following typing rule.

$$\textsc{Choice}\quad \frac{\Delta(D) = n \qquad \Delta,\Gamma \vdash e_1 : T_1 \quad \cdots \quad \Delta,\Gamma \vdash e_n : T_n}{\Delta,\Gamma \vdash D\langle e_1,\ldots,e_n\rangle : D\langle T_1,\ldots,T_n\rangle}$$

The first premise ensures that the choice contains the correct number of alternatives while the remaining premises verify that all alternatives are well typed individually.

Do we really need to add the dimension name to the inferred choice type? To understand the need for dimension names, compare the expression $A\langle \texttt{succ}, \texttt{not}\rangle\ e$, which is type correct, with the expression $B\langle \texttt{succ}, \texttt{not}\rangle\ e$, which isn't. The difference lies in the fact that both choices are synchronized in the first expression, which ensures that whenever `succ` is selected, it will be applied to `3` (and correspondingly for `not` and `True`), whereas the choices in the second expression are in different dimensions and are therefore unrelated, which means we can produce, for example, the variation `not 3`, which is not type correct. If we didn't represent the dimension names on the type level, we couldn't distinguish between the two cases.

Since the dimension names that appear in the types are scoped, we have to include a dimension binding construct on the type level as well. The typing rule for dimensions is rather straightforward: It extends the dimension environment with the size information for the dimension and then type checks the scope of the declaration.

$$\textsc{Dim}\quad \frac{(\Delta,(D,n)),\Gamma \vdash e : T}{\Delta,\Gamma \vdash \textbf{dim}\ D\langle t_1,\ldots,t_n\rangle\ \textbf{in}\ e : \textbf{dim}\ D\langle t_1,\ldots,t_n\rangle\ \textbf{in}\ T}$$

These two rules seem to illustrate that the type checking effort for this type system is not multiplicative in the sizes of dimensions, but rather additive, which is exactly the goal we started out with. However, the challenge does not lie in the two shown rules, but in the typing rule for function application, which is the place where the combination of types across different dimensions must be considered.

The traditional typing rule for function application requires that the expression that is applied has a function type, but this is too rigid in the presence of choice types, because it would prevent the obviously type-correct expression $A\langle \texttt{succ}, \texttt{even}\rangle\ \texttt{3}$ from passing type checking. This is because the type of the choice expression is required to be `Int -> T`, whereas it has in fact the following choice type.

$$\Delta,\Gamma \vdash A\langle \texttt{succ}, \texttt{even}\rangle : A\langle \texttt{Int -> Int}, \texttt{Int -> Bool}\rangle$$

We therefore need to relax the condition so that, for example, choices of function types can also be accepted in applications. We do this by requiring that the applied expression have a type that is *equivalent* to a function type.

$$\textsc{App}\quad \frac{\Delta,\Gamma \vdash e : T'' \qquad \Delta,\Gamma \vdash e' : T' \qquad T'' \equiv T' \to T}{\Delta,\Gamma \vdash e\ e' : T}$$

Much of the type systems strength and flexibility depends on the definition of $\equiv$, and the efficiency of the type checking process requires a generalization of unification that can efficiently unify

types modulo $\equiv$. It turns out that $\equiv$ is generically defined and is valid for any object language instance of the choice calculus, which means that unification, normalization algorithms, etc. can be reused in different application areas.

In our example the rule FACTOR that commutes choices with syntactic structure gives rise to the following equivalence.

$$A\langle \texttt{Int -> Int}, \texttt{Int -> Bool}\rangle \equiv \texttt{Int -> } A\langle \texttt{Int}, \texttt{Bool}\rangle$$

Now the APP rule can be applied to our example to derive the following type.

$$\Delta, \Gamma \vdash A\langle \texttt{succ}, \texttt{even}\rangle \ \ \texttt{3} : A\langle \texttt{Int}, \texttt{Bool}\rangle$$

We observe that the notion of variation and dimension types mirrors the corresponding concepts in choice calculus expressions. As the rules CHOICE and DIM illustrate, the choice calculus helps to structure the variation aspect of both the expression language and the type language in a systematic way that supports the formulation of typing rules and that suggests a standardized approach to investigating the preservation of properties under variation.

4.5 Variation Maintenance

The variation annotations of programs add a layer of complexity on top of the program representation that makes reading, understanding, and editing programs more difficult. It is therefore important to develop tool support that can provide all the benefits of the variation annotations without adding too much burden to the programming activity. Currently, programs such as `diff`, program editors, and IDEs, serve that function. However, with increasing degree of variability in program representation, more sophisticated tools and support environments are needed to manage the additional complexity. For example, while `diff` can report the raw differences between two programs, its output is often too low level and requires non-trivial interpretation to make sense. As I have argued in Section 4.3, the choice calculus can support the advancement of tools, such as `diff`, to produce more sophisticated high-level explanations of program changes.

Reporting and explaining changes between program versions is just one aspect of the more general task of maintaining software variations. In addition, we need to be able to add new variations or change existing ones. Moreover, given a variation repository, we would like to be able to navigate among variations and express queries. We can envision a whole new class of query and transformation languages for the choice calculus that process variation representations for various purposes, much like XQuery or XSLT for XML.

Since most editing of programs probably happens by using some editor or IDE and is based on the textual representation of programs, the first step is to find a textual representation of variation programs that supports editing operations well. One possible approach is illustrated in the IDE mockup shown in Figure 1. Dimensions and their tags are shown color-coded on the left of the editing window. Each dimension contains a "current choice" of one of its tags, which selects a current variation to be displayed in the main window. In the example, it is the program obtained by selecting *long* and *adj*. All parts of the program that do not vary across the different versions are shown on white background, whereas those parts that do vary are shown with a background color corresponding to the dimension in which the variation occurs. For example, `Node` varies in the *Name* dimension, and `head` and `ws` vary in the *Rep* dimension. The fact that the type `[Node]` varies on both dimensions is indicated by nested colors.

While such an editor representation inherently shows only one version, it is vital to make the context of that current version clear (as it is done here through the dimension sidebar) and also support the easy transition between variants and, in particular, the

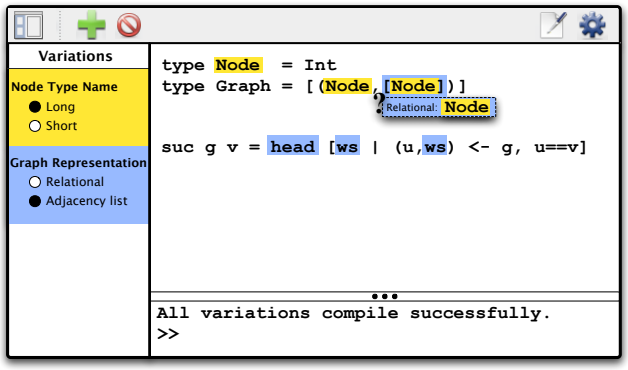

Figure 1. Variation editor mockup.

exploration of alternatives in the code. In Figure 1 this is done by interpreting mouse-over events to shows alternatives for the pointed-at code.

This envisioned user interface is very similar to the CIDE tool [16], which allows the association of parts of code with features of a feature model. In CIDE, features are represented by colors, and code implementing a feature is shown using the feature's color as background color. However, CIDE can only represent optional code and not alternatives.

While this small example looks quite nice, it is not at all clear whether this visual representation scales well. In particular, the nested coloring seems to break down quickly when more and more independent dimensions come into play. Certainly, more HCI research with user studies is needed to find an effective representation. The purpose of Figure 1 is not to advocate this particular design, but rather to illustrate some important issues of variation editing that have to be addressed.

For example, suppose we change in the editing window the name `suc` to `successor`. Which variations does this change affect? Does this mean to change the name in all variations, or only in the currently selected one? In the latter case, the changed text would need to be displayed using the nested blue/yellow background color. A third possible interpretation is to create a new variation, much like we have done for the names `Node` and `N`. In that case, a further question is, whether we also want to create a new dimension or whether this variation should rather be be part of the already existing *Name* dimension. It seems we need to equip editing actions with more details to specify their intended scope and meaning. This will certainly complicate the user interface, and it is not obvious what the best design is. In any case, the choice calculus representation can serve as a guide in investigating these options. Enumerating all the possible ways to represent the changed program using the choice calculus provides examples for different editing actions that may or may not become part of a user interface.

A different and potentially more user-friendly approach to the editing of variations would be to not require up front any additional specification from the user for editing actions, except for marking the beginning and the end of an editing transaction. Then simply take an edited piece of text and merge it "intelligently" into the existing choice calculus expression. Of course, an automatic merging algorithm cannot guess the right decisions in cases where changes can be ambiguously attributed to different dimensions. Therefore some form of user interaction is required to accomplish the merging properly according to the intent of the user. But doing this in a demand-driven way, while exploiting clues from other changes that have been performed as part of the same transaction, might make this approach on the whole less burdensome. This problem is very similar to the "view update" problem in relational databases [6],

which also has become popular in the programming language field under the name of "bidirectional programming" [12].

The design of editors and other variation maintenance tools is supported by the choice calculus since it defines a clean interface for variation annotation to be used by editors. The succinct representation also encourages the search for basic editing principles for variations, which can be derived from a systematic investigation of update operations for choice calculus expressions.

5. Related Work

Languages for describing variations have been proposed in the different specific research areas of variation, such as feature modeling or configuration management, and they are too numerous to list and discuss them all here. Most of these proposals are domain specific in that they exploit the particular structures and requirements of their application or variation domain. This also applies to the many metaprogramming languages, which can also be used to express variation. In contrast, the choice calculus provides a generic representation for variation that is applicable, in principle, in any domain. Therefore, most closely related to the choice calculus are language proposals that are, in principle, agnostic about the language or domain for which they describe variation.

Probably the most well known variation description language is provided by the C Preprocessor (CPP) [13]. It is also the most widely used tool for annotating variations [7]. In addition to its capabilities to define and use macros, CPP offers a set of directives, such as `#ifdef`, to mark parts of a document and to conditionally include or exclude these parts based on given definitions of macros. While CPP is extremely simple and versatile, it also lacks structure, which makes the maintenance of CPP-annotated programs difficult and prone to errors. The two major differences between CPP and the choice calculus are this. First, CPP works on the textual representation of a language and not its abstract syntax, which means that we can easily represent syntactically incorrect programs with CPP. In contrast, the choice calculus guarantees syntactic correctness of the annotated object language. Second, CPP provides no means to represent relationships or constraints among macros used for describing variations, which leads to a flat decision structure that basically consists of a huge unstructured set of choices. In contrast, using the choice calculus we can structure variations along dependent and independent dimensions.

Another closely related approach is CIDE [16], which is also based on code annotations and also operates on abstract syntax. However, CIDE's underlying annotation (which seems to be based on a mapping from XPath-like pointers into the code to sets of features) is used only internally and is not formalized and defined for use by other tools.

6. Conclusions

Variation plays an important role in the software development process. The choice calculus is a well formalized language for representing and reasoning about variation that enjoys many useful properties and shows promise for a wide range of applications. The choice calculus is intended to be a new tool to describe and solve problems in all areas of variation research.

Acknowledgments

I would like to thank the organizers for inviting me to give the 2010 joint GPCE and SLE keynote address and write this paper.

Special thanks go to Eric Walkingshaw, the co-designer of the choice calculus, who provided me with detailed comments on an earlier version of this paper.

This work is supported by the Air Force Office of Scientific Research under the grant FA9550-09-1-0229 and by the National Science Foundation under the grant CCF-0917092.

References

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

[2] D. Batory. Feature Models, Grammars, and Propositional Formulas. In *Int. Software Product Line Conf.*, LNCS 3714, pages 7–20, 2005.

[3] D. Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. on Software Engineering*, 30(6):355–371, 2004.

[4] K. Czarnecki and U. Eisenecker. *Generative Programming: Method, Tools, and Applications*. Addison-Wesley, 2000.

[5] C. J. Date. *Database in Depth: Relational Theory for Practitioners*. O'Reilly Media, Inc., 2005.

[6] 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.

[7] M. D. Ernst, G. J. Badros, and D. Notkin. An Empirical Analysis of C Preprocessor Use. *IEEE Trans. on Software Engineering*, 28(12):1146–1170, 2002.

[8] M. Erwig and D. Le. Supporting Feature Modeling with the Choice Calculus. 2010. In preparation.

[9] M. Erwig and D. Ren. An Update Calculus for Expressing Type-Safe Program Updates. *Science of Computer Programming*, 67(2-3):199–222, 2007.

[10] M. Erwig and E. Walkingshaw. The Choice Calculus: A Representation for Software Variation. *ACM Trans. on Software Engineering and Methodology*, 2010. To appear.

[11] J. Estublier, D. Leblang, A. van der Hoek, R. Conradi, G. Clemm, W. Tichy, and D. Wiborg-Weber. Impact of Software Engineering Research on the Practice of Software Configuration Management. *ACM Trans. on Software Engineering and Methodology*, 14(4):383–430, 2005.

[12] 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):17, 2007.

[13] GNU Project. *The C Preprocessor*. Free Software Foundation, 2009. `http://gcc.gnu.org/onlinedocs/cpp/`.

[14] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[15] C. Kästner and S. Apel. Type Checking Software Product Lines—A Formal Approach. In *IEEE Int. Conf. on Automated Software Engineering*, pages 258–267, 2008.

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

[17] M. Kim. *Analyzing and Inferring the Structure of Code Changes*. PhD thesis, University of Washington, 2008.

[18] H. Ossher and P. Tarr. Hyper/J: Multi-Dimensional Separation of Concerns for Java. In *IEEE Int. Conf. on Software Engineering*, pages 734–737, 2000.

[19] D. L. Parnas. On the Design and Development of Program Families. *IEEE Trans. on Software Engineering*, 2(1):1–9, 1976.

[20] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer-Verlang, Berlin Heidelberg, 2005.

[21] W. F. Tichy. Design, Implementation, and Evaluation of a Revision Control System. *IEEE Int. Conf. on Software Engineering*, pages 58–67, 1982.