The Choice Calculus: A Representation for Software Variation

MARTIN ERWIG and ERIC WALKINGSHAW
Oregon State University

Many areas of computer science are concerned with some form of variation in software—from managing changes to software over time, to supporting families of related artifacts. We present 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 the lambda calculus in programming language research. We also develop an associated theory of software variation, including sound transformations of variation artifacts, the definition of strategic normal forms, and a design theory for variation structures, which will support the development of better algorithms and tools.

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

Additional Key Words and Phrases: Variation, Representation

1. INTRODUCTION

Effectively dealing with variation is a fundamental problem in software engineering that emerges in many different ways throughout the field. As such, equally many tools have been developed for managing variation. Virtually all non-disposable software projects use some sort of revision control system for managing variation in software over time, and tools like the C Preprocessor and software-product-line systems manage variation in many other dimensions. Each of these tools is a concrete solution to a particular view of this common problem, each with its own way of indicating which parts of a system vary, and how a particular variant is produced. While the diversity of variation representations is not inherently a problem, it would be nice if advances in one line of research could be easily understood and incorporated in others. Additionally, by focusing on domain-specific applications, researchers may miss insights that can only be gained by examining the problem from a broader, more abstract point of view.

In this paper we develop the *choice calculus*, a language that provides a general representation for software variation, intended to support research in variation man-

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.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 20YY ACM 1049-331X/20YY/0700-0001 $5.00

agement, and to provide a foundation for variation management tools with widely varying goals. We provide a semantics for this representation and a set of semantics-preserving transformations which will be of use in tools. We also identify useful normal forms and a set of quality criteria for choice calculus expressions, which together provide a foundation for a robust variation design theory.

Explicitly, our goals for this work are as follows:

—To provide a common language of discourse, the choice calculus, for software variation management, to facilitate the sharing of ideas and tools between fields. Ideally, this would fill a role similar to that of the lambda calculus for the representation and discussion of programming languages and language features.

—To provide a theoretical foundation on which tools and future research can be built. Lowering the barrier to entry and providing solutions for the mundane details common to all variation management problems allows subsequent researchers to focus only on the parts most relevant to their problem.

—To develop theoretical results that can inform existing research, lead to the development of provably correct operations on variational artifacts, and form the basis of a variation design theory.

—To reveal previously unexplored areas of research by approaching the problem of software variation from first principles. While solving a concrete problem has the benefit of being immediately useful, a purely theoretical perspective can expose underlying patterns and uncovered territory.

To provide a theoretical foundation for diverse variation management research, a representation must have several qualities; these will mostly be introduced throughout the next two sections. The most fundamental qualities, however, follow directly from the above goals. In particular, the representation should be a *formal* and *general* representation of software variation. In Section 4 we provide the requisite formality with precise definitions of both the syntax and semantics of choice calculus expressions. Generality is a more subjective quality, but is often supported in representations with a small number of simple, but highly composable fundamental elements. The lambda calculus, for example, has only three constructs and is highly recursive; we have emulated this approach in the choice calculus.

In the next section we will provide a brief overview of variation management research, touching on the strengths and weaknesses of some existing approaches, and establishing a frame of reference for the rest of the paper. Section 3 incrementally introduces and motivates the choice calculus, while Section 4 provides the formal syntax and semantics. In Section 5, we present several semantics-preserving transformations of choice calculus expressions, and we identify theoretically important normal forms in Section 6. In Section 7 we provide formal quality criteria for choice calculus expressions, and transformations for improving them. We provide a deeper comparison with related work in Section 8, discuss future work in Section 9, and offer conclusions in Section 10.

2. BACKGROUND

Existing work on variation management can be generally split into two categories. The first is primarily concerned with managing program variation over time and includes *revision control systems* [Tichy 1982] and the larger field of *software configuration management* (SCM), an overview of which can be found in [Estublier et al. 2005]. There have been several efforts to capture general principles of SCM, for example through the creation of taxonomies [Buckley et al. 2004] and metamodels [Westfechtel et al. 2001]. While these works strive for generality by examining many different existing systems, we provide a different view by approaching the problem from first principles.

In this paper, we focus more on the second category of existing work: managing software variation in multiple non-temporal dimensions. SCM systems often provide support for this type of variation as well, through branching features. However, branches are a highly *redundant* representation—code and data common to multiple branches are usually duplicated across each branch. Besides being inefficient, this often forces users to manually copy changes in one branch to other, related branches. This significantly increases the costs of changes and creates a high potential for the introduction of errors. An ideal variation representation should thus seek to *minimize redundancy*. The management of branching problems in SCM is largely institutional rather than technical; branching structures and change patterns must adhere to one of many sets of best practices to be manageable [Wingerd and Seiwald 1998; Walrad and Strom 2002].

The C Preprocessor (CPP) [GNU Project 2009] is the most widely used tool for larger-scale, multi-dimensional variation management [Ernst et al. 2002]. Although CPP can do other things, like macro definition and expansion, we are interested only in its ability to conditionally include parts of files. CPP provides a set of directives for this purpose—`#if`, `#elif`, `#else`, etc.—which combine to form conditional statements in the obvious way. Text between these directives is then included or not depending on the settings of various macros. The biggest advantage of CPP is that it is very general. Despite the name, CPP is almost completely indifferent to the type of underlying artifact—it must simply be a text file and not contain text that resembles CPP syntax. Another advantage is that CPP can capture very fine-grained variation, since any text that can be isolated on a line can be conditionally included.

However, CPP is also a very *unstructured* language that leads to code which is difficult to read and edit, and is often a source of errors [Spencer and Collyer 1992]. The ability to vary practically any piece of text can easily lead to situations where only some variants of a CPP-annotated program even compile (e.g. one could conditionally include the closing brace of a procedure definition in C). Worse, these situations cannot be easily detected without additional tool support—one must explicitly generate a faulty variant and then attempt to compile it.

Another major deficiency of CPP is that it does not capture the relationships and constraints that exist between macros used in conditional compilation. To demonstrate this, in a very simple case study we looked at the conditional compilation structure of the source code for the open-source MySQL database.[1] Considering only the 557 C source code files (i.e. disregarding header files, which often use conditional compilation for non-variational purposes), we found 938 unique CPP macros used in conditional compilation directives. Even if we assume only two possible values for each, "defined" or "undefined", this leads to 2^{938} potential variants (roughly the

[1] `http://mysql.com`

number of atoms in the universe to the fourth power). Clearly, only a tiny subset of these represent unique programs, and only a smaller subset of these are valid. But determining which settings of macros correspond to valid variants is extremely difficult to derive from the CPP representation.

More rigorous approaches to multi-dimensional variation management have been developed in the context of *software product lines* (SPLs) [Parnas 1976; Pohl et al. 2005]. Very generally, a SPL is a collection of related programs generated from a common set of resources. Most technical research on SPLs is focused on the representation and management of *features*, where a feature is some piece of functionality which can be included or not in a program. A *feature model* is a way of describing relationships and constraints that exist between features in a product line. Feature models address the variant explosion problem described in CPP-annotated programs above; only certain combinations of features produce valid programs, and these sets are defined by the feature model. Feature models can be expressed as diagrams [Kang et al. 1990], algebras [Höfner et al. 2006], propositional formulas [Batory 2005], and more. Section 8 compares these approaches with each other and the choice calculus.

SPL systems vary widely in their representation of features. *Feature-oriented programming* (FOP) systems like AHEAD [Batory et al. 2004] and Hyper/J [Ossher and Tarr 2000] are rooted in object-oriented programming and represent a feature by a set of classes, subclasses, and mixins [Bracha and Cook 1990] which can be added or not to some base program. This approach is significantly more structured than CPP, but obviously captures only much coarser-grained variation.

The graphical CIDE tool [Kästner et al. 2008] is closer in spirit to CPP, allowing snippets of code to be highlighted and associated with a particular feature. When a feature is included in a program, all of the associated pieces of code are included, and when a feature is excluded its associated code is as well. CIDE limits these optional pieces of code to syntactically optional elements (e.g. a statement in a statement block, or a method in a class definition). Code and most other types of artifacts that make up software are inherently structured, and CIDE works with this structure rather than discarding it as CPP does. This approach allows it to capture finer-grained variation than FOP systems, while avoiding some of the pitfalls of CPP. Of course, one must have a syntax definition for each type of artifact under variation management in order to get these benefits (or fall back on the line-based approach otherwise).

Most existing SPL research seems to consider feature representation and feature modeling to be two fundamentally different problems, and Section 8 discusses several more approaches to each. In contrast, the choice calculus is a more integrated approach. The concepts of features and feature models are replaced by the concept of choices—rather than optionally including some piece of code, one chooses between alternative pieces of code. This distinction is subtle but significant. For example, using the choice calculus, one can lift the constraint that variation in CIDE be limited to optional syntactic elements; instead, any syntactic element can be varied by simply requiring that every alternative in a choice be in the same syntactic category.[2]

All of the work discussed so far is focused on *long-term* variation; that is, variation

[2]Recent work on CIDE has begun to extend it in this direction also [Rosenthal 2009].

which is planned and intended to be maintained indefinitely. However, there is a second category of *short-term* variation which is currently very poorly supported by tools despite evidence that it is needed [Szekely et al. 1992]. Short-term variation represents an exploration of alternatives. When faced with a design decision which would otherwise force a premature commitment—for example, choosing a data structure when the operations required of it are not yet known—short-term variation allows the user to make a tentative decision and add alternatives as they present themselves. The impact of these alternatives can then be analyzed before a final decision is made and the variation is removed. A general model of variation can lead to tools which better support this exploration process, making the contemplation of alternatives easy and providing fundamental support for decision making processes.

Finally, we also want to support the development of tools that assist *understanding* and *reasoning* about variational structures. This influences the representation's design in several ways. For example, we have stated that minimizing redundancy is a primary goal of a good representation, but from a tool user's perspective, it may be more helpful to see both implementations of a short method with two variants side-by-side than it would be to see a combined, redundancy-free representation. These sorts of tradeoffs between technical goals and usability are common and suggest that, more important than, for example, supporting or minimizing redundancy, is that a representation be highly *flexible*. That is, it should be able to represent the same set of variations in different ways, and provide transformations for moving between forms that are desirable for different reasons. Such transformations for the choice calculus are presented in Section 5. In the next section, we walk through and motivate the basic design of the choice calculus.

3. DESIGN OF A VARIATION REPRESENTATION

In this section we will derive the choice calculus by considering the qualities of a good representation of software variation (identified in the previous two sections), then by incrementally manipulating our representation to maximize these qualities. Although we explored many more branches of the design-decision tree and many other intermediate representations, this section illustrates a sort of idealized design process to help motivate the final representation. But first, in order to talk more concretely about the representation, it will help to have an example and establish some terminology.

Consider the following four implementations of the function `twice` which takes a numerical argument and returns that value doubled.

		Implementation	
		plus	*times*
Name	*x*	`int twice(int x) {` `return x+x;` `}`	`int twice(int x) {` `return 2*x;` `}`
	y	`int twice(int y) {` `return y+y;` `}`	`int twice(int y) {` `return 2*y;` `}`

These functions vary in two independent *dimensions* with two *options* each. The functions in the top row differ from the bottom row in the choice of parameter name, with options "*x*" and "*y*". The functions in the left column column differ from the right column in the choice of implementation method, with options "*plus*" and "*times*".

Together, these two dimensions with two options produce four *variants* of the program. If we were to add a third option, "*z*", to the parameter name dimension, we would have six total variants. If we were to then add a new dimension, for example, "function name" with options "*twice*" and "*double*", we would have twelve total variants. Clearly, the number of variants grows multiplicatively with respect to the number of options in each independent dimension. Together with the MySQL/CPP case study in Section 2, this suggests that another quality of a successful representation is that it be highly *scalable*. As the number of dimensions in a varying piece of software increases, the number of variants grows very quickly, making it infeasible to consider every possible variant. Of critical importance to managing this complexity is representational *modularity*. One must be able to work on and integrate a part of the structure without considering the whole.

The goal of our representation is to provide a formal model to represent variation in all kinds of languages and documents. To this end, we employ a simple tree model to represent the underlying artifact. This allows us to focus on the variational aspects of the representation, while providing a *general* and *structured* model to build on. Therefore, we define that an expression is given by some constant information a and a possibly empty list of subexpressions, written as $a{\prec}e_1,\ldots,e_n{\succ}$.[3] For example, the first variant of the `twice` function above could be represented by the following (simplified abstract syntax) tree.

$$\texttt{twice}{\prec}\texttt{int},{\prec}\texttt{int},\texttt{x}{\succ},{\prec}\texttt{return}{\prec}\texttt{+}{\prec}\texttt{x},\texttt{x}{\succ}{\succ}{\succ}{\succ}$$

In the following we rarely show this tree structure explicitly, instead using concrete syntax whenever there is no danger of ambiguity.

The fundamental concept of our representation is that of *choice*. A choice is a set of *alternative* expressions from which one can be selected. This selection can be facilitated by associating *tags* with each alternative expression in the choice; one then selects a tag to replace a choice with the corresponding alternative. There are (at least[4]) two ways to implement this association of alternatives and tags.

(1) A choice is represented as a mapping from tags to alternative expressions. We call this the *direct tagging* approach.
(2) The introduction of tags and their association with expressions are separated. Since the separate introduction of the tags amounts to the definition of a dimension of variation, we call this the *dimension* approach.

Representing the variation of the parameter name in the definition of the `twice` function using direct tagging would look as follows. We write sets of alternatives as mappings from tags to code alternatives. Such a set is called a *choice*.

```
int twice(int ⟨x: x, y: y⟩) {
    return 2*⟨x: x, y: y⟩;
}
```

[3]The term "expression" includes all kinds of tree-structured documents.
[4]We will mention briefly a few other approaches in Section 10.

To represent both dimensions of variation, name and implementation method, we could use nested choices in the body of the function, as shown below.

```
int twice(int ⟨x: x,y: y⟩) {
    return ⟨plus: ⟨x: x+x,y: y+y⟩,times: ⟨x: 2*x,y: 2*y⟩⟩;
}
```

Unfortunately, this representation results in a lot of redundant code. For example, the code associated with each implementation method (+ and 2*) is represented twice each. In fact, had we not only two but 17 parameter name options, we would need 17 copies of the code for each implementation method. Changing the nesting of choices can change the number of choices involved,[5] but has no effect on the amount of redundant code. Neither dimension, naming or implementation method, can be extended or changed independently of the other. For example, if we decide to add a new parameter name, we are forced to extend *both* implementation alternatives. Similarly, if we want to add a new implementation method, we have to add it twice (or 17 times), once for each naming variant.

Such redundancy in the representation can easily lead to "update anomalies" (a term used in the database field to motivate and explain normalization of schemas [Date 2005]). If we want to change the name `x` to `z`, we have to do it in both implementation alternatives. Or if we want to change one implementation method, say change `x+x` to `plus(x,x)`, we have to do it for all parameter name alternatives. It is very easy to forget such a change or do it inconsistently. We can avoid the redundant representation by factoring common parts using a **let** construct. For example, the `twice` function can be represented as follows.

```
let v=⟨x: x,y: y⟩ in
int twice(int v) {
    return ⟨plus: v+v,times: 2*v⟩;
}
```

This representation can be easily extended by a new name or a new implementation method. Moreover, the example doesn't suffer from update anomalies anymore. For example, we can easily rename `x` to `z` in only one place.

Although the direct tagging approach is very simple and flexible, it has a few significant shortcomings. To demonstrate these, suppose we add the function `thrice` to our program. In the following, whenever we show two expressions in sequence, it is assumed that they are children of a containing structure node, according to the object language. For example, two expressions representing Java statements would be children of a structure node representing a statement-block and two expressions representing methods would be children of a class node. Our program with the added `thrice` function follows.

```
let v=⟨x: x,y: y⟩ in
int twice(int v) {
    return ⟨plus: v+v,times: 2*v⟩;
}
```

[5]An effect known as the "tyranny of the dominant decomposition" [Tarr et al. 1999].

```
let v=⟨x: x, y: y⟩ in
int thrice(int v) {
    return ⟨times: 3*v⟩;
}
```

If we assume that tag selection is applied globally, there are a couple of things to note about this program. First, notice that the implementation of `thrice` is missing the *plus* option. Selecting the *plus* tag will therefore leave the implementation of `thrice` unresolved, while the selection of *times* will perform a selection in the implementation of both `twice` and `thrice`. This construction seems odd; since *plus* and *times* are related (that is, they are in the same dimension), we would expect them to have to appear together in choices, but there is nothing in the syntax to enforce this. This demonstrates that the direct tagging approach is too *unstructured*. It allows the creation of choices which are not fully defined with respect to some dimension.

The second thing to note is that, since tags are globally defined, it is not possible to choose the x parameter name for one function and the y for the other. Perhaps this is what we want for this small example (to select x or y only once and have it apply to both functions), but if not, the only resolution is to come up with new tag names for the parameters of every function we want to vary independently. Global tag names are an even bigger problem when tags from separate dimensions collide. This can cause the extremely unexpected result of a selection in one dimension affecting choices in another dimension. These problems demonstrate that the direct tagging approach is *not modular*.

As a solution to both of these problems, we introduce constructs for explicit, local dimension declarations. We also modify the syntax for choices so that each choice has an associated dimension and must be fully specified in terms of that dimension. Dimensions, therefore, are similar to a type system for choices. Below are the `twice` and `thrice` examples from above encoded with explicit dimension declarations.

```
dim Par⟨x, y⟩ in
dim Impl⟨plus, times⟩ in
let v=Par⟨x, y⟩ in
int twice(int v) {
    return Impl⟨v+v, 2*v⟩;
}
let v=Par⟨x, y⟩ in
int thrice(int v) {
    return Impl⟨v+v+v, 3*v⟩;
}
```

In this syntax, each choice has a dimension name associated with it, and the number of elements in the choice must be equal to the number of tags in the corresponding dimension declaration. This means that omitted alternatives, as in our original definition of `thrice`, are syntactic errors.

In this example, the selection of a parameter name will be applied to both functions. If we want to vary each function's parameter name independently, we can instead define two local dimension declarations as in the alternative program below.

```
dim Impl⟨plus,times⟩ in
let v=(dim Par⟨x,y⟩ in Par⟨x,y⟩) in
int twice(int v) {
    return  Impl⟨v+v,2*v⟩;
}
let v=(dim Par⟨x,y⟩ in Par⟨x,y⟩) in
int thrice(int v) {
    return  Impl⟨v+v+v,3*v⟩;
}
```

For the choice calculus we have adopted the second, *dimensioned* approach because it provides a more structured representation, is more modular, and obeys a richer set of laws.

4. THE CHOICE CALCULUS

In Section 4.1 we present the syntax of the choice calculus together with some conditions for well-formed choice expressions. Based on an operation for the systematic elimination of choices described in Section 4.2 we define the formal semantics of choice calculus in Section 4.3.

4.1 Syntax

The syntax of the choice calculus is based on four disjoint sets: (1) a set of symbols, a, to be stored in expressions, (2) a set of tags, t, to label alternatives of choices, (3) a set of dimension names, D, used to identify choice expressions, and (4) a set of variable names, v, to bind and refer to expressions.

$$\begin{array}{rcll}
e & ::= & a\prec e,\ldots,e\succ & \textit{Structure} \\
& | & \textbf{let}\ v{=}e\ \textbf{in}\ e & \textit{Binding} \\
& | & v & \textit{Reference} \\
& | & \textbf{dim}\ D\langle t,\ldots,t\rangle\ \textbf{in}\ e & \textit{Dimension} \\
& | & D\langle e,\ldots,e\rangle & \textit{Choice}
\end{array}$$

We require that a dimension declaration contain at least one tag and that all the tags in one dimension are pairwise different. We call a $D.t$ a *qualified tag*, and we use the metavariable q to range over qualified tags.

For expressions we use the standard definitions of free and bound variables, and $FV(e)$ is used to denote the set of free variables in expression e. In addition, we can define a similar notion for dimensions, which are bound by **dim** expressions and referenced by choices. Specifically, $FD(e)$ denotes the set of free dimension names contained in e.

$$\begin{aligned}
FD(a\prec e_1,\ldots,e_n\succ) &= FD(e_1)\cup\ldots\cup FD(e_n) \\
FD(\textbf{let}\ v{=}e\ \textbf{in}\ e') &= FD(e)\cup FD(e') \\
FD(v) &= \varnothing \\
FD(\textbf{dim}\ D\langle t_1,\ldots,t_n\rangle\ \textbf{in}\ e) &= FD(e)-\{D\} \\
FD(D\langle e_1,\ldots,e_n\rangle) &= \{D\}\cup FD(e_1)\cup\ldots\cup FD(e_n)
\end{aligned}$$

Similarly, we have a definition of the set of bound dimension names $BD(e)$. The definition is identical except for the last two cases.

$$BD(\textbf{dim}\ D\langle t_1,\ldots,t_n\rangle\ \textbf{in}\ e) = \{D\}$$
$$BD(D\langle e_1,\ldots,e_n\rangle) = BD(e_1)\cup\ldots\cup BD(e_n)$$

We call an expression e *well dimensioned* if each choice $D\langle e_1,\ldots,e_n\rangle$ is in scope of a corresponding dimension definition **dim** $D\langle t_1,\ldots,t_n\rangle$, that is, the dimension declaration that binds d introduces exactly as many tags as each choice that references d has alternatives. Thus, being well dimensioned is a stronger property than just having no unbound dimensions.

This property is expressed formally by the judgment $\Delta \vdash e$, which says that for each choice $D\langle e_1,\ldots,e_n\rangle$ used in e, $n = \Delta(D)$ where Δ is an environment that maps dimension names to numbers. We use Δ as a stack when extending it with new bindings, which means that when we use Δ as a function to look up entries, we start searching from the top of the stack. We use the notation $\Delta\oplus(D,n)$ to push the binding (D,n) onto the stack Δ.

The judgment is defined by the rule system given below in which we employ the following notational abbreviations for writing expressions involving sequences. A list of n expressions (or other syntactic elements) $e_1,\ldots,e_n$ is written as e^n. This notation can be generalized to allow for context around the enumerated elements: we write $C(e_i)^{i:1..n}$ for $C(e_1),\ldots,C(e_n)$. For example, $(x_i+1)^{i:1..n}$ represents $x_1+1,\ldots,x_n+1$.

$$\frac{(\Delta\vdash e_i)^{i:1..n}}{\Delta\vdash a\prec e^n\succ} \qquad \frac{\Delta\vdash e \qquad \Delta\vdash e'}{\Delta\vdash \textbf{let}\ v{=}e\ \textbf{in}\ e'} \qquad \Delta\vdash v$$

$$\frac{\Delta\oplus(D,n)\vdash e}{\Delta\vdash \textbf{dim}\ D\langle t^n\rangle\ \textbf{in}\ e} \qquad \frac{\Delta(D)=n \qquad (\Delta\vdash e_i)^{i:1..n}}{\Delta\vdash D\langle e^n\rangle}$$

We say that a choice calculus expression e is *well formed* if it is well dimensioned and contains no free variables, or more formally, if $\varnothing\vdash e$ and $FV(e)=\varnothing$.

Finally, an expression e is called *dimension linear* if all dimension names in e that are introduced by a **dim** construct are pairwise different. The rationale for this definition is that a dimension linear expression can be transformed into a useful normal form in which dimension declarations are maximally factored (see Section 6). Note that any expression can be made dimension linear by simply renaming conflicting dimensions and their bound choices.

4.2 Choice Elimination

Choice calculus expressions represent choices, and the process of choosing alternatives from these choices is called *choice elimination*. Sets of choices can be synchronized through the use of dimensions. An operation called *tag selection* formalizes choice elimination, allowing one to simultaneously choose alternatives in all choices in one dimension with a single tag. More specifically, when a qualified tag $D.t$ is selected, a choice $D\langle e_1,\ldots,e_n\rangle$ yields e_i if the choice is in scope of a dimension declaration **dim** $D\langle t_1,\ldots,t_n\rangle$ with $t=t_i$.

We write $\lfloor e\rfloor_{D.t}$ for selecting from choices in e with the qualified tag $D.t$. The semantics of tag selection is defined in two steps. First, we have to find a dimension definition for D that includes the tag t and determine t's index, say i, in that definition. After that we find all choices tagged by D in the scope of the identified dimension definition and replace each of them by their ith alternative.

The first step, finding a dimension definition for the tag $D.t$, can be defined in two principally different ways.

(1) *Arbitrary dimension access.* We can match an arbitrary dimension definition and perform tag selection in its scope.
(2) *Ordered dimension access.* We require that selections in dimensions be made in a specific order, for example, in the order in which they are encountered during a preorder traversal.

The first approach is appealing since it allows users to make decisions in different (although not completely arbitrary) orders. However, it also complicates the semantics and severely limits semantics-preserving transformations. Conversely, the second approach prescribes an ordering on decisions, but has the advantage of supporting more transformations and, in particular, guarantees the existence of important normal forms.

In this paper we pursue the second approach, because it better supports modularity of the choice representation. Therefore, we will define below a function V for computing variants by traversing (in preorder) a choice calculus expression and creating a mapping from encountered dimensions and tags to expressions that are obtained by performing tag selection along the way.

The second step of the semantics definition, replacing each bound choice by its ith alternative, can be defined by traversing the subexpression of the declaration for dimension D with the selector, or qualified index, $D.i$. This traversal will reach all choices tagged by D, but recursion will stop whenever a nested dimension declaration of the same name is encountered.

$$\begin{aligned}
\lfloor v \rfloor_{D.i} &= v \\
\lfloor a\prec e_1,\ldots,e_n \succ \rfloor_{D.i} &= a\prec \lfloor e_1 \rfloor_{D.i},\ldots,\lfloor e_n \rfloor_{D.i}\succ \\
\lfloor \mathbf{let}\ v{=}e\ \mathbf{in}\ e' \rfloor_{D.i} &= \mathbf{let}\ v{=}\lfloor e \rfloor_{D.i}\ \mathbf{in}\ \lfloor e' \rfloor_{D.i} \\
\lfloor \mathbf{dim}\ D'\langle t^n \rangle\ \mathbf{in}\ e \rfloor_{D.i} &= \begin{cases} \mathbf{dim}\ D'\langle t^n \rangle\ \mathbf{in}\ e & \text{if } D = D' \\ \mathbf{dim}\ D'\langle t^n \rangle\ \mathbf{in}\ \lfloor e \rfloor_{D.i} & \text{otherwise} \end{cases} \\
\lfloor D'\langle e_1,\ldots,e_n \rangle \rfloor_{D.i} &= \begin{cases} \lfloor e_i \rfloor_{D.i} & \text{if } D = D' \\ D'\langle \lfloor e_1 \rfloor_{D.i},\ldots,\lfloor e_n \rfloor_{D.i} \rangle & \text{otherwise} \end{cases}
\end{aligned}$$

In general, tag selection will result in an expression that still contains dimensions and choices. To talk about the different forms that an expression can have we introduce the property of being *s free*, which means that an expression does not include a syntactic category s. For example, a dimension-free expression does not contain any dimension declarations (but may still contain choices, bindings, or any other syntactic category). Additionally, an expression is called *variation free* if it is dimension free and choice free, and it is called *sharing free* if it is reference free and binding free. Finally, an expression is called *plain* if it is variation free and sharing free.

With these definitions we can say that the goal of repeated tag selection is to obtain variation-free expressions, but we observe that, in general, this property cannot be guaranteed by any particular tag selection.

4.3 Choice Semantics

The meaning of a choice expression is the total set of decisions it represents and the results of those decisions. We represent the semantics as a mapping from sequences of tags to expressions. We define the semantics in two steps. First, we define the set $V(e)$ of variants denoted by e as a mapping from sequences of tags to variation-free expressions. After that we define the semantics based on $V(e)$ by expanding all **let** expressions. We use $\bar{q}$ to range over tuples of qualified tags. We also employ an auxiliary function Π that computes the variants for a tuple of n expressions and combines the results. The combination is done by taking one pair $(\bar{q}_i, e'_i)$ from each set, and forming a new pair $(\bar{q}, e'^n)$ by concatenating all tuples $\bar{q}_i$ and forming a sequence of the expressions e'_i.

$$\Pi(e^n) = \{(\bar{q}^n, e'^n) \mid ((\bar{q}_i, e'_i) \in V(e_i))^{i:1..n}\}$$

The function V is then defined as follows.

$$\begin{aligned}
V(v) &= \{(\langle\rangle, v)\} \\
V(a{\prec}{\succ}) &= \{(\langle\rangle, a{\prec}{\succ})\} \\
V(a{\prec}e^n{\succ}) &= \{(\bar{q}, a{\prec}e'^n{\succ}) \mid (\bar{q}, e'^n) \in \Pi(e^n)\} \\
V(\textbf{let } v{=}e_1 \textbf{ in } e_2) &= \{(\bar{q}, \textbf{let } v{=}e'_1 \textbf{ in } e'_2) \mid (\bar{q}, (e'_1, e'_2)) \in \Pi(e_1, e_2)\} \\
V(\textbf{dim } D\langle t^n\rangle \textbf{ in } e) &= \{((D.t_i, \bar{q}), e') \mid i \in \{1, \ldots, n\},\ (\bar{q}, e') \in V(\lfloor e\rfloor_{D.i})\} \\
V(D\langle e^n\rangle) &= \{(\bar{q}, D\langle e'^n\rangle) \mid (\bar{q}, e'^n) \in \Pi(e^n)\}
\end{aligned}$$

Note that V leaves choices unchanged. In a well-dimensioned expression, V will not encounter any choices because they will have been resolved by tag selections triggered by their binding dimensions. We define V to propagate over choices anyway, for completeness.

By definition, expressions in the range of $V(e)$ do not contain dimensions; if e is well-dimensioned, they will not contain choices either. However, expressions in the range of $V(e)$ may still contain **let** expressions and variables references. To obtain plain expressions, we can eliminate bindings and references with the function μ, which takes as a parameter an environment ρ for storing variable bindings.

$$\begin{aligned}
\mu_\rho(a{\prec}e_1, \ldots, e_n{\succ}) &= a{\prec}\mu_\rho(e_1), \ldots, \mu_\rho(e_n){\succ} \\
\mu_\rho(\textbf{dim } D\langle t^n\rangle \textbf{ in } e) &= \textbf{dim } D\langle t^n\rangle \textbf{ in } \mu_\rho(e) \\
\mu_\rho(D\langle e_1, \ldots, e_n\rangle) &= D\langle \mu_\rho(e_1), \ldots, \mu_\rho(e_n)\rangle \\
\mu_\rho(\textbf{let } v{=}e \textbf{ in } e') &= \mu_{\rho\oplus(v,\mu_\rho(e))}(e') \\
\mu_\rho(v) &= \rho(v)
\end{aligned}$$

Although they are not strictly needed for defining the semantics of well-dimensioned expressions, the cases for dimensions and choices are provided for completeness.

The semantics of well-formed choice expressions can now be defined as a mapping from sequences of tags to plain expressions.

$$[\![e]\!] = \{(\bar{q}, \mu_\varnothing(e')) \mid (\bar{q}, e') \in V(e)\}$$

Because the semantic function can also be applied to expressions which are not well formed, we capture the relationship between well-formedness and the semantics in the following lemma.

LEMMA 1. *If e is well formed, then $\forall e' \in rng(\llbracket e \rrbracket) : e'$ is plain.*

PROOF. The semantics definition is based on V. We can observe from the definition of V that all dimension definitions and all bound choices are eliminated. Therefore, if e is well dimensioned, $\llbracket e \rrbracket$ will be variation free.

Moreover, the definition of the semantics employs μ to remove all **let** binding in all variants produced by V for e. If e does not contain any free variables, all references will be removed through the last case of μ, and thus $\llbracket e \rrbracket$ is also sharing free and therefore plain. □

We show some examples of the semantics later in Section 7.

5. FACTORIZATION AND DISTRIBUTION OF DIMENSIONS AND CHOICES

We observe that the choice representation is not unique, that choices can be generally represented on different levels of granularity, and that dimension definitions can be moved around too. For example, the following three expression are all equivalent in the sense that $\llbracket e \rrbracket = \llbracket e' \rrbracket = \llbracket e'' \rrbracket$.

$$\begin{aligned} e &= \mathbf{dim}\ A\langle a,b\rangle\ \mathbf{in}\ 5 + A\langle 1,2\rangle \\ e' &= \mathbf{dim}\ A\langle a,b\rangle\ \mathbf{in}\ A\langle 5+1, 5+2\rangle \\ e'' &= 5 + \mathbf{dim}\ A\langle a,b\rangle\ \mathbf{in}\ A\langle 1,2\rangle \end{aligned}$$

This observation raises several questions: Does it matter which representation is chosen? Is there a preferred representation that should be chosen, or do we need different representations and operations to transform between them?

It turns out that different representations are useful for different purposes. For example, maximally factored choices (as in e and e'') keep common parts out of alternatives as much as possible and thus simplify the editing of these common parts, avoiding update anomalies. On the other hand, fewer and bigger choices that repeat common parts are sometimes better suited to compare alternatives than a huge collection of fine-grained representations. Moreover, having dimensions as far at the top as possible (as in e and e') reveals the variational structure better than deeply nested dimensions. This might be desirable or not, depending on the context.

In this section we identify a number of semantics-preserving relationships between choice-calculus expressions that can be used to transform expressions into a desired form. A complete set of relationships can be obtained by observing that in principle any syntactic form, that is, *Structure*, *Binding*, *Reference*, *Dimension*, or *Choice*, can be commuted with any other. An attempt to systematically enumerate all possibilities reveals further that (1) we need two rules for commutation with bindings since we have two recursive occurrences of expressions (that, unlike in the structure and choice case, cannot be dealt with in one rule using our pattern notation) and (2) that for the case of self-commutation we can in some cases consider the merging of the two identical constructs into one. The naming of rules follows this system and uses the initials of the syntactic constructs being commuted, possibly with a suffix indicating further detail.

We present the rules in several groups. First, we show rules for factoring and distributing choices across other syntactic constructs in Figure 1. In the rules we

C-S
$$a\prec e^n[i:D\langle e'^{j:1..k}\rangle]\succ \equiv D\langle a\prec e^n[i:e'_j]\succ^{j:1..k}\rangle$$

C-B-DEF
$$\textbf{let } v{=}D\langle e^n\rangle \textbf{ in } e \equiv D\langle(\textbf{let } v{=}e_i \textbf{ in } e)^{i:1..n}\rangle$$

C-B-USE
$$\textbf{let } v{=}e \textbf{ in } D\langle e^n\rangle \equiv D\langle(\textbf{let } v{=}e \textbf{ in } e_i)^{i:1..n}\rangle$$

C-D
$$\frac{D \neq D'}{\textbf{dim } D'\langle t^m\rangle \textbf{ in } D\langle e^n\rangle \equiv D\langle(\textbf{dim } D'\langle t^m\rangle \textbf{ in } e_i)^{i:1..n}\rangle}$$

C-C-SWAP
$$D'\langle e^n[i:D\langle e'^{j:1..k}\rangle]\rangle \equiv D\langle D'\langle [e^n[i:e'_j]\rangle^{j:1..k}\rangle$$

C-C-MERGE
$$D\langle e^n[i:e'_i]\rangle \equiv D\langle e^n[i:D\langle e'^n\rangle]\rangle$$

Fig. 1. Choice commutation rules.

make use of a further notational convention to expose the ith element of a sequence. The pattern notation $e^n[i:e']$ expresses the requirement that e_i has the form given by the expression (or pattern) e'. For example, $e^n[i:e'+1]$ says that e_i must be an expression that matches $e'+1$.

In the case of nested choices for the same dimension D (rule C-C-MERGE), distribution amounts to merging the two choices into one. In that case the nested choice ($D\langle e'^n\rangle$) does not really present a choice since all alternatives except e'_i are dead and cannot be reached because the semantics of tag selection recursively selects the same component from a nested choice. That is, if tag selection selects $D\langle e'^n\rangle$ as the ith alternative of the outer choice, it will also select e'_i from the inner one.

To apply the rules C-B-DEF and C-B-USE from right to left it is required that all alternatives in the choice contain a **let** binding. Whenever that is not the case, we can employ the relationships in Figure 2, for manipulating **let** expressions, to prepare for the application of the C-B-* rules. In rule premises we use $j \neq i$ as an abbreviation for $j \in \{1,\ldots,n\} - \{i\}$. The first two rules can be used to both introduce new **let** expressions and eliminate redundant ones, the two B-USE-USE rules are for commuting **let** bindings, and the two B-S rules are for moving bindings into and out of structures.

Note that all semantics-preserving transformations must not introduce or remove dimension declarations, nor change the *order* in which they appear in an in-order traversal of the expression's abstract syntax tree. The binding rules contain conditions which ensure that these constraints on dimension declarations are enforced, in addition to the obvious conditions that prevent the capture of free variables. Consider, for example, the application of the B-S rules from right to left. Pulling the binding at position i out of the structure changes the ordering of the subexpressions: expression e now precedes expressions e_1 through e_{i-1}. This transformation is only

$$\textsc{Naming}\qquad e \equiv \textbf{let}\ v{=}e\ \textbf{in}\ v$$

$$\textsc{B-New}\quad \frac{v \notin FV(e) \qquad BD(e') = \varnothing}{e \equiv \textbf{let}\ v{=}e'\ \textbf{in}\ e}$$

$$\textsc{B-Use-Use}\quad \frac{v \neq w \qquad v \notin FV(e') \qquad w \notin FV(e) \qquad BD(e) = \varnothing}{\textbf{let}\ v{=}e\ \textbf{in}\ (\textbf{let}\ w{=}e'\ \textbf{in}\ e'') \equiv \textbf{let}\ w{=}e'\ \textbf{in}\ (\textbf{let}\ v{=}e\ \textbf{in}\ e'')}$$

$$\textsc{B-Use-Use'}\quad \frac{v \neq w \qquad v \notin FV(e') \qquad w \notin FV(e) \qquad BD(e') = \varnothing}{\textbf{let}\ v{=}e\ \textbf{in}\ (\textbf{let}\ w{=}e'\ \textbf{in}\ e'') \equiv \textbf{let}\ w{=}e'\ \textbf{in}\ (\textbf{let}\ v{=}e\ \textbf{in}\ e'')}$$

$$\textsc{B-S}\quad \frac{v \notin \cup_{j\neq i} FV(e_j) \qquad BD(e) = \varnothing}{\textbf{let}\ v{=}e\ \textbf{in}\ a{\prec}e^n[i:e']{\succ} \equiv a{\prec}e^n[i:\textbf{let}\ v{=}e\ \textbf{in}\ e']{\succ}}$$

$$\textsc{B-S'}\quad \frac{v \notin \cup_{j\neq i} FV(e_j) \qquad (BD(e_j) = \varnothing)^{j:1..i-1}}{\textbf{let}\ v{=}e\ \textbf{in}\ a{\prec}e^n[i:e']{\succ} \equiv a{\prec}e^n[i:\textbf{let}\ v{=}e\ \textbf{in}\ e']{\succ}}$$

Fig. 2. Binding commutation rules.

semantics-preserving if it does not affect the ordering of the dimensions, so either e does not contain any dimension declarations (rule B-S) or none of e_1 through e_{i-1} do (rule B-S').

Also note that unlike the C-B-* rules, we do *not* merge similar **let** expressions in the B-S rules. Such a transformation would not be semantics preserving if the bound expressions contain dimension declarations. Similarly, we do not have a rule S-D*, which would lift and combine several dimension declarations (with the same dimension name and tags) out of a structure. When commuting **let** expressions or dimension declarations with choices (as in rule C-D and the C-B-* rules), creating or merging duplicates does not affect the semantics since only one alternative can ever be selected. In other cases, duplicating an expression containing a dimension declaration increases the number of decisions that must be made and increases the number of variants.

Next we present the rules for factoring and distributing dimensions in Figure 3. We have omitted the rule D-C since, due to symmetry, it is identical to the rule C-D already given in Figure 1. Like the binding commutation rules, these rules take similar care to prevent the reordering of dimension declarations and the capture of free dimensions.

We also add reflexivity, symmetry, and transitivity rules for $\equiv$ to make it an equivalence relationship, and add a congruence rule to transform expressions within the same context.

$$\textsc{Refl}\quad e \equiv e \qquad \textsc{Symm}\quad \frac{e \equiv e'}{e' \equiv e} \qquad \textsc{Trans}\quad \frac{e_1 \equiv e_2 \qquad e_2 \equiv e_3}{e_1 \equiv e_3} \qquad \textsc{Cong}\quad \frac{e \equiv e'}{C[e] \equiv C[e']}$$

D-S
$$\frac{D \notin \cup_{j \neq i} FD(e_j) \qquad (BD(e_j) = \varnothing)^{j:1..i-1}}{a\prec e^n[i:\mathbf{dim}\ D\langle t^m\rangle\ \mathbf{in}\ e]\succ\ \equiv \mathbf{dim}\ D\langle t^m\rangle\ \mathbf{in}\ a\prec e^n[i:e]\succ}$$

D-B-DEF
$$\frac{D \notin FD(e')}{\mathbf{let}\ v{=}(\mathbf{dim}\ D\langle t^n\rangle\ \mathbf{in}\ e)\ \mathbf{in}\ e' \equiv \mathbf{dim}\ D\langle t^n\rangle\ \mathbf{in}\ (\mathbf{let}\ v{=}e\ \mathbf{in}\ e')}$$

D-B-USE
$$\frac{D \notin FD(e) \qquad BD(e) = \varnothing}{\mathbf{let}\ v{=}e\ \mathbf{in}\ (\mathbf{dim}\ D\langle t^n\rangle\ \mathbf{in}\ e') \equiv \mathbf{dim}\ D\langle t^n\rangle\ \mathbf{in}\ (\mathbf{let}\ v{=}e\ \mathbf{in}\ e')}$$

Fig. 3. Dimension commutation rules.

An important property of the transformation rules is that they are semantics preserving. This property is captured in the following theorem.

THEOREM 1. *If e is well formed, then $e \equiv e' \implies [\![e]\!] = [\![e']\!]$*

A proof of this theorem is provided in the appendix. The factorization/distribution rules provide a powerful means to transform choice expression and make the representation very flexible.

6. DIMENSION AND CHOICE NORMAL FORMS

The rules presented in Section 5 can be used to transform expressions in many different ways. In this section we identify three strategically significant representations: *dimension normal form*, *choice normal form*, and *dimension-choice normal form*. We then show that any expression can be transformed into choice normal form, and that any dimension-linear expression can be transformed into dimension normal form and consequently, dimension-choice normal form.

We say that an expression e is in *choice normal form* (*CNF*) if it contains only choices that are maximally factored. That is, e is in CNF if no subexpression of e matches the right-hand side of any of the rules given in Figure 1 (without violating a premise). CNF is significant because it reduces redundancy in the representation. This is an important feature for the development of variation editing tools because it decreases the risk of update anomalies.

We similarly say that an expression e is in *dimension normal form* (*DNF*) if all dimensions are maximally factored. We consider a dimension maximally factored if its declaration appears at the top of the expression, at the top of an alternative within a choice, or directly beneath another maximally-factored dimension. DNF is convenient because it groups dimension declarations according to their dependencies. For example, all dimensions at the top of an expression are *independent*—the selection of any tag in any independent dimension does not affect the possible selections in other independent dimensions. Dimensions grouped within an alternative are *dependent* on the corresponding tag being chosen in the enclosing choice—if the tag is not chosen, we need not make a selection in any dimensions in the group.

Finally, we say that an expression is in *dimension-choice normal form* (*DCNF*) if it is in choice normal form and in dimension normal form. Naturally, DCNF combines the benefits of both CNF and DNF, avoiding redundancy and clearly revealing the dimension structure. It is therefore a prime candidate for a variation representation in an editor tool or IDE since it avoids update anomalies while editing and groups related dimensions.

We call an expression that is well dimensioned and dimension linear *linearly dimensioned*. In the following we will show that any linearly dimensioned expression can be transformed into DCNF. Any expression e can be transformed into an equivalent expression e' that is maximally choice factored (that is, in CNF). This can be achieved by repeatedly applying the rules from Figure 1 from right to left.

LEMMA 2. *$\forall e.\exists e'.e \equiv e' \wedge e'$ is in CNF.*

PROOF. The definition of CNF is based on the applicability of transformation rules. For an expression e, there are two possibilities: Either no rule is applicable, in which case e is in CNF already. Otherwise, a rule can be applied, which yields an expression e' to which the same reasoning can be applied inductively.

To show that the transformation process must end after a finite number of steps, we define a measure, called the *choice depth*, as follows. If n is the number of choices in an expression, and m is the distance of the choice furthest from the root of that expression, the choice depth is $\frac{1}{n}+m$. It is clear that all of the C-* rules, when applied from right to left, increase the choice depth of an expression without increasing the size of the expression. □

Lemma 2 is significant on its own, demonstrating that any e can be transformed into *CNF*, minimizing redundancy. However, only expressions that are linearly dimensioned can, in general, be brought into dimension normal form.

LEMMA 3. *If e is linearly dimensioned, then $\exists e'$ in DNF such that $e \equiv e'$.*

PROOF. This result follows from the fact that we have a rule for moving a dimension out of each syntactic category. The premises that would prevent the application of a rule are of two forms. Either they prevent the capture of free choices or they constrain the order in which the rules can be applied. The first case does not apply since e is well dimensioned and also dimension linear. The conditions about bound dimensions can be met by simply moving out those "earlier" dimensions before the current one. □

From Lemmas 2 and 3 the following result about dimension-choice normal form follows directly.

THEOREM 2. *If e is linearly dimensioned, then $\exists e'$ in DCNF such that $e \equiv e'$.*

As a final illustration of the three types of normal form, recall the following expressions from Section 5.

$$\begin{aligned} e &= \mathbf{dim}\ A\langle a,b\rangle\ \mathbf{in}\ 5+A\langle 1,2\rangle \\ e' &= \mathbf{dim}\ A\langle a,b\rangle\ \mathbf{in}\ A\langle 5+1,5+2\rangle \\ e'' &= 5+\mathbf{dim}\ A\langle a,b\rangle\ \mathbf{in}\ A\langle 1,2\rangle \end{aligned}$$

Comparing these to the definitions above, we see that e is in DCNF, while e' is (only) in DNF and e'' is (only) in CNF.

7. VARIATION DESIGN THEORY

We can observe that not every choice expression is a good variation representation. A trivial example is a choice of the form $A\langle e,e\rangle$ that contains two identical alternatives. Since it does not matter which alternative we select, this is a “false choice” that could be simply replaced by e.

In this section we formalize several quality criteria for choices and dimensions that can serve as guidelines for the design of variation structures. In Section 7.1 we compare a syntactic and a semantic approach to deriving design criteria. In Section 7.2 we then develop a semantic criterion for identifying equivalent alternatives in choices and equivalent tags in dimensions. We also define transformations to remove redundant alternatives and tags. In Section 7.3 we give semantic criteria for identifying spurious choices and dimensions, and transformations to remove them. Finally, in Section 7.4 we provide a transformation for eliminating undesireable nestings of choices by removing unreachable alternatives.

7.1 Syntactic vs. Semantic Design Criteria

There are two ways to approach the formalization of variation design criteria. First, we can pursue a syntactic approach and identify patterns of dimension and choice expressions directly, as we have done with the example $A\langle e,e\rangle$. However, it is not always syntactically obvious when two expressions are equivalent. Consider the following choice expression abc.

$$\begin{array}{l}\textbf{dim}\ A\langle a,b\rangle\ \textbf{in}\\ \textbf{dim}\ B\langle c,d\rangle\ \textbf{in}\\ \textbf{dim}\ C\langle e,f\rangle\ \textbf{in}\\ \quad A\langle B\langle C\langle 1,2\rangle,C\langle 3,4\rangle\rangle,C\langle B\langle 1,3\rangle,B\langle 2,4\rangle\rangle\rangle \end{array} \qquad (abc)$$

In the above example, the A dimension is irrelevant, but this is very difficult to see in the syntax.

We can apply transformations to make the situation more obvious. For example, applying the rule C-C-SWAP to $C\langle B\langle 1,3\rangle,B\langle 2,4\rangle\rangle$, the second alternative of A, and fixing i to 1, yields the following choice.

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

Now we apply C-C-SWAP two more times, swapping $B\langle 2,4\rangle$ twice with C. After swapping the first occurrence of $B\langle 2,4\rangle$, we obtain the following expression.

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

Swapping the second occurrence gives us the following.

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

Next we can apply the rule C-C-MERGE twice, which simplifies the first alternative by dropping its second alternative $C\langle 1,4\rangle$ as follows.

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

Similarly, if we apply C-C-MERGE again, we can simplify the second alternative by dropping its first alternative $C\langle 3,2\rangle$ to obtain the following.

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

Altogether we obtain the following transformed expression, in which it is obvious that both alternatives of A are identical.

$$\begin{array}{l} \textbf{dim}\ A\langle a,b\rangle\ \textbf{in} \\ \textbf{dim}\ B\langle c,d\rangle\ \textbf{in} \\ \textbf{dim}\ C\langle e,f\rangle\ \textbf{in} \\ \quad A\langle B\langle C\langle 1,2\rangle,C\langle 3,4\rangle\rangle,B\langle C\langle 1,2\rangle,C\langle 3,4\rangle\rangle\rangle \end{array}$$

This shows that we could replace *abc* with the following expression *bc*.

$$\begin{array}{l} \textbf{dim}\ B\langle c,d\rangle\ \textbf{in} \\ \textbf{dim}\ C\langle e,f\rangle\ \textbf{in} \\ \quad B\langle C\langle 1,2\rangle,C\langle 3,4\rangle\rangle \end{array} \tag{bc}$$

Alternatively, we can formulate criteria based on the semantics of choice expressions. However, requiring semantics preservation would be too strong a criterion. To illustrate this point let us consider the following choice expression *ab*.

$$\textbf{dim}\ A\langle a,b\rangle\ \textbf{in}\ A\langle 1,1\rangle \tag{ab}$$

It is easy to see that *ab* and the expression 1 do *not* have the same semantics since $[\![1]\!] = \{((),1)\}$ whereas $[\![ab]\!] = \{(A.a,1),(A.b,1)\}$. Nevertheless, it seems justified to replace *ab* by 1. Even though such a transformation is not semantics preserving, it is *variant preserving* in the following sense. First, we define the *variants* of an expression e to be $rng([\![e]\!])$, that is, the expressions contained in the range of e's denotation. From Lemma 1 we know that if $FD(e) = \varnothing$ and $FV(e) = \varnothing$, then all variants of e are plain. Based on this notion of variants we can define that two expressions e and e' are *variant equivalent*, written as $e \sim e'$, if $rng([\![e]\!]) = rng([\![e']\!])$, and we can call a transformation that maps e into e' *variant preserving* if $e \sim e'$.

We can now see that $ab \sim 1$ because *ab* and 1 have the same variants. Therefore, the replacement of *ab* by 1 is variant preserving. Likewise, the variants of *abc* and *bc* are both $\{1,2,3,4\}$. Therefore, they too are variant equivalent and the simplification of *abc* to *bc* is variant preserving and thus justified according to this criterion.

The semantic approach provides a simpler and more general avenue to a design theory since the semantics flattens the potentially complex, nested dimension and choice structure of an expression into a plain relation.

Redundancies and corresponding simplification opportunities can occur in choice expressions on different levels and in different forms. Employing semantics-based criteria, we will investigate the binary equivalence of alternatives and tags in Section 7.2. For equivalent tags, we can identify a simplification transformation that reduces the size of the corresponding dimension and choices. In Section 7.3 we lift those relationships to sets of entities, that is, choices and dimensions, and we identify simplification transformations that can eliminate whole choices and dimensions.

7.2 Equivalent Alternatives and Tags

The simplest form of equivalence that we can observe is that of different alternatives in individual choices, as in $A\langle 1,1\rangle$. We say that two alternatives e_i and e_j of a choice $D\langle e^n\rangle$ are *equivalent* in context C, written as $e_i \sim_C^D e_j$, if $[\![C[D\langle e^n\rangle]]\!]$ is unchanged by swapping alternatives e_i and e_j; that is, if

$$[\![C[D\langle e^n\rangle]]\!] = [\![C[D\langle e_1,\ldots,e_j,\ldots,e_i,\ldots,e_n\rangle]]\!]$$

This definition seems overly complicated. Why can't we just define equivalence to hold if $[\![e_i]\!] = [\![e_j]\!]$? The problem is that requiring the equality of semantics is too strong a condition. Consider, for example, the choice $A\langle v,v\rangle$. The semantics of both alternatives is undefined since v is unbound. Still, we would like to say that both alternatives are equivalent. We can't use plain term equality either since this would miss the equivalent alternatives in $A\langle \mathbf{dim}\ B\langle a,b\rangle\ \mathbf{in}\ B\langle 1,2\rangle, \mathbf{dim}\ B\langle b,a\rangle\ \mathbf{in}\ B\langle 2,1\rangle\rangle$.

The chosen context-dependent definition leads to situations that might seem a bit unintuitive at first. Consider, for example, the following expression ab_1.

$$\mathbf{dim}\ A\langle a,b\rangle\ \mathbf{in}\ A\langle A\langle 1,1\rangle,1\rangle \qquad (ab_1)$$

The semantics for this expression and the version in which $A\langle 1,1\rangle$ and 1 are swapped are $\{(A.a,1),(A.b,1)\}$, which means that the alternatives $A\langle 1,1\rangle$ and 1 are equivalent even though they are different. However, considering that both alternatives ultimately always produce the same results, the definition seems appropriate.

We define the function $\bar{\alpha}_{C/i}(e)$ to perform the removal of the ith alternative of a choice in context C within expression e. This function is defined only if C is a context that matches a choice in e with at least i alternatives; that is, we assume $e = C[D\langle e^n\rangle]$ with $n \geq i$. Then we obtain the following, obvious definition.

$$\bar{\alpha}_{C/i}(e) = C[D\langle e_1,\ldots,e_{i-1},e_{i+1},\ldots,e_n\rangle]$$

As an example, consider the following expression abx.

$$\mathbf{dim}\ A\langle a,b,x\rangle\ \mathbf{in}\ A\langle 1,1,9\rangle \qquad (abx)$$

With $C = \mathbf{dim}\ A\langle a,b,x\rangle\ \mathbf{in}\ []$, we can remove the second alternative in the choice in abx by applying $\bar{\alpha}_{C/2}(abx)$. This yields the expression $\mathbf{dim}\ A\langle a,b,x\rangle\ \mathbf{in}\ A\langle 1,9\rangle$, which is not well dimensioned. This example demonstrates that we cannot, in general, simplify a choice that contains equivalent alternatives in isolation. Since the number of alternatives in a choice must match the number of tags in its binding dimension, reducing the number of alternatives in a choice requires the removal of the corresponding tag in the binding dimension to maintain well dimensionedness. In this example, this would actually work since we have only one choice that is bound by dimension A. But in cases where we have other choices, the removal of the tag is possible only if all corresponding pairs of alternatives in all those other choices are redundant too, which is generally not the case.

On the dimension level, we can consider the equivalence of tags. As an example, consider again the expressions ab and ab_1. The tags a and b are equivalent in the sense that selection with one tag yields the same result as selection with the other. Therefore, we define that two tags t_i and t_j are *equivalent* in context C, written as $t_i \sim_C t_j$, if $[\![C[\mathbf{dim}\ D\langle t^n\rangle\ \mathbf{in}\ e]]\!]$ is unchanged by swapping tags t_i and t_j; that is, if

$$[\![C[\mathbf{dim}\ D\langle t^n\rangle\ \mathbf{in}\ e]]\!] = [\![C[\mathbf{dim}\ D\langle t_1,\ldots,t_j,\ldots,t_i,\ldots,t_n\rangle\ \mathbf{in}\ e]]\!]$$

Tag equivalence is in a sense a stronger property than equivalence of alternatives since a dimension that defines two equivalent tags can bind many choices, and thus the equivalence has a broader scope. However, equivalent tags do *not* imply equivalent alternatives, which can be seen in the following simple example.

$$\mathbf{dim}\ A\langle a,b\rangle\ \mathbf{in}\ A\langle A\langle 1,2\rangle,1\rangle$$

It is clear that selection with either $A.a$ or $A.b$ produces 1 as a result; that is, tags a and b are equivalent. However, neither pair of alternatives in either of the two bound choices is equivalent.

Two equivalent tags are redundant with respect to each other, and therefore, one of them can be safely removed, because the removal is variant preserving. Removing a tag amounts to reducing the size of a dimension and all of its bound choices by one and is done as follows. If $t_i \sim_C t_j$, replace the dimension declaration $\mathbf{dim}\ D\langle t^n\rangle$ with $\mathbf{dim}\ D\langle t_1,\ldots,t_{j-1},t_{j+1},\ldots,t_n\rangle$ and every choice $D\langle e^n\rangle$ bound by that dimension with $D\langle e_1,\ldots,e_{j-1},e_{j+1},\ldots,e_n\rangle$. We write $\bar{\tau}_{C/t_j}(e)$ for applying this simplification operation to an expression e with $e = C[\mathbf{dim}\ D\langle t^n\rangle\ \mathbf{in}\ e']$. For our examples we obtain the following possible removals.

$$\begin{aligned}
\bar{\tau}_{C/a}(ab) &= \mathbf{dim}\ A\langle b\rangle\ \mathbf{in}\ A\langle 1\rangle & \bar{\tau}_{C/b}(ab) &= \mathbf{dim}\ A\langle a\rangle\ \mathbf{in}\ A\langle 1\rangle\\
\bar{\tau}_{C/a}(ab_1) &= \mathbf{dim}\ A\langle b\rangle\ \mathbf{in}\ A\langle 1\rangle & \bar{\tau}_{C/b}(ab_1) &= \mathbf{dim}\ A\langle a\rangle\ \mathbf{in}\ A\langle A\langle 1,1\rangle\rangle\\
\bar{\tau}_{C/a}(abx) &= \mathbf{dim}\ A\langle b,x\rangle\ \mathbf{in}\ A\langle 1,9\rangle & \bar{\tau}_{C/b}(abx) &= \mathbf{dim}\ A\langle a,x\rangle\ \mathbf{in}\ A\langle 1,9\rangle
\end{aligned}$$

Since an equivalent tag represents a redundancy in an expression, the systematic removal of equivalent tags does not affect the represented variants. This is summarized in the following theorem.

THEOREM 3. $t_i \sim_C t_j \implies \bar{\tau}_{C/t_j}(e) \sim e$

Obviously, Theorem 3 applies to our examples ab, ab_1, and abx. We can also apply it to the choice expression abc. In all examples except for expression abx, the tag removal results in a dimension that contains only one tag and corresponding choices with only one alternative each. Such dimensions and choices are trivially superfluous and can thus be eliminated. We will consider this kind of transformation in the next subsection.

7.3 Removing Pseudo-Choices and Pseudo-Dimensions

A choice in which all alternatives are pairwise equivalent is not really a choice since the decision of what alternative to pick has no impact on the semantics of the expression. We call a choice with this property a *pseudo-choice*. In the previous subsection we have observed that a single pair of equivalent alternatives cannot be removed in general. In contrast, a pseudo-choice can be safely replaced by any one of its alternatives, a fact that is summarized in the following lemma.

LEMMA 4. $\forall i,j \in \{1,\ldots,n\} : e_i \sim_C^D e_j \implies [\![C[D\langle e^n\rangle]]\!] = [\![C[e_i]]\!]$

We write $\bar{\gamma}_C(e)$ for removing the choice in the context C within expression e. This operation is, of course, defined only when C matches a choice expression in e, that is, when $e = C[D\langle e^n\rangle]$.

As an example, consider again the expression ab_1. Since both alternatives $A\langle 1,1\rangle$ and 1 are equivalent (that is, $A\langle 1,1\rangle \sim_{C'}^D 1$ with $C' = \mathbf{dim}\ A\langle a,b\rangle\ \mathbf{in}\ []$), we can replace $A\langle A\langle 1,1\rangle,1\rangle$ by 1 (or $A\langle 1,1\rangle$), that is, we apply $\bar{\gamma}_{C'}(ab_1)$. The same also trivially applies to the expression ab.

For both expressions, the application of $\bar{\gamma}$ results in the expression $\mathbf{dim}\ A\langle a,b\rangle\ \mathbf{in}\ 1$, which can be obviously further simplified to 1 by removing the dimension definition. This simplification opportunity is captured more generally in the following lemma.

LEMMA 5. $D \notin FD(e) \implies \mathbf{dim}\, D\langle t^n \rangle\ \mathbf{in}\ e \sim e$

The rationale for this kind of transformation is that the semantics of the expression is completely independent of the tags in the dimension. Therefore, we can take the idea of pairwise equivalence to the dimension level and arrive at the notion of a *pseudo-dimension*, which is a dimension in which all tags are equivalent. For example, dimension A is indeed a pseudo-dimension in expressions ab and ab_1 since tags a and b, the only tags of A, are equivalent. In contrast, dimension A is not a pseudo-dimension in the expression abx.

Like pseudo-choices, pseudo-dimensions are not needed at all and can be removed, but we must suitably replace all of their bound choices as well. A pseudo-dimension can be removed as follows. First, replace $\mathbf{dim}\, D\langle t^n \rangle\ \mathbf{in}\ e$ by e. Then replace all free choices $D\langle e^n \rangle$ in e by some e_i (in a well-dimensioned expression the free choices in e are exactly those that were previously bound by the just removed dimension). The tag-selection operation $\lfloor \mathbf{dim}\, D\langle t^n \rangle\ \mathbf{in}\ e \rfloor_{D.t_i}$ defined in Section 4.2 performs exactly this function. We define the simplification operation $\bar{\delta}$ on an expression e with $e = C[\mathbf{dim}\, D\langle t^n \rangle\ \mathbf{in}\ e']$ as follows.

$$\bar{\delta}_C(e) = C[\lfloor e' \rfloor_{D.1}]$$

The case for pseudo-dimensions can be made by induction over redundant tags. That is, we can repeatedly apply Theorem 3, followed by an application of Lemma 5. We capture the validity of pseudo-dimension removal in the following theorem.

THEOREM 4. $\forall i,j \in \{1,\ldots,n\} : t_i \sim_C t_j \implies \bar{\delta}_C(e) \sim e$

Since any tag is equivalent to itself, the premise of the theorem is trivially fulfilled for dimensions that contain only one tag, which means that those dimensions can always be safely removed. This fact is summarized in the following corollary.

COROLLARY 1. $\mathbf{dim}\, D\langle t \rangle\ \mathbf{in}\ e \sim \lfloor e \rfloor_{D.t}$

7.4 Choice Fusion

The rule C-C-MERGE shown in Section 5 allows, when applied from right to left, the removal of unreachable alternatives from a nested choice. It is therefore well suited as a basis for improving choice expressions. However, the rule as given is of a syntactic nature, which means that, in general, other transformations have to take place for it to be applicable. We have illustrated in Section 7.1 that this can be a tedious process and have therefore also investigated semantics-based descriptions of transformations. So a natural question is: Can we express the idea of choice merging in a more declarative way based on semantics criteria?

A simple way to achieve this is by employing tag selection. We can safely project on the ith alternative in all choices in dimension D that occur nested within the ith alternative of a choice in dimension D. This is due to the definition of selection with a qualified index. We can summarize this idea in the following lemma.

LEMMA 6. $[\![C[D\langle e^n \rangle]]\!] = [\![C[D\langle \lfloor e_1 \rfloor_{D.1}, \ldots, \lfloor e_n \rfloor_{D.n} \rangle]]\!]$

This lemma describes a form of algebraic optimization reminiscent of constant folding and propagation in compiler theory. Note that the described transformation is actually semantics preserving and not just variant preserving.

8. RELATED WORK

Section 2 provided an overview of existing work in software variation management, their strengths and weaknesses, and hinted at differences between existing work and the choice calculus. In this section we complete this discussion by presenting other related work and providing deeper comparisons of the existing work to the choice calculus.

Most current research on multi-dimensional variation management is focused on the development and maintenance of software product lines, as introduced in Section 2. Recall that this research revolves around the notion of optional functionality encapsulated as *features* and constraints on the sets of features that can be incorporated into a program, captured in *feature models*. Representing and implementing feature models are usually approached as two fundamentally different problems. Implementation is lower-level, concerned with annotating or encapsulating and combining code, while feature modeling is significantly more abstract, usually treating features as fundamental elements. Ostensibly, one could choose an implementation strategy and a feature modeling technique independently, then combine them to form a complete software-product-line system. In practice, certain implementation methods work better with certain model representations; for example, the order in which features are incorporated is significant in some implementations (most FOP systems), but not with others (CIDE), and only some types of feature models include ordering constraints.

There are two different ways of relating the choice calculus to the feature model view of variation management. The first is to consider the choice calculus primarily as a means of implementing feature models. This could be augmented by traditional feature modeling techniques or a higher-level "choice-model" that enforces additional, inter-dimensional constraints on selections applied to a choice expression. For example, one could specify that the selection of a tag in one dimension implies the selection of a tag in another, or that two particular tags in two different dimensions cannot both be selected. The choice calculus compares favorably as an implementation strategy. Its use of an underlying tree model and local dimension declarations make it more structured than most annotation systems, but it is able to capture significantly finer-grained variation than FOP systems. Also, by basing variation on choices among alternatives, rather than on optionality, the choice calculus is not restricted to varying only optional syntactic elements (as are most FOP systems and the currently available version of CIDE, for example).

The second view is to consider the choice calculus a representation that spans both the implementation and feature modeling levels. In a choice calculus expression, tags can be considered to correspond to features, and dimensions and the structure of choices constitute the feature model. This view does not support inter-dimensional constraints, but it is still appealing since it unifies the representation and implementation of feature models. In particular, it does not suffer from the "optional feature problem" [Kästner et al. 2009] that can arise when there are constraints in the implementation of a feature model not present in the model itself. Fortunately, we are not restricted to just one of these views of how the choice calculus relates to existing work, and we will alternate between them in the following discussion.

In Section 2 we introduced FOP (feature-oriented programming) as one approach to representing features. In FOP, features are represented by sets of classes, subclasses, and mixins, which can be optionally added to the program. Due to their reliance on inheritance to introduce variation, FOP systems are good at capturing (and limited to) relatively ad hoc, coarse-grained, and hierarchically structured variation. Aspect-oriented programming (AOP) [Kiczales et al. 1997] excels at capturing a complementary class of regular, finer-grained, and non-local variation. Naturally, recent work has sought to combine the two [Mezini and Ostermann 2004, Liu et al. 2006, Apel et al. 2008a]. Feature representations relying on FOP and AOP are related to traditional ideals of software reuse, and particularly the pursuit of a *separation of concerns* [Tarr et al. 1999]. These approaches emphasize the complete modularization of features—that code comprising a feature should be self-contained and separate from code describing other features or the "base program"—leading to a highly distributed feature representation. This contrasts sharply with the approach of tools like CPP, CIDE, and the choice calculus, which represent variation in a way that is integrated with the rest of the program.[6] While there are several trade-offs between the two approaches, perhaps the most interesting is in comprehensibility. In general, with the distributed approach we might expect that as the complexity of a variational structure increases, its comprehensibility will degrade more slowly thanks to a separation of concerns. However, with the integrated approach we might expect higher comprehensibility in smaller programs, or when trying to understand the interaction of multiple features, since variable parts can be seen in the context of the rest of the program. CIDE offers an elegant compromise through a *virtual separation of concerns*, allowing one to optionally hide or show the code associated with certain features [Kästner et al. 2008]. Thus, with tool support, the integrated approach is able to gain many of the benefits of both approaches.

Like features, feature models are represented in a variety of ways. At the simplest end of the spectrum is the algebraic feature model used by the FOP system GenVoca [Batory and O'Malley 1992]. GenVoca's algebra includes two types of values: a program is represented as a constant, while a feature is represented as a function that takes a program and produces a new program (with the corresponding feature included); the only operation is function application. Valid combinations of features are simply enumerated, and each of these variants is bound to a unique name. Apel et al. [2008b] generalize and formalize this basic idea into a sophisticated algebra for feature composition, where features are represented by trees and operations describe various ways of composing these trees. This approach provides explicit support for feature implementations with ordering constraints, and is general in the sense that any subset of all possible variants can be expressed. However, the drawbacks of this approach relative to the choice calculus and other types of feature models are obvious: enumerating all valid combinations of features is tedious and potentially error-prone, especially as the number of variants grows large.[7]

Feature diagrams [Kang et al. 1990] are a simple graphical notation for expressing

[6]The XML-based Variant Configuration Language [Zhang and Jarzabek 2004] and "invasive software composition" [Aßmann 2003] attempt to straddle both the separated and integrated approaches, to gain the benefits and mitigate the weaknesses of each.
[7]This problem is mitigated by the invention of "Origami matrices" that allow features to be organized into a grid that can be "folded" to produce several different products [Batory et al. 2002; 2003].

relationships between features. Although many different feature diagramming notations and extensions have been developed [Schobbens et al. 2006], the core language has remained constant. In a feature diagram, features are arranged hierarchically, such that children are dependent on their parents. A feature can also be marked as mandatory or optional, and groups of features with a common parent can be joined together as alternatives, exactly one of which may be included. Each of these relationships have corresponding structures in the choice calculus: Dependency relationships are expressed by nesting choices within other choices, the alternative relationship corresponds directly to dimensions in the choice calculus, and optional features can be simulated by choices with two alternatives, one which includes the code and one which does not. Many feature diagram extensions are less straightforward to represent in the choice calculus, however. For example, many feature diagram notations allow arbitrary connections between features to indicate AND or NAND relationships (that is, that two features must always be included together, or can never be included together, respectively). These sorts of constraints between arbitrary features cannot be directly expressed in the choice calculus, and so fall in the domain of a higher-level choice model.

Unlike the algebras described above, feature diagrams are usually not a processable part of a SPL system but rather employed as design documents and developer specifications. In other words, the constraints on features that they specify are usually not technically binding, leaving room for error in their realization in the code. The static analyses supported by the choice calculus (in particular, the well-formed property defined in Section 4.1) can help this situation.

The core feature diagram notation is equivalent to several other representations of feature models, as outlined in [Batory 2005]. Höfner et al. have taken a substantially different approach, presenting an algebraic description of feature models based on semirings [Höfner et al. 2006]. This allows common problems to be expressed as algebraic operations. For example, finding commonalities between products to support reuse can be expressed as the greatest common divisor problem. This work has an emphasis on formalism in common with the choice calculus, but is considerably higher level, with no means of associating features with the underlying artifacts.

9. FUTURE WORK

The choice calculus provides a rich platform for further research in software variation management. In particular, the theoretical results presented in Sections 5, 6, and 7 can support the development of variation management tools. In this section we will place our work in this context and discuss potential directions for future work and how the choice calculus and theoretical results will support this research.

We have begun work on methods and tools for improving variation management in existing artifacts. In particular, we are considering how the choice calculus can be useful to the large base of existing CPP users. We can translate a program with CPP annotations into the choice calculus by initially considering each macro to correspond to a dimension with two tags (corresponding to whether the macro is defined or not), then translating conditional statements into (potentially nested) choices. Consider, for example, the following CPP-annotated code.

```
#if LINUX || MAC
  newline = "\n";
#elif WINDOWS
  newline = "\r\n";
#else
  error("Unknown OS!");
#endif
```

We can straightforwardly translate the above code into the following choice calculus expression.

dim $\mathit{Linux}\langle t, f\rangle$ **in**
dim $\mathit{Mac}\langle t, f\rangle$ **in**
dim $\mathit{Windows}\langle t, f\rangle$ **in**
  let v=`newline = "\n";`
  in $\mathit{Linux}\langle v,\mathit{Mac}\langle v,\mathit{Windows}\langle$`newline = "\r\n";,error("Unknown OS!");`$\rangle\rangle\rangle$

A tag t indicates that the corresponding macro is defined, while f indicates that it is undefined. A sequence of tag selections then corresponds to a configuration of macros passed to the preprocessor.

Once translated, we can already apply several of the results from Section 7. For example, we can identify and remove dead macros with pseudo-dimension removal and remove dead code with choice fusion. However, a significant problem with CPP, that is still present in the direct-translation to the choice calculus, is that relationships between macros are only *implicitly* defined (and almost never documented). This exacerbates the variant explosion problem and leads to code which is difficult to understand and error-prone. We hope to identify sets of macros which may constitute a dimension by analyzing patterns in the translated choice expressions. In the example above, choices nested in the "undefined" alternatives of other choices may indicate that the macros are part of an implicit dimension. We can present this set of macros to the user and once verified, we can merge the corresponding dimensions and choices. The example above could be transformed into the following.

dim $\mathit{OS}\langle \mathit{Linux},\mathit{Mac},\mathit{Windows},\mathit{Unknown}\rangle$ **in**
  let v=`newline = "\n";`
  in $\mathit{OS}\langle v,v,$`newline = "\r\n";,error("Unknown OS!");`$\rangle$

This work could form the foundation of several different tools: (1) a static checker for verifying that CPP code conforms to some expected macro structure and is otherwise well formed, (2) a code optimizer, which eliminates bad or redundant code, or transforms it into more consistent or understandable state, or (3) a documentation tool for describing the current conditional compilation structure of the code and the relationships between macros.

Ultimately, we would like to develop an IDE for creating, managing, and explaining variation. This requires developing a set of principles for interacting with dimensions and choices, and designing visualization techniques for complex choice calculus expressions. We believe the results from Section 5 and 6 will be essential here. The transformation rules can be used in editing operations, and by factoring expressions into CNF we can remove redundancy and therefore avoid update anomalies. The rules for moving dimension definitions could be employed to hide or reveal dimension information in specific contexts, depending on the intents and needs of the user.

Other directions for future work include developing an exchange format for variations. This could be used, for example, as a mode of communication between tools, and as way of managing variation in a more distributed manner. The normal forms described in Section 6 could be useful here. Exchanging expressions in DCNF would minimize redundancy and expose dimensions at the top level, perhaps simplifying the task of identifying conflicts when combining documents.

10. CONCLUSIONS

We have proposed the choice calculus as a general representation platform for variation structures. In Section 3 we have briefly compared the chosen syntax with an alternative, more direct representation. We have argued that the chosen representation is superior. But this is only one example of a number of alternative representations that we have considered. For example, in the course of developing the choice calculus, we have investigated the following three alternative representations: one that enforced that all dimensions are kept at the top level, one that required that all choices for a dimension are stored in one place, and one that did both. However, while these representations simplified some aspects of the theory (for example, the selection semantics), they all suffered from a lack of modularity, making it very difficult to combine choice expressions. The additional constraints also led to inflexibility, making transformations considerably more complex. By providing local dimension declarations and flexible choice constructs, we attain a highly modular and much more flexible and scalable representation.

Therefore, we believe that the choice calculus provides a general, flexible, and modular representation for software variation. This representation has wide applicability in computer science and software engineering, but also in a much broader range of fields. In addition to providing a common language of discourse for variation researchers and facilitating the development of sophisticated variation-supporting tools, the choice calculus could be used to represent variation in mechanical designs, project plans, workflows, travel plans, resource planning, laws/policies, human resource management, and much more.

The theoretical results presented in this paper are significant for supporting the development of sound algorithms and advanced tools for analyzing, manipulating, and explaining variations. Using these results directly we can represent variation with minimal redundancy; avoid update anomalies; recognize and purge dead alternatives, choices, and dimensions; and describe complex, semantics-preserving transformations.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for many helpful comments on earlier drafts; they have improved this paper greatly.

REFERENCES

APEL, S., LEICH, T., AND SAAKE, G. 2008a. Aspectual feature modules. *IEEE Trans. on Software Engineering 34,* 2, 162–180.

APEL, S., LENGAUER, C., MÖLLER, B., AND KÄSTNER, C. 2008b. An Algebra for Features and Feature Composition. In *Int. Conf. on Algebraic Methodology and Software Technology*. LNCS, vol. 5140. Springer-Verlang, 36–50.

ASSMANN, U. 2003. *Invasive Software Composition*. Springer-Verlang, New York.

BATORY, D. 2005. Feature Models, Grammars, and Propositional Formulas. In *Int. Software Product Line Conf.* LNCS, vol. 3714. Springer-Verlang, 7–20.

BATORY, D., LIU, J., AND SARVELA, J. N. 2003. Refinements and Multi-Dimensional Separation of Concerns. In *TODO!* 48–57.

BATORY, D., LOPEZ-HERREJON, R. E., AND MARTIN, J.-P. 2002. Generating Product-Lines of Product-Families. In *IEEE Int. Conf. on Automated Software Engineering*. 81–92.

BATORY, D. AND O'MALLEY, S. 1992. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Trans. on Software Engineering and Methodology 1,* 4, 355–398.

BATORY, D., SARVELA, J. N., AND RAUSCHMAYER, A. 2004. Scaling Step-Wise Refinement. *IEEE Trans. on Software Engineering 30,* 6, 355–371.

BRACHA, G. AND COOK, W. 1990. Mixin-Based Inheritance. In *ACM SIGPLAN Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*. 303–311.

BUCKLEY, J., MENS, T., ZENGER, M., RASHID, A., AND KNIESEL, G. 2004. Towards a Taxonomy of Software Change. *Journal of Software Maintenance and Evolution: Research and Practice*.

DATE, C. J. 2005. *Database in Depth: Relational Theory for Practitioners*. O'Reilly Media, Inc.

ERNST, M. D., BADROS, G. J., AND NOTKIN, D. 2002. An Empirical Analysis of C Preprocessor Use. *IEEE Trans. on Software Engineering 28,* 12, 1146–1170.

ESTUBLIER, J., LEBLANG, D., VAN DER HOEK, A., CONRADI, R., CLEMM, G., TICHY, W., AND WIBORG-WEBER, D. 2005. Impact of Software Engineering Research on the Practice of Software Configuration Management. *ACM Trans. on Software Engineering and Methodology 14,* 4, 383–430.

GNU PROJECT. 2009. *The C Preprocessor*. Free Software Foundation. `http://gcc.gnu.org/onlinedocs/cpp/`.

HÖFNER, P., KHEDRI, R., AND MÖLLER, B. 2006. Feature Algebra. In *Int. Symp. on Formal Methods*. LNCS, vol. 4085. Springer-Verlang, 300–315.

KANG, K. C., COHEN, S. G., HESS, J. A., NOVAK, W. E., AND PETERSON, A. S. 1990. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Tech. Rep. CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University. Nov.

KÄSTNER, C., APEL, S., AND KUHLEMANN, M. 2008. Granularity in Software Product Lines. In *IEEE Int. Conf. on Software Engineering*. 311–320.

KÄSTNER, C., APEL, S., UR RAHMAN, S. S., ROSENMÜLLER, M., BATORY, D., AND SAAKE, G. 2009. On the Impact of the Optional Feature Problem: Analysis and Case Studies. In *Int. Software Product Line Conf.* 181–190.

KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C. V., LOINGTIER, J.-M., AND IRWIN, J. 1997. Aspect-Oriented Programming. In *European Conf. on Object-Oriented Programming*. LNCS, vol. 1241. Springer-Verlang, 220–242.

LIU, J., BATORY, D., AND LENGAUER, C. 2006. Feature Oriented Refactoring of Legacy Applications. In *IEEE Int. Conf. on Software Engineering*. 112–121.

MEZINI, M. AND OSTERMANN, K. 2004. Variability Management with Feature-Oriented Programming and Aspects. *ACM SIGSOFT Software Engineering Notes 29,* 6, 127–136.

OSSHER, H. AND TARR, P. 2000. Hyper/J: Multi-Dimensional Separation of Concerns for Java. In *IEEE Int. Conf. on Software Engineering*. 734–737.

PARNAS, D. L. 1976. On the Design and Development of Program Families. *IEEE Trans. on Software Engineering 2,* 1, 1–9.

POHL, K., BÖCKLE, G., AND VAN DER LINDEN, F. 2005. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer-Verlang, Berlin Heidelberg.

ROSENTHAL, M. 2009. Alternative Features in Colored Featherweight Java. M.S. thesis, University of Passau, Germany. (In German).

SCHOBBENS, P.-Y., HEYMANS, P., AND TRIGAUX, J.-C. 2006. Feature Diagrams: A Survey and a Formal Semantics. In *IEEE Int. Requirements Engineering Conf.* 139–148.

SPENCER, H. AND COLLYER, G. 1992. #ifdef Considered Harmful, or Portability Experience With C News. In *Proc. of the USENIX Summer Conf.* 185–198.

SZEKELY, P., LUO, P., AND NECHES, R. 1992. Facilitating the Exploration of Interface Design Alternatives: The HUMANOID Model of Interface Design. In *ACM SIGCHI Conf. on Human Factors in Computing Systems*. 507–515.

TARR, P., OSSHER, H., HARRISON, W., AND SUTTON JR., S. M. 1999. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *IEEE Int. Conf. on Software Engineering*. 107–119.

TICHY, W. F. 1982. Design, Implementation, and Evaluation of a Revision Control System. *Proceedings of the 6th International Conference on Software Engineering*, 58–67.

WALRAD, C. AND STROM, D. 2002. The Importance of Branching Models in SCM. *Computer 35,* 9, 31–38.

WESTFECHTEL, B., MUNCH, B. P., AND CONRADI, R. 2001. A Layered Architecture for Uniform Version Management. *IEEE Transactions on Software Engineering 27,* 12, 1111–1133.

WINGERD, L. AND SEIWALD, C. 1998. High-level Best Practices in Software Configuration Management. *Lecture notes in computer science*, 57–66.

ZHANG, H. AND JARZABEK, S. 2004. XVCL: A Mechanism for Handling Variants in Software Product Lines. *Science of Computer Programming 53,* 3, 381–407.

APPENDIX

In this appendix we provide a proof of Theorem 1, which states that all of the transformation rules presented in Section 5 are semantics preserving. The theorem is repeated below for convenience.

THEOREM 1. *If e_L is well formed, then* $e_L \equiv e_R \implies [\![e_L]\!] = [\![e_R]\!]$

PROOF. There are two principal ways in which rules can change the semantics. First, a rule could directly change a dimension or choice; second, a rule could move a dimension declaration, choice, or let expression, which could potentially alter the binding status of choices or variables, or the ordering of tags in the domain of the semantics. Since none of this happens for the rule NAMING, the theorem holds in this case. The theorem also holds for the rules REFL, SYMM, and TRANS since equality is an equivalence relationship.

Next we consider the other rules that potentially could change the semantics. In cases where the ordering of alternatives within choices is not altered, it suffices to show that the rule in question does not change any variants, which will ensure the preservation of semantics. Note that we can always assume that choices are resolved in the semantics since e is well formed.

For the first group of choice commutation rules (C-*) we have to postulate the existence of a context C such that $C[e]$ is well formed. In particular, C has to provide one (or more) dimension definition(s) for the choice(s) involved in a rule. The choice commutation rules can only work in combination with the CONG rule, which provides this required context C.

$\boxed{\text{C-S}}$ The variations that can be produced from $a\prec e^n[i:D\langle e'^{j:1..k}\rangle]\succ$ are obtained from choices in the expressions e^n and e'^k. Lifting the choice out of e_i has two potential effects.

First, it could change the variations produced by e_i. But it is easy to see that this is not the case, because if we pick alternative j from $D\langle e'^{j:1..k}\rangle$, we obtain the variation $a\prec e^n[i:e'_j]\succ$, which is the same as when we pick alternative j from $D\langle a\prec e^n[i:e'_j]\succ^{j:1..k}\rangle$.

Second, the lifted choice could potentially shadow and deactivate choices for d in expressions $e_{j \neq i}$, that is, if we pick j in d, this will force the selection of all jth alternatives in all nested d choices and effectively synchronizes the different d choices. However, in this case this effect does not change the possible variations since all d choices in all e^n are already synchronized. Therefore, the variations and semantics are preserved.

$\boxed{\text{C-B-DEF}}$ Consider selecting the ith alternative of d. The definition of μ will produce $[e_i/v]e$ for both e_L and e_R. Lifting the choice out of the definition may shadow other choices in e. But here the same argument as in the case C-S applies, which proves this case. The case for C-B-USE runs completely analogous.

$\boxed{\text{C-D}}$ Selecting the ith alternative of d of either e_L or e_R yields the expression **dim** $d'\langle t^m \rangle$ **in** e_i. Since no subexpression is moved, the ordering of dimensions is unaffected by this rule, which guarantees that the tags in the domain of the semantics are not changed.

$\boxed{\text{C-C-SWAP}}$ Consider the selection of alternative x in choice d' and alternative y in d. We have to consider two different cases for x.

First, assume $x \neq i$. In this case e_L simply evaluates to e_x, and the choice in d is not relevant. On the other hand, e_R evaluates first to $D\langle (e_x)^k \rangle$ since the same selection is made in each of the k choices for d'. Then the selection of y yields e_x, the same as e_L.

Second, assume $x = i$. In this case e_L first evaluates to $D\langle e'^k \rangle$, for which the selection of y than produces e'_y. For e_R the selection of $x = i$ produces a different value e'_j in each of the k choices, that is, it evaluates to $D\langle e'^k \rangle$. As for e_L, selection with y yields e'_y.

Finally, since the two selections remove all other expressions, changes in the semantics domain cannot occur due to reordering of expressions.

$\boxed{\text{C-C-MERGE}}$ Selection of the ith alternative in e_L yields e'_i. For e_R we first obtain $D\langle e'^n \rangle$, which reduces through a second selection with i also to e'_i.

$\boxed{\text{B-NEW}}$ Since v is not used in e, the newly introduced **let** binding does not change the semantics of e, no matter what e' is. Moreover, since e' does not contain a dimension, the domain of the semantics will not be extended or altered in any way either.

$\boxed{\text{B-USE-USE}}$ Since $v \neq w$, exchanging the bindings for v and w does not cause any shadowing of definitions. Moreover, since neither variable occurs freely in the defining expression of the other, variable capture cannot happen after the transformation. Finally, since e does not contain a dimension definition, the reordering of expressions e and e' does not lead to a change in the ordering of the tags in the domain of the semantics. The case for B-USE-USE' is the same.

$\boxed{\text{B-S}}$ The first premise ensures that moving the **let** binding doesn't lead to the capture of variables and thus preserves the semantics. The second premise guarantees that no reordering of dimension definitions is caused by the transformation. The case for B-S' is identical, except that the second premise is replaced by $i-1$ premises that provide the same guarantee. The ordering of dimensions is preserved if e does not contain any dimension declarations, or if none of e_1 through e_{i-1} contain dimension declarations.

$\boxed{\text{B-S'}}$ This case is identical to rule B-S The other $i-1$ premises guarantee that no reordering of dimension definitions is caused by this transformation, which preserves the ordering of tags in the domain of the semantics.

$\boxed{\text{D-S}}$ This case is similar in structure to the rule B-S. The first premise ensures that moving the **dim** binding doesn't lead to the capture of choices in any of the other subexpressions, which preserves the semantics. The other $i-1$ premises again guarantee that no reordering of dimension definitions is caused by this transformation, which preserves the ordering of tags in the domain of the semantics.

$\boxed{\text{D-B-DEF}}$ Since e' does not contain a choice in dimension d, lifting the dimension definition does not lead to choice capture. Moreover, since all dimensions will be eliminated by the semantics *before* **let** expression are expanded, dimension definitions cannot be duplicated, that is, they cannot be eliminated multiple times due to multiple references to the bound variable. Therefore, lifting an expression out of a **let** definition does not change the semantics.

$\boxed{\text{D-B-USE}}$ To move a dimension expression into or out of the body of a **let** expression we have to ensure that the defining expression e does not contain any dimension definitions so that the tags in the domain of the semantics are not reordered. Moreover, we require that e does not contain any choices for d, which prevents capturing.

$\boxed{\text{CONG}}$ We know that $e_L = C[e'_L]$ and $e_R = C[e'_R]$ with $e'_L \equiv e'_R$. We can consider two cases.

First, if e'_L is well formed, we can assume by induction that $[\![e'_L]\!] = [\![e'_R]\!]$. To prove this case we use the following auxiliary lemma.

LEMMA 7. *If $\tilde{e}$ is a plain expression and both e and $C[e]$ are well formed, then* $V(C[e]) = \{(\bar{q}_1\bar{q}_e\bar{q}_2, C'[e']) \mid (\bar{q}_C, C'[\tilde{e}]) \in V(C[\tilde{e}]), (\bar{q}_e, e') \in V(e), \xi_{C'/e_0}(\bar{q}_C) = (\bar{q}_1, \bar{q}_2)\}$.

The expression $\xi_{C'/e_0}(\bar{q}_C)$ decomposes the tag sequence into two parts, those tags ($\bar{q}_1$) that are produced by V during the traversal before e_0 is encountered and those ($\bar{q}_2$) that are produced afterwards.

Now according to Lemma 7, $V(e_L) = V(C[e'_L])= \{(\bar{q}_1\bar{q}_e\bar{q}_2, C'[e'])\}$ where $(\bar{q}_C, C'[\tilde{e}]) \in V(C[\tilde{e}])$ and $(\bar{q}_e, e') \in V(e'_L)$. Since $[\![e'_L]\!] = [\![e'_R]\!]$, we also know that $V(e'_L) = V(e'_R)$ up to expansion of **let** expressions, that is, $V(e'_L)$ and $V(e'_R)$ differ, if at all, only insofar as one or the other may contain **let** expressions that the other expression doesn't, but the expansion of those **let** expressions will remove any difference. Therefore, substituting e'_R for e'_L doesn't change the result, and therefore $V(e_L) = V(e_R)$ (up to expansion of **let** expressions). Since the semantics definition is directly based on V and only performs the expansion of **let** expressions, it then follows that $[\![e_L]\!] = [\![e_R]\!]$.

Second, if e'_L is not well formed, C will provide dimension declarations and variable bindings for free choices and free variables in e'_L. Now, for all those cases in which a rule transforms an e'_L containing free choices or free variables, we have already argued that the rule does not change the semantics (cf. rules C-*). □