An Abstract Representation of Variational Graphs[†]

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

Eric Walkingshaw
Department of Computer Science
University of Marburg
walkiner@informatik.uni-marburg.de

Sheng Chen
School of EECS
Oregon State University
chensh@eecs.oregonstate.edu

Abstract

In the context of software product lines, there is often a need to represent graphs containing variability. For example, extending traditional modeling techniques or program analyses to variational software requires a corresponding notion of variational graphs. In this paper, we introduce a general model of variational graphs and a theoretical framework for discussing variational graph algorithms. Specifically, we present an abstract syntax based on tagging for succinctly representing variational graphs and other data types relevant to variational graph algorithms, such as variational sets and paths. We demonstrate how (non-variational) graph algorithms can be generalized to operate on variational graphs, to accept variational inputs, and produce variational outputs. Finally, we discuss a filtering operation on variational graphs and how this interacts with variational graph algorithms.

Categories and Subject Descriptors D.2.9 [*Software Engineering*]: Management—Software configuration management; E.1 [*Data Structures*]: Graphs and networks

Keywords Choice calculus, variational data structures, variational algorithms

1. Introduction

Graphs are a fundamental data structure in computer science and are useful for representing and solving a huge range of problems. In the context of feature-oriented software development [1] and software product lines [28], graphs are used for a wide variety of purposes. For example, graphs are used to describe the structure and relationships of the components or features that make up a system [15, 31] and to support separation of concerns [29]. Graphs in the form of UML models [25] are used to support a broad range of tasks related to the construction of component-based software product lines [2, 16].

A major goal of all of these fields is to support the creation and maintenance of *variational software*—that is, software that represents not just a single program, but can be used to generate many different program variants that provide different features and run in different environments. Therefore, it is not surprising that there is often a need to represent variation in these supporting graph representations. For example, many different extensions to UML have been proposed for incorporating variability [8, 22, 24, 33].

The need for variation in graphs is perhaps most strongly observed in the implementation of *variability-aware analyses* [21], which are the extension of traditional single-program static analyses to variational software. For example, extending traditional data-flow or control-flow analyses to variational software leads to correspondingly variational data-flow and control-flow graphs [4, 5].

Analogous to variational software, a *variational graph* represents not one, but many different plain graphs. Although many variational graph representations have been devised already, these have all been adaptations of existing representations—situated solutions that address a specific problem, with no shared underlying theory. This paper attempts to fill this gap by presenting an abstract representation of variational graphs and a theoretical framework for discussing *variational graph algorithms*. Such an underlying theory is useful since it supports the reuse of operations and theoretical results between tools, problems, and fields.

In the following we establish a set of principles on which to base an abstract representation of variational graphs and also outline the rest of the paper.

(1) The model should provide a way to *structure the variation space* to support the scalability of variation. This is typically done through a feature model that can specify relationships between different decisions, such as dependency or mutual exclusiveness. It should be possible in our model to express and precisely characterize such relationships. This principle is the focus of Section 2.

(2) It should provide a *general model of variation* that applies not only to graphs, but to the values accepted and returned by graph algorithms. This is necessary to support variational graph queries. For example, variational path analysis relies not only on variational graphs, but also on variational nodes (passed as inputs to the path-finding algorithm), and variational paths (as ouptut). Section 3 provides this model.

(3) It should suggest representations for *variational data structures* that take advantage of inherent *sharing* between alternatives. For example, the paths computed by a variational path analysis may have many nodes and edges in common. As the variability in a structure increases, it becomes increasingly important to share these common sub-parts due to a multiplicative explosion in the total number of variants. Sharing is also important for making sense of variational output; for example, the differences between alternative paths are easier to see when they are overlaid on a single graph than when examined separately. *Tagged sets* are introduced in Section 4 as a specific data structure for efficiently representing variation in sets, and this representation is extended to *tagged graphs* in Section 5. These representations realize the general variation model from Section 3 for sets and graphs, respectively.

(4) It should provide a way to *lift* algorithms to the variational setting to support the reuse of algorithms. In the case of graphs, *variational graph algorithms* should accept variational input, operate on variational graphs, and produce variational output. For exam-

[†] This work is supported by the National Science Foundation under the grants CCF-0917092 and CCF-1219165.

FOSD'13 October 26, 2013, Indianapolis, IN, USA
Copyright © 2013 ACM Copyright is held by the owner/author(s). Publication rights licensed to ACM. ACM 978-1-4503-2168-6/13/10
http://dx.doi.org/10.1145/2528265.2528270. . . $15.00

ple, we should be able to take a simple path-finding algorithm and mechanically lift it to a variational algorithm that supports finding variational paths. We show how to do this in general, in Section 6, and also show how lifted graph algorithms can take advantage of sharing in tagged graphs.

(5) Finally, it should provide a way to *filter* variational results. For example, we wanted to be able to discard alternative paths during the exploration process. We did this by making *selections* on the input, restricting the variation in the output. This feature is also important since a few independent sources of variation can lead to a potentially huge number of variants. Filters provide a way to project on variational output, in order to focus on and understand a related subset of the variants at a time. In Section 7 we describe *variational filters*, which provide a general way to project on variational values.

2. Dimensions and Decisions

In this section we describe how the variation space of a variational type can be organized in terms of its *dimensions* of variation, and how to identify particular points in this space by *decisions*. This way of structuring variability is motivated in part by our previous work on the choice calculus [10] (see Section 8).

A *dimension* describes one way in which something varies. A *dimension definition* assigns a *dimension name* to a non-empty set of *tags*, which correspond to the alternatives in that dimension. A dimension definition is written as $D = \{t_1, \ldots, t_n\}$, where D is the dimension's name and $t_1, \ldots, t_n$ are its tags. A *qualified tag* is a tag prefixed by the name of the dimension it is in, written $D.t_i$. Qualified tags are used to distinguish between tags of the same name from different dimensions. Given a qualified tag $q = D.t$, we can extract the dimension name with the function $dim(q) = D$.

A *decision space* of *degree n* is given by a set of n dimension definitions $D^n = \{D_1 = T_1, \ldots, D_n = T_n\}$ where T_i is the set of tags in dimension D_i. We define the function $dims(D^n) = \{D_1, \ldots, D_n\}$ to return the set of all dimension names in a decision space. The *tag universe* of decision space D^n, written Q_{D^n}, is the set of all qualified tags in D^n, defined as $Q_{D^n} = \{D.t \mid D \in dims(D^n) \wedge t \in D\}$.

A *decision* in a decision space D^n is a set of qualified tags $\delta \subseteq Q_{D^n}$ that contains at most one tag for each dimension, that is, $q, q' \in \delta \implies q = q' \vee dim(q) \neq dim(q')$. We overload the function *dims* to also denote the dimension names of a decision, that is, $dims(\delta) = \bigcup_{q \in \delta} dim(q)$. A decision $\delta \subseteq Q_{D^n}$ is *complete* if it contains a qualified tag from every dimension in D^n, that is, if $dims(\delta) = dims(D^n)$, otherwise it is *partial*.

However, not all dimensions are independent of one another. a *decision structure* $\mathscr{D} = (D^n, R)$ is given by a dimension space D^n and a (potentially empty) set of dependency relationships $R \subseteq 2^{Q_{D^n}} \times D^n$, where $2^{Q_{D^n}}$ is the power set of the tag universe for D^n. Given a dependency relationship $(\{D_i.t_j, \ldots, D_k.t_k\}, D') \in R$, which we write also as $D_i.t_j, \ldots, D_k.t_l \to D'$, we say that D' is a *dependent dimension* and that D' is *dependent on* the qualified tags $D_i.t_j, \ldots, D_k.t_l$. The tag universe of $\mathscr{D}$ is given by that of D^n, that is, $Q_{\mathscr{D}} = Q_{D^n}$.

A decision structure refines the concept of decision completeness given above, and it also gives rise to the idea of "overdetermination" of a decision. The formal definitions are based on the notions of *decision covering* and *dimension triggering*. First, we say that a decision δ *covers* another decision δ' if $\delta' \subseteq \delta$. Second, we say that a decision δ *triggers* the dimension D if it covers a decision δ' for a dependency $\delta' \to D \in R$.

With these two concepts we can define the completeness of decisions with respect to a decision structure as follows. A decision δ is complete if it contains a tag for each non-dependent dimension and a tag for each dependent dimension that it triggers. Formally, $\delta \subseteq Q_{\mathscr{D}}$ is *complete* with respect to $\mathscr{D}$ if the following is true.

$$\forall D \in dims(D^n) - dims(\delta).\ \forall \delta' \to D.\ \delta' \nsubseteq \delta$$

A decision structure can thus reduce the number of dimensions required in a decision to be complete. The flip side of this aspect is that dimensions in a decision can also become superfluous. We say that a decision δ is *overdetermined* if δ contains a dependent dimension D without containing the tags that D is dependent on. Formally, δ is overdetermined with respect to a decision structure $\mathscr{D} = (D^n, R)$ if the following is true.

$$\exists D.t \in \delta.\ \delta' \to D \wedge \delta' \nsubseteq \delta$$

Note that the concepts of completeness and overdetermination are independent of one another, that is, a partial as well as a complete decision can be overdetermined. Since the semantics of variational values will be defined to map to values only those decisions that contain exactly the tags needed for selecting a value and not more, we also identify the set of *exact* decisions, which are decisions that are complete, but not overdetermined. To summarize, a decision is:

- *partial* if it lacks tags to select a plain value;
- *complete* if it is not partial, that is, it has enough tags to select a plain value;
- *overdetermined* if it contains tags that are not required to select a plain value; and
- *exact* if it is complete but not overdetermined, that is, it contains exactly the tags needed to select a plain value.

We write $\Delta_{\mathscr{D}}$ for the set of all exact decisions with respect to the decision structure $\mathscr{D}$.

3. Variational Values

A *variational value* represents several different *plain values* that can be obtained by making different decisions. Given a set of plain values A, a *variational A* value over a decision structure $\mathscr{D} = (D^n, R)$ is a partial function of type $V_{\mathscr{D}}(A) = \Delta_{\mathscr{D}} \to A$ that maps exact decisions to plain values in A. Given a variational value[1] $\vec{v}$, the function *dom* returns the decisions that can be made for $\vec{v}$ and *rng* returns the variants contained in $\vec{v}$. A variational value may not be defined for all decisions, and the set $\Delta_{\mathscr{D}} - dom(\vec{v})$ contains all the *uncovered decisions* of $\vec{v}$.

The process of obtaining a plain value from a variational value $\vec{v}$ is called *selection*. A plain value is selected by applying $\vec{v}$ to an exact decision.

In addition to selecting plain values, we can also *refine* variational values by applying them to partial decisions. The result of refining a variational value is a new variational value that contains fewer variants. The refinement of a variational value $\vec{v}$ with respect to a partial decision δ is defined as a selection on the domain of $\vec{v}$.

$$\vec{v}!\delta = \{(\delta' - \delta, v) \mid (\delta', v) \in \vec{v} \wedge \delta \subseteq \delta'\}$$

Note that refinement with a complete decision δ will produce a trivial function $\{(\varnothing, v)\}$ where $v = \vec{v}(\delta)$, and refinement with an overdetermined decision will produce the empty set.

Note also that the refinement operation changes the *type* of the variational value. Dimensions contained in δ are removed from the decision space, as well as dependent dimensions that cannot be triggered anymore. Moreover, triggered dependencies will be removed, as well as dependencies whose antecedent tags contain dimensions that are also contained in δ.

In cases of successive refinements, the order of refining does not matter. This property is captured in the following lemma.

LEMMA 1. $\vec{v}!\delta!\delta' = \vec{v}!\delta'!\delta$

[1] We use arrows over variables that range over variational values to indicate that these are functions.

4. Tagged and Variational Sets

The definition of variational values is completely generic in the type of values that are being varied. While this is a useful feature that provides a common semantic basis, it does not automatically lead to succinct representations of non-atomic, structured values. Specifically, variation in structured values can be local, and the duplication of all non-varying parts in the semantic representation of a variational value is not space efficient. It is also probably hard to understand and work with in many cases.

Let us consider variational sets as a simple example of a structured value that illustrates the problem well. A solution to the representation of variational sets also forms the basis for the representation of more complicated variational structures, such as graphs.

Given a set $A = \{a,b,c,d\}$, we consider the representation of two variant A-sets $S_1 = \{a,b,c\} \in 2^A$ and $S_2 = \{a,b,d\} \in 2^A$ as a variational A-set over the decision structure $(\{D = \{t,u\}\},\varnothing)$.

The variational semantics described in Section 3 yields the following variational set.

$$\vec{S} = \{(\{D.t\},\{a,b,c\}),(\{D.u\},\{a,b,d\})\}$$

We can observe the repeated representation of $\{a,b\}$ in both variants. In this small example, this is not a problem, but if the set of shared values gets large, say n, and if the number of variant sets also grows, say to k, then this representation produces $n(k-1)$ unnecessary copies of the shared data.

A more economic representation that shares common parts would be something like this.

$$\bar{S} = \{a,b,c^{D.t},d^{D.u}\}$$

This representation assumes that unlabeled values appear in all variants of the set, whereas a value that is labeled by a qualified tag appears only in those variants that are mapped to by decisions containing that tag.

Labeling values with single tags is a bit limited and does not exploit the full potential of this approach to sharing. For example, we might want to express that a value is included only if two or more tags (from different dimensions) are selected. This could be expressed by a conjunction of tags, which allows the inclusion of a value to be restricted to more specific cases. Similarly, we can imagine cases where the same value is included only if one of some number of different tags is selected. In this case, instead of repeating the value for each tag, we could attach a corresponding disjunction of tags and represent it only once.

These ideas are combined by the concept of *tag expression* over a tag universe $Q_{\mathscr{D}}$. Tag expressions are defined by the following grammar (where q ranges over qualified tags drawn from $Q_{\mathscr{D}}$).[2]

$$e \ ::= \ q \mid e \wedge e \mid e \vee e$$

Additionally, we introduce a separate annotation $\star$ to label elements that are included in all variants. Now we can define a *tagged A set* over a decision structure $\mathscr{D}$ as a mapping of type $A \to e_\star$ where $e_\star$ represents the set of tag expressions that can be generated by the above grammar based on $\mathscr{D}$'s tag universe $Q_{\mathscr{D}}$, extended by the element $\star$. We write $\bar{S}$ for a variable representing a tagged set and $\tau_{\mathscr{D}}(A)$ for the type of tagged A sets over $\mathscr{D}$. Moreover, as indicated above, we write elements $(v,e) \in \bar{S}$ of the tagged set $\bar{S}$ using exponent notation, that is, as v^e, and we typically omit the $\star$ exponent and simply write v for $v^\star$.

The purpose of tagged A sets is to provide an economical syntax for variational A sets. In the following we therefore define the semantics of this notation and explain what variational set is denoted by a tagged set. First, using a simple qualified tag v^q means to include v in the set only if the tag q is selected. Conversely, an element $v^\star$ is included for any selection. Second, the meaning of a conjunction of tags $v^{e_1 \wedge e_2}$ is that v is included for a selection that would cause v^{e_1} and v^{e_2} to be included. Dually, a disjunction of tags $v^{e_1 \vee e_2}$ causes v to be included for selections that would cause v^{e_1} or v^{e_2} to be included.

We can define the semantics of tagged sets in two steps by first mapping single tag elements into variational sets and then combining those sets. Since variational sets are represented by functions, the combination of those functions requires a careful distinction of what to do for overlapping and non-overlapping decisions in the sets to be combined. Specifically, we can consider the two cases of computing the union and intersection of functions. The *variational union* of two variational sets $\vec{S}$ and $\vec{S}'$ takes the union of the non-overlapping parts of $\vec{S}$ and $\vec{S}'$ and computes the union of the sets mapped to by the overlapping parts. Conversely, the *variational intersection* of $\vec{S}$ and $\vec{S}'$ takes only the intersection of the sets mapped to by the overlapping parts of $\vec{S}$ and $\vec{S}'$ and drops the non-overlapping parts.

To define these operations formally we introduce an auxiliary function $\curlywedge$ that decomposes two functions into their overlapping and non-overlapping parts. The result of $\vec{S} \curlywedge \vec{S}'$ is a triple of sets, where the first and third sets capture the non-overlapping parts of $\vec{S}$ and $\vec{S}'$, respectively, and the second captures the overlapping parts.

$$\begin{aligned}\vec{S} \curlywedge \vec{S}' = (&\{(\delta,\vec{S}(\delta)) \mid \delta \in dom(\vec{S}) - dom(\vec{S}')\},\\ &\{(\delta,\vec{S}(\delta),\vec{S}'(\delta)) \mid \delta \in dom(\vec{S}) \cap dom(\vec{S}')\},\\ &\{(\delta,\vec{S}'(\delta)) \mid \delta \in dom(\vec{S}') - dom(\vec{S})\})\end{aligned}$$

With this auxiliary function we define variational union as follows.

$$\begin{aligned}\vec{S} \mathbin{\vec{\cup}} \vec{S}' &= L \cup \{(\delta,M_L \cup M_R) \mid (\delta,M_L,M_R) \in M\} \cup R\\ &\text{where } (L,M,R) = \vec{S} \curlywedge \vec{S}'\end{aligned}$$

In our example we can use $\vec{S}_1 = \{(\{D.t\},\{a,b,c\})\}$ and $\vec{S}_2 = \{(\{D.u\},\{a,b,d\})\}$, and we find that $\vec{S}_1 \mathbin{\vec{\cup}} \vec{S}_2$ yields $\vec{S}$ since $\vec{S}_1 \curlywedge \vec{S}_2$ yields $L = \vec{S}_1$, $R = \vec{S}_2$, and $M = \varnothing$.

In a similar way we can define variational intersection using the auxiliary decomposition function.

$$\begin{aligned}\vec{S} \mathbin{\vec{\cap}} \vec{S}' &= \{(\delta,M_L \cap M_R) \mid (\delta,M_L,M_R) \in M\}\\ &\text{where } (L,M,R) = \vec{S} \curlywedge \vec{S}'\end{aligned}$$

In our example, $\vec{S}_1 \mathbin{\vec{\cap}} \vec{S}_2$ yields $\varnothing$.

Now we can define the semantics of tagged values. First, we define the semantics of $\star$ tagged values and singly tagged values. Then we use the union and intersection operations to define the semantics of values that are tagged by disjunctions and conjunctions of tags, respectively.

$$\begin{aligned}[\![v^\star]\!]_{\mathscr{D}} &= \{(\delta,\{v\}) \mid \delta \in \Delta_{\mathscr{D}}\}\\ [\![v^q]\!]_{\mathscr{D}} &= \{(\delta,\{v\}) \mid \delta \in \Delta_{\mathscr{D}} \wedge q \in \delta\}\\ [\![v^{e_1 \wedge e_2}]\!]_{\mathscr{D}} &= [\![v^{e_1}]\!]_{\mathscr{D}} \mathbin{\vec{\cap}} [\![v^{e_2}]\!]_{\mathscr{D}}\\ [\![v^{e_1 \vee e_2}]\!]_{\mathscr{D}} &= [\![v^{e_1}]\!]_{\mathscr{D}} \mathbin{\vec{\cup}} [\![v^{e_2}]\!]_{\mathscr{D}}\end{aligned}$$

Finally, we can again use the variational union operation to define the semantics of tagged sets below. The type of the semantic function is $[\![\cdot]\!]_{\mathscr{D}} : \tau_{\mathscr{D}}(A) \to V_{\mathscr{D}}(2^A)$.

$$[\![\bar{S}]\!]_{\mathscr{D}} = \vec{\bigcup}_{v^e \in \bar{S}} [\![v^e]\!]_{\mathscr{D}}$$

Note that the decision structure determines the domain of the semantics. Thus, one tagged value or set will generally have different semantics under different decision structures. Specifically, an element in a tagged set will materialize as a plain element in the range of the semantics only if its tag expression respects the dependencies

[2] It is possible to extend tag expressions also by "negated tags", that is, tags whose selection will exclude elements, but we omit these here for brevity.

in the decision structure. An example of this effect can be seen in the graph example shown in Figure 1(a) (explained in more depth in the next section). The edge $(3,1)$ that is tagged $A.t \wedge B.r$ would not appear in any plain graph if the dependency was instead $\{A.u\} \to B$.

We observe that the semantics for tagged values and sets are well defined in the sense that the definition always produces variational values whose domain respects the decision structure. Consider as an example the tagged set $\{v^{D.t \wedge D.u}\}$. We expect the corresponding variational set to be empty since v cannot appear in a decision that contains both $D.t$ *and* $D.u$ since the two tags belong to the same dimension. We can verify our expectation by computing the semantics, which, as expected, produces the empty set.

$$\begin{aligned}
[\![\{v^{D.t \wedge D.u}\}]\!]_{\mathscr{D}} &= [\![v^{D.t \wedge D.u}]\!]_{\mathscr{D}} = [\![v^{D.t}]\!]_{\mathscr{D}} \vec{\cap} [\![v^{D.u}]\!]_{\mathscr{D}} \\
&= \{(\{D.t\},\{v\})\} \vec{\cap} \{(\{D.u\},\{v\})\} = \varnothing
\end{aligned}$$

Finally note that the function representation requires that in a tagged set each element can occur at most once. Thus in order to represent an element in different alternatives one must combine the corresponding tags with a disjunction.

Tagged and variational sets provide the basis for the succinct representation of a wide range of variational data structures, and thus support the two principles of *generality* and *sharing*.

5. Tagged and Variational Graphs

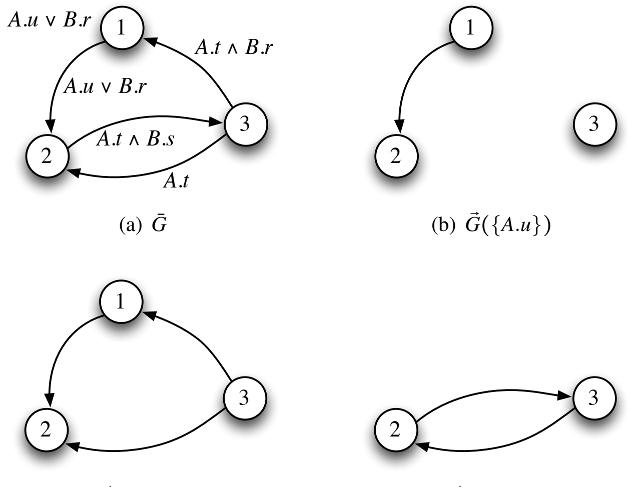
A representation for variational graphs that is flexible, allows the exploitation of sharing, and lends itself to succinct visualizations, is obtained by drawing a "supergraph" of all graph variants and labeling nodes, edges, and labels with tags to express the variational parts. This supergraph representation strategy is in principle an instance of the approach that we have already employed in the representation of variational sets using tagged sets.

Before we can define tagged graphs, however, we have to decide on a definition of (plain) directed, labeled graphs to work with. We can obtain a succinct representation when we combine the set of vertices and edges with their respective labeling functions. We therefore represent a directed, node- and edge-labeled graph by a pair of mappings (N,E) where $N : V \to L$ and $E : V \times V \to L$ represent the sets of labeled vertices and edges, respectively, and where V and L are sets that provide universes of node and label values, respectively. Such a graph is said to be *well defined* if edges connect only existing nodes, that is, $(v,w) \in dom(E) \implies \{v,w\} \subseteq dom(N)$.

Following the definition in Section 4 we now define a *tagged graph* over a decision structure $\mathscr{D}$ as a pair of mappings $(\bar{N},\bar{E})$ where the supergraph (N,E), obtained by $N = rng(\bar{N})$ and $E = rng(\bar{E})$, is a directed graph. An example is shown in Figure 1(a). To simplify the following discussion, we use, without loss of generality, an unlabeled graph.

The definition of the semantics of a tagged graph can be based on the semantics of tagged sets. Since labeled nodes and edges are functions (and thus sets), their variational versions represented as tagged functions/sets is directly amenable to the semantics of tagged sets as defined in Section 4. However, this is not all there is to do. The two resulting variational sets must still be combined into a variational graph; that is, we have to transform an element of type $(V_{\mathscr{D}}(2^N), V_{\mathscr{D}}(2^E))$ into one of type $V_{\mathscr{D}}(2^{N \times E})$. This requires some care since the tag structures of nodes and edges are principally independent of one another; that is, the variational nodes $(V_{\mathscr{D}}(2^N))$ and variational edges $(V_{\mathscr{D}}(2^E))$ that result form the semantics are not guaranteed to be synchronized. For example, there may be variants corresponding to specific decisions in one that are missing in the other.

This problem can be avoided by taking decisions only from the intersection of the domains of both variational sets, an idea



(a) $\bar{G}$ (b) $\vec{G}(\{A.u\})$

(c) $\vec{G}(\{A.t, B.r\})$ (d) $\vec{G}(\{A.t, B.s\})$

Figure 1. A tagged (unlabeled) graph $\bar{G}$ over the decision structure $(\{A = \{t,u\}, B = \{r,s\}\}, \{(\{A.t\},B)\})$ and its variants.

captured in the following definition of the semantics.

$$[\![(\bar{N},\bar{E})]\!]_{\mathscr{D}} = \{(\delta,(\vec{N}(\delta),\vec{E}(\delta))) \mid \delta \in dom(\vec{N}) \cap dom(\vec{E})\}$$
$$\text{where } \vec{N} = [\![\bar{N}]\!]_{\mathscr{D}} \text{ and } \vec{E} = [\![\bar{E}]\!]_{\mathscr{D}}$$

As an example, consider the tagged graph $\bar{G}$ shown in Figure 1(a). Its semantics $[\![\bar{G}]\!]_{\mathscr{D}}$ is shown in Figures 1(b) - (d).

Even though the semantics definition ensures that each decision leads to two sets of labeled nodes and edges, it is not guaranteed that any such graph is itself well defined. Consider, for example, the following tagged graph $\bar{G} = (\bar{N},\bar{E})$ with $\bar{N} = \{(v,l)^t, (w,k)^u\}$ and $\bar{E} = \{((v,w),m)^{\star}\}$. We assume the simple decision structure $(D = \{t,u\}, \varnothing)$, and since we have only one dimension, we use unqualified tags for brevity. The semantics of the tagged node and edge label functions are as follows.

$$\begin{aligned}
\vec{N} &= \{(\{t\},\{(v,l)\}),(\{u\},(w,k))\} \\
\vec{E} &= \{(\{t\},\{((v,w),m)\}),(\{u\},\{((v,w),m)\})\}
\end{aligned}$$

The semantics of $\bar{G}$ is thus the following function (note that $dom(\vec{N}) = dom(\vec{E}) = \{\{t\},\{u\}\}$).

$$\begin{aligned}
\vec{G} = [\![\bar{G}]\!]_{\mathscr{D}} = \{&(\{t\},(\{(v,l)\},\{((v,w),m)\})), \\
&(\{u\},(\{(w,k)\},\{((v,w),m)\}))\}
\end{aligned}$$

Now we can see that neither $\vec{G}(\{t\})$ nor $\vec{G}(\{u\})$ is well defined. For example, in the graph $(N,E) = \vec{G}(\{t\})$, we observe that $((v,w),m) \in E$ and while $(v,w) \in dom(E)$, it is not the case that $\{v,w\} \subseteq dom(N) = \{v\}$. (The situation is similar for $\vec{G}(\{u\})$ which contains the same edge but only the node w.)

Similarly, consider what happens when we change the tag expression of node 1 in Figure 1(a) from $A.u \vee B.r$ to just $B.r$. Then the graph generated by the decision $\{A.u\}$ is not well-defined since it includes the edge $(1,2)$ but not the node 1.

To ensure that the graph variants denoted by tagged graphs are well defined, we will add a condition that the tags of nodes must include the decisions of their incident edges. To this end, in Figure 5, we define a "subsumes" relationship $e \sqsubseteq e'$ on tag expressions that holds if e is more inclusive with respect to the decisions it denotes than e'. A crucial property of $\sqsubseteq$ is expressed in the following lemma that holds for arbitrary values v_1 and v_2.

$$\star \sqsubseteq e \qquad e \sqsubseteq e \qquad \frac{e \sqsubseteq e_1 \qquad e \sqsubseteq e_2}{e \sqsubseteq e_1 \vee e_2} \qquad \frac{e_1 \sqsubseteq e \qquad e_2 \sqsubseteq e}{e_1 \wedge e_2 \sqsubseteq e}$$

$$\frac{e \sqsubseteq e_1}{e \sqsubseteq e_1 \wedge e_2} \qquad \frac{e \sqsubseteq e_2}{e \sqsubseteq e_1 \wedge e_2} \qquad \frac{e_1 \sqsubseteq e}{e_1 \vee e_2 \sqsubseteq e} \qquad \frac{e_2 \sqsubseteq e}{e_1 \vee e_2 \sqsubseteq e}$$

Figure 2. Subsumes relationship on tag expressions.

LEMMA 2. $e_2 \sqsubseteq e_1 \implies \forall v_1, v_2.\ dom(\llbracket v_1^{e_1} \rrbracket_{\mathscr{D}}) \subseteq dom(\llbracket v_2^{e_2} \rrbracket_{\mathscr{D}})$

This property allows us to formulate the following theorem that places a generality condition on node tags, ensuring that nodes will be included in all decisions that produce incident edges.

THEOREM 1. *If* $\forall ((v,w),e) \in \bar{E} : \bar{N}(v) \sqsubseteq e \wedge \bar{N}(w) \sqsubseteq e$, *then* $\forall G \in rng(\llbracket (\bar{N},\bar{E}) \rrbracket_{\mathscr{D}}) : G$ *is well defined.*

We can apply this theorem to the graph $\bar{G}$ shown in Figure 1(a). Only node 1 is not tagged $\star$ and needs to be considered. Due to the reflexivity of $\sqsubseteq$, we have $\bar{N}(1) \sqsubseteq \bar{E}(1,2)$, and $\bar{N}(1) \sqsubseteq \bar{E}(3,1)$ holds since $B.r \sqsubseteq B.r$ implies $B.r \sqsubseteq A.t \wedge B.r$, which in turn implies $A.u \vee B.r \sqsubseteq A.t \wedge B.r$. We can then observe that indeed all graph variants in Figure 1(b) - (d) are well defined.

Note, however, that the condition in the theorem is an approximation, that is, there are tagged graphs that do not satisfy the condition but are still well formed.

Tagged graphs directly support the third design principle that we identified in Section 1 since they offer a succinct representation for variational graphs that exploits sharing.

6. Variational Graph Algorithms

Since the V type constructor (see Section 3) is a functor [27], it can automatically lift any function f of type $A \to B$ to a variational function $\vec{f}$ of type $V(A) \to V(B)$. We can provide the definition of $\vec{f}$ using a plain set comprehension that iterates over all decisions in the variational domain of the function.

$$\vec{f} = \{(\delta, f(a)) \mid (\delta, a) \in V(A)\}$$

However, this is generally not a very efficient method for computing $\vec{f}$ since it does not exploit any potential sharing among the different alternatives of values in $V(A)$.[3]

What we would like to do instead is lift f based on a more economical representation, such as the one provided by tagging, which can exploit the sharing in the representation. In other words, we would like to find a definition for $\bar{f}$ that is correct with respect to $\vec{f}$. We will illustrate this process with a simple example.

Consider the following function *path* for finding all directed paths between two nodes (in some given graph G). Here [] denotes an empty path, and $v\!:\!p$ puts the node v at the beginning of the path p, but only if p doesn't already contain v, to avoid cycles.

$$path(v,v) = \{[\,]\}$$
$$path(v,w) = \{v\!:\!p \mid p \in path(u,w) \wedge (v,u,l) \in E\}$$

To generalize this function to work on tagged graphs, we simply copy tag expressions from traversed edges to the nodes that are reached via those edges in the constructed paths. In the constructed paths we use the conjunction of the edge tag and the tag of the target node to find only paths that satisfy both tag constraints.

$$p\bar{a}th(v,v) = \{[\,]\}$$
$$p\bar{a}th(v,w) = \{v\!:\!u^{e \wedge \bar{N}(u)}\!:\!p \mid u\!:\!p \in p\bar{a}th(u,w) \wedge ((v,u,l),e) \in \bar{E}\}$$

For the tagged graph $\bar{G}$ in Figure 1, *p̄ath* computes one tagged path between nodes 1 and 2, namely $[1, 2^{A.u \vee B.r}]$, and the following two paths between nodes 3 and 2.

$$p\bar{a}th(3,2) = \{[3, 1^{(A.t \wedge B.r) \wedge (A.u \vee B.r)}, 2^{A.u \vee B.r}], [3, 2^{A.t}]\}$$

To judge the correctness of the algorithm, we need a semantics for tagged paths, which is given by variational paths, that is, by functions from decisions to plain paths. For the semantics of a tagged path $\bar{p}$ computed in a tagged graph $\bar{G}$ the following consistency criterion must hold.

$$\forall (\delta, p) \in \llbracket \bar{p} \rrbracket_{\mathscr{D}}.\ p \text{ is a path in } \llbracket \bar{G} \rrbracket_{\mathscr{D}}(\delta)$$

To ensure this property we gather the tags of all nodes in $\bar{p}$ and form a conjunction of all of them, effectively computing the most restrictive tag from all tags, written as $\bigwedge_{(v,e) \in \bar{p}} e$. We attach this tag to the plain path p, which is obtained from $\bar{p}$ by removing the tags from all nodes, and then apply the standard semantics for tagged values.

$$\llbracket \bar{p} \rrbracket_{\mathscr{D}} = \llbracket p^{\bigwedge_{(v,e) \in \bar{p}} e} \rrbracket_{\mathscr{D}}$$

With this definition we can now define the semantics of the tagged path algorithm as follows.

$$\llbracket p\bar{a}th(v,w) \rrbracket_{\mathscr{D}} = \vec{\bigcup}_{\bar{p} \in p\bar{a}th(v,w)} \llbracket \bar{p} \rrbracket_{\mathscr{D}}$$

As an example, consider the computation of $\llbracket p\bar{a}th(3,2) \rrbracket_{\mathscr{D}}$. Given $\bar{p} = [3, 1^{(A.t \wedge B.r) \wedge (A.u \vee B.r)}, 2^{A.u \vee B.r}]$, we obtain $p = [3,1,2]$ as the plain path and $((A.t \wedge B.r) \wedge (A.u \vee B.r)) \wedge (A.u \vee B.r)$ for its aggregated tag, which can be simplified to $e = (A.t \wedge B.r) \wedge (A.u \vee B.r)$. Now we can compute $\llbracket \bar{p} \rrbracket_{\mathscr{D}}$ as follows.

$$\begin{aligned}
\llbracket \bar{p} \rrbracket_{\mathscr{D}} &= \llbracket p^{e} \rrbracket_{\mathscr{D}} \\
&= \llbracket p^{A.t \wedge B.r} \rrbracket_{\mathscr{D}} \mathbin{\vec{\cap}} \llbracket p^{A.u \vee B.r} \rrbracket_{\mathscr{D}} \\
&= \llbracket p^{A.t} \rrbracket_{\mathscr{D}} \mathbin{\vec{\cap}} \llbracket p^{B.r} \rrbracket_{\mathscr{D}} \mathbin{\vec{\cap}} (\llbracket p^{A.u} \rrbracket_{\mathscr{D}} \mathbin{\vec{\cup}} \llbracket p^{B.r} \rrbracket_{\mathscr{D}}) \\
&= \{(\{A.t, B.r\}, \{p\}), (\{A.t, B.s\}, \{p\})\} \mathbin{\vec{\cap}} \{(\{A.t, B.r\}, \{p\})\} \\
&\quad \mathbin{\vec{\cap}} (\{(\{A.u\}, \{p\})\} \mathbin{\vec{\cup}} \{(\{A.t, B.r\}, \{p\})\}) \\
&= \{(\{A.t, B.r\}, \{p\})\} && \text{by definition of } \vec{\cap} \\
&\quad \mathbin{\vec{\cap}} \{(\{A.u\}, \{p\}), (\{A.t, B.r\}, \{p\})\} && \text{by definition of } \vec{\cup} \\
&= \{(\{A.t, B.r\}, \{p\})\} && \text{by definition of } \curlywedge
\end{aligned}$$

This complicated computation can be simplified significantly if we first simplify the tag expression e as follows.

$$\begin{aligned}
e &= (A.t \wedge B.r) \wedge (A.u \vee B.r) \\
&= A.t \wedge B.r \wedge A.u \vee A.t \wedge B.r \wedge B.r && \text{distributivity} \\
&= A.t \wedge B.r \wedge B.r && A.t \wedge A.u = \text{false} \\
&= A.t \wedge B.r && \text{absorption}
\end{aligned}$$

Note that the tag expression $A.t \wedge B.r \wedge A.u$ is unsatisfiable because it needs both tags $A.t$ and $A.u$, whereas we can choose at most one tag from each dimension. Thus, this expression can be viewed as the zero identity to the connective $\vee$.

With the simplified tag expression, the semantics of $\llbracket p^{A.t \wedge B.r} \rrbracket_{\mathscr{D}}$ can be easily computed. Similarly, given $\bar{p} = [3, 2^{A.t}]$ and thus $p = [3,2]$ we obtain $\llbracket \bar{p} \rrbracket_{\mathscr{D}} = \{(\{A.t, B.r\}, \{p\}), (\{A.t, B.s\}, \{p\})\}$. We can put these two results together and obtain the following.

$$\begin{aligned}
\llbracket p\bar{a}th(3,2) \rrbracket_{\mathscr{D}} = \{&(\{A.t, B.r\}, \{[3,1,2], [3,2]\}), \\
&(\{A.t, B.s\}, \{[3,2]\})\}
\end{aligned}$$

[3] Note, however, that we can always compute $\vec{f}$ lazily, that is, only when $\vec{f}$ is applied to some decision δ will we actually compute f for the value $A(\delta)$. We can then also memoize the result.

The semantics tell us that when the tags $A.t$ and $B.r$ are chosen, the resulting graph will have two different paths from node 3 to node 1, namely the paths $[3,1,2]$ and $[3,2]$. The correctness can be easily verified against Figure 1(c). Similarly, when $A.t$ and $B.s$ are chosen, the resulting graph has one path $[3,2]$, which can again be verified against Figure 1(c).

The correctness of the algorithm $\bar{path}$ is expressed in the following lemma where $\vec{path}$ gives the "semantic standard" by lifting the path function to the variational case.

LEMMA 3. $[\![\bar{path}(v,w)]\!]_{\mathscr{D}} = \vec{path}(v,w)$

Just like *path*, the algorithm $\bar{path}$ traverses the graph only once and is thus in general more efficient than the variational version $\vec{path}$. The expansion of tagged paths into a set of paths does cause some additional effort, but the efficiency gains are obvious in cases where one sub(path) occurs in *n* variants. The brute-force algorithm $\vec{path}$ will compute that path *n* times, whereas $\bar{path}$ does this only once. However, the efficiency gain of $\bar{path}$ over $\vec{path}$ depends on the graph structure. For a given tagged graph $\bar{G}$, the efficiency gain is significant if there is a lot of sharing among different variants.

Lemma 3, together with all of its efficiency benefits, extends naturally to many other graph algorithms. This result is important since it supports our fourth principle from Section 1 by providing a general roadmap to extend algorithms employed in graph applications to the variational case, along with their graph representations.

7. Variational Filters

Variational graphs, and variational values in general, are binary relations between decisions and plain values. Filters on variational graphs can therefore be conveniently expressed by pairs of predicates on the domain and range of these relations. We have already encountered refinement in Section 3 as an example of one such query operation. Refinement was basically a selection on a relation's domain. Similarly, we can envision selection on the range.

In general, given a variational value $\vec{v}$ of type $V_{\mathscr{D}}(A)$, a $V_{\mathscr{D}}(A)$-*filter* on $\vec{v}$ is a given by a pair of predicates $\phi = \langle\alpha,\beta\rangle$ where $\alpha : \Delta_{\mathscr{D}} \to \mathbb{B}$ and $\beta : A \to \mathbb{B}$ represent the domain and range predicates, respectively. We call α the *decision predicate* and β the *value predicate*. The semantics of a filter is the transformation of a variational value obtained by applying the decision and value predicates to the two respective parts of the relation representing the variational value. Since the filtering step might leave redundant fragments in the decisions—that is, qualified tags that occur in every entry—those parts are removed from the decisions of the resulting relation.

$$\begin{aligned} [\![\langle\alpha,\beta\rangle]\!]_{\mathscr{D}}(\vec{v}) &= \{(\delta - K, v) \mid (\delta, v) \in R\} \\ \text{where } R &= \{(\delta,v) \in \vec{v} \mid \alpha(\delta) \wedge \beta(v)\} \\ K &= \textstyle\bigcap_{\delta \in dom(R)} \delta \end{aligned}$$

With the "accept everything" predicate $\iota = \lambda x.true$ that always returns true, we can identify some special cases of filters. For example, refinement with a decision δ (defined earlier in Section 3) is given by the filter $\langle \lambda x.x = \delta, \iota\rangle$. Refinement corresponds to partial decision making. The dual operation, which restricts a variational value by a predicate β on values is given by $\langle\iota,\beta\rangle$.

From the semantics definition of filters it follows directly that the composition of filters can be performed component-wise, that is, by forming conjunctions of the decision and value predicates. (In the following definition, we lift conjunction from values to predicates, that is, we write more shortly $\alpha \wedge \alpha'$ for the expression $\lambda x.\alpha(x) \wedge \beta(x)$.)

LEMMA 4. $\langle\alpha,\beta\rangle \circ \langle\alpha',\beta'\rangle = \langle\alpha\wedge\alpha',\beta\wedge\beta'\rangle$

Since lifted variation computations do not change the domain of variational values, that is, $dom(\vec{f}(\vec{v})) = dom(\vec{v})$, they are unaffected by changes to the decisions in variational values and thus commute with decision refinements, a result summarized in Lemma 5.

LEMMA 5. $\langle\alpha,\iota\rangle \circ \vec{f} = \vec{f} \circ \langle\alpha,\iota\rangle$

The importance of this lemma lies in the fact that, when applied from left to right, it can save potentially costly computations of f for all those decisions that are eliminated by α.

Commuting a filter with a non-trivial value predicate and a transformation is generally much more difficult, and for lack of space we will not explore this aspect here in detail. Instead we provide some preliminary results.

If we apply a filter after a transformation, as in $\langle\alpha,\beta\rangle \circ \vec{f}$, the predicate $\beta : B \to \mathbb{B}$ will generally reduce the set of values produced by the transformation $\vec{f} : A \to B$. Therefore, in order to apply β before f, we have to apply a modified version of it that excludes only those A values whose image under f would be excluded by β, that is, we need a predicate $\beta_f : A \to \mathbb{B}$ that has the following property.

$$\neg\beta_f(a) \Leftrightarrow \neg\beta(f(a))$$

We call such a predicate the *f-prefilter* for β. Having such a predicate available, we can move the filter as follows. Since the value predicate is not needed anymore, we can remove the whole filter following f.

LEMMA 6. $\langle\iota,\beta\rangle \circ \vec{f} = \vec{f} \circ \langle\iota,\beta_f\rangle$

This lemma is probably not very useful in practice since it is generally difficult to construct a prefilter. A more promising approach for exploiting knowledge about the presence of post-algorithmic filters is to integrate the filter into the algorithm itself. Unfortunately, this is not a general solution and requires much effort. In many cases, a better approach is to apply an algorithm that modifies the plain values directly instead of applying a filter. For example, if $\vec{f}$ is a shortest path algorithm and β requires that certain nodes not be in the resulting paths, it is generally not possible to achieve this by removing some graphs from $\vec{v}$ altogether. However, removing the nodes from the graph would cause the shortest path algorithm to produce the desired results.

As a corollary of Lemmas 5 and 6 we obtain the following result.

THEOREM 2. $\langle\alpha,\beta\rangle \circ \vec{f} = \vec{f} \circ \langle\alpha,\beta_f\rangle$

As already mentioned, the construction of prefilters is a difficult problem. Therefore, Lemma 5 will probably be applicable more often and thus be more relevant.

On the other hand, we observe that we can always at least partially commute a filter with an algorithm by employing Lemma 4, a result captured in the following theorem.

THEOREM 3. $\langle\alpha,\beta\rangle \circ \vec{f} = \langle\iota,\beta\rangle \circ \vec{f} \circ \langle\alpha,\iota\rangle$

This result reveals a general strategy for evaluating queries in a variational setting, namely to first apply refinements, then run the algorithm, and finally apply value filters. The fact that variational filters are open to optimization, as illustrated by the lemmas and theorem, support the fifth principle we identified in Section 1.

8. Related Work

The feature-oriented software development (FOSD) [1] and software product line (SPL) [28] communities are intimately concerned with creating and managing variational software. Since the construction of plain software is often supported by various graph representations, there is a need to represent variation in these supporting graph representations as well. There are many examples of such adaptations already, which will be described in this section. Compared to the model presented in this paper, these variational graph representations are in some sense domain-specific, adapting specific object languages in order to solve a particular problem.

An important distinction to make, however, is between graph representations that *contain* variability and graph representations that *describe* variability. In FOSD terms, the first is related to the problem of *feature implementation* while the second is related to the problem of *feature modeling* [20].

Feature diagrams [18] are a common notation for feature modeling. Although feature diagrams are typically described as trees, they are actually directed graphs when cross-tree constraints are taken into account [30]. However, a feature diagram does not itself contain variation—instead it *describes* the high-level structure of the variation in some other artifact, typically a software system. In this way, feature diagrams are more similar to our dimensions of variation and decisions structures (Section 2) than to variational graphs. In terms of expressiveness, many feature diagram notations are functionally complete, making them more expressive than our decision structure representation. Our decisions structures provide only implications and disjunctions of implications, which are not by themselves functionally complete [32]. Our representation of decisions structures would be complete if extended by a negation operation. Feature diagrams are equivalent to many other representations of feature models, such as propositional formulas [3].

On the other hand, the tagged graph representation (Section 5) *contains* variation—by making selections we can generate many different plain graph variants. Similarly, to support efficient model checking of SPLs, Classen et al. [6, 7] extend the directed graph notation of *transition systems* with a way to tag edges that correspond only to certain features in a SPL, which may be optional. They call this representation *featured transition systems* (FTS). The FTS notation resembles our tagged graphs, although there are several important differences. First, the role of decision structures in tagged graphs is played in FTS by an associated feature diagram. Each node in the feature diagram corresponds to a potential tag in FTS. Second, tagged graphs support a more flexible and precise form of variation than FTS. In FTS, each edge can be associated with only a single tag, and nodes are not variational at all. Tagged graphs associate tag *expressions* (Section 4) with both nodes and edges. Finally, to overcome the limitations of tagging in FTS, a notion of edge priority is introduced, where some edges, if present, supersede and remove other edges. Although this solution supported their variational model-checking algorithm, it is rather ad hoc and demonstrates the need for a consistent, general model for variational graphs. A similar but less expressive representation was introduced by Lauenroth et al. [19], also in the context of model checking product lines. Compared to FTS, this representation allows only simple tags on edges rather than boolean expressions. Like FTS, but unlike tagged graphs, nodes cannot be varied.

Relatedly, Fischbein et al. [14] describe how the graph-based representation of *modal transition systems* (MTS) [17] can be used to model variational behaviors in software product lines. However, compared to FTS and our tagged graph representation, the variability in these systems is quite unstructured since there is no corresponding feature model or decision structure. In other words, all variational transitions are considered independent of one another. This means that, in general, the MTS encoding of a SPL contains many more variants than the SPL. Fantechi and Gnesi [12, 13] extend MTS with new variation operators that allow many kinds of sophisticated cardinality-based constraints on variational transitions, such as, at least k of a set of n transitions must be included. However, it still does not provide a general mechanism for synchronizing variational transitions in different parts of a MTS. Finally, Muschevici et al. [23] extended petri nets with variational constructs. An interesting aspect of their approach is that it allows the elimination of variability through the execution of petri nets.

Multiple researchers have extended UML class diagrams and sequence diagrams to incorporate variability [22, 24, 33]. These extensions again resemble our tagged graph representation, except that only nodes are tagged. Also, the set of available tags is fixed and not determined by an associated decision structure (or feature diagram, as in Classen et al.). For example, optional class nodes can be tagged directly as *optional*. A class tagged with the *variation* tag denotes a variation point, whose alternatives are denoted with *variant* tags. Constraints between different variation points and optional classes, such as mutual requirement and exclusion, can be expressed using the Object Constraint Language of UML [25].

The Common Variability Language (CVL) by the Object Management Group (OMG) is an ongoing effort to standardize the representation of variation in model-based engineering [26]. The overarching goal of CVL is to achieve orthogonality [28] with respect to the domain and language of the underlying model. To this end, CVL adds variability modeling constructs to any modeling language based on OMG standards. CVL consists of several different kinds of models. A *variability model* is linked to nodes and edges in the underlying base model. The links denote potential actions to modify the base model. The modifications are controlled by a *variability specification tree*, which is similar to a feature model. An individual configuration is given by a so-called *resolution model*, which describes the modifications to be applied to the base model, resulting in a *resolved model*. Compared to CVL, the representations proposed in this paper are simpler and more abstract. While variational graphs and tagged graphs offer a straightforward and systematic approach to formulate variational graph algorithms, it is not clear how to achieve this with CVL.

Czarnecki and Antkiewicz present another approach to add variability to modeling languages in an othogonal way [8]. Their approach associates presence conditions with model elements, similar to tagged graphs. When generating graph variants, the elements whose presence conditions evaluate to false are removed. Unlike tagged graphs, presence conditions are externally associated with nodes (addressing them via XPath), so the variation structure is not apparent when looking at the graph itself.

Brabrand et al. [5] describe several different ways of adapting existing static data-flow and control-flow analyses to variational software. This is supported by extending the representation of data/control-flow graphs with an inclusion condition associated with each node, describing under which configurations the node is present in the graph. This is similar to our own tagged graph representation except that only nodes are variational, not edges. Additionally, it is assumed that if nodes A and C are connected through node B, A and C will still be connected even if B is excluded. An alternative approach to lifting flow-based analyses to SPLs is described by Bodden et al. [4]. In this approach, the data/control-flow graphs are computed in the usual way, then supplemented by conditional edges. While this representation has some benefits over the representation used by Brabrand et al., the result does not really represent a variational data/control-flow graph in the sense described in this paper since it is not possible to generate variant flow graphs corresponding to each product of the SPL.

In our own previous work on the *choice calculus* [10], we have developed a formal variation representation that offers many algebraic laws to support the flexible transformations of variation representations. Additionally, we have introduced the idea of variational data structures, and described how to extend algebraic data types with choice calculus-based variation [11]. One difference between the choice calculus and this work is that dimensions of variation can be locally scoped in the choice calculus, while they are globally defined in the decision structure corresponding to a variational graph. Even more importantly, the choice calculus operates on tree-structured data (such as a program's abstract syntax tree), and is therefore not directly applicable to graphs. However, it could be applied to an inductive representation of graphs by an algebraic data type [9].

9. Conclusions and Future Work

We have introduced a model of variational graphs that can serve as a formal foundation for extending graph-based applications with variation. The ability to consider different versions of graph data side by side, to structure these versions systematically, and observe the corresponding variation in the results opens many exciting opportunities in these areas. The variational graph model can serve as a framework for systematically introducing variation into graph-based applications.

We have also identified tagged graphs as a succinct syntactic representation for variational graphs and demonstrated how this may provide an efficient representation for variational graph algorithms. However, we have barely scratched the surface of this area. The question of finding efficient variational and tagged graph algorithms for all kinds of graph problems is a wide open research field that can be the subject of future research efforts.

References

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

[2] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Peach, J. Wust, and J. Zettel. *Component-Based Product Line Engineering with UML*. Addison-Wesley Professional, 2002.

[3] D. Batory. Feature Models, Grammars, and Propositional Formulas. In *Int. Software Product Line Conf.*, volume 3714 of *LNCS*, pages 7–20. Springer-Verlang, 2005.

[4] E. Bodden, M. Mezini, C. Brabrand, T. Tolêdo, M. Ribeiro, and P. Borba. SPLlift - Statically Analyzing Software Product Lines in Minutes Instead of Years. In *ACM Conf. on Programming Languages Design and Implementation*, June 2013.

[5] C. Brabrand, M. Ribeiro, T. Tolêdo, and P. Borba. Intraprocedural Dataflow Analysis for Software Product Lines. In *Int. Conf. on Aspect-Oriented Software Development*, pages 13–24, 2012.

[6] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. Symbolic Model Checking of Software Product Lines. In *Int. Conf. on Software Engineering*, pages 321–330, 2011.

[7] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines. In *ACM/IEEE Int. Conf. on Software Engineering*, pages 335–344, 2010.

[8] K. Czarnecki and M. Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *Int. Conf. on Generative Programming and Component Engineering*, pages 422–437, 2005.

[9] M. Erwig. Inductive Graphs and Functional Graph Algorithms. *Journal of Functional Programming*, 11(5):467–492, 2001.

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

[11] M. Erwig and E. Walkingshaw. Variation Programming with the Choice Calculus. In *Generative and Transformational Techniques in Software Engineering*, pages 55–99, 2012.

[12] A. Fantechi and S. Gnesi. A Behavioural Model for Product Families. In *European Software Engineering Conf. and ACM SIGSOFT Symp. on the Foundations of Software Engineering, Companion Papers*, pages 521–524, 2007.

[13] A. Fantechi and S. Gnesi. Formal modeling for product families engineering. In *Int. Software Product Line Conf.*, pages 193–202, 2008.

[14] D. Fischbein, S. Uchitel, and V. Braberman. A foundation for behavioural conformance in software product line architectures. In *ISSTA Workshop on the Role of Software Architecture for Testing and Analysis*, pages 39–48, 2006.

[15] D. Garlan, R. T. Monroe, and D. Wile. Acme: Architectural Description of Component-Based Systems. *Foundations of Component-Based Systems*, 68:47–68, 2000.

[16] H. Gomaa. *Designing Software Product Lines with UML*. Addison-Wesley, Reading, MA, 2004.

[17] M. Huth, R. Jagadeesan, and D. Schmidt. Modal Transition Systems: A Foundation for Three-Valued Program Analysis. In D. Sands, editor, *Programming Languages and Systems*, volume 2028 of *LNCS*, pages 155–169. Springer, 2001.

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

[19] K. Lauenroth, K. Pohl, and S. Toehning. Model Checking of Domain Artifacts in Product Line Engineering. In *Int. Conf. on Automated Software Engineering*, pages 269–280, 2009.

[20] K. Lee, K. C. Kang, and J. Lee. Concepts and Guidelines of Feature Modeling for Product Line Software Engineering. In *Software Reuse: Methods, Techniques, and Tools*, volume 2319 of *LNCS*, pages 62–77. Springer-Verlang, 2002.

[21] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Scalable Analysis of Variable Software. In *European Software Engineering Conf. and ACM SIGSOFT Symp. on the Foundations of Software Engineering*, 2013. To appear.

[22] J. Liu, J. Dehlinger, and R. Lutz. Safety analysis of software product lines using state-based modeling. *J. Syst. Softw.*, 80(11):1879–1892, Nov. 2007.

[23] R. Muschevici, D. Clarke, and J. Proenca. Feature petri nets. In *14th Int. Software Product Line Conference (SPLC 2010)*, volume 2, 2010.

[24] D. Muthig and C. Atkinson. Model-Driven Product Line Architectures. In *Int. Conf. on Software Product Lines*, pages 110–129, 2002.

[25] Object Management Group. Unified Modeling Language Specification Version 2.0: Superstructure. Technical Report PCT/03-08-02, OMG, 2003.

[26] Object Management Group. Common Variability Language (CVL). Technical Report ad/2012-08-05, OMG, 2012.

[27] B. C. Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, Cambridge, MA, 1991.

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

[29] M. P. Robillard and G. C. Murphy. Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies. In *Int. Conf. on Software Engineering*, pages 406–416, 2002.

[30] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. Generic Semantics of Feature Diagrams. *Computer Networks*, 51(2):456–479, 2007.

[31] J. van Gurp, J. Bosch, and M. Svahnberg. On the Notion of Variability in Software Product Lines. In *IEEE/IFIP Conf. on Software Architecture*, pages 45–54, 2001.

[32] J. E. Whitesitt. *Boolean Algebra and its Applications*. Addison-Wesley, Reading, MA, 1961. Reprint, Dover, Mineola, NY, 1995, p. 68.

[33] T. Ziadi, L. Hélouët, and J.-M. Jézéquel. Towards a UML Profile for Software Product Lines. In *Software Product-Family Engineering*, pages 129–139. Springer, 2004.