H-0285 (H1007-003) June 30, 2010
Computer Science

IBM Research Report

SPLGraph: Towards a Graph-Based Formalism for Software Product Lines

Itay Maman
IBM Research Division
Haifa Research Laboratory
Mt. Carmel 31905
Haifa, Israel

Goetz Botterweck
Lero - The Irish Sofware Engineering Research Centre
Limerick, Ireland

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

SPLGraph: Towards a Graph-based Formalism for Software Product Lines

Itay Maman
IBM Haifa Research Lab
Haifa, Israel
imaman@il.ibm.com

Goetz Botterweck
Lero – The Irish Software Engineering Research Centre
Limerick, Ireland
goetz.botterweck@lero.ie

ABSTRACT

This paper presents SPLGraph a graph-based model for Software Product Lines, including (1) a formal definition; (2) an algorithm that applies configuration decisions to an SPLGraph thus yielding a product specific graph; (3) a set of patterns for typical SPLGraph structures, such as Boolean operators, reuse of expressions, named configurations, optional and alternative features and staged configuration; and (4) an algorithm that infers product configuration per a variability point.

SPLGraph is generic, simple, and self sustaining in the sense that an SPLGraph instance can apply variability to itself. These properties make SPLGraph a basis for a solid and complete formalism for Software Product Lines.

Keywords

Software product lines; variability modeling; graph;

1. INTRODUCTION

> *A software product line (SPL) is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way* [1].

Like other definitions of SPL, this definition states an SPL is developed from a "common set of core assets" without imposing any concrete requirements on the actual assets—both core (shared, common) assets and non-core assets—making up the SPL.

Despite the wide spectrum of possibilities allowed by this definition, most modern SPL systems share a stricter common denominator formed by the usage of technologies such as variability modeling, feature modeling, variability patterns, configuration parameters, etc.

[1]http://www.sei.cmu.edu/productlines

While some of these concepts have been properly formalized (e.g., feature-modeling [13, 3, 1, 7, 15]) we have yet to see a precise model that captures the overall relationships between products, product families, assets and variability in a generic (i.e., non domain specific) manner. This is a predicament for both tool vendors and researchers.

For tool vendors, lack of a unifying formalization means they need to "reinvent the wheel" whenever they orchestrate the different mechanisms in a new tool. This leads to ad-hoc, incomplete, solutions which are hard to standardize. In the absence of a unifying formalism the vendor's solution is often coupled—too tightly—to a specific market or a customer. Finally, interoperability across tools and transfer of solutions between tools is also difficult.

For researchers, lack of formalization means lack of terminology, semantics and building blocks which are essential for devising new algorithms. More importantly, this implies the absence of well defined ways for capturing the problems that the researchers are asked to solve.

A case in point is that of modern development platforms such as IBM's Jazz project [9], which attempt to unify and streamline multiple software life-cycle activities. While maintaining traceability between artifacts that describe distinct aspects of the system (user stories, unit tests, source code, user manuals, deployment scripts, etc.) is already a challenging problem, doing so in the presence of variability is nearly impossible if every aspect/tool has its own way of representing variability, features or products.

It seems that the handling of SPL issues is a concern that crosscuts all other which should be treated by a dedicated mechanism that is capable of serving each aspect-specific tool with the appropriate product-specific variants of the relevant artifacts. This is not unlike the way versioning is handled by source control tools. Instead of letting each (aspect-specific) tool implement its own support for source control, a uniform "source control" concept was devised. This concept, which is built around notions such as revisions, branches, and baselines, is agnostic of the particularities of a specific aspect. A source control system is a realization of this concept. It frees other tool vendors (which are not focusing on source control) from reimplementing such support and, at the same time, it fosters interoperability by offering consistent semantics for the said notions.

It is our belief that SPL support should follow a similar path in the sense that variability (and constraints on the available variability options) should be described in a generic form and independently from the particular type of artifact. Eventually this may lead to the appearance of a new breed

of source management tools that will encompass both source control and SPL capabilities.

This paper presents *SPLGraph* , A graph-based mathematical structure that attempts to pave the road toward such a unifying formalism. SPLGraph is accompanied with several algorithms, working on top of an SPLGraph instance, providing services such as materialization, extraction of unresolved decision (per a variability point), etc. While we do discuss a prototype implementation (Sec.5) SPLGraph is not an SPL tool but rather a formal underpinning of SPL.

Background. Modern SPL tools are based on two key mechanisms: (i) *Variability Mechanism*: Operators that manipulate SPL assets, transforming them from their generic form into their product-specific form; (ii) *Control Mechanism*: A language through which one can specify which variability operators will be applied in a specific product, typically via high-level abstractions. A particular example for such a language is that of feature diagrams [11].

We observe that the question of expressive power is solved in both mechanisms. Variability—If we treat all assets as sequences of binary bits, then a set of operators such as: *replace-bit*, *delete-bit*, *insert-bit* is enough for expressing any possible transformation. Control—By using a Turing-complete language one can express arbitrarily rich control logic.

Clearly, both solutions are far from being optimal. Developing SPL tools that are based on manipulation of assets in granularity of bits is highly error prone and practically infeasible. Using a Turing-complete language as a control mechanism prevents meaningful reasoning on the configurations (products). In particular, questions such as: *are two configurations equivalent?* or *is this a legal configuration?* cannot be answered without running the corresponding materialization (which, alas, is not guaranteed to terminate under such circumstances).

We are therefore looking at a double trade off. In variability operators we trade the power to transform for higher level of abstraction. In the control mechanism we trade the power to configure for the ability to reason about the configurations.

Design Considerations. SPLGraph chooses a directed, labeled, graph as its data representation medium, where rerouting of edges is its single variability operator. Clearly, graphs stand at a higher level of abstraction than sequences of bits and thus are more descriptive and less prone to errors. Still, the price that we pay for this higher abstraction is minimal: by rerouting edges one can change almost any piece of information communicated by a graph. We would argue that in a way graphs are the sweet-spot for striking the variability mechanism trade off.

SPLGraph's control language, realized by *decision vertices*, is standard boolean algebra. As shown in the past [1] such a language is sufficiently powerful for expressing the common constructs of feature diagrams.

The second decision that drove the design of SPLGraph is that of *self sustainability*. In addition to the "plain" data being represented as a graph, both variability and control are represented as vertices/edges (namely: as *mutation* and decision vertices). This allows one to use SPLGraph's power to enrich SPLGraph itself, as will be illustrated—for instance—by the *Named Configuration Reuse* pattern (Sec.3.3).

Self sustainability is paramount to ensuring flexibility and future adaptability. To see that let us imagine a scenario where decision and mutation vertices were some mathematical structure foreign to the graph. If we were to provide a way to customize decisions or mutations we had to introduce another construct dedicated to this purpose which, in turn, would require the introduction of yet another construct for its own customization.

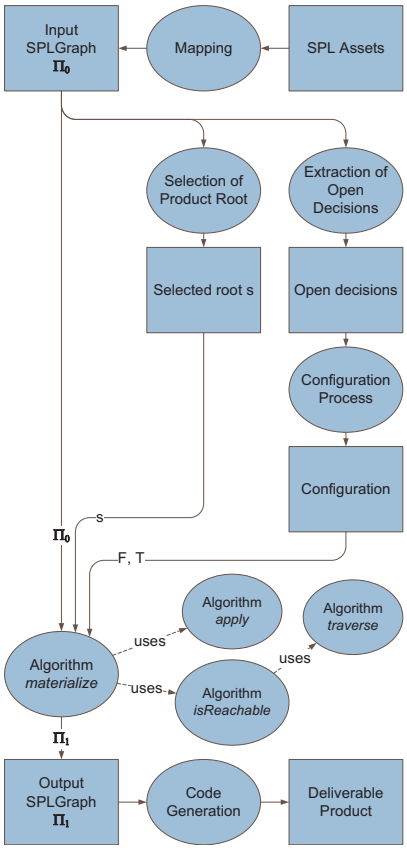

Figure 1: Overview of the engineering process. Given an SPLGraph user selects a product root and resolves open decisions. The *materialize* algorithm will produce a new SPLGraph by the application of various mutations.

Self sustainability implies no new customization means are needed for customizing the customization means already built into SPLGraph. This is similar, in principle, to the object model underlying modern object oriented languages [4]. In Smalltalk, for example, an object is an instance of a class, but a class is also an object, thereby making mechanisms such as inheritance, overriding, etc. applicable not only for objects but for the classes themselves.

Finally, we intentionally kept the formalism simple: a single variability operator, and only two kinds of special vertices. This resulted in simple algorithms, in simple correctness proofs, and in the ability to express a wide spectrum of high level patterns in a precise, terse, manner.

Overall process. The fundamental operation of SPL is that of *materialization: the act of deriving a specific product from the assets making up the product line.* The activities pertaining to the materialization are described in Fig.1.

The process shown in the figure starts from an existing SPLGraph, Π_0, which represents, via some mapping, assets of the SPL. While this mapping is largely domain specific, the

representation of variability constructs (boolean conditions, optional, alternative, etc.) is uniform across all SPLs and is presented in this paper as a set of SPLGraph patterns. After choosing Π_0, the user chooses a product by selecting a *start vertex*, extracts open decisions, and configures the product according to his needs by resolving (some of) these decisions.

Finally, the materialization algorithm is applied. This yields another SPLGraph, Π_1, which in turn may be further materialized if it sports its own set of open decisions. Otherwise, Π_1 has no open decision and it describes a concrete product with zero variability. In this case, one can apply code generation to translate the SPLGraph into target assets which form the final product.

The focus of this paper is on the parts of the process (Fig.1) that are generic and do not require domain specific knowledge. Specifically, we formally define: the SPLGraph mathematical structure (Sec.2); the *materialize* algorithm (Sec.2.1) along with its auxiliary algorithms *apply*, *traverse* and *isReachable*; the patterns (Sec.3) that are central to the mapping of SPL variability onto an SPLGraph instance; and, finally, the *justify* algorithm (Sec.4) which is the backbone of the *extraction of open decisions* activity.

2. A FORMAL MODEL

An *SPLGraph* is a 7-tuple $\Pi \equiv \langle G, M, \mathbb{F}, \mathbb{T}, \mathbb{X}, L, \delta \rangle$:

- $G = \langle V, E \rangle$ is a finite directed graph.
- $M \subseteq V$ is a set of *mutation vertices*.
- $\mathbb{F} \in V$ is the *False vertex*.
- $\mathbb{T} \in V$, $\mathbb{T} \neq \mathbb{F}$ is the *True vertex*.
- $\mathbb{X} \in V$, $\mathbb{X} \neq \mathbb{F}$, $\mathbb{X} \neq \mathbb{T}$ is the *Unknown vertex*. $\mathbb{X}$ is a sink: it has no out-going edges.
- L is a finite set of words over some alphabet. We assume the existence of some order over L (or a lexicographic order over the alphabet of L). This order will make enumeration over a set of vertices or edges well defined: enumeration will follow the ordering of the respective labels (see δ below).
- $\delta : V \cup E \rightarrow L$ is a *labeling function* associating a word from L with each vertex or edge. Each vertex has a unique label, i.e., for any two vertices $v \in V$, $u \in V$, $\delta(v) = \delta(u) \Leftrightarrow v = u$. Two edges $e_1 = \langle v_1, w_1 \rangle$, $e_2 = \langle v_2, w_2 \rangle$ may share the same label only if they go out from two different vertices: $\delta(e_1) = \delta(e_2) \Rightarrow v_1 \neq v_2 \vee e_1 = e_2$.

We will use the square brackets notation (similar to the array access operator found in many programming languages) to denote the association of vertices via edges. This operator is defined as follows: for a vertex v and a label l, the expression $v[l]$ evaluate to u if there is an edge from v to u whose label is l, or $\mathbb{X}$ otherwise.

This definition implies that we do not distinct between an edge going to $\mathbb{X}$ and a missing edge. $a[\text{“}x\text{”}] = \mathbb{X}$ can mean either (i) the existence of an edge labeled “x” from a to $\mathbb{X}$; or (ii) that there is no edge labeled “x” going out from a. For our purposes there is no need to distinct between the two cases: For the remainder of this paper, an edge going to $\mathbb{X}$ is a non-existing edge.

SPLGraph makes no assumption on the information represented by the graph. User is free to choose any translation scheme for mapping domain entities (source code, requirements, tests, manuals, etc.) to graph elements.

To illustrate the versatility of the graph structure let us outline a particular translation scheme that maps source code of an object-oriented program to a directed labeled graph. Note that our model does not rely on this translation scheme in any way and it is provided here purely for illustrative purposes.

Each class will be represented by a single vertex. Inter-class relationships, such as the subclassing relationship, will be represented by an edge associating the corresponding classes. For instance, class inheritance is depicted by an edge labeled “*extends*” that connects the subclass to the superclass. Additional relationships (e.g., class-field) can be expressed by introducing a vertex for each field and associating the declaring class with these via edges such as “$field_1$”, “$field_2$”, etc.

We will now turn our attention to two kinds of vertices that provide SPLGraph with the ability to express variability. Mutation vertices describe potential changes to the graph while *decision vertices* describe conditions which, ultimately, control which changes are to be applied. This behavior is dictated by the *materialize* algorithm which is subsequently presented.

Mutation vertices. Vertices in M are referred to as *G's mutation vertices* and they formalize the notion of *variability* or *variation points* that is central to many SPL systems. In SPLGraph, a mutation vertex is a vertex that describes a change to the SPLGraph itself.

A mutation vertex must have at least three outgoing edges carrying the labels “*subject*”, “*key*” and “*target*”.

The semantics of such vertices is realized by algorithm *apply* from Alg. 1. The reader should note that this algorithm has little utility on its own: as shown in Fig.1 it is a building block of the *materialize* algorithm.

Algorithm: $apply(\Pi, m)$

Input $\Pi = \langle G = \langle V, E \rangle, M, \mathbb{F}, \mathbb{T}, \mathbb{X}, L, \delta \rangle$, $m \in M$

1: $s \leftarrow m[\text{“}subject\text{”}]$
2: $k \leftarrow m[\text{“}key\text{”}]$
3: $l \leftarrow \delta(k)$
4: $s[l] \leftarrow m[\text{“}target\text{”}]$

Algorithm 1: Enactment of the mutation represented by a mutation vertex m: An edge in the subject vertex, s, is rerouted to the target vertex, $m[\text{“}target\text{”}]$.

The *Apply* algorithm in Alg. 1 accepts, as input, a vertex m from which it finds s, the subject vertex (the vertex that will be mutated), by $m[\text{“}subject\text{”}]$. It then finds the label of the edge that will be mutated, l, by going to the vertex $m[\text{“}key\text{”}]$ and extracting its label. Finally, *apply* makes the actual mutation by rerouting the edge with the label l (going out from s) to $m[\text{“}target\text{”}]$.

Decision vertices. Decision vertices provide the ability to define conditions. Structurally, a decision vertex is indicated by a self edge carrying the label “d” (for “decision”) and by the existence of three additional edges with the following labels: “*condition*”, “f” and “t”.

Alg.2 depicts a graph traversal algorithm that takes into

account the decisions embodied by decision vertices [2].

Algorithm: *traverse*(Π, s)

Input $\Pi = \langle G = \langle V, E\rangle, M, \mathbb{F}, \mathbb{T}, \mathbb{X}, L, \delta\rangle$, $s \in V$
Output An ordered set of vertices
1: $T \leftarrow \phi$ //*an ordered set*
2: **if** s is not marked as "explored" **then**
3: mark s as "explored"
4: $T \leftarrow T \cup \{s\}$
5: **if** $s[\text{“}d\text{”}] \neq s$ **then**
6: **for all** v such that $v \in neighbors(\Pi, s)$ **do**
7: $T \leftarrow T \cup traverse(\Pi, v)$
8: **else**
9: **if** $s[\text{“}condition\text{”}] = \mathbb{T}$ **then**
10: $v \leftarrow s[\text{“}t\text{”}]$
11: $T \leftarrow T \cup traverse(\Pi, v)$
12: **else if** $s[\text{“}condition\text{”}] = \mathbb{F}$ **then**
13: $v \leftarrow s[\text{“}f\text{”}]$
14: $T \leftarrow T \cup traverse(\Pi, v)$
15: **return** T

Algorithm 2: Traverse an SPLGraph from a given vertex, s. T is an ordered set. $T \cup Q$ indicates the sequence concatenation. *neighbors* computes all immediate neighbors of the given vertex, ignoring edges to $\mathbb{X}$. When a decision vertex is discovered traversal continues through the "t" edge or the "f" edge, determined by $s[\text{“}condition\text{”}]$. The algorithm assumes that initially all vertices are marked as "unexplored".

The algorithm in Alg.2 is based on the well known (recursive) Depth First Search (DFS) algorithm [5]. Specifically, it computes a sequence of vertices specifying order of traversal throughout the graph. Unlike classic DFS, it does not guarantee that all reachable vertices will be traversed, which is due to the existence of decision vertices.

The algorithm assumes that a vertex can be marked as either "unexplored" or "explored" and that initially all vertices are "unexplored". If the current vertex, s, is "explored" the algorithm will do nothing. Otherwise, it immediately marks s as "explored" and then decides if s is a decision vertex, by checking for a self edge labeled "d". If s is not a decision vertex, all immediate neighbors of s will be traversed by a recursive application of the algorithm on each.

In a decision vertex, the algorithm checks the "*condition*" edge going out of s. If it leads to $\mathbb{T}$ traversal will continue recursively on $s[\text{“}t\text{”}]$. If it leads to $\mathbb{F}$ traversal will continue recursively on $s[\text{“}f\text{”}]$. If neither of these two cases apply, no further traversal takes place.

LEMMA 1. *Algorithm traversal from Alg.2 always terminates.*

PROOF. There is only one loop in the body Alg.2. Given that the graph is finite by definition, this loop is bound to be finite. Non-termination is thus possible only via infinite recursion. Prior to any recursive call the algorithm changes the status of a vertex from "unexplored" to "explored". Thus, an infinite recursion implies an infinite number of unexplored vertices, which contradicts the finiteness of the graph. □

LEMMA 2. *Algorithm traversal from Alg.2 always returns a finite set.*

PROOF. Each application of Alg.2 directly contributes one element to the result. Given that the number of applications is finite (see Lem. 1), and that the result is initially empty, we have that the result is also finite. □

Note that unlike mutation vertices, decision vertices are not part of the SPLGraph definition. Any vertex that has a self "d" edge is considered to be a decision vertex. This looseness is intentional: we want the formalism to be as flexible as possible. In particular, a mutation vertex can change a plain vertex into a decision vertex or vice-versa, which were not possible if decision vertices were fixed at the SPLGraph definition (for example, as a set D of decision vertices, similar to the set, M, of mutation vertices).

The *isReachable* algorithm, depicted in Alg.3 is built on top of the *traverse* algorithm. It will return "true" iff a traversal (as per *traverse*) that starts at s passes through t.

Algorithm: *isReachable*(Π, s, t)

Input $\Pi = \langle G = \langle V, E\rangle, M, \mathbb{F}, \mathbb{T}, \mathbb{X}, L, \delta\rangle$, $s \in V$, $t \in V$
1: **for all** $v \in V$ **do**
2: mark v as "unexplored"
3: $T \leftarrow traverse(\Pi, s)$
4: **return** $t \in T$ //either True of False

Algorithm 3: Determine if a traversal (as-per the *traversal* algorithm) from s will pass through t.

2.1 Materialization

We can now define the main algorithm *materialize* that relies on the algorithms presented so far. As shown in Fig.1 this algorithm expects its user to supply an SPLGraph (Π), a *configuration* that resolves open decisions (F, T) and a root vertex for the product that the user is interested in (s). This algorithm is depicted in Alg.4.

Algorithm: *materialize*(Π, F, T, s)

Input $\Pi = \langle G = \langle V, E\rangle, M, \mathbb{F}, \mathbb{T}, \mathbb{X}, L, \delta\rangle$, $F \subseteq V$, $T \subseteq V$, $s \in V$
1: **for all** $v \in F$ **do**
2: $v[\text{“}condition\text{”}] \leftarrow \mathbb{F}$
3: **for all** $v \in T$ **do**
4: $v[\text{“}condition\text{”}] \leftarrow \mathbb{T}$
5: $R \leftarrow \{v \in M \mid isReachable(\Pi, s, v)\}$
6: **for all** $v \in R$ **do**
7: $apply(\Pi, v)$
8: **return** Π

Algorithm 4: Derive an SPLGraph by mutating an existing SPLGraph, Π. F, T specify the configuration: sets of decision vertices that are resolved to $\mathbb{F}$, $\mathbb{T}$ (respectively). s is the root vertex of the product.

The algorithm starts by rerouting the "*condition*" edge of all vertices in the input sets F, T to $\mathbb{F}$, $\mathbb{T}$ (respectively). In step (5) the algorithm finds all mutation vertices that are reachable from s via traversals that respect decision vertices. It then actuates the mutations therein by invoking *apply* on each such vertex (7).

Note that *materialize* is expressed in imperative terms: it mutates its input, Π, and then returns it. As convention, we will assume that the algorithm always accepts a

[2]Just like *apply*, this is also an auxiliary algorithm that is used by the main algorithm *materialize*

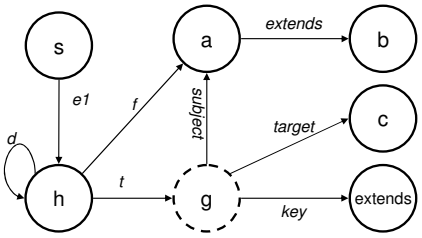

Figure 2: Π_0: An SPLGraph with three plain vertices a, b, c; a decision vertex, h; a mutation vertex, g, representing the following mutation: a[“*extends*”] $\leftarrow$ c. Given that h[“t”] $= g$ then g will be applied only if h[“*condition*”] were rerouted to $\mathbb{T}$.

copy of its input, that is: a call to $materialize(\Pi, F, T, s)$ is a shorthand for $materialize(copy(\Pi), F, T, s)$ where *copy* is a function that duplicates its input. This convention will allow expression such as $\Pi_1 \leftarrow materialize(\Pi_0, F, T, s)$ to be meaningful as it will make sure the mutations applied during materialization will not affect the input Π_0.

LEMMA 3. *Algorithm materialize from Alg.4 always terminates.*

PROOF. Call to *isReachable* terminates (trivial, by relying on Lem.1). M is finite by definition so number of calls to *isReachable* is also finite. Every element of R is also an element of M so R is finite and so is the iteration over its elements (loop in steps (6)-(7)). □

2.2 A Simple Example

Let us now present a small SPL, Π_0, defined over the domain of Java code capable of producing two products that differ only in the superclass of one of the classes (see Fig.2). Π_0 uses the program-to-graph translation scheme presented earlier (Sec.2).

Convention. *We omit, from the figures in this paper, edges that arrive at the $\mathbb{X}$ vertex. This is in alignment with our definition of the v[“e”] operator. We also omit any of the vertices $\mathbb{X}$, $\mathbb{F}$ or $\mathbb{T}$ if it is not needed.*

Examining Fig.2 we note that Π_0 defines an empty class a that subclasses (“extends”) the empty class b. It also defined the empty class c.

The dashed border of vertex g indicates that it is a mutation vertex ($g \in M$). g's outgoing edges realize the following mutation: a[“*extends*”] $\leftarrow$ c (change the superclass of a to c). h is a decision vertex since it has a self edge labeled “d”. Its “t” edge goes to g. Therefore, h states that if h[“*condition*”] $= \mathbb{T}$ then a traversal starting from s will reach g.

We can now examine the effect of materialization (Alg.4) on Π_0. Our materialization will bind h to the logical truth value (h[“*condition*”] $\leftarrow \mathbb{T}$) by materializing with the configuration $\langle F, T\rangle = \langle \phi, \{h\}\rangle$. Specifically, we define Π_1 as $\Pi_1 \equiv materialize(\Pi_0, \phi, \{h\}, s)$.

Examining Π_1 (see Fig. 3) we see that now a[“*extends*”] $=$ c whereas in Π_0 a[“*extends*”] $= b$. Π_1 therefore represents one of the products that can be obtained from Π_0.

In order to get the other product obtainable from Π_0, one needs to reroute h's “*condition*” edge to $\mathbb{F}$. This rerouting is expressed by the configuration $\langle F, T\rangle = \langle \{h\}, \phi\rangle$ which we use in $\Pi_2 \equiv materialize(\Pi_0, \{h\}, \phi, s)$

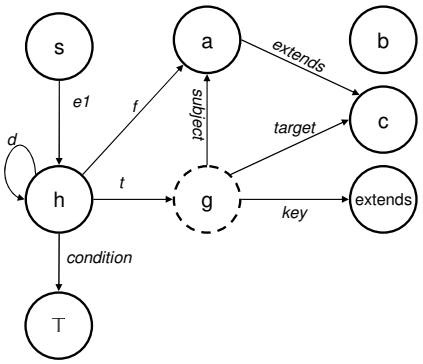

Figure 3: Π_1: An SPLGraph that is a product of Π_0: $\Pi_1 \equiv materialize(\Pi_0, \phi, \{h\}, s)$. Π_1 is different from Π_0 in h[“*condition*”] $= \mathbb{T}$ and in a[“*extends*”] $= c$, due to the application of the mutation vertex g.

This change prevents the materialization algorithm from passing through g thereby preventing the mutation from taking place. In the resulting SPL, Π_2, a[“*extends*”] will be identical to a[“*extends*”] in Π_0, that is: a[“*extends*”] $= b$

This illustrates how changes in configuration affect the materialized SPL.

3. PATTERNS

In this section we show several patterns that capture recurring SPL structures. Note that all these patterns can be expressed in terms of SPLGraph as presented so far.

3.1 Boolean Operators

Although decision vertices realize only a simple if-else decision, they are sufficiently powerful to express all boolean formulae. We will justify this claim by showing how decision vertices can express disjunction and negation.

Fig.4 shows a (partial) graph expressing disjunction.

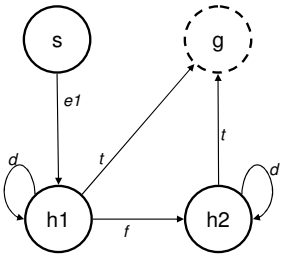

Figure 4: Π_2: An SPLGraph showing the disjunction pattern. h_1 and h_2 are decision vertices. g is a mutation vertex whose outgoing edges were omitted from the figure. g will be applied if either h_1[“*condition*”] or h_2[“*condition*”] are rerouted to $\mathbb{T}$.

In the figure, h_1 and h_2 are decision vertices where h_1[“f”] $=$ h_2 and h_1[“t”] $= h_2$[“t”] $= g$. Traversal from h_1 will include g (a mutation vertex, outgoing edges omitted) if either h_1[“*condition*”] $= \mathbb{T}$ or h_2[“*condition*”] $= \mathbb{T}$.

Negation of a decision vertex is even simpler and can be done by switching the labels “f” and “t” of the outgoing

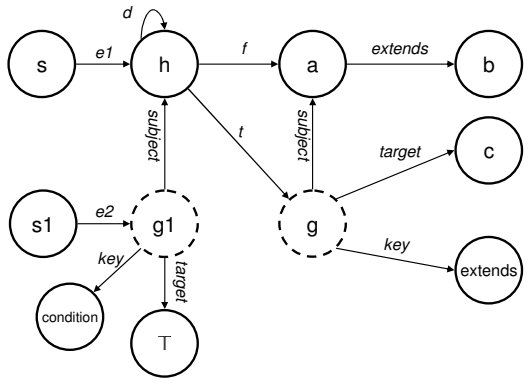

Figure 5: Π_4: An extension of Π_0 from Fig.2. Materialization from s_1 will unconditionally apply the g_1 mutation: $h[\text{“}condition\text{”}] \leftarrow \mathbb{T}$ thereby forcing the application of g in subsequent materializations.

edges. Clearly, a system that is capable of expressing disjunction and negation is sufficiently powerful for expressing any boolean formula.

3.2 Named Configurations

So far, the distinction between the different products that can be derived from a given SPL was determined by the configuration ($\langle F, T\rangle$ parameters) specified at materialization time. A recurring practical need is that of defining several canned configurations.

One can express such configurations by defining additional vertices that will be used instead of the "original" start vertex. An example is Π_4 (Fig.5) which is an extension of Π_0 (Fig.2). The only additions are three new vertices s_1, g_1 and *condition* along with their adjacent edges.

The new mutation vertex g_1 will mutate h as follows: $h[\text{“}condition\text{”}] \leftarrow \mathbb{T}$. The vertex s_1 is connected to g_1 via the "e_2" edge. If we materialize Π_4 from s_1 we will get an SPLGraph, Π'_4, where $h[\text{“}condition\text{”}]$ is already wired to $\mathbb{T}$. A subsequent materialization of Π'_4 will produce an SPLGraph that is equivalent to Π_1 even if we choose an empty configuration $\langle F, T\rangle = \langle \phi, \phi\rangle$.

The additions made by Π_4 show how a configuration of a specific product can be expressed inside an SPLGraph. The s_1 vertex provides an access point to this configuration, thus serving as its "name".

Note that a start vertex has no special, formal distinction, i.e., any vertex can be passed as s when calling *materialize*. However, from a practical point of view, the engineer designing the SPL is likely to treat one of the vertices as the root of the complete product family. The same goes for s_1: it has no formal role, but it serves, from the user's point of view, as the root of a specific product.

3.3 Named Configuration Reuse

It is quite common for the configurations of two (or more) products to exhibit a high degree of overlap. In a similar manner to reuse of boolean expression one can use plain edges to express this type of reuse.

To see that let us assume that we add an s_2 vertex to Π_4 (Fig.5). We want s_2 to reuse the product configuration of s_1. We add an edge from s_2 to s_1: $s_2[\text{“}e_3\text{”}] = s_1$.

Materialization from s_2 will start at s_2 and then follow the exact same steps as those of s_1. This was achieved without duplicating s_1's specific logic (namely: the g_1 vertex). Moreover, subsequent evolution of the SPL and of s_1 (such as: s_1 will have additional edges going out to other mutation vertices) will automatically be reflected in s_2.

Of course, s_2 can also define his own mutation[3] that will be applied either before or after the mutations of s_1 are taking place, depending on the labeling on the edges and the order defined upon these labels.

3.4 Optional and Alternative

Many SPL tools support, at least, two variability operators: Optional and Alternative. The former allows the (conditional) exclusion of a certain artifact from the final product. The latter allows the inclusion of a single artifact out of a designated set of two or more artifacts.

In our model, optionality of a vertex v can be expressed by a path of mutation vertices each one removing one of the edges going into v (an edge can be removed if the "*target*" of the mutation vertex is $\mathbb{X}$) The path starts from one or more decision vertices expressing the boolean formula controlling the inclusion/exclusion.

The alternative operator can be modeled by a repeated application of the optional operator. An alternative of n becomes n instances of optional, each time with a different condition along with a constraint that specifies the mutual exclusion: $(c_1 \wedge \neg c_2 \wedge \neg c_3 \ldots \neg c_n) \vee (\neg c_1 \wedge c_2 \wedge \neg c_3 \ldots \neg c_n) \vee \ldots (\neg c_1 \wedge \neg c_2 \wedge \neg c_3 \ldots c_n)$

3.5 Staged Materialization

A common SPL scenarios, is that of *Supply Chain* development. In such settings, a vendor configures an SPL not for the sake of obtaining a product, but for the purpose of obtaining a partially configured SPL that is then passed to another vendor, which will configure it further.

This type of development is supported by our formalism because there is no distinction between the input and the output of the *materialize* algorithm. Specifically, one can form a chain of materializations $materialize(\ldots materialize($ $materialize(\Pi, F, T, s), F_1, T_1, s_1), \ldots F_n, T_n, s_n)$ to express a multi-staged materialization process.

4. JUSTIFICATION

A central question in SPL development is that of extracting open decisions: *what decisions are still unresolved?*

The algorithm presented here goes a step beyond that. It determines what configurations will ensure that a given mutation vertex is applied.

Formally, the *justification problem* can be defined as follows: *given an SPLGraph, Π, a vertex s and a vertex $m \in M$, describe all possible assignments into F and T such that $materialize(\Pi, F, T, s)$ will induce the invocation $apply(m)$.*

As an example let us consider Π_2 from Fig.4. Looking at vertex g in Π_2 we intuitively note that it will be applied in a materialization from s if either h_1 is wired to $\mathbb{T}$ or h_2 is wired to $\mathbb{T}$ (that is: $h_1[\text{“}condition\text{”}] = \mathbb{T}$ or $h_2[\text{“}condition\text{”}] = \mathbb{T}$. We therefore conclude that any configuration $\langle F, T\rangle$ wherein either $h_1 \in T$ or $h_2 \in T$ will induce the application of g.

[3]Alas, the graph in the figure is too simple for such mutations to be meaningful

The *justify* algorithm depicted in Alg.5 computes the justification of a vertex m. It returns a boolean formula which is satisfied by a configuration if and only if the configuration will lead to the application of m in the materialization. The algorithm works by computing a set of *paths*. A path is a *sequence of pairs* $\langle v, l\rangle$ *where* v *is a vertex and* l *is a label of an edge leaving* v *(that is:* $v[l] \neq \mathbb{X}$*).*

Algorithm: *justify*(m, s)

Input $m \in M$, $s \in V$

1: let P = set of all cycle free paths s to m
2: **for all** $p_i \in P$ **do**
3: remove from p_i all pairs $\langle v, l\rangle$ where $v[\text{“}d\text{”}] \neq v$ or $l \neq$ “*true*” or $l \neq$ “*false*”
4: **for all** $p_i \in P$ **do**
5: **for all** $\langle v_j, l_j\rangle \in p_i$ **do**
6: **if** $l_j =$“*true*” **then**
7: let $t_{i,j} = v_j$
8: **else**
9: let $t_{i,j} = \neg v_j$
10: **return** a formula $(t_{1,1} \wedge t_{1,2} \wedge \ldots) \vee (t_{2,1} \wedge t_{2,2} \wedge \ldots) \vee \ldots$

Algorithm 5: Obtain a boolean formula that specifies the conditions under which the mutation vertex, m, will be applied in a materialization.

Algorithm *jutsify* starts with calculating all paths from s (a starting vertex) to m, the mutation vertex we are interested in. It then considers only decision vertices (on these paths) and only decision-related edges and translates them into a boolean formula using these rules:(i) vertices on the same path are conjuncted; (ii) Different paths are disjuncted. Negation is determined by the edge associated with a vertex (on the path at hand): an “f” edge will induce negation[4].

When computing *justify*(Π_2, s, g) the algorithm will build two paths that start as s and end at g: $\langle\langle s, e_1\rangle, \langle h_1, true\rangle\rangle$ and $\langle\langle s, e_1\rangle, \langle h_1, false\rangle, \langle h_2, true\rangle\rangle$.

Discarding non-decision vertices we get: $\langle\langle h_1, true\rangle\rangle$ and $\langle\langle h_1, false\rangle, \langle h_2, true\rangle\rangle$. These paths are then translated into the following boolean formula: $h_1 \vee \neg h_1 \wedge h_2$ which is equivalent, as expected, to $h_1 \vee h_2$.

5. IMPLEMENTATION

In order to evaluate our approach we implemented a prototype of an SPLGraph-based product lines tools. The tool is implemented as a Java GUI application. The user can import Java source files into the tool, browse the various program elements (packages, classes, methods, fields) and mark them as *optional*. An optional element is realized as a decision vertex combined with a mutation vertex that will delete the edge attaching that element to its owner in the Java hierarchy.

The underlying SPLGraph is implemented as an in-memory data structure which maps edges to vertices. The Java-to-SPLGraph translation scheme that we used represents packages, classes, methods, fields, and parameters as distinct vertices. Bodies of methods (as well as field initializers) are stored as text.

Although we did not try to optimize the current representation (neither for space nor speed) it can cope with

[4]Correctness and termination proofs of this algorithm are left out of this paper due to space limitations.

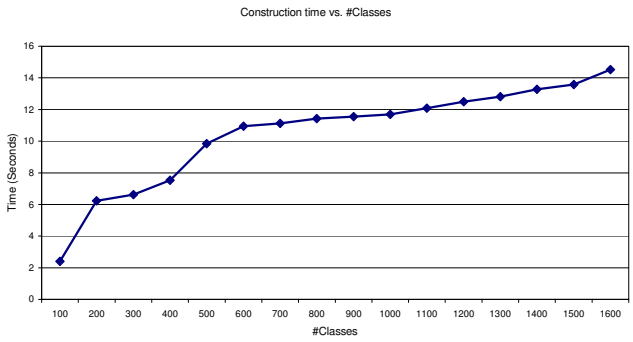

Figure 6: Construction time of an SPLGraph of a sample Java program as a function of number of classes.

Java programs of several thousand classes. For instance, a Java program[5] comprising 1,661 classes, 4,826 methods and 10,189 fields and parameters requires 215MB of memory when represented as an SPLGraph structure.

Fig.6 shows the time needed for building an SPLGraph from the sources of the said code[6].

The figure shows the time needed for parsing and creating an SPLGraph as a function of the number of parsed Java files. We note that after initial 500 classes construction rate was quite steady. Overall time for 1,600 classes was 14.524 seconds. Note that sizes of classes in a real-life Java programs are not uniform and are typically following a power-law distribution. This incurs occasional fluctuations in construction time.

We then turned to evaluate materialization time. We generated several SPLGraphs off of the the SPLGraph of the said Java programs by randomly selecting a subset of n vertices (n goes from 20,000 to 27,500).

For each of these generated graphs we chose m vertices to be optional (m goes from 5,000 to 20,000 in steps of 5,000). We then measured the (median) time needed for materialization. In each materialization half of the decision vertices were defined to be true and half were defined to be false. Results are depicted in Fig.7.

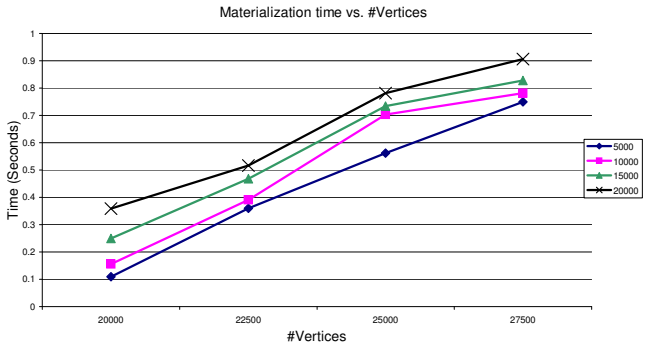

Figure 7: Materialization time as a function of number of vertices in the graph. Each curve corresponds to a different number of decision vertices.

Each curve in the figure corresponds to a different value

[5]http://openjtl.sourceforge.net

[6]Our experiments were executed on a 2GB Windows XP machine running on a 2.39GHz Intel processor with two cores.

of m. The X-axis values corresponds to n. Measured time is represented by the Y-axis. Materialization time of our largest input ($n = 27,500$, $m = 20,000$) was 1.57 seconds.

We also note that materialization time depends primarily on the size of the graph. Changes in the number of decisions, m, had only a secondary effect on materialization time.

6. RELATED WORK

Examining existing research we note that a large body of work concentrates on formalizing particular aspects of SPLs. A specific aspect where significant formalization attempts have been made is that of type checking. Kastner and Apel's CFJ calculus [12] which extends the *Feathrerweight Java* [14] calculus with variability constructs, is a recent example.

Formal foundations of control mechanisms have been studied in works ([1, 8]) that showed the connection between well known mathematical structures (grammars, propositional logic, boolean formulae) and features.

Our work goes beyond these as it tries to formalize SPL as a whole, rather than specific parts thereof. In that sense its goal is similar to that of Bayer et-al's Consolidated metamodel [2].

The two works represent different approaches to the same topic. In particular, Bayer's model focuses mostly on the variability mechanisms. The exploration of the control mechanism, resolution model in Bayer's terminology, is limited (a similar difference arises when comparing SPLGraph with the work of Haugen, Møller-Pedersen et-al [10]). Thus, it does not support the type of reasoning that is offered by our *justify* algorithm.

Another difference, is in the size of the core. In SPLGraph there are only a few core constructs. Additional constructs are expressed as patterns built on top of the core. In the consolidated model, these patterns are defined as core entities (meta-model classes).

7. FUTURE WORK AND CONCLUSIONS

In this paper we presented the *justify* algorithm as an application algorithm that allows an engineer to better understand the SPL at hands. A future extension of *justify* will provide the justification not for a mutation vertex, but for a triplet of vertex, edge label, vertex: $\langle v, l, u\rangle$. The output will be the justification that will cause the mutation $v[l] \leftarrow u$ to take place.

Another useful service is an algorithm that computes all possible mutations that can be applied to a given vertex. This algorithm will provide the information that is needed for validation purposes, such as SPL type checking [12].

Another question is that of scalability in terms of performance and size. A possible solution to such issue will be that of using a tree-oriented data base (such such as Neo4j[7]) for storing the underlying data.

Another topic for future research is that of enhancing the decision vertices. This new control construct will allow certain boolean formulae, such as the ones needed for the *Or* pattern [6], to be expressed more succinctly, a-la the $choose_{n,m}(e_1, \ldots, e_k)$ operator of propositional logic [1].

Overall, it seems that the versatility of SPLGraph, as shown by the broad spectrum of patterns it can express, is largely due to what is informally called the *Lego Principle: the smaller the building blocks, the more reusable they are.* This versatility is what turns, in our minds, SPLGraph into a solid basis for a generic underpinning for SPL tools.

[7]http://neo4j.org

Acknowledgments. This work is partially supported by Science Foundation Ireland under grant no. 03/CE2/I303_1 to Lero – The Irish Software Engineering Research Centre, http://www.lero.ie. The authors thank Keren Lenz for her thorough feedback.

8. REFERENCES

[1] D. Batory. Feature models, grammars, and propositional formulas. In *SPLC 2005*, pages 7–20, Rennes, France, 2005. Springer Verlag.

[2] J. Bayer, S. Gerard, Ø. Haugen, J. X. Mansell, B. Møller-Pedersen, J. Oldevik, P. Tessier, J.-P. Thibault, and T. Widen. Consolidated product line variability modeling. In *Software Product Lines*, pages 195–241. Springer, 2006.

[3] Y. Bontemps, P. Heymans, P.-Y. Schobbens, and J.-C. Trigaux. The semantics of FODA feature diagrams. In *Workshop on Software Variability Management for Product Derivation*, Boston, MA, August 2004.

[4] P. Cointe. Metaclasses are first class: The ObjVlisp model. In *OOPSLA 1987*, pages 156–162, New York, NY, USA, 1987. ACM.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd revised edition edition, September 2001.

[6] K. Czarnecki and U. W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[7] K. Czarnecki, S. Helsen, and U. Eisenecker. Formalizing cardinality-based feature models and their specialization. *SPI&P*, 10(1):7–29, 2005.

[8] M. de Jonge and J. Visser. Grammars as feature diagrams. manuscript, 2002.

[9] R. Frost. Jazz and the Eclipse Way of Collaboration. *IEEE Software*, 24(6):114–117, 2007.

[10] Ø. Haugen, B. Møller-Pedersen, J. Oldevik, G. K. Olsen, and A. Svendsen. Adding standardized variability to domain specific languages. In *SPLC 2008*, pages 139–148, 2008.

[11] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, CMU/SEI-90TR-21, 1990.

[12] C. Kästner and S. Apel. Type-checking software product lines - a formal approach. In *ASE 2008*, pages 258–267, 2008.

[13] M. Mannion. Using first-order logic for product line model validation. In *SPLC 2002*, pages 176–187, August 2002.

[14] T. Nipkow and D. von Oheimb. Javalight is type-safe—definitely. In *POPL 1998*, pages 161–170, New York, NY, USA, 1998. ACM.

[15] P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux. Feature diagrams: A survey and a formal semantics. In *RE 2006*, pages 136–145, 2006.