December 13, 2016

A Mathematical Formulation of a Product Family

Grady H. Campbell, Jr.
domain-specific.com
Annandale, VA, USA

Table of Contents

Abstract

A product family is an envisioned set of similar products. Products are similar if they provide similar solutions to the same or similar problems. The products comprising a product family can be explicitly specified and differentiated by a mathematical formulation based on set and metric space theories. Such a formulation provides a framework for the systematic engineering of a product family and an associated mechanism for deriving any encompassed product, enabling the streamlined manufacture of customized products.

Keywords

program family · similarity · set theory · metric space · product · product family · decision model · domain · market.

1. Introduction

A *product family* is an envisioned set of similar products, alternative solutions to one problem or similar solutions to a set of similar problems[1]. This concept provides a framework for building and evolving a set of high quality, customized products with effort comparable to building and maintaining a single product conventionally.

As a realization of the manufacturing concept of mass customization, the product family concept derives its power from focusing developers on problems that are intuitively similar but that are nevertheless characterized by diversity, uncertainty, and change. Each product must be customized to achieve a proper fit to a particular customer's specific needs.

The concept of a product family has long been understood, in practical terms, as the representation in aggregate of a set of similar products, with an associated mechanism for deriving instance products from that representation. After more than 25 years experience with product families, this paper defines a mathematical formulation of the product family concept.

The essence of a product is its behavior, the capabilities realized through its use. This behavior is expressed in the guise of a "program". To define the product family concept, we first present a framework for characterizing programs as abstract objects and a set of similar programs as a family. We then integrate these views to enable representing a program family in the form of a metric space in terms of differences (i.e., variability) among instance programs. Finally, we generalize this formulation of programs and program families to characterize products and product families as the basis for the derivation of customized products.

[1] A "product" is some realization of a solution to a problem, corresponding to a point in a universal problem-solution space. "Problem" is used, in a general sense, to mean a set of needs that will be met by a product that conveys a "solution" to those needs.

2. Background

In 1972, Dijkstra informally defined the concept of a (computer software) program family as a set of "related programs either as alternative programs for the same task or as similar programs for similar tasks" [1]. He argued that the proper solution to a problem should be justified based on an understanding of possible alternative solutions. Furthermore, as Parnas also argued [2], when an imperfect understanding of the problem or a changed or new but similar problem requires a different solution, that solution should not be viewed as merely a change in the program's source text but as a rederivation that retains common aspects of the prior solution.

In 1990, this viewpoint became a core premise of the Synthesis methodology (later refined and extended as the Domain-specific Engineering (DsE) methodology) for the development and use of product families [3, 4]. The motivation for Synthesis was to define a practical approach for building multiple similar products for customers having differing needs, but without the effort required conventionally to build, tailor, and maintain each product individually. The approach was also motivated by the view that engineering is a discipline of identifying and resolving uncertainties and tradeoffs to find a suitable fit to needs among feasible alternatives in a problem-solution space.

Synthesis defined the concept of a product family as a generalization of the program family concept: although actual development of a product is concerned primarily with creating an operational software program that provides some capability to an enterprise, it entails creating more than just the program – it also requires the production of specifications, documentation, test materials, and installation and training materials. All of these materials in combination, and mutually consistent with the software program as developed, constitutes a product that is then able to be delivered to a customer and used. Similarly, as a customer's needs change, not only the software program but the product as a whole must be revised/replaced with a product that better suits then-current needs.

Subsequent "product line"[2] efforts by the Software Engineering Institute and various European initiatives, as reported in numerous conferences and workshops, have shown that the product family concept is the basis for a viable approach to the development of software-based products. This paper augments those experiences by providing a definition of the mathematical properties of a product family.

3. A Set Theoretical Framework For Program Families

A program family is "a set of similar programs". To understand what this means, we have to understand the concept of a 'program', what is meant by 'similar', and how such a set is conceived.

3.1 On the Characterization of a Program

A *program* is an expression of behavior that a computer is able to enact[3]. Such expression must be well-formed but can be in any notation that can be translated into a form that a computer has been built to recognize. An expression is well-formed if it conforms to the rules of the conveying notation (being well-formed implies that the expression is, or can be correctly translated into, an expression that a computer can enact; it does not imply that the expression will result in expected ("correct") behavior when enacted by a computer).

Walenstein identifies two categories of program characterization [8]: representational (syntactic) and behavioral (semantic). These categories provide a framework not only for describing the different aspects of a program but also for characterizing how two programs differ.

Representational characterizations, corresponding to the physical realization of a program, can be further divided into:

- Source – the expression (syntax, structure, and layout) of a program in a human-readable notation
- Object – the expression (semantics and physical realization in a computational environment) of a program in a computer-enactable notation (a machine-independent intermediate notation or a machine-dependent binary encoding)

Although a program is most simply understood in terms of its expression in a given source notation, its utility comes through its expression in object form that is then realized as the observable behavior of a computer. The object form of a program is translated from and therefore derivative of its source form, according to the conventions of a targeted computational environment (i.e., processor, devices, and

[2] Many people have conflated "product line" with "product family"; this paper retains the original distinction made in Synthesis between the two, that a product line comprises only products that have been built and deployed into use to solve similar problems. Although a product line may be a subset of a product family, it may include dissimilar products if they solve similar problems. (This is formally clarified in section 6 below.)

[3] Although in this context a program is conceived of as instructions to be executed by a computer, we can more generally think of a program as an abstract definition of behavior that a given natural or constructed mechanism is capable of exhibiting.

operating software). A program may have multiple object form expressions of its source notation if it is intended to operate in differing environments.

Behavioral characterizations, designating the observable properties associated with a program, can be divided into:

- Functional – an expression of a program's intended purpose, meaning, properties, and constraints (e.g., the algorithm, business practices, and ancillary conventions that it implements)
- Computational – an expression of a program's observable behavior in a particular operational context of usage

The essential distinction between these two types of behavioral characterization is that functional assumes an idealized approximation of the operational environment whereas computational presumes a fully realized environment having empirically predictable effects on a program's behavior.

These bipartite divisions (of representational and behavioral characterizations) are correlated. A program's functional properties are a realization of its source expression: both express intrinsic aspects of a program independent of computational context. Likewise, a program's computational properties are a realization of its object expression: both are extrinsically influenced by properties of a targeted computational environment. The essential challenge of software engineering is achieving a consistent realization of all four of these forms of expression.

3.2 Programs, Envisioned and Realized

Although we tend to think of a program as corresponding to a source expression that engenders some intended behavior, when we set out to create a program, we do not know its source form. We imagine a program that has the anticipated behavior and then must derive a source expression that, after translation into a suitable object expression, will cause a computer to enact that behavior. Even if we have the source expression of an existing program having similar behavior, we typically will know only approximately how its source differs from what is needed to produce the intended behavior.

If a program's source and object forms are known, these can be analyzed to produce behavioral characterizations. A program whose source form is not known can still be characterized in terms of its envisioned behavior, but then corresponding source and object expressions that will actually produce (an acceptable approximation of) that behavior must be discovered.

A program family, whose motivation is the ability to derive programs with intended behaviors, leverages both of these perspectives. In this, we are interested in how differences in source expression correspond to differences in functionality and how, via object expressions, these effect differences in computational behavior. Moreover, our interest is in how, starting with an envisioned behavior, we can systematically derive a program that verifiably produces that behavior.

In conventional practice, a needed program is envisioned and derived by creating an incomplete or otherwise flawed variant that is then iteratively modified, creating other variants that differ in order to enhance or correct particular aspects of its predecessors' behaviors, until an acceptable variant is achieved.

Similarly, when an existing program misbehaves or no longer meets changed needs, its replacement is envisioned as being just the old program but with some parts of its source expression changed.

Conversely, we can think of such efforts not as "changing" an existing program but rather as replacing it with a different, more suitable one, without consideration of specifically how the source forms of the two differ. It is as if we had independently realized two programs that exhibit similar behavior and we are selecting the one that is the best fit to perceived needs.

3.3 Distinguishing Among Programs

Two programs may be deemed to be more or less alike depending on which form of expression is used as the basis for comparison. At first, it may seem sufficient to presume that differences between two programs are exactly the differences between their source expressions. After all, in a constructive sense, the other forms can be thought of as deriving from the source form. However, a particular source expression may result in differing object expressions when translated for different computational environments. Similarly, programs having different source expressions (in the same or different notations) may correspond to identical functional or object expressions. Even more significantly, we can express behavioral characterizations of envisioned programs and explore differences between them entirely in terms of their differing behaviors and tradeoffs, without having existing representational expressions of those programs.

Still, comparing source expressions of programs provides a good foundation for understanding how programs can differ. A common prescriptive expression of the source differences between two programs is the series of changes required to transform one program's source into the other's. In this, a distinction between essential and incidental differences warrants consideration. An incidental difference results when the same meaning can be expressed in different forms. For example, in most notations, different spacing would be an incidental difference between two source expressions. Similarly, different developers may differently name and represent the same data. More significantly, notations often provide alternative ways of expressing the same functionality; these alternate expressions will differ in appearance but may (or may not) produce the same behavior depending on how each form is translated into object form. Consequently, a capability implemented independently in the same source notation by two different developers might not be recognized as similar because of incidental differences in expression even if they happen to produce identical behavior. Conversely, a difference would be considered essential if it results in different behaviors.

Comparing object expressions as an indicator of program difference, whether for different object expressions of the same source or for object expressions of different source expressions, is more complex. A program's object expression(s), being derivative of its source form, is a less reliable conveyor of essential difference because of the different forms that a translation can introduce. However as Batory has noted, correlating behavioral differences between two programs to source differences can provide a basis for understanding how changes in a program's source representation will affect its behavior [9]. Different translations from source to object form can introduce differences that may or may not correspond to differences in behavior. Two object expressions of a single program may differ as a result of how the source form is translated to achieve differing computational qualities or to conform to conventions and capabilities of each specific environment. Conversely, two programs may differ in their source form expressions and yet translate into identical object form expressions for a given computational environment.

Analogous to how source and object forms can impart different views of how much two programs differ, the functional expression of a program can differ from corresponding behavioral expressions due to the effects of different operational contexts (e.g., a multi-thread source program translated onto a single processor versus a multi-core multiprocessor versus a distributed network of processors). Similarly, two programs that are different implementations of the same functionality may exhibit behaviors that are more, or less, similar depending on how well they fit in a particular computational environment. The functionality of a single program may even exhibit different computational results in different environments. For example, one environment may implement some capability in hardware which in another environment must be emulated in software, resulting in behavioral differences in what from a functional perspective is the same program.

A proper determination of whether two programs differ may need to take all of these perspectives into account.

3.4 The Concepts of Similarity and Equivalence

The premise of program families is that we are interested in programs that are "similar". Similarity is informally the basis upon which humans intuitively categorize objects. Lin offers a formal definition, consistent with our intuition, of similarity as a fundamental concept [7]. We take *similarity* in the context of a program family as being a measure of the degree to which two programs are interchangeable for a specified purpose.

More precisely, we define similarity (~) to be a non-negative real-valued binary function over the instances of a specified set (the value of ~ is undefined for objects not in the set). The value of ~ expresses the degree to which two instances differ (larger values of ~ indicate more differences). Equality (=) is the zero-valued special case of similarity, denoting instances that are identical.

Equivalence (~~), being a generalization of equality, is a related concept to similarity, for when two objects are sufficiently similar for a given purpose that they are effectively interchangeable (i.e., their differences are inconsequential with respect to the intended purpose).

Equivalence is a binary relation over instances (s1, s2, s3) of a set such that:

(1) s1 ~~ s1 [reflexive]

(2) if s1 ~~ s2, then s2 ~~ s1 [symmetric]

(3) if s1 ~~ s2 and s2 ~~ s3, then s1 ~~ s3 [transitive]

Similarity in Practice

As a general concept, similarity has increasing utility in many specialized fields, such as information retrieval (full-text searching to identify articles having similar content[4]), genomics (analyzing genetic material for correspondences in sequence encoding or function, such as similarity in genetic features of different types of cancer[5]), or pharmaceuticals (differentiating drugs that treat the same condition but with differing costs, benefits, effectiveness, and effects[6]).

[4] access the abstract for any article in <`http://dmd.aspetjournals.org`>, then click in the associated services sidebar on "similar articles in this journal"

[5] "Implementation of a Functional Semantic Similarity Measure between Gene-Products" `http://hdl.handle.net/10451/14233` and Study reveals genomic similarities between breast cancer and ovarian cancers" <`http://www.nih.gov/news/health/sep2012/nci-23.htm`>

[6] "Incontinence Drugs: Benefits and Harms Compared" <`http://www.webmd.com/urinary-incontinence-oab/news/20120409/incontinence-drugs-benefits-and-harms-compared`>

In software, similarity has been used as a basis for detecting clones, malware, and duplicate code within and across programs. This has typically taken the form of analyzing and comparing the source or object representations of existing programs for indications of similarity in their content or structure [11]. Another approach involved characterizing programs in terms of measures of relevant qualities/aspects exhibited in a program's design and code, each program corresponding to a point in a multi-dimensional 'semantic' metric space, in which distance between two programs indicated the degree to which their properties were similar [12].

Conversely, conventional software reuse involves searching among already realized components for ones that will provide needed functionality. An investigation into how similarities in component interfaces can be indicative of similarities in functionality has resulted in a metric for identifying potential matches and a refactoring method for reducing differences [10]. Another study explored analyzing benchmark program code to discover similarities in relevant properties of different programs as a means of reducing the number needing to be used in evaluating the performance of a microprocessor design [13].

As these examples suggest, the information and criteria needed to determine whether objects are similar vary, depending on the objects being considered and the use to be made of such determination.

For programs, there has long been the perception that similar problems accommodate similar solutions. Often the degree of change required to modify a solution to fit a new problem has been thought of informally as a measure of similarity. The concept of a program family takes this to its logical conclusion, that we can envision a set of programs that constitute similar solutions to similar problems. This usually arises from practical experience with existing programs whose behaviors are expected to be useful as a basis for future programs.

3.5 Defining a Program Family as a Set

The set of all programs is the universal set U [5]. A subset Ux of U can be defined in two ways:

- Extension – enumeration of the members of U that belong to Ux
- Intension – application of a characteristic predicate that designates which members of U belong to Ux.

Any arbitrary set of programs can be designated as an extensional subset of U. One such sort of subset comes about through the common industry practice of deriving new programs as variants of an existing program. The resulting programs are, both by intent and by construction, likely to be somewhat similar, alike in some respects but differing significantly in others. Regardless, these programs constitute a coherent extensional subset of U, but its membership can change over time as the need for other similar solutions arise or as existing programs are modified or discarded.

In contrast, a *program family* f_i is defined as an intensional subset Uf_i of U, corresponding to programs that are similar in accordance with a characteristic predicate associated with f_i. Programs that satisfy the intensional predicate criteria are included in Uf_i; all other instances of U are excluded. If the predicate associated with f_i is changed, the membership of Uf_i changes accordingly. We can also define a program subfamily f_{ij} of any family f_i by conjunctive extension of the Uf_i predicate with a predicate for Uf_{ij}, selecting only those instances of Uf_i that satisfy the criteria specified for Uf_{ij}.

3.6 Distinguishing Among Instances of a Program Family

By definition, all programs in Uf_i are similar, in accordance with the intensional predicate for f_i. However, within the context of a program family, "similar" becomes a euphemism for different. Any two instances of f_i remain similar with respect to the intensional predicate but will differ in other essential and incidental ways.

In principle, any program that satisfies the intensional predicate is an instance of and can be derived from f_i by reduction of Uf_i to a subset that includes only the needed program. Reduction of an intensionally-defined set entails the conjunctive extension of the predicate with additional terms. The basis for these terms are deferred decisions that express differences in behavior that customers in a target market need a program to exhibit (and that establish why a single program will not suffice). The motivations for having different programs are sufficient as the determining factors that allow us to distinguish among them. Conversely, two programs will be considered equivalent, no matter how different they are otherwise, if their targeted market sees no purposeful difference in behavior between them.

As noted by Dijkstra and Parnas, such decisions guide engineers in determining how a particular program is to be realized. A particular program can be realized only after these deferred decisions have been resolved, expressing the needs of a particular customer. Different decisions lead to different programs being realized.

An advantage of using deferred decisions to characterize differences among programs is that there is no need to appeal to the physical realizations (source or object expressions) of those programs. Instead, the representational and behavioral expressions of each program are determined, within the context of a family, by how these decisions are resolved.

3.7 A Decision Model for a Program Family

Formalization of the set of decisions that discriminate among the instances of a program family f_i is referred to as the family's *decision model*. There is a precise canonical form for expressing a decision model.

A decision model Df_i for program family f_i is a set of decisions (d1, d2, ..., dn) that formalize how programs in f_i differ. A fully resolved decision model is one for which all required decisions have been resolved. The mapping from Df_i to Uf_i is one to many: every resolution of Df_i determines one or more instances of Uf_i and each instance of Uf_i is designated by exactly one resolution of Df_i.

It is presumed that the need for a program in f_i is known but which of those programs is not known. Decisions can be thought of informally as questions that have been deferred and must be answered in order to select the instance of f_i that best fits the perceived need.

There is a default scheme for characterizing individual decisions. Each decision is named and can be either required or optional. Each decision is one of three types:

- discrete: a simple choice, optionally having an associated value in a designated target notation (e.g., the notation in which a program's source is expressed)
- composite: a set of M constituent decisions, each constituent being optional or required
- repeating: an ordered, indeterminate number of uniformly defined constituent decisions

Any (partial or complete) resolution of the decision model Df_i for f_i corresponds to a subset of Uf_i. Each such subset comprises a set of equivalent instances of f_i relative to resolved decisions. The set of all possible complete resolutions of Df_i defines a partitioning of Uf_i such that each instance of f_i is a member of exactly one subfamily whose instances constitute an equivalence set within Uf_i.

Instances of a subfamily, defined by a partial resolution of Df_i, can later be distinguished from each other through the resolution of remaining unresolved decisions. In this way, by progressively resolving elements of Df_i, continuing to defer decisions in areas of uncertainty, the set of programs can be narrowed to those that are a better fit to a particular customer's needs; potentially, multiple alternative solutions can be derived to empirically evaluate and resolve uncertainties concerning the best fit to actual needs.

Similarly, instances of a fully resolved equivalence subset can be distinguished only by extending Df_i. Although the programs in such a subset are characterized by the same resolution of Df_i, they would nevertheless differ in some unspecified, presumably incidental, aspects. If meaningful differences in these programs are discovered, the ambiguity can be resolved simply by adding decisions to Df_i (possibly only with respect to the programs in that subset), so as to further partition this subset to account for those differences.

4. Formulating a Metric Space

A *metric space* defines a topology over an abstract space containing the members of a set S of objects [6]. The properties of a metric space define a relationship among members s1, s2, and s3 of S in terms of an associated non-negative "distance" metric d (S, S):

(1) d (s1, s2) = 0 if and only if s1 = s2 [identity]

(2) d (s1, s2) + d (s2, s3) >= d (s1, s3) [triangularity]

(3) d (s1, s2) = d (s2, s1) [symmetry]

These properties define the criteria for any measure of distance between members of S within the containing space.

4.1 Object Identity in a Metric Space

To satisfy the identity property for a metric space, a metric must exist that can determine the equality of contained objects.

There are in general two ways of framing object identity. The more common is to view objects as mutable, having identities that are extrinsically fixed and independent of any changes in an object's properties or composition. For example, a river retains its identity even as its constituent water changes or its course varies over time. The same holds for people: changing any particular attribute such as hair color, an organ, or finger prints does not constitute a change in identity.

Conventionally, this view of identity has prevailed for software as well: modifying a program is not viewed as creating a different program but merely as changing only selected aspects of that program. Analogously, a "new" program can be created as an identical copy of an existing program and then exist with a separate identity independent of the copied program.

This conception of programs as mutable presents a dilemma for an intensionally-defined set. With mutable programs, set membership would be unstable: changes to a program's properties could cause it to move into or out of set membership or equivalence to other members of the set. Furthermore, changing a program could have the indirect effect of making two different programs identical, violating the identity property for a metric space. We would not know whether two programs were actually equal except by knowing how they came to exist, whether as the same object with differing properties or as different objects even if their properties are identical.

With objective framing, identity is determined based strictly on the (immutable) properties of each program considered. The properties of a program cannot "change" – different values connote a different program. With this framing of identity, we can distinguish two programs simply by comparing their properties.

Conventional software practices are clearly sufficient to build a needed program with some degree of certainty.

The reason for this is that developers do not actually know which program they are building until it is complete: they know neither the exact form of its representation nor its exact behavioral properties. Instead they iteratively refine the representational expression of the needed program until it sufficiently approximates specified behavioral properties. The way to think of this if programs are viewed as immutable is that developers iteratively search the space of potential programs to find one that adequately satisfies their needs.

The intent with a program family is to in fact approach derivation of programs as being a search through a space of similar immutable programs. This search is guided by what from a customer perspective are the properties of a needed program for a best (possibly approximate) fit within that space. Selecting a program becomes a process of progressively reducing the space, through the tightening of similarity criteria, until a single program is identified.

4.2 Pseudo-Metric Spaces

The identity property for a metric space is strict: the distance between two objects cannot be zero unless the objects are identical. For some purposes, this is overly strict. It can be useful to associate objects that are not identical but that are similar enough to be considered interchangeable. Substituting the equivalence relation in place of equality for a more flexible definition of identity determines a relaxed form of metric space known as a "pseudo" metric space:

(1) d (s1, s2) = 0 if and only if s1 ~~ s2 [identity]

In a pseudo-metric space, the distance between any two members of S will be zero if they are either identical or equivalent; the distance metric for a space must encapsulate the criteria for what constitutes equivalent objects in that space.

Given a pseudo-metric space for a set Ux, an equivalent metric space can be derived by defining a subset Ux′ of Ux in which collections of equivalent members of Ux are replaced by single archetypal instances. This entails partitioning the members of Ux into a collection of subsets Ux_i (i = 1..n) such that each Ux_i contains only equivalent members. By this, each member of Ux is mapped into exactly one Ux_i. For each Ux_i, one member would be chosen as its archetype and added to the Ux′ subset of Ux. The resulting set Ux′ will have exactly the same number of members as Ux has equivalence subsets. The pseudo-metric d can then be reframed as a metric for Ux′, which containing no equivalent instances, will satisfy the stricter identity property required for a metric space.

5. A Distance Metric For A Program Family

To define a metric space for a set of programs comprising a program family, we will interpret difference as being a measure of distance: the distance between two programs is a measure of how different they are. The distance between a program and itself is zero (also between any two equivalent programs in a pseudo-metric space). The distance between any other pair of programs will be greater than zero. We will show how differences between instances of a program family can be expressed and used to compute a distance metric for a pseudo-metric space. We can then apply the simple pseudo-metric-to-metric transformation to create a condensed but equivalent program family that, using the same distance metric, will constitute a proper metric space.

With a program family, we have a set of programs that we have established as being similar due to the intensional predicate for the set; programs that are not in that set are of no further interest. From a decision model for distinguishing instances of a family f_i, we have criteria that can be used to describe how any two programs in Uf_i differ.

A decision model only indirectly expresses how programs differ; what it expresses directly are the considerations that cause customers to need different programs. This is a sufficient basis for describing program differences because all more intrinsic differences must be traceable to the differences that matter to customers. We have no reason to distinguish between programs whose differences are not of interest to the customers in a targeted market. A key feature of this approach is that programs can be characterized and related to each other within the metric space without the programs having been built.

To define a pseudo-metric space for the family, we only need a means to compute a distance pseudo-metric for the set based on the decision model. By distinguishing between programs by the differences in how the decision model is resolved for each program and defining a computation that quantifies this difference, we create a metric that can be interpreted as the distance between any two instances of a program family.

5.1 A Simple Metric

From a decision model associated with f_i, there is a simple way and a complex way to define a metric that will satisfy pseudo-metric criteria. The decision model defines the choices that a developer needs to have resolved in order to know which member of a program family corresponds to what a customer needs. Every program in Uf_i is characterized by a specific resolution of the decision model. Any complete resolution of the decision model will designate a subset of Uf_i that satisfies the indicated criteria.

The simple alternative is to define what is known as a discrete metric for the family: for each pair of programs p1 and p2 that are members of the family, d (p1, p2) = 0 if and only if p1 and p2 are equivalent programs (i.e., characterized by identical resolutions of the decision model), otherwise 1.

The limitation of the discrete metric is that it is overly simple, treating all differences as being equally weighted. However, it does define a partitioning of Uf_i into equivalence subsets. Programs in an equivalence subset are interchangeable with respect to decision model criteria for f_i, that is, they do not differ in any way that matters to customers. If a customer's needs are met by the instances of a particular equivalence subset, any instance of that subset can be chosen arbitrarily as all provide equivalent behavior, in the view of customers in the targeted market.

5.2 A Complex Metric

A more useful metric can be defined as the computation of an actual difference value between any two instances p1 and p2 of a program family. Because each instance of a family is characterized by a resolution of the family's decision model, an analysis of the differences in the decision model resolutions for p1 and p2 is a sufficient basis for such a metric. This has the advantage over the simple metric of giving a more accurate approximation of how much difference there is, from a customer perspective, between p1 and p2. In addition, it may be feasible to identify programs that are a better fit to some need by first finding an inferior fit and then examining "near by" programs that differ favorably with respect to that need.

For a complex metric, the distance between two programs is computed in terms of the difference between their associated decision model resolutions. The relative importance of a decision is determined by its level in the hierarchy of decisions represented by the decision model. A first approximation to computing the value of a more precise distance metric has four elements:

(1) For any optional decision di, d (p1, p2) = 0 if both di (p1) and di (p2) are omitted, or 1 if one is omitted but not the other, otherwise evaluate according to decision type

(2) For each discrete decision di, d (p1, p2) = 0 if di (p1) = di (p2), otherwise 1

(3) For each composite decision di, d (p1, p2) = the sum of constituent decision metric values divided by the total number of constituent decisions M

(4) For each repeating decision di, d (p1, p2) = the pairwise sum of constituent decision metric values divided by the greater of the number of component decision values comprising p1 and p2

Each of these computations will result in a rational (real) value between 0 (identical decisions) and 1 (no decisions in common at all). For the purpose of computing the aggregate distance metric for programs p1 and p2, the decision model itself is treated as a composite decision.

5.3 Further Metric Refinements

The complex method as described yields a metric that may suffice but is still not an entirely comprehensive or precise measure of the differences between two programs. Rather it provides an approximation of how much two programs differ in the aspects (only and most importantly) that matter most from a customer perspective.

Three minor refinements, and perhaps others, could make this metric more precise:

- Treat any required decision that is unresolved as if it were optional in computing an interim metric; this will result in a value that is the lower bound on the difference between two programs.
- For a target-valued discrete decision, compute the metric to be proportional to the degree of difference in the decision's values (i.e., using a string metric to compute a value between 0, a perfect match, and 1, no match at all).
- For a repeating decision, discount component ordering by computing the metric as an average of the metric values obtained under all reorderings of the decision value having more elements.

5.4 A Strategy for Defining a Program Family Metric Space

The set of programs Uf_i comprising program family f_i and the simple distance metric applied to the decision model associated with f_i defines a pseudo-metric space. The simple metric effectively partitions Uf_i into equivalence subsets. Each of these subsets is collapsed into a singleton by the arbitrary selection of any one of its equivalent instances.

The programs from these singleton subsets constitute a subset Uf_i' of Uf_i, such that Uf_i' consists entirely of programs that are one-to-one uniquely selected by resolutions of Df_i. Each possible resolution of Df_i will select exactly one and only one instance of Uf_i', satisfying the strict identity property for a metric. Associating the complex metric with the reduced set Uf_i', which is behaviorally equivalent to Uf_i, defines a metric space that quantifies the substantive differences among instances of f_i'.

This formulation of a program family as the set of programs Uf_i' and the complex metric, with all instances of the family characterized in terms of a single decision model, satisfies the identity, triangularity, and symmetry properties for a metric space. Per the identity property of a metric space, and due to the reduction of all equivalent subsets to singleton sets, each full resolution of the decision model selects exactly one program.

5.5 Practical Relevance of a Program Family Metric Space

The intrinsic value of formulating a program family as a metric space is that it gives an objective mathematical interpretation to the concept of similarity among programs. It provides a basis for viewing programs as abstract conceptions that have many physical realizations, differing in both essential and incidental aspects. It provides a medium for exploring how and why programs differ, based on why different programs are needed rather than on the superficial basis of their physical representation or the more complex basis of how to accurately formalize the many facets of behavior.

The practical value of the metric comes in using it as a measure of the distance between a "theory" of the needed program and individual instances of the program family that can be built (i.e., how well each program satisfies a particular set of needs). Using the decision model, we describe an envisioned program that will satisfy a customer's needs. We then (in principle) determine the distance between that envisioned program and each of the instances of the program family in order to find an instance that best matches those needs. Ideally we will find a program that has a distance metric of zero to the envisioned program.

An incomplete resolution of the decision model, corresponding to an incomplete program theory and suggesting uncertainty about needs, inherently implies multiple alternative programs (just as the whole decision model implies the entire set of constructible programs comprising the family). Choosing among multiple programs presumes being able to specify and compare multiple resolutions of the decision model, possibly going so far as building models of alternative programs for analysis or even full realizations for comparative empirical evaluations. The distance from the incomplete program theory to each of these programs can be determined inversely by describing the decisions that would result in each program. Another alternative is to generalize the program theory by deferring resolution of some decisions beyond program derivation, emulating a subset of the program family as a hybrid program that requires resolution of deferred decisions during program operation.

6. Product Families As A Generalization Of Program Families

The generalization from programs to products is straightforward. A product family is simply a set of similar products, each of which conveys one instance of a corresponding program family. Beyond being the conveyance of a program, a product includes all of the elements and artifacts entailed in creating and instituting the means for operation of a business process within an enterprise.

In simple terms, a product is everything (e.g., specifications, a program realized as software and/or hardware components, quality criteria, materials for installation/validation/training/use) needed for the provision of a capability to an enterprise: a product is a means for instituting changes in how an enterprise operates.

Each product, by convention, extends and frames a particular program, but the same program may be conveyed by multiple products. With a product family being the extension of a conveyed program family, the decision model associated with the program family is also a sufficient starting point for distinguishing among the instances of the product family. Resolution of this decision model determines a specific program and, at least partially, determines its conveying product (or more precisely, a set of potential conveying products).

Analogous to the program family it conveys, a product family f_i is conceived as an intensionally-defined subset Uf_i of the set U of all such products. Existing products that belong to this set, having each been physically realized and deployed into use, constitute an extensionally-defined subset of Uf_i. (Synthesis defined a *product line* as being this subset.) The membership of this subset evolves as new instances of the product family are built.

If there is only a single instance product for conveying each program, the product family constitutes a metric space based on the same decision model-derived metric as the subsumed program family. If any programs are conveyed by multiple products, we can view the set of products associated with each such program as an equivalence subset of the product family. In this case, the product family reverts to being a pseudo-metric space.

By deferring considerations that are discriminators among products but not of the programs that they convey, we can still rely on the program family metric for determining the best fitting program and then choose among the set of conveying products based on how they differ. Product differences are typically independent of conveyed program differences and concern conventions of product packaging or business process transition as instituted by the targeted enterprise. A product-options extension to the program family decision model is easily defined to account for these product discriminators, resulting in a metric space for the family of complete products.

On First Conceiving a Product Family

Synthesis was conceived based on an understanding of how to express a product family in aggregate as the basis for a product generator and a streamlined product manufacturing (i.e., application engineering) process. However, lacking an effective criteria for limiting the scope of a product family, there was not a disciplined way to limit the effort required to do this. The answer lay in two determinations that any effective manufacturing enterprise must make:

- (*domain)* What type of products do we have the competence (knowledge, experience, and expertise) to build?
- (*coherent market)* What capabilities would customers, having similar needs for such products, need now and in the future?

These suggest a potential convergence between a supplier's capabilities and their customers' needs. A coherent market represents what customers need, a domain represents similar products that a supplier has the ability to build. A domain that aligns to a market constitutes a business opportunity. An effective domain-market pair will be mutually defining, co-dependent, and co-evolving.

Domain knowledge is expressed in the aggregate representation of the product family; domain expertise is expressed in the process for the manufacture (specification, evaluation, and generation) and deployment of instances of the product family; and domain experience resides in the people who comprise the organization.

As a practical matter, in the realization of a product family, formalization of the intensional predicate that characterizes included products would be extremely complex and entail significant effort (consider that this predicate would have to characterize how the envisioned products differ from all other conceivable products). Instead, in practice, it suffices to express this predicate informally in a set of assumptions of "commonality" that define what distinguishes included products from those excluded. (In fact, these assumptions might reasonably be formalized as terms in a corresponding, though possibly incomplete, predicate, but the benefit of doing so is not evident.) These assumptions suffice as the basis for creating a concrete, aggregate realization of the corresponding product family. Customized products can then be "selected" (i.e., derived) from this representation based on resolution of the associated decision model.

In reality, the aggregate realization of a product family serves as a de facto formalization of the intensional predicate, establishing exactly what products are included: any product that can be derived from the family is included and all others are excluded. In practice, however, a domain is typically conceived as including products that are not initially derivable. One option in this case is to derive an approximation of a needed product by modifying the corresponding resolution of the decision model so that it describes a product that can be derived but is a "close", rather than exact, fit to customer needs. In any case, a domain will inevitably evolve over time both to accommodate initially unsupported products and, along with its corresponding intensional predicate, to account for changing market needs.

7. SUMMARY

This paper has offered a mathematical formulation for understanding the concept of a product family. Based on set and metric space theories, this formulation establishes similarity as a mathematical relationship over a set of products that are perceived intuitively as being similar. This may allow us to better formalize the process and mechanisms by which customized products as a best fit to a customer's needs can be efficiently manufactured and revised as those needs change.

REFERENCES

1. E. Dijkstra, "Notes on Structured Programming: On Program Families", *Structured Programming*. London: Academic Press, 1972, 39-41.
2. D. Parnas, "On the Design and Development of Program Families", *IEEE Trans on Software Eng.* SE-2, 1976, 1-9.
3. G. Campbell, S. Faulk, and D. Weiss, *Introduction to Synthesis*. Herndon, Va: Software Productivity Consortium, 1990. <www.domain-specific.com/PDFfiles/IntroSyn.pdf>
4. G. Campbell, "Domain-specific Engineering", Proceedings Embedded Systems Conference, 1997. <www.domain-specific.com/PDFfiles/DsE-RSP.pdf>
5. S. Hu, *Introduction to Contemporary Mathematics*, Holden-Day, Inc., 1966.
6. J. Kelley, *General Topology*, D. Van Nostrand Company, Inc., 1955.
7. D. Lin, "An Information-Theoretic Definition of Similarity", *Proceedings of the Fifteenth International Conference on Machine Learning*, 1998, 296-304.
8. A. Walenstein, et.al., "Similarity in Programs" *Duplication, Redundancy, and Similarity in Software* (Dagstuhl Seminar Proceedings 06301), 2006.
9. D. Batory, "The Challenges and Science of Variability", *Duplication, Redundancy, and Similarity in Software* (Dagstuhl Seminar Proceedings 06301), 2006.
10. B. Kratz, R. Reussner, and W. van den Heuvel, "Empirical Research On Similarity Metrics For Software Component Interfaces", *Transactions of the Society for Design and Process Science* 8 (4), 2004, 1-17.
11. S. Cesare and Y. Jiang, *Software Similarity and Classification*, Springer, 2012.
12. E. Kapetanios and S. Black, *On the Notion of Semantic Metric Spaces for Object and Aspect Oriented Software Design*. University of Westminster, London, UK, 2008.
13. A. Phansalkar, A. Joshi, L. Eeckhout, and L. John, "Measuring Program Similarity", *IEEE International Symposium on Performance Analysis of Systems and Software*, 2005, 10-20.