Modeling Product Line Architectures through Change Sets and Relationships

Scott A. Hendrickson and André van der Hoek
Department of Informatics
University of California, Irvine
Irvine, CA 92697-3440 U.S.A.
{shendric, andre}@ics.uci.edu

Abstract

The essence of any modeling approach for product line architectures lies in its ability to express variability. Existing approaches do so by explicitly specifying variation points inside the architectural specification of the entire product line, usually with optional and alternative elements of some form. This, however, leads to a sizable mismatch between conceptual variability (i.e., the features through which architects logically view and interpret differences in product architectures) and actual variability (i.e., the modeling constructs through which the logical differences must be expressed). We contribute a new product line architecture modeling approach that unites the two. Our approach uses change sets to group related architectural differences and relationships to govern which change set combinations are valid when composed into a particular product architecture. The result lifts modeling of variability out of modeling architectural structure, consolidates related variation points, and explicitly and separately manages their compatibilities.

1. Introduction

The use of product lines as a principled form of software reuse has significantly increased over the past decade [6, 7, 22, 27]. This paper is concerned with how to define and evolve a product line and the individual products that constitute it. While a few exceptions exist [5, 15], the predominant way of doing so is through a product line architecture (PLA): an abstraction in which the high-level structure of the product line is defined in terms of architectural elements that are (eventually) mapped onto the source code that implements the product line [6].

A key aspect of any PLA is that it must capture the variabilities that exist in the product line. To date, a variety of different modeling languages have been proposed for capturing PLAs [2, 23, 32, 34]. Common to all of these modeling languages is that they capture a PLA as a single, monolithic architecture containing variation points to differentiate among products. These variation points, architectural elements themselves, take on several forms. Koala uses switches [23], Ménage allows optional, variant, and optional variant elements [11], and COVAMOF utilizes optionals, alternatives, optional variants, variants, and values [32]. In all of these cases, the result is what could be considered a configurable architectural specification: by making appropriate choices to resolve the variation points, a single product architecture describing a single product is selected from the PLA [30].

While insertion of variation points in an architectural description can adequately model PLA variation, it also exhibits a sizable mismatch between conceptual and actual variability. For instance, consider an optional feature, which is conceptually described as involving one or more components and links that are included or excluded in unison. However, a traditional approach models each component and link separately as a variation point in and of itself, each governed by its own redundant (often Boolean) clause to determine inclusion or exclusion. Thus, the way in which an architect logically views and interprets a PLA, i.e., in terms of the features that determine differences among individual products, does not match the modeling constructs available to express those differences.

The problem of redundancy is compounded and transformed into an intricate problem of relationship management when additional kinds of variabilities are introduced (e.g., variants, optional variants, alternatives), and especially when those variabilities interact (e.g., selection of one variant requires inclusion of another). Keeping track of which individual variation points belong together quickly becomes a nightmare of repetitive, brittle, and non-intuitive expressions.

The solution presented in this paper alleviates these difficulties by bringing together conceptual and actual variability. Specifically, we contribute a new modeling approach that uses *change sets* to group related architectural differences and *relationships* to constrain

which combinations of change sets are valid when composed into a particular product architecture.

A change set consolidates related variation points into a single conceptual variation. Instead of embedding variation points directly in the architecture, our approach promotes change sets to be first-class entities consisting of sets of closely-related additions, removals, and (property) changes performed on the architectural description. Doing so eliminates the need for variation points entirely. Furthermore, change sets are independently manipulatable, making them the primary representation mechanism and focus for architects.

Relationships ensure that only desired product architectures are constructed. They allow an architect to (1) explicitly set the rules according to which change sets may be combined, and to (2) form the basis for the modeling of variants, includes, excludes, and other kinds of concepts common to PLA modeling [34]. Note that relationships are specified at the same level as change sets, thereby removing the redundancy existent in previous approaches and separating out each individual relationship into its own entity that can be independently manipulated and managed.

Overall, our approach turns the existing approach of "architecture first, variability second" into one of "architecture first, variability first." Both an architecture and its variability are equally important and must be equally available to the architect. Matching the modeling facilities with conceptual thinking is a critical first step in realizing this equality. We believe the results presented in this paper demonstrate that our approach successfully achieves this first step.

2. Background

The work presented in this paper relies on concepts from the areas of software architecture, which includes product line architecture, and configuration management. We introduce these concepts here.

2.1 Software Architecture

The field of *software architecture* provides high-level abstractions for representing the structure, behavior, and key properties of a software system. Software architectures involve descriptions of the elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns [26].

Architecture description languages (ADLs) [20] aid architecture-based development by providing formal notations for describing software systems. Examples of ADLs include C2SADEL [19], Darwin [18], Rapide [17], UniCon [31], and Wright [1]. Some are supported by an architecture design environment through which an architect can graphically design and manipulate architectures, e.g., ArchStudio [12] or AcmeStudio [33].

Whereas a "regular" software architecture only defines the structure of a single software system, a *product line architecture* (PLA) defines the architectural structure for a set of closely-related products [6]. As such, a PLA serves a dual role. First, it must support understanding and manipulating the commonalities and variabilities that exist among individual product architectures constituting the PLA. Second, it must support identification of one or more individual product architectures from the PLA, for instance to deploy one or more individual products to a client.

A number of ADLs support the specification of product line architectures, including xADL 2.0 [9], Koala [23], and GenVoca [3]. These ADLs typically distinguish *core elements* from *variation points*, the latter specifying places in the PLA where differences exist among specific product architectures. Most express these differences in some form of configuration or constraint language and promote commonly-used rules to first-class language constructs. For instance, xADL 2.0 uses Boolean expressions and Koala has a language construct for switches that routes connections to one of several alternative interfaces.

2.2 Configuration Management

The discipline of *configuration management* (CM) has been primarily concerned with capturing the evolution of a software system at the *source code* level [10]. For this, it has extensive and detailed mechanisms and procedures for storing multiple versions of code and allowing multiple developers parallel access to that code [8]. Automated conflict detection and merge routines help in reconciling overlapping changes that may arise as a result of parallel development [21].

Of interest to this paper are the concepts of extensional and intensional versioning [8]. In *extensional versioning*, the configuration management system focuses on managing versions of artifacts that result after making changes. Typically, a version graph is used to relate different versions of an artifact; developers retrieve a particular version, modify it, and then add the new version to the graph when complete.

In contrast, *intensional versioning* makes changes a first class entity, inverting the relationship between versions and changes [8]. Instead of ensuring that each version is uniquely stored and accessible, intensional versioning stores each change as a change set (a "delta") independently from the other changes. So, instead of requesting a version of an artifact, developers retrieve an artifact by requesting a series of change

sets from which a "version" is constructed. Similarly, after modification of this "version," the delta between this new and the original version is stored as an individually-identifiable change set. This has the advantage that new incarnations of an artifact can be composed by mixing and matching different change sets.

3. Motivating Example

To understand the problems with current PLA modeling approaches, we introduce a motivating example: a hypothetical software system implementing an audio player available in a *Free*, *Trial*, and *Pro* version. As shown in Table 1, each version differs in the set of components that constitute its software architecture. Each architecture is shown using boxes for components, small boxes with directional arrows for interfaces, and lines connecting interfaces for connections.

Figure 1 shows a representation of the audio player as a traditional PLA. The notation is as follows: core elements belonging to all product architectures are shown as solid boxes or lines, optional elements as dashed boxes or lines, and variant elements as large boxes containing the variants. Additionally, but not shown, each variation point is annotated with a Boolean guard that specifies how to construct desired products. For example, the *CD Reader* component (part of a variant component) has a guard that reads:

ProductType == 'free' || ProductType == 'trial'

indicating that this particular variant is included in both the *Free* and *Trial* versions of the player.

Modeling product variation as individual variation points in a PLA reveals the first difficulty: *high-level conceptual desires must be expressed in terms of multiple, low-level variation points.* In our example, we *conceptually* desire "a *Free* product that plays only CD's" and "a *Trial* product with a purchase reminder that plays both CD's and MP3's." However, these must *actually* be expressed disjointly, using many individual variation points. For example, modeling the MP3 variability involves the *MP3 Variant*, its two optional interfaces, links, and an optional *Sound Source* interface.

Additionally, each low-level variation point must be governed by its own constraint expression to determine which parts of the PLA are included in which products. These are highly redundant since, as just discussed, multiple variation points may be required to express a single feature. Thus, the second difficulty: *constraint expressions are redundantly spread throughout the architecture*. While these expressions can sometimes be updated automatically [11], consider what happens when creating the new *Basic* version of the player shown in Figure 2. This version is similar to the *Pro* version, except that it includes the *CD Reader* from the *Free* version. When expressing this new product, the architect must tediously check, and potentially update, each and every constraint expression. Here, capturing the *Basic* product would involve updating the *CD Reader*, *MP3 Encoder / Decoder*, and *Sound Source* components, their interfaces, and links.

The third issue is that *constraint expressions imply relationships, hiding dependencies among features*. This can be seen with the additional interface on the *Sound Source* component, as highlighted in Figure 2. It was previously only present in the *Pro* version of the product. However, it is not the version of the product that determines its inclusion, but the presence of the *CD Reader / Writer* or *MP3 Encoder / Decoder*. Current techniques only allow one to model this interface as optional, hiding the dependency in the constraint and, thus, losing the relationship.

Table 1. Different audio player versions.

Product	Architecture
Free	CD Reader, Sound Source, Player
Trial	CD Reader, Sound Source, Player, Purchase Reminder, MP3 Decoder
Pro	CD Reader / Writer, Sound Source, Player, MP3 Encoder / Decoder

4. Approach

Our work was sparked by the observation that existing approaches to modeling product line architectures are predominantly extensional in nature. However, as the example in Section 3 shows, conceptual differences in product features and their interrelationships are not

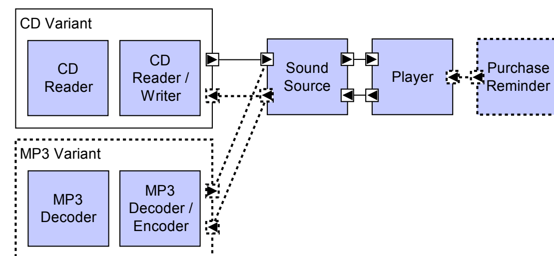


Figure 1. Audio player product line architecture.

easily expressed in the available modeling constructs.

The first key insight to our solution is that an intensional approach based on change sets provides a natural mapping from conceptual intent and understanding to actual realization. The second key insight is that "straight" change sets, as they are applied in the configuration management field, are not sufficient: explicit and detailed management of change set relationships must complement their use. Below, we detail these two insights and outline our solution in the context of the motivational example.

4.1 Change Sets

The traditional model of intensional versioning as used in the field of configuration management distinguishes baselines from change sets. A baseline is a *beginning*, formed by a configuration of artifacts that represents a stable state of the software being managed. Change sets represent independent *increments* from the baseline, such as a bug fix or a new feature addition. Each change set is captured as a "diff," detailing the exact lines of code added, removed, or changed with respect to the baseline [8]. A particular version of the software is then constructed by merging a desired set of change sets to the baseline.

The advantages of this approach are twofold:

1. Because each change set encapsulates a logically-related set of changes, it provides developers with a natural model of interacting with the configuration management system. No longer must they mentally map desired conceptual features onto specific artifact versions; they can request features by name.
2. Because each change set is built from the baseline and stored independently from other change sets, it is possible to combine change sets in new ways. This is the key to intensional versioning: combining different change sets produces different results.

These are the exact advantages that we would like to achieve with respect to PLAs. However, there is one major disadvantage to using intensional versioning:

3. Because each change set is independent, change sets may exhibit conflicting changes, which produce invalid or incomplete results when combined. For instance, one change set may remove a line of code that another happens to modify.

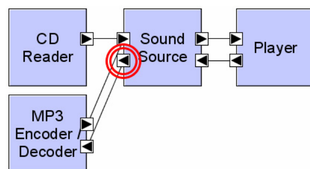


Figure 2. New *Basic* audio player version.

We discuss how we address this disadvantage using the concept of relationships and a specialized merge process in the next sections. Here, we describe the foundation of modeling PLAs using intensional versioning.

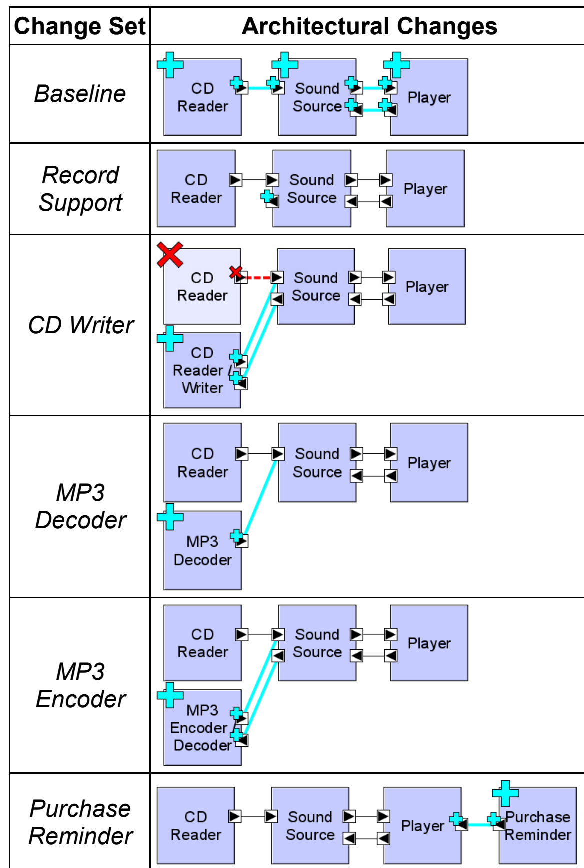
We first modify the code-centric concepts of baselines and change sets to capture architectural concepts. This is done by making architectural elements the constituents of change set additions and removals. Second, we increase a change set's independence by abandoning the concept of a (required) baseline, instead always starting out with an empty (virtual) baseline. This allows an architect to use change sets in new contexts without requiring a core (baseline) architecture as a starting point. Of course, an architect may still use a change set as a baseline if desired, for example, to capture elements common to all products. Finally, we allow a change set to be modified at any time. Because PLA design is not linear, it must be possible to update all change sets at all times, so features stay coherent. This differs from change sets in the configuration management world. Those change sets only capture the evolution of a baseline over time and are not intended to be modified once they are created. While malleable change sets may introduce more conflicts during the design process, we address this issue with relationships and a specialized merge process to be described later.

To illustrate the use of change sets in our approach, consider the example presented in the previous section, but now restated using change sets. The approach is flexible enough that variability can be captured at many levels of granularity. Exactly how is up to the architect and their preferences. For instance, a very fine-grained approach could be used where each component, interface, and link is separately modeled as optional, each represented by its own change set.

Alternatively, a very coarse-grained approach could differentiate entire products. Suppose we begin with the *Basic* audio player as a first (baseline) change set. To turn the *Basic* player into the *Trial* version requires a change set that: (1) adds the *Purchase Reminder* component, its link and interfaces, (2) removes the *MP3 Encoder / Decoder*, replacing it with the *MP3 Decoder* component, and (3) removes the now defunct link to, and interface on, *Sound Source*. Similarly, another change set could capture the changes that morph the *Basic* player into the *Free* version, and still another would morph the *Basic* player into the *Pro* version.

These two approaches, fine-grained change sets and product-oriented change sets, technically work, but perhaps do not have as many benefits from an architect's point of view as others. A better approach is to map features onto change sets. This is demonstrated by the change sets listed in Table 2, where each change set

Table 2. Audio player PLA change sets.



Change Set	Architectural Changes
Baseline	CD Reader, Sound Source, Player
Record Support	CD Reader, Sound Source, Player
CD Writer	CD Reader, Sound Source, Player, CD Reader / Writer
MP3 Decoder	CD Reader, Sound Source, Player, MP3 Decoder
MP3 Encoder	CD Reader, Sound Source, Player, MP3 Encoder / Decoder
Purchase Reminder	CD Reader, Sound Source, Player, Purchase Reminder

adds a particular feature to the audio player. In each of these change sets, an annotation with a "+" means that it adds that particular element and an annotation with an "×" means that it removes that particular element. Elements without any annotations are there for the sole benefit of the reader, placing the changes within context. For example, the *CD Writer* change set removes the *CD Reader* component with its link, and adds the *CD Reader / Writer* component with its links.

Note that these feature-oriented change sets have an implicit hierarchical arrangement. The *Record Support* change set, which only adds a single outgoing interface to the *Sound Source* component, prepares this component for use by other components that record audio streams (in addition to playing them). This was separated out from the *MP3 Encoder* and *CD Writer* change sets, which both rely on its presence. The other change sets depend only on the *Baseline* change set. We will see in the next section that these implicit hierarchical arrangements can be conveniently and explicitly modeled using relationships.

Our approach does not differentiate between common elements and variable elements. Instead, our approach uses changes sets to capture *both* concepts. One advantage to this approach is that common elements themselves may be modified by later change sets. For instance, the *Baseline* change set captures elements initially thought to be common to all product architectures, while the *CD Writer* change set removes the *CD Reader* component. This allows new variability or updated common functionality to be introduced without the need to go back and modify all other product variants. Moreover, it also supports modeling product populations [24], which involve entirely different products to which similar changes are applied.

Finally, we note that change sets explicitly address the first issue presented by our motivating example. They consolidate related changes into a single conceptual variation, uniting concepts with the actual modeling constructs, therefore allowing an architect to think in terms of features rather than variation points. Of course, the architect is the one who ultimately decides the exact breakdown of a PLA into change sets; they may prefer alternative breakdowns than by feature. Our approach still supports them fully in doing so.

4.2 Relationships

Capturing product lines using an intensional approach introduces new challenges during composition. Invalid products may be composed out of change sets that contain conflicting or dependent changes. We use relationships to enable an architect to express and reason about such dependencies, to support an architect in creating only desired product architectures.

One type of relationship, *structural dependencies*, operate at the level of the architecture structure, and arise when elements introduced by one change set depend on elements introduced by another. For instance, the *MP3 Encoder* change set structurally depends on the *Record Support* change set because it adds a link that connects to an interface introduced by the *Record Support* change set. By the same reasoning, one change set can conflict with another if elements it introduces depend on elements the other removes.

While structural dependencies are of a syntactic nature, a PLA also involves modeling semantic relationships. One such type, *compatibilities,* dictates which change sets are and are not compatible with one another, based on conceptual design knowledge of the architect. For instance, our example has a trial version of the product without record support, but with a nagging purchase reminder. The intent is that users will try the product and decide to purchase the full version for the extra feature (or to remove the nag screen). Consequently, the *Record Support* and *Purchase Reminder* change sets are mutually exclusive: both change sets should never be included at the same time.

Another semantic relationship, *composition*, is used to group change sets into higher-level concepts, such as a subsystem or entire products. For instance, it would be useful to record that the *Trial* version of the audio player is composed of the *Baseline*, *MP3 Decoder,* and *Purchase Reminder* change sets. We can do so by defining a *Trial* change set and a corresponding (composition) relationship stating that when including the *Trial* change set, the *Baseline*, *MP3 Decoder,* and *Purchase Reminder* change sets must also be included. Composition relationships may also refer to other change sets, which represent compositions themselves, allowing their hierarchical composition. It is also possible to establish relationships among composition relationships to model high-level conceptual compatibilities.

Structural dependencies, compatibilities, and compositions are expressed using the following constructs:

1. *and relationships* state that if *all* change sets *a*, *b*, and *c* are included, then *d* must also be included;
2. *or relationships* state that if *any* change set *a*, *b*, or *c* is included, then *d* must also be included; and
3. *variant relationships* state that from a set of change sets *a*, *b*, and *c*, only a certain minimum and maximum number may be included at the same time.

The first two relationships can designate the inclusion of multiple change sets (e.g., if *a*, *b*, and *c* are included, then *d*, *e*, and *f* must also be included), can have negated "source" change sets (e.g., if *a* and *b* are included and *c* is *not* included, then *d* must be included), and can have negated "destination" change sets (e.g., if *a* or *b* are included then d must *not* be included). They may also be specified without "destination" change sets, in which case the *and relationship* is interpreted as a disjunction (e.g., *a*, *b*, and *c* must always be included) and the *or relationship* as a conjunction (e.g., *a*, *b*, or *c* must always be included). The variant relationship may contain an arbitrary number of change sets, and may limit the number of change sets selected concurrently to one (making a group of change sets mutually exclusive, creating a switch [23]) or multiple (creating what COVAMOF terms an alternative [32]).

As with change sets, different ways exist to choose and organize relationships. This is influenced by the choice of change sets, but also by personal preferences of the architect. For instance, we previously discussed modeling the *Record Support* and *Purchase Reminder* change sets as mutually exclusive. However, it would be as valid to create a relationship stating that including the *Free* or *Trial* change sets may requires excluding the *Record Support* change set. This is a matter of taste, and usually evolves as the PLA evolves.

Finally, we note that relationships, combined with change sets, address the remaining issues presented by our motivating example. First, redundancy is avoided because relationships refer to change sets rather than individual architectural elements. Also, a relationship only needs to be modeled once, after which any valid product architectures must satisfy it. Second, because relationships are modeled explicitly, conceptual interdependencies among change sets are no longer hidden.

4.3 Merging

Relationships aid an architect in composing only valid collections of change sets. However, since we are capturing PLA concepts and allow an architect to "go back" and modify a change set at any time, it is possible to produce circular (or cyclic) dependencies, which requires a special merge algorithm, compared to traditional configuration management approaches.

If we used a sequenced merging algorithm with circular and cyclic structural dependencies, two problems arise. The first we refer to as "ghost additions," and occurs when a link from the first change set relies on the presence of an interface from the second change set and, vice versa, a link from the second change set relies on the presence of an interface from the first change set. Regardless of which change set is applied first, a necessary element will not be there. The second problem is "ghost removals:" if two change sets each remove an element established by the other, one of those removed items is bound to erroneously reappear.

To address these two problems, we use a special merge algorithm. First, we apply all additions of all change sets, in the order of components first, interfaces next, and links last. Then we perform all of the removals, in the order of links first, interfaces next, and components last. The result is that all necessary elements are always there, avoiding ghost additions, and elements that are intended to be removed are always removed, avoiding ghost removals. The benefit of this approach is that cyclic change sets are applied predictably and consistently.

Note, this advocates an automatable approach with no human intervention. We can see some situations, though, in which an architect may want to have a slightly different behavior or even want to exhibit manual control. Other merge algorithms and implementations can be envisioned accordingly and inserted into our EASEL environment (see Section 5).

5. EASEL

To demonstrate our approach, we have implemented it in EASEL, a new product line architecture modeling environment. As illustrated in Figure 3, EASEL is partitioned into two separate areas: a drawing canvas for

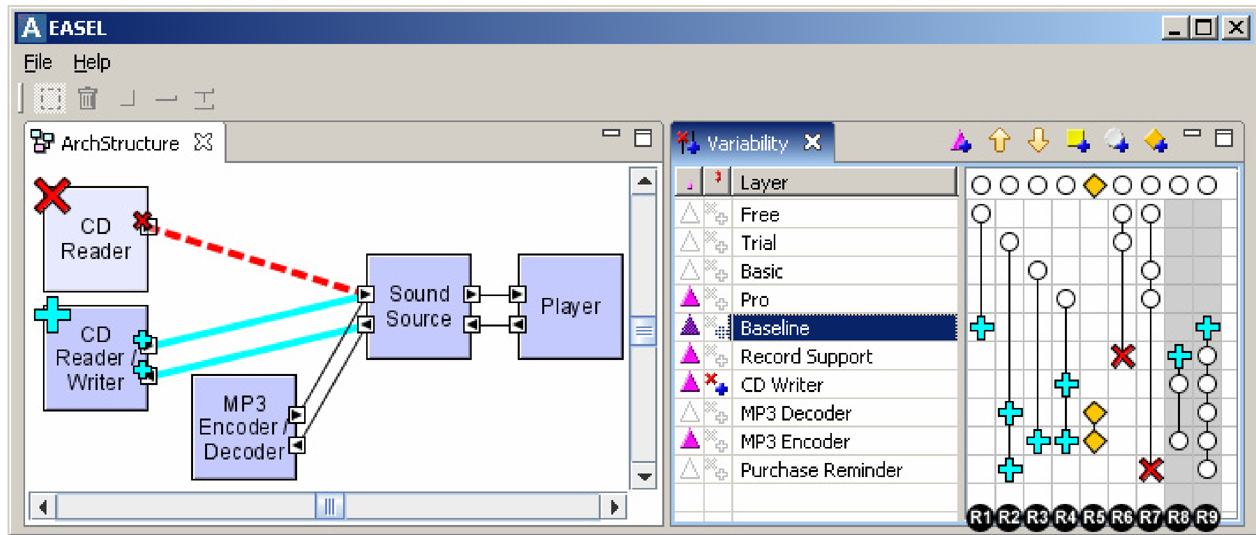


Figure 3. EASEL. Relationships are Labeled R1 to R9 for Reference.

specifying PLAs and a variability spreadsheet for managing change sets and relationships.

If none of the special features of EASEL are used, the drawing canvas of EASEL operates similarly to ArchStudio [12] or AcmeStudio [33], allowing an architect to define single architectures. But, the drawing canvas has additional behaviors that bring it into the realm of PLA modeling by implementing the concepts discussed in Section 4. First, it is actually a "layered canvas," with each layer representing a change set. The visible architecture, then, is constructed from the change sets selected by the first column of the variability spreadsheet. For example, the architecture shown in Figure 3 is a result of the *Pro*, *Baseline*, *Record Support*, *CD Writer*, and *MP3 Encoder* change sets.

To realize our approach, EASEL stores change sets, relationships, and the resulting architecture internally using XML. Thus, changes can be consistently recorded and reapplied using the XML element ID's and property names, which are globally unique and remain constant. It would be difficult to use textual diffs for this purpose, since they use adjacent lines of text to determine where a change is to be applied, and these can potentially be modified by other change sets. Still, and even with the specialized merging process, conflicts may occur when change sets modify the exact same element in an incompatible way, i.e., two change sets that each rename a component, but to a different name. In such cases, the order in which change sets are listed is used to resolve the conflict.

The second additional behavior is that each of the elements on the drawing canvas may be annotated to explicitly illustrate what specific changes are recorded. In Figure 3, the *CD Writer* change set is selected as such (see second column of the variability spreadsheet), annotating the *CD Reader / Writer* component, its interfaces, and its links with a "+" to show that these elements are added, and the *CD Reader* component, its interface, and its link with an "×" to show that these elements are removed. Elements that are not a part of the final result, but that are changed by a selected change set, are drawn using lighter, transparent colors. Selecting multiple change sets results in annotations that would result from logically combining them.

The third and final additional behavior lies in how an architect populates the contents of change set. In EASEL, modifications are incorporated into the change set currently selected for editing. In Figure 3, any addition or removal made by an architect at this time would be incorporated into the *Baseline* change set, as highlighted. This allows an architect to revisit and update a change set as easily as creating a new one.

The variability spreadsheet shown on the right hand side of Figure 3 provides an architect with a graphical representation through which they manage relationships. Rows represent change sets and the columns relationships. The following symbols are used:

- A *circle* is a source of an *or* relationship (e.g., A in "if A *or* B are included, then C must be also").
- A *square* is a source of an *and* relationship (e.g., A in "if A *and* B are included, then C must be also").
- A *slash* superimposed over a circle or square represents a source negation (e.g., B in "if A is included and B is *not* included, then C must be included").
- A *plus* is a change set implied by a relationship (e.g., C in "if A is included, then C must be *included*").
- An *X* is a change set excluded by a relationship (e.g., C in "if A is included, then C must be *excluded*").

- A *diamond* represents a variant in a variant relationship. The minimum and maximum number of variants that may be selected concurrently is viewable and editable using context menus.

Returning to the example in Figure 3, the way to read some of the relationships, then, is as follows:

R2. The *Trial* change set is logically composed of the *MP3 Decoder* and *Purchase Reminder* change sets (i.e., if the *Trial* change set is included, then the *MP3 Decoder* and *Purchase Reminder* change sets must also be included).

R5. The *MP3 Decoder* and *MP3 Encoder* change sets are variants of each other; only one can be included at a time.

R6. If either the *Free* or *Trial* change set is included, then the *Record Support* change set should not be.

Note that the composition relationship for the *Trial* product architecture (R2) appears to be incomplete; it is missing the *Baseline* change set. This, however, is covered by *R9*, which states that the *Baseline* change set must be included when any of the specified change sets below it are included; it therefore does not have to be repeated for the *Trial* feature composition.

Finally, EASEL automatically detects structural dependencies and conflicts between change sets, which are added to the variability spreadsheet with a darker background. These act as critics [29], and disappear when no longer applicable. Additionally, EASEL does not prevent an architect from combining change sets that violate relationships. However, it does inform them of violated relationships by highlighting them.

6. Evaluation

We evaluated our approach in three ways. First, we compared the compactness of models produced using our environment, EASEL, with those produced by a representative of traditional, extensional environments, Ménage [11]. We then discuss the expressiveness of our approach. Finally, we comment on its usability by making four typical changes to a PLA, as performed using both approaches.

To compare model compactness, we modeled three PLAs in both EASEL and Ménage: (1) the motivating example in Section 3, (2) the entertainment system used to evaluate Ménage [11], and (3) an actual system, ArchStudio [12]. Table 3 presents these results, showing that our approach resulted in more compact, and significantly less complex, PLAs. Specifically, the excessive number of Boolean guards were replaced with fewer change sets and relationships. This is exactly what we hoped for, giving PLA designers a representation that is significantly less burdensome to maintain.

Table 3. Modeling compactness.

	Change Sets		Relationships			Traditional		
	Compositions	Features	Compositions	Compatibilities	Structural	Product Definitions	Boolean Guards	Variables
Example in Section 3	4	6	4	3	2	4	26	3
Entertainment System	0	19	0	4	5	0	114	18
ArchStudio	1	17	1	2	1	1	175	15

Our approach must be adequately expressive, able to model intensionally all PLAs that can be modeled extensionally using variation points. Fortunately, the field of configuration management has already shown a general equivalence in expressiveness of extensional and intentional approaches [8].

It should also be possible to express any PLA that is modeled in a feature model or through a propositional formula. In response, we observe that these can both be rewritten in conjunctive normal form (CNF) and therefore captured in EASEL as a series of *or relationships*. Of course, the *and relationship* and *variant relationship* of EASEL enable much more compact expressions, just as the mandatory, optional, and alternative relationships of feature models and the various analogous operators of propositional formulas.

Finally, we believe that it is in the usability of our approach that its greatest benefits are seen. A full study is ultimately necessary to make this claim. However, we are encouraged by the initial evidence gained from modeling the three PLAs using both approaches. First, the modeling of entire systems results in compact notations. But second, we informally found that our approach was easier to use for changing PLAs as well. This was confirmed when we performed four typical changes, which included: (1) creating a new product architecture out of existing features, (2) adding a new optional feature, (3) adding this feature to some products, and (4) updating this feature for a few of the products. Though subjective and personal, combined with the advantages of compactness, explicitness, and equal expressiveness, we believe that we have sufficiently shown EASEL to be a new and promising alternative for PLA modeling.

7. Related Work

Our work is related to several different efforts. AHEAD, an example of feature-oriented programming

[28], uses a compositional expression language that can be generically mapped onto rules that govern the underlying composition process [4]. AHEAD has been applied to the domain of product line architectures. However, AHEAD is only "additive" and prohibits element deletion, which is essential to our approach.

Likewise, our work is related to multi-dimensional separation of concerns [35]. Change sets are close to modules, and change set selection with our merge algorithm, is close to a hypermodule. However, most research in this area has focused on the source code level (e.g., Hyper/J [25]) and only on adding functionality.

Aspect-oriented programming [14] is also related to our work. By their very nature, change sets are close to aspects in their compositional capabilities; in fact, one could probably support the other and vice versa. However, compared to aspects, our work has taken the generic change set idea, modified it to support PLAs, and built enhanced support through detailed relationships.

COVAMOF [32] provides perhaps the best support for modeling variability in the extensional approach. It certainly has the most comprehensive set of language concepts with which an architect can model PLAs. Because our approach elevates variability to a level equal to architecture, all COVAMOF language concepts can be subsumed by just two concepts, change sets and relationships. Other extensional approaches to modeling PLAs [2, 23, 32, 34], including our own previous work on Ménage [11], are similarly subsumed.

Finally, the field of feature engineering has worked on modeling features and their relationships for a number of years [5, 13]. These models usually stay at the conceptual level, although some exceptions exist in which attempts are made to map the conceptual features to concrete implementations [36]. Of most interest here is the work by Lago and Van Vliet, who built an extensive mapping from conceptual features to architectural components [16]. We believe our work complements their work by, for the first time, providing a PLA approach that reduces the complexity of such mappings. In providing a modeling mechanism in which features and variability can be naturally expressed, the mapping becomes one-to-one.

8. Conclusions

This paper contributes a new approach to modeling product line architectures that leverages change sets and relationships to bring together conceptual variability and the actual modeling of this variability. By adopting an intensional modeling approach, variability is placed at the same level of importance as architecture and the resulting modeling concepts of change sets and relationships support architects in modeling their PLA in a much more concise and natural sense. As implemented in EASEL, the approach supports the modeling of overall PLAs and the composition of individual product architectures out of these PLAs. It particularly enables exploration of different compositions by informing the user of whether these compositions adhere to the rules of the relationships.

Our future work involves several different strands. First, we wish to enhance the usability of EASEL. For instance, it should be easy to split or combine change sets, or move elements from one change set to another. Second, we recognize that for a large PLA, the set of relationships can grow large; it is incumbent to address this problem. We will explore filters that display only relevant relationships and attempt to group and summarize them. Third, we wish to enhance EASEL's automatic analysis of structural relationships to deduce "unless" change sets, which resolve structural conflicts. For instance, if one change set adds a link to an interface removed by another change set, it structurally conflicts with that change set *unless* a third change set is included that removes the offending link. Automatically finding this change set among the existing change sets would help users develop consistent product architectures. Fourth, at a more conceptual level, we wish to explore how our approach integrates with substructure (i.e., hierarchical composition of components and connectors), strong typing (which would cut across change sets), and versioning (both at the level of the entire PLA and individual types). Finally, we wish to fully evaluate the usability of EASEL. These are non-trivial cognitive implications of an approach like this that we ignored to first accomplish our basic goal of representing PLAs as change sets and relationships.

9. Acknowledgements

Effort partially funded by the National Science Foundation under grant number CCR-0093489, DUE-0536203, and IIS-0205724.

10. References

[1] Allen, R. and Garlan, D. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*. 6(3), p. 213-249, Jul., 1997.

[2] Asikainen, T., Soininen, T., et al. A Koala-Based Approach for Modelling and Deploying Configurable Software Product Families. *5th International Workshop on Product-Family Engineering*. p. 225-249, Siena, Italy, Nov. 4-6, 2003.

[3] Batory, D. and Geraci, B.J. Composition Validation and Subjectivity in GenVoca Generators. *IEEE Transactions on Software Engineering*. 23(2), p. 67-82, Feb., 1997.

[4] Batory, D. Feature Models, Grammars, and Propositional Formulas. *9th International Software Product Line Conference*. p. 7-20, Rennes, France, Sept., 2005.

[5] Beuche, D., Papajewski, H., et al. Variability Management with Feature Models. *1st Workshop on Software Variability Management*. p. 72-83, Groningen, The Netherlands, Feb. 13-14, 2003.

[6] Bosch, J. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Addison-Wesley Professional: Reading, USA, 2000.

[7] Clements, P. and Northrop, L. *Software Product Lines: Practices and Patterns*. Addison-Wesley: New York, NY, 2002.

[8] Conradi, R. and Westfechtel, B. Version Models for Software Configuration Management. *ACM Computing Surveys*. 30(2), p. 232-282, Jun., 1998.

[9] Dashofy, E., van der Hoek, A., et al. A Comprehensive Approach for the Development of Modular Software Architecture Description Languages. *ACM Transactions on Software Engineering and Methodology*. 14(2), p. 199-245, Apr., 2005.

[10] Estublier, J., Leblang, D.B., et al. Impact of the Research Community on the Field of Software Configuration Management. *ACM Transactions on Software Engineering Methodology*. 14(4), p. 383-430, Oct., 2005.

[11] Garg, A., Critchlow, M., et al. An Environment for Managing Evolving Product Line Architectures. *IEEE International Conference on Software Maintenance*. p. 358-367, Amsterdam, The Netherlands, Sept., 2003.

[12] Institute for Software Research. *ArchStudio, An Architecture-based Development Environment*. http://www.isr.uci.edu/projects/archstudio/, UC, Irvine.

[13] Kang, K.C., Cohen, S.G., et al. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Software Engineering Institute, Technical Report CMU/SEI-90-TR-21, Nov., 1990.

[14] Kiczales, G., Lamping, J., et al. Aspect-Oriented Programming. *11th European Conference on Object-Oriented Programming*. p. 220-242, Jyväskylä, Finland, Jun. 9-13, 1997.

[15] Krueger, C.W. Variation Management for Software Production Lines. *2nd International Software Product Line Conference*. p. 37-48, San Diego, USA, Aug. 19-22, 2002.

[16] Lago, P., Niemelä, E., et al. Tool Support for Traceable Product Evolution. *8th European Conference on Software Maintenance and Reengineering*. p. 261-269, Tampere, Finland, Mar. 24-26, 2004.

[17] Luckham, D.C. and Vera, J. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*. 21(9), p. 717-734, Sept., 1995.

[18] Magee, J. and Kramer, J. Dynamic Structure in Software Architectures. *4th International Symposium on the Foundations of Software Engineering*. p. 3-14. San Francisco, USA, Oct. 16-18, 1996.

[19] Medvidovic, N., Rosenblum, D.S., et al. A Language and Environment for Architecture-Based Software Development and Evolution. *21st International Conference on Software Engineering*. p. 44-53. Los Angeles, USA, May 16-22, 1999.

[20] Medvidovic, N. and Taylor, R.N. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*. 26(1), p. 70-93, Jan., 2000.

[21] Mens, T. A State-of-the-Art Survey on Software Merging. *IEEE Transactions on Software Engineering*. 28(5), p. 449-462, May, 2002.

[22] Northrop, L. Reuse That Pays: ICSE Keynote Presentation. *23rd International Conference on Software Engineering*. p. 667, Toronto, Canada, May 12-19, 2001.

[23] Ommering, R.v., Linden, F.v.d., et al. The Koala Component Model for Consumer Electronics Software. *IEEE Computer*. 33(3), p. 78-85, Mar., 2000.

[24] Ommering, R.v. Building Product Populations with Software Components. *24th International Conference on Software Engineering*. p. 255-265, Orlando, USA, May 19-25, 2002.

[25] Ossher, H. and Tarr, P. Hyper/J™: Multi-Dimensional Separation of Concerns for Java™. *23rd International Conference on Software Engineering*. p. 729-730, Toronto, Canada, May 12-19, 2001.

[26] Perry, D.E. and Wolf, A.L. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*. 17(4), p. 40-52, Oct., 1992.

[27] Pohl, K., Böckle, G., et al. *Software Product Line Engineering: Foundations, Principles and Techniques*. 1 ed. 468 pgs., Springer: New York, NY, 2005.

[28] Prehofer, C. Feature-Oriented Programming: A Fresh Look at Objects. *11th European Conference on Object-Oriented Programming*. p. 419-443, Jyväskylä, Finland, Jun. 9-13, 1997.

[29] Robbins, J. and Redmiles, D. Software Architecture Critics in the Argo Design Environment. *International Conference on Intelligent User Interfaces*. p. 47-60, San Francisco, USA, Jan. 6-9, 1998.

[30] Roshandel, R., van der Hoek, A., et al. Mae - A System Model and Environment for Managing Architectural Evolution. *ACM Transactions on Software Engineering and Methodology*. 13(2), p. 240-276, Apr., 2004.

[31] Shaw, M., DeLine, R., et al. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*. 21(4), p. 314-335, Apr., 1995.

[32] Sinnema, M., Deelstra, S., et al. COVAMOF: A Framework for Modeling Variability in Software Product Families. *3rd International Software Product Lines Conference*. p. 197-213, Boston, USA, Aug., 2004.

[33] Software Engineering Institute. *ACMEStudio*. http://www.cs.cmu.edu/~acme/AcmeStudio, Carnegie Mellon University.

[34] Svahnberg, M., van Gurp, J., et al. A Taxonomy of Variability Realization Techniques. *Software Practice and Experience*. 35(8), p. 705-754, Jul., 2005.

[35] Tarr, P., Ossher, H., et al. N Degrees of Separation: Multi-dimensional Separation of Concerns. *21st International Conference on Software Engineering*. p. 107-119, Los Angeles, USA, May 16-22, 1999.

[36] Turner, C.R., Fuggetta, A., et al. A Conceptual Basis for Feature Engineering. *Journal of Systems and Software*. 49(1), p. 3-15, Dec. 15, 1999.