

Using Service Utilization Metrics to Assess the Structure of Product Line Architectures

André van der Hoek

*Institute for Software Research
University of California, Irvine
Irvine, CA 92697 USA
andre@ics.uci.edu*

Ebru Dincel

*Computer Science Department
University of Southern California
Los Angeles, CA 90089 USA
edincel@usc.edu*

Nenad Medvidović

*Computer Science Department
University of Southern California
Los Angeles, CA 90089 USA
nenom@usc.edu*

Abstract

Metrics have long been used to measure and evaluate software products and processes. Many metrics have been developed that have led to different degrees of success. Software architecture is a discipline in which few metrics have been applied, a surprising fact given the critical role of software architecture in software development. Software product line architectures represent one area of software architecture in which we believe metrics can be of especially great use. The critical importance of the structure defined by a product line architecture requires that its properties be meaningfully assessed and that informed architectural decisions be made to guide its evolution. To begin addressing this issue, we have developed a class of closely related metrics that specifically target product line architectures. The metrics are based on the concept of service utilization and explicitly take into account the context in which individual architectural elements are placed. In this paper, we define the metrics, illustrate their use, and evaluate their strengths and weaknesses through their application on three example product line architectures.

1. Introduction

Considering the increasing importance of product line architectures (PLAs) in today's software development [5], a reasonable question one may want to ask is "What is the overall quality of my PLA?" Unfortunately, one is left with very few means of answering this question. Although experts may be able to provide an answer based on their experience, such an answer is generally not easily attained. As a result, the presence of fundamental flaws in a PLA sometimes may not be revealed until late in its use.

The field of software architecture, and product line architecture in particular, is not alone in asking a question such as the above. Throughout the history of software engineering, similar questions have been asked in many of its different domains. To answer these questions, the use of metrics has proven helpful [11]. Typically focused on one aspect of system quality (e.g., size, complexity, reliability),

metrics help in learning from past experiences, evaluating present situations, and sometimes predicting future trends.

The work presented in this paper addresses the question of whether metrics can help in assessing one particular quality of PLAs, namely their structural soundness. Although other qualities are certainly important as well, our focus on structure is guided by the critical role it plays in a PLA. The structure created by the components constituting a PLA forms the central abstraction upon which all other information regarding a PLA is based. Precisely managing all aspects of such architectural structure, then, is one of the main activities in product line engineering [5].

Metrics are available that assess a number of structural aspects of software systems (e.g., fan-in/fan-out [15], cyclomatic complexity [22], cohesion [25], coupling [16]), but their direct application to PLAs is difficult. This is caused by two characteristics inherent to a PLA's structure.

- **Optionality.** Whereas some components form the core of a PLA and are present in all of its product instances, other components may or may not be included in a final product depending on, for example, customer preference. These components are normally not critical in nature and simply provide some auxiliary functionality.
- **Variability.** At the heart of any PLA are variation points [5], which capture logical alternatives as a set of components from which only one can be selected for incorporation in a specific product instance. These components typically provide similar functionality, but have different properties (e.g., platform, performance).

Although both of these characteristics are conceptually simple in nature, they violate the critical assumption underlying existing metrics, namely that the system under measurement has a single, fixed structure.

This paper reports on our initial experiences in designing and using a novel class of closely related metrics that, while based on existing structural metrics, specifically address the PLA characteristics of optionality and variability. Rooted in the concept of *service utilization*, which measures the percentage of the provided or required services of a component (i.e., resources such as public methods or func-

tions) that are used in an architectural configuration, our metrics are capable of highlighting potential structural deficiencies in PLAs and assisting an architect in evaluating different solutions that may correct such problems. An important design decision underlying the metrics is that they do not provide an absolute measure of structural “goodness” or “badness”, but rather highlight anomalies based on relative values. While requiring human interpretation of the results, it makes the metrics much more informative.

It should be noted that our metrics measure only one dimension along which a PLA could be improved. Other factors (such as cost, performance, or reusability) and other methods (such as design reviews) provide additional input to a designer faced with the challenge of maintaining a PLA. A designer indeed may choose a less-than-optimal structural solution based upon these other factors. Nonetheless, our metrics provide one additional (and previously not available) input datum in this process: a measure of the structural soundness of a PLA. This is a critical piece of information, since the structure of a PLA forms the basis for the individual products that populate it.

To evaluate the strengths and weaknesses of the metrics, we apply them to three example PLAs. The first is a reverse engineered representation of an implicit PLA used in a course at the University of Southern California (USC); the second is a new PLA for a troops deployment system designed by a different team in our research group; and the third is an externally provided case study of a real-world library PLA. Initial results are encouraging: using our metrics, we were able to identify several structural problems in each of the PLAs and evaluate suggested alternative solutions for those problems. At the same time, the evaluation naturally exposed some weaknesses in the metrics, suggesting that further research is needed to make them more accurate in diagnosing potential problems.

2. Background

The disciplines of metrics and software architecture form the basis for the work described in this paper. Below, we summarize relevant contributions in each discipline.

2.1 Software metrics

Software metrics have been studied extensively over the past several decades and many metrics have been proposed [11, 19]. These metrics are classified into two general categories: *process metrics* to assess the state and resource usage of the development process and *product metrics* to assess the state and complexity of the resulting artifacts (e.g., requirements, designs, code). While the two categories are complementary and usually used in tandem, the remainder of the discussion focuses on product metrics.

Product metrics are categorized based on whether they measure external or internal properties of software systems.

External product metrics assess properties that are visible to users of a software system, such as complexity of functionality [8] and resource usage [10]. *Internal product metrics* measure properties visible only to the developers of a software system. Internal product metrics are further subdivided into *cognitive complexity metrics* and *structural complexity metrics*. Cognitive complexity metrics measure the effort required by developers to understand a system. They aim at discovering the cause of the complexity, which requires understanding human mental processes and details of the software system under development [3, 7].

Structural complexity metrics use the interactions within and among modules to measure a system’s complexity. One of the oldest and most commonly used structural complexity metrics is the number of lines of code [10, 18]. Unfortunately, this metric suffers from lack of a standard definition, indifference to redundancy and reuse, and sensitivity to programming language. Size/complexity metrics [26], software science metrics [14], and function points [1] are more sophisticated but at the same time more complex to use. These static metrics are complemented by graph-based *control flow metrics*, such as cyclomatic complexity [22], which map the dynamic control flow of a program onto a connected graph. A hybrid metric makes use of both static and dynamic information about a system. For example, weighted methods per class [6] sums the complexity of the methods in a class, where method complexity is determined using the cyclomatic complexity metric.

The emergence of modular and object-oriented systems led to the introduction of such metrics as number of children [6] and depth of inheritance tree [6], as well as adaptation of existing metrics that were originally proposed for procedural systems (*cohesion* [25] and *coupling* [16]). Cohesion is the degree to which a module works well as a unit, while coupling is the degree to which modules are interdependent. High cohesion and low coupling are considered indications of a good design. An inverse metric, lack of cohesion [6], is the typical metric used to assess the structural cohesion of modules. Coupling is typically measured with *inter-module metrics*. Information flow metrics [15], for example, use the measures of fan-in and fan-out to assess the strength of association between modules. Other widely used coupling metrics include response for a class [6] and coupling between objects [6].

2.2 Software architecture

Software architectures provide high-level abstractions for representing the structure, behavior, and key properties of a software system. These abstractions involve: (1) descriptions of the elements from which systems are built, (2) interactions among those elements, (3) patterns that guide their composition, and (4) constraints on those patterns [24]. In general, a system is defined as a set of *components*,

their interconnections (*connectors*), and the overall organization of the components and connectors (*configuration*).

An *architectural style* defines a vocabulary of component and connector types and a set of constraints on how instances of these types may be combined in a system [13, 29]. Styles do not prescribe the actual components and connectors that may be used. For example, an architecture adhering to the client-server style [29] may in fact be a set of Web clients and servers, a database, or a banking system. Styles also influence architectural evolution by restricting the possible changes an architect is allowed to make.

Whereas a “regular” architecture defines the structure of a single product, a product line architecture (PLA) defines the common architecture for a set of related products [5]. A PLA explicitly specifies: (1) elements that are present in all products, (2) elements that are optional, and (3) variation points among products [31, 32, 33]. Particular product instances are selected by choosing the desired optional components and selecting one element per variation point. Perry [27] outlined the space of possibilities for modeling PLAs and observed that a PLA modeling technique must be both generic enough to encompass all members of a product line and specific enough to provide developers with adequate support for instantiating and implementing individual products. While it is technically possible to reuse styles for this purpose [29], experience has shown a need for higher-level support in terms of explicit facilities for modeling optionality and variability [33].

To date, many *architecture description languages* (ADLs) have been proposed to aid architecture-based development [24]. ADLs provide formal notations to describe software systems and are usually accompanied by various tools for parsing, analysis, simulation, and code generation of the modeled systems. Examples of ADLs include C2SADEL [23], Darwin [21], Rapide [20], UniCon [28], and Wright [2]. A number of these ADLs also provide extensive support for modeling *behaviors* and *constraints* on the properties of components and connectors [24], which can be leveraged to ensure the consistency of an architecture (e.g., by establishing conformance between the services of interacting components). Unfortunately, with the exception of Mae [32], Koala [33], and GenVoca [9] few existing ADLs explicitly support the specification of all aspects of PLAs.

3. Service utilization metrics

This section presents the metrics we have defined to assess the structural quality of a PLA. The metrics are based upon the concept of *service utilization*. Under the term service, we include such things as public methods or functions, directly accessible data structures, and any other kind of publicly accessible resource one may be able to express in an ADL. We use the generic concept of a service to avoid tying our metrics to a specific ADL. Instead, given

a proper mapping of ADL constructs to our service concept, the metrics can be applied to any ADL in which a PLA can be modeled. As many ADLs tend to model component interaction using methods [24], such a mapping is trivial in most cases. Even for an event-based ADL [20], a mapping can be conceived in which provided services are the events to which a component will react and required services are the events emitted by a component.

A match between a provided and required service is defined as a match according to the specific rules of an ADL [34]. A match of public methods, for instance, can be based on any of four levels: name, interface, behavior, or protocol [23]. Name matching only requires names of services to match; interface matching requires parameters and their types to match as well; behavior matching requires service semantics (described by, for example, pre- and postconditions) to match; and protocol matching requires the order of invocations to match. In the remainder of this paper, we use name matching of interface methods for the purposes of our examples, but it should be understood that the metrics are just as readily calculated for more complicated types of service conformance. All that is needed in such cases is a proper matching algorithm—something already provided by existing ADLs and their toolsets [24].

Our metrics are designed to operate on PLA descriptions as described in some ADL. While we believe that, in principle, there are no obstacles to applying them on implementation architectures (i.e., as embodied in actual code), we have not done so as of yet. Note, however, that our metrics are equally applicable during the initial design and maintenance of a PLA. Information on structural quality is needed at both times, and can be provided by our metrics as long as an architectural description is available.

Below, we define our metrics in an incremental fashion. We first define the basic service utilization metrics as they apply to an individual architecture with a fixed configuration. Then, we refine the metrics to address the unique characteristics of PLAs by extending them to support optionality and variability. We conclude the section with a critical look at the metrics.

3.1 Individual components

The basic building blocks of our metrics are the concepts of *Provided Service Utilization (PSU)* and *Required Service Utilization (RSU)*, defined on a per-component basis as follows:

$$PSU_X = \frac{P_{actual}}{P_{total}} \quad RSU_X = \frac{R_{actual}}{R_{total}}$$

In these formulas, P_{actual} is the number of services provided by component X that is actually used by other components in the architecture and P_{total} is the total number of services provided by component X. R_{actual} and R_{total} have analogous meanings for required services. While it is clear

that P_{actual} may be less than P_{total} , the fact that R_{actual} can be less than R_{total} is counter-intuitive. Required services normally must be met for a system to properly operate. In a PLA, however, it is common to design components with required services that, if not provided by other components, will not lead to system failure. When matched, however, these services enhance overall system functionality [33]. For instance, a numerical component may “require” a service through which it reports statistical information regarding its progress. Connecting this required service to a component that gathers and presents such statistical information would result in a system that is more informative to its users. Not connecting the required service, however, does not need to lead to system failure since the core functionality of the system is still in tact. In essence, the fulfillment of these required services is optional.¹ Of note is that a number of well-known techniques can be used to realize and implement this behavior, including dynamic linking, event-based architectures [20] and the advanced compilation techniques employed by Koala [33].

PSU and RSU are context-dependent. A given component will have different PSU and RSU values depending on the particular architectural configuration of which it is a part. Consider the example architecture shown in Figure 1a. Component B provides services to component A1 and requires services from component C1. Of the three services provided by component B ($foo()$, $bar()$, and $foobar()$), only one is actually used by component A1, namely $foo()$. Therefore, PSU_B is $1/3$. Because component C1 provides only one of the three services required by component B, RSU_B is $1/3$ as well. PSU_B and RSU_B are low compared to their counterparts in the architecture shown in Figure 1b. There, all of the services provided by component B are used by component A2, leading to a PSU_B of 1. Similarly, RSU_B in Figure 1b is $2/3$ since two out of the three services required by component B are provided by component C2.

Although absolute PSU and RSU values may indicate potential problems (e.g., a component with RSU close to 0 may not function well within an architecture; a PSU close to 0 signifies a lot of extra functionality that is not used by other components), relative values are even more useful in comparing two related, but different architectures. PSU_B and RSU_B , for example, show that component B is a much better fit in the architecture of Figure 1b than in the architecture shown in Figure 1a.

We note once more that many other considerations besides structural soundness—such as cost, performance, or reusability—influence architects in designing a particular PLA that may or may not be optimal in structure. PSU and

RSU values are useful in assessing the chosen structure, but they are not the sole determinants. For instance, while a low PSU may indicate a “bloated” component, the benefits of reusing such a component may outweigh the desire to maintain a “lean” PLA.

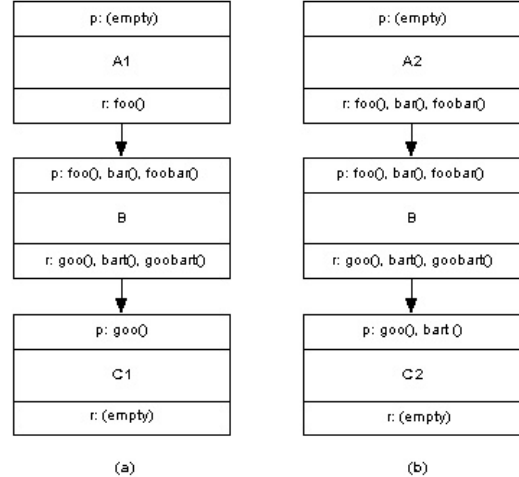


Figure 1. Two example architectures.

To determine the role of each component in an architecture, PSU and RSU of all components should be calculated and analyzed. In doing so for the components in Figure 1, we observe that some of those components have an empty set of provided or required services. The PSU and RSU of such a set remains undefined, indicating a need for attention to ensure the empty set of services is intentional.

PSU and RSU are unbiased with respect to the size of a component. High values can be achieved by both small and large components, as long as their provided and required services are actually used. Therefore, for example, a set of well-used small components is equivalent to one well-used large component.

3.2 Architectures as a whole

In addition to assessing each individual component, it is useful to analyze an architecture as a whole. To this end, the *Compound PSU (CPSU)* and *Compound RSU (CRSU)* are defined as follows:

$$CPSU = \frac{\sum_{i=1}^n P_{\text{actual}}^i}{\sum_{i=1}^n P_{\text{total}}^i} \quad CRSU = \frac{\sum_{i=1}^n R_{\text{actual}}^i}{\sum_{i=1}^n R_{\text{total}}^i}$$

P_{actual} , P_{total} , R_{actual} , and R_{total} are defined as in PSU and RSU, while n is the number of components in the architecture. In the examples of Figure 1a and Figure 1b, CPSUs are $1/2$ and 1 , respectively, whereas CRSUs are $1/2$ and $5/6$.

CPSU and CRSU are used to assess the internal cohesion of an architecture: CPSU and CRSU values close to 1

¹ Within the context of a product line architecture, of course, one would expect the required service to be fulfilled by a provided service in at least one product—otherwise the required service would be superfluous and represent obsolete functionality.

signify self-contained, fully functional architectures. In our example, the values of CPSU and CRSU demonstrate that the architecture in Figure 1b is more cohesive than the architecture in Figure 1a (since no superfluous services are present and all but one of the required services are met).

Low CPSU or CRSU indicates an unbalanced architecture. In particular, low CPSU combined with high CRSU implies that the architecture is larger than it needs to be, i.e., its components provide more services than are actually needed for the system to accomplish its purpose. Conversely, high CPSU with low CRSU indicates an architecture with significantly degraded functionality: components must be added for all of the required services to be present.

These observations can only be interpreted as guidelines. It is quite possible to construct functioning architectures for which, for example, CPSU is low and CRSU is high. The importance of the measures does not lie in their absolute values. In comparing different, yet related solutions, the relative values of CPSU and CRSU are best used in assessing the structural quality of alternative architectures.

3.3 Optionality

Whereas the discussion in the previous two sections focused on defining our basic metrics in terms of “conventional” architectures, we now turn our attention to applying the metrics in the context of PLAs. The first challenge put forth by PLAs is optionality, which allows a particular product to either include or exclude certain non-critical components. We look at the problems posed by optionality from the perspective of the: (1) optional component itself, (2) other, non-optional components, and (3) overall PLA.

3.3.1 Optional component perspective. PSU and RSU as defined in Section 3.1 are sufficient for assessment purposes from the perspective of an optional component. In particular, calculating the PSU and RSU of an optional component is only relevant when it is actually included in a product instance. At that point, it can simply be treated as any other (non-optional) component. When an optional component is not included, it is not useful to analyze its PSU and RSU—the component is disconnected.

3.3.2 Impact on non-optional components. Inclusion of an optional component in a PLA poses a problem in calculating PSU and RSU of non-optional components. The question seems simple: “Should optional components be included or excluded in the calculation of PSU and RSU of other components?” In reality, however, the issue is more complex and the question is whether the current placement of the optional components in a PLA is structurally proper.

To answer this question, we calculate, per the formulas of Section 3.1, the *span* of PSU values and the *span* of RSU values for each non-optional component. Consider the ex-

ample in Figure 2, which shows the original architecture of Figure 1a as expanded by two optional components (indicated by the dashed boxes and arrows). To analyze the span of RSU_B , we calculate RSU_B in four instances of the PLA: no optional component included, D1 included, D2 included, and both D1 and D2 included. This leads to a span of the following four values for RSU_B : $1/3$, $2/3$, $2/3$, and 1. A graphical view of this span is shown below.



The inclusion of progressively more optional components should show a steady increase in the values contained in the PSU span of a non-optional component, indicating incremental use of its provided services. The RSU span of a non-optional component should show a similar pattern, indicating increased functionality as more optional components are included. While this is not an absolute rule, any deviation from this pattern may be a cause for concern (for example, it may signify optional components that are no longer compatible with the overall PLA) and should be examined by the architect.

Inclusion of an optional component never leads to lower values for PSU and RSU of other components, since existing connections are never taken away. Therefore, it is important to analyze the magnitude of increase in PSU or RSU brought forth by the inclusion of an optional component: the higher the increase, the better the structural fit and value added of an optional component in a PLA.

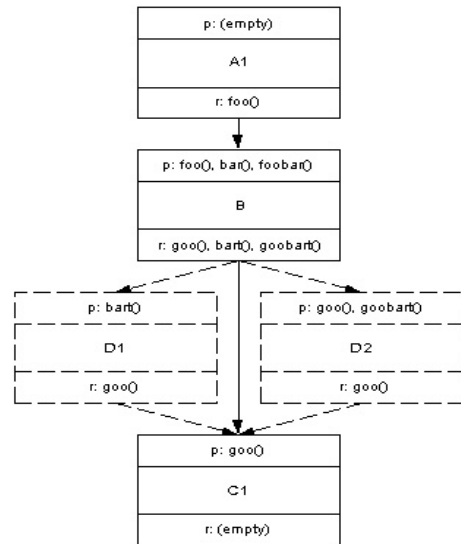


Figure 2. Example PLA with optional components.

We note that in the case of our example, an analysis of PSU and RSU of optional components D1 and D2 already would have revealed that each is a good fit in the PLA. An overall analysis of other components is still necessary for two reasons: (a) optional components may interact directly

with each other, making it impossible to assess the influence of a single optional component in isolation, and (b) a small optional component with high PSU and RSU may actually fulfill a lower number of services provided and required by other components than a large optional component with low PSU and RSU. In such a case, inclusion of the larger component (which, judging only by its own PSU and RSU, is not as good of a fit as the smaller component) results in a PLA that may be structurally better (as measured by the PSU and RSU of the other components). Clearly, an architect would need to examine such a case in more detail, but our metrics provide a way of highlighting these issues.

3.3.3 Impact on product line architecture. Many of the arguments regarding spans of PSU and RSU as discussed in Section 3.3.2 apply to the spans of CPSU and CRSU from the perspective of a PLA as a whole. Inclusion of progressively more optional components should result in spans containing steadily increasing CPSU and CRSU values. If such a pattern is not found, it may indicate a problem. For example, a CPSU span in which values actually decline as more optional components are included signifies that those optional components all have low PSU and do not fit very well with the remainder of the PLA.

3.4 Variability

The second challenge unique to PLAs is variability, the influence of which we examine again from the perspective of the: (1) variant component, (2) other components, and (3) overall PLA.

3.4.1 Variant component perspective. A variant component is a component that comprises several other components, termed *variants*, only one of which may be chosen at any time for incorporation in a product instance. To analyze the structural fit of a variant component, we compute the spans and averages of its PSU and RSU values. The resulting spans should be small (indicating a cohesive set of variants) while the average should be high (indicating that each variant is structurally a good fit with the remainder of the PLA). As an example, Figure 3 shows the architecture of Figure 1b as expanded by a variant component E (indicated by the shaded box) that consists of four variants. The RSU span of component E contains the values 1, 1, 1, and 1, and its RSU average is 1. From the perspective of required services, thus, variant component E is an ideal fit. The PSU span of component E, on the other hand, contains the values 1, 0, 1, and 0, and its average is 1/2. The PSU span is broad and its average is relatively low, indicating that, from the perspective of provided services, component E is probably not a good fit.

Outliers in a computed span represent variants that may require attention. For example, if the PSU of one variant is

significantly lower than the other PSU values in the span, it indicates a non-relevant variant that perhaps has to be removed altogether or merged with another variant. The converse, a variant with a significantly higher PSU, may require that variant to be split into a “basic” variant with a similar PSU value as the other variants and an optional (or even required) component that inherits the remaining functionality. This extra component can then be shared by all variants, raising the overall CPSU. The presence of multiple clusters of PSU or RSU values indicates similar kinds of restructuring opportunities.

3.4.2 Impact on non-variant components. The inclusion of a variant component represents a challenge in calculating PSUs and RSUs of other, non-variant components. To address this challenge, we analyze the span of PSUs and the span of RSUs of each non-variant component. These spans are complementary to those of the variant component and ideally show a similar pattern: a narrow profile with a high average. The RSU span of component B in Figure 3, for example, contains the following values: 1, 2/3, 1, and 2/3, and its average is 5/6.

The observations made in Section 3.3.2 regarding optional components also hold for variant components: inclusion of a variant component never leads to lower values of PSUs or RSUs for other components; analysis of other components is still necessary even if analysis of a variant component by itself does not reveal any structural problems; and the span is particularly useful when multiple variant components are present.

The interplay among options and variants (e.g., optional variant components and components that are connected to a set of variant and optional components) is a potentially difficult problem that, however, our metrics naturally address. Because we analyze both options and variants using spans of PSU and RSU values, we can perform a single analysis in which each point on a PSU or RSU span represents a particular selection of options and variants (e.g., a product). In such an analysis, it is important to identify and analyze patterns relating to using a particular option or variant. For example, if all products that include a particular option are significantly lower than other products on the span, this option may present a potential problem. Patterns related to combinations of options and/or variants must similarly be examined for potential problems.

3.4.3 Impact on product line architecture. To analyze variability at the level of PLAs, we use the span and average of CPSU and CRSU. Arguments analogous to those presented in Section 3.4.2 hold: the average CPSU and CRSU should stay high while the span remains narrow, indicating a cohesive, structurally sound PLA.

Because CPSU and CRSU provide a summarization, a low average or broad span often will not immediately help in determining the exact structural weakness in a PLA. Fur-

ther study, in the form of the analysis of individual PSU and RSU values, is then necessary. Nonetheless, the overall average is useful when comparing different, but related PLAs. If structural changes have been made, a higher average indicates overall structural improvement.

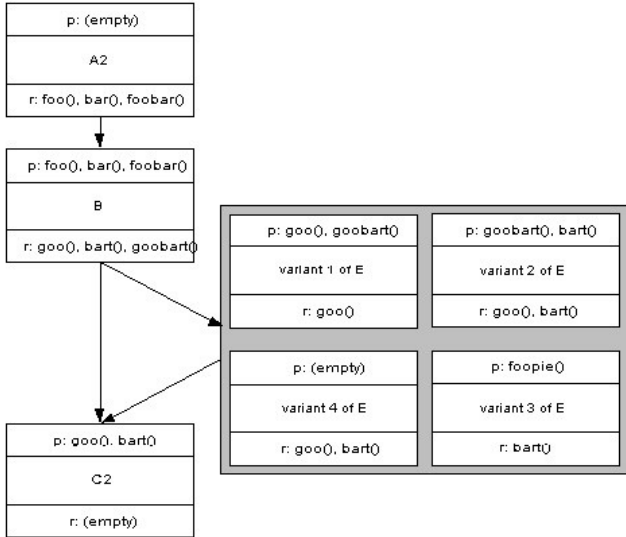


Figure 3. Example PLA with a variant component.

3.5 Critical evaluation

As is typically the case with software metrics, (C)PSU and (C)RSU are not perfect. They do not provide exact guidelines (e.g., they do not provide an absolute number for deciding whether or not a PLA is structurally sound); they cannot identify all structural weaknesses; and they may draw attention to minor structural weaknesses. Nonetheless, they represent a start at addressing the important issue of assessing structure in PLAs. Specifically, our metrics help in identifying potential structural problems and evaluating alternative solutions to those problems. The strength of the metrics is that they are incremental: the base metric of PSU and RSU used to assess an individual component is carried all the way through to an entire PLA that may contain many options and variants.

Although our metrics resemble such traditional metrics as fan-in/fan-out [15], the explicit definition of the (C)PSU and (C)RSU as ratios makes them well-suited for our specific purpose. The absolute numbers provided by fan-in/fan-out and other similar metrics [6, 16, 25] excel at assessing a single architecture, but fail when addressing multiple architectures since absolute numbers cannot be compared across different products without being placed in context (e.g., a fan-in of 6 for a component in one product cannot be compared to a fan-in of 8 for that same component in a different product unless one knows which other components are present in each respective product). Our metrics, thus, are modeled after other relative metrics (such as, for example, package visibility metrics [12]) to address the compli-

cations of multiple products, but uniquely focus their relative nature to address optionality and variability—complications that are not addressed by traditional metrics. (Even refactoring metrics [30], which help in restructuring object-oriented classes, do not currently handle optionality or variability.)

A critical characteristic of our metrics is that they are unbiased: they do not favor large or small components; they do not favor large or small sets of services; and they do not favor options or variants. Instead, the metrics rely solely on analyzing percentages of service utilization, and in doing so guide an architect towards structurally sound PLAs.

Calculating our metrics and performing analyses on the results is a repetitive and time-consuming effort. The presence of a large number of options and variants could theoretically result in spans that contain hundreds or even thousands of points. We observe, however, that calculations should be performed on a per-product basis. Rather than arbitrarily combining options and variants, an architect should iterate over each known product in the product line and calculate the metrics for each of those products. Not only does this reduce the number of points on a span to a maximum of the number of products in the product line, but it also ensures that dependencies among options and variants (e.g., inclusion of one option requires inclusion of another option) and known incompatibilities (e.g., selection of one particular variant prevents inclusion of a particular option) are taken into account. Each product specification, after all, has to be consistent.

Additionally, we observe that the metrics are amenable to the creation and application of an automated tool. We, in fact, have built a preliminary version of such a tool, which we used to calculate the metrics for the example product line architectures discussed in the next section.

4. Experiences

To examine the utility of our metrics, we have applied them in assessing the structure of three PLAs. The first is a reverse engineered representation of an implicit PLA used in a course at the University of Southern California (USC); the second is a PLA for a troops deployment system designed by another team in our research group; and the third is an externally provided case study of a real-world library PLA. Below, we summarize our experience in using our metrics on each of the three PLAs. Due to space constraints, we can only highlight representative examples of how we applied our metrics. Full details of each study can be found in a series of three technical reports at <http://sunset.usc.edu/publications/TECHRPTS/2002/>.

4.1 Digital library projects

CS577 is a two-semester project course for computer science graduate students at USC. Assigned teams develop

a variety of applications for a real client, the USC Information Sciences Division. A unique feature of the course is its focus on the use of product lines: applications build upon one of a select few semi-standard architectures that already contain COTS components serving as reusable building blocks. To date, about sixty applications have been developed as part of the course, five of which constitute the digital library PLA we discuss in this section.

While CS577 focuses on product lines, it does so in an informal manner. Graphical depictions of the various PLAs are available, but explicit architectural descriptions are lacking. To examine the structure of the digital library PLA, we therefore reverse engineered its description by studying project documentation, interviewing course instructors, and combining individual architectures into one single PLA. In this process, we did not always have access to the complete interfaces of the COTS components used. In such cases, we provided approximations of those components' interfaces in order to still be able to apply our metrics. The approximations were "best-case" and assume that all provided and required services are present in those components. As such, we did not introduce any artificial problems in the PLA and the PSU and RSU spans provided below represent a theoretical upper bound on the actual PSU and RSU spans; the actual spans likely will be lower.

Figure 4 shows an excerpt of the PSU and RSU spans we calculated for each of the components in the PLA. Most spans exhibit a set of high values similar to the spans for the SEARCH MANAGER and REPOSITORY components, indicating that the majority of the PLA has good structural quality. Note that some of the spans have fewer than five values, since the respective provided or required interfaces are only connected to other components in a few of the products.

Figure 4 also indicates the presence of some potential problems. The first is highlighted by the RSU span of the ADMIN INTERFACE component, which shows that in one product one fourth of the required services of this component are not met by the rest of the PLA. A further investigation reveals that ADMIN INTERFACE is a variant component and that one of the variants requires certain account management services. To address this problem, we examined several alternatives, including removing the account management requirements from the variant, adding an optional account management component, and adding a required account management component. Calculating our metrics for each case reveals that the second option leads to RSU values of 1 for all variants of ADMIN INTERFACE. Intuitively, this is indeed the best structural solution since it matches up services only as needed.

A second potential problem is highlighted by the PSU span of the component MEDIA UTILITIES. The span is wide and the individual values are very low. A further investigation reveals that the component exposes two classes of services: (1) displaying an image and (2) manipulating an image. Moreover, different products always use one class of

services only, never both. To address the resulting unnecessary bloat of the individual products, we considered three alternatives: (1) split the component into two variants, (2) design a customized variant for each product, and (3) keep the component unchanged. In this case, even though our assessment of the first and second options indicated 25-30% increases in average CPSU values, it turns out that the third option is most preferable. The reason is simple: while our metrics correctly indicate the potential for structural improvements, actually carrying out these improvements would be impossible since MEDIA UTILITIES is a COTS component for which source code is not available. Since redeveloping the component from scratch is costly, reuse considerations outweigh structural quality in this case.

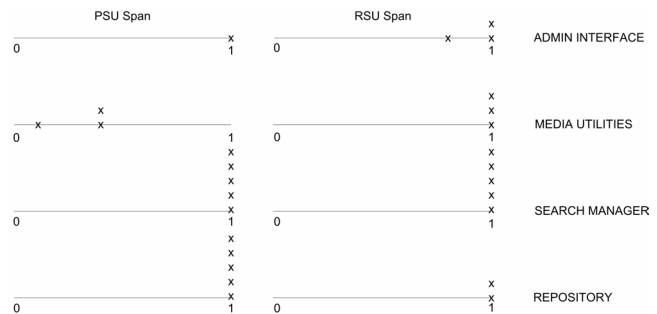


Figure 4. Selected spans for digital library PLA.

4.2 Troops deployment system

Troops Deployment System (TDS) is a PLA designed and implemented to support distributed deployment of personnel in situations such as natural disasters, search-and-rescue efforts, and military crises. Each of the nine products in the PLA is a distributed application that executes on multiple, heterogeneous, resource-constrained devices.

In applying our metrics to this PLA, the RSU spans of three components stood out (see Figure 5). The first component we examine is CLOCK, which stands out because it has no RSU span. Upon examination of the PLA, this is easily explained since CLOCK has no required services. The PSU span of CLOCK, however, is also interesting: while CLOCK is an optional component, the span shows that all of its provided services are used in each of the products. CLOCK, thus, can be turned into a core, non-optional component. After doing so, our metrics do not show an increase in the overall average CPSU or CRSU because no restructuring of services takes place. In this particular case, the metrics help in identifying a minor structural problem (an optional component that does not need to be optional), but cannot determine whether the proposed solution is structurally better.

The RSU span of the RENDERING_AGENT_HQ component shows RSU values distributed in two clusters of mediocre values. Further investigations reveal a mismatch be-

tween the required services of this component and the provided services of the DEPLOYMENT ADVISOR component. In fact, four variants of the DEPLOYMENT ADVISOR component exist, each of which provides a specific type of advice (fire-fighting, flood, military offensive, and military defensive).

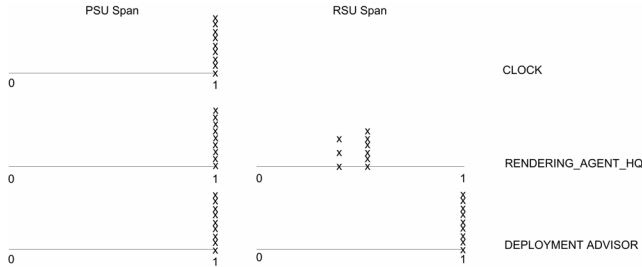


Figure 5. Selected spans for TDS PLA.

We initially examined two alternatives to remedy this problem: (1) merge all variants of DEPLOYMENT ADVISOR into one required component and (2) combine its four variants into two variants. Given the clusters in Figure 5, one would expect the second solution to be structurally most desired. In recalculating our metrics for each alternative, however, the first alternative displayed higher values. This illustrates both a drawback and a benefit of our metrics.

The drawback lies in the fact that the metrics do not differentiate the four variants of DEPLOYMENT ADVISOR, but instead place them into two clusters on the RSU span of RENDERING_AGENT_HQ. Further investigation reveals that this particular clustering is caused by the fact that the component RENDERING_AGENT_HQ also interacts with a number of other components. A closer look at those other components shows that they are the source of the clusters because they strictly differentiate between either military or civilian services. The services of the other components therefore visually dominate the RSU span of RENDERING_AGENT_HQ and the different variants of DEPLOYMENT ADVISOR cannot be distinguished.

The benefit of our metrics lies in their use for evaluating alternative solutions. In evaluating more than one approach, we were able to challenge our second solution and instead opted for the first alternative, resulting in a 7% increase in average CRSU.

Our final observation for the TDS example concerns iterative use of our metrics. Because, after these first two improvements, average CRSU still was not very high, we continued making enhancements, resulting in eventual increases of 33% (average CRSU) and 7% (average CPSU).

4.3 The library system

The Library System PLA was developed at Fraunhofer Institute in Germany [4]. The scope of the PLA encompasses city, university, and research libraries. The PLA is in actual use with a number of diverse customers and users.

In analyzing the library system PLA, we discovered a number of problems similar to those we discussed for the other PLAs, but here we focus on two problems that were unique to this PLA. The first regards the PSU span of the LOAN MANAGER component as shown in Figure 6. An examination reveals that the low values are caused by the fact that some of the provided services are never used in any of the products. The obvious solution is to remove the extra functionality (after verifying that it indeed represents legacy functionality that is no longer is needed; as opposed to recently added functionality that is planned for future use), resulting in a 7% increase in average CPSU and a less bloated PLA.

The second issue regards the LIBRARY SYSTEM component, which has some low PSU and RSU values. In this case, it turns out that the city library product does not need to support some functionality that must be supported by the university and research libraries. To address this issue, we tried a number of alternative solutions. Splitting the LIBRARY SYSTEM component into one required component (containing standard functionality) and an optional component (containing extra functionality for university and research libraries) resulted in the highest increases, namely 17% in average CPSU and 10% in average CRSU. Structurally, this is also the best solution since it aligns the necessary functionality in each product perfectly without adding much complexity.

In both these improvements, changes are at the architectural design level. Given that the PLA has already been implemented, we would have to consult with the implementers and consider other factors (e.g., cost, effort) regarding the actual feasibility of realizing the improvements. Nonetheless, this example highlights how our metrics can illustrate structural problems in a real-life PLA.

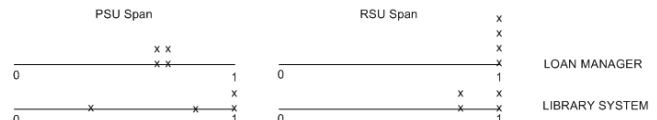


Figure 6. Selected spans for library system PLA.

5. Conclusions

The service utilization metrics introduced in this paper represent a first step toward examining the feasibility of using metrics to improve the structural quality of product line architectures. Our positive experiences in using the metrics on three example PLAs indicate that the metrics, while not perfect, provide a solid foundation to not only identify potential structural problems in a PLA, but also evaluate different solutions to these problems. As such, the metrics complement and enhance design reviews, design parameters and constraints, and an architect's intuition, with a previously unavailable mechanism to assess the structure of a PLA. Of course, other influences, such as cost, per-

formance, or reusability, may lead to a choice of a less-than-optimal structural solution. Nonetheless, our metrics add one critical input datum to this selection process, namely the structural soundness of a PLA.

As expected, applying our metrics on actual examples reveals that they have some strengths and some weaknesses in assessing the structural quality of PLAs. In general, we observe that real problems are discovered, both in the two academic PLAs and in the industrial PLA. The problems are diverse in nature, and solutions include splitting a component into variants, splitting a component into a required and an optional component, merging variants together into a single required component, and simply removing unused services from an interface.

We believe the metrics could be especially useful in helping to maintain a structurally sound PLA when multiple developers make parallel changes to the PLA. In such cases, a PLA's structure tends to degrade over time and gaining an insight into the quality of the overall structure becomes an increasingly difficult process [17]. Our metrics provide one particular abstraction mechanism for gaining such insight. By comparing patterns in the results for older versions of the PLA with patterns in the results for newer versions, the metrics can highlight issues such as decreasing levels of component reuse, (partially) obsolete components, unbalanced variation points, and so on.

Our long-term plans focus on the development of more sophisticated metrics that take into account the presence of software connectors, frequency of interaction among components, and architectural topology and style in assessing structural soundness of a PLA. Using these concepts, we hope to be able to overcome some of the shortcomings reported in Section 4. More immediately, we intend to extend our prototype PLA metric calculation tool beyond its current, limited ability of being able to calculate metrics for a single product only. In particular, we intend to investigate heuristics that automatically highlight potential trouble spots in a PLA and suggest corresponding remedies. The development of these heuristics, combined with our investigation into more sophisticated metrics, forms the focal point of our future research.

Acknowledgments

This research is supported by the National Science Foundation under Grant Numbers CCR-9985441 and CCR-0093489. Effort also sponsored by the Defense Advanced Research Projects Agency, Rome Laboratory, Air Force Materiel Command, USAF under agreement numbers F30602-00-2-0615 and F30602-00-2-0599. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorse-

ments, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory or the U.S. Government. Effort also supported in part by Xerox.

References

- [1] Albrecht, A., and Gaffney, J., *Software Function, Source Lines of Code, and Development Effort Prediction: a Software Science Validation*. IEEE Transactions on Software Engineering, 1983. **9**(6): p. 639-648.
- [2] Allen, R. and D. Garlan, *A Formal Basis for Architectural Connection*. ACM Transactions on Software Engineering and Methodology, 1997. **6**(3): p. 213-249.
- [3] Bastani, F.B., *An Approach to Measuring Program Complexity*, in *COMPSAC'83*. 1983, IEEE Computer Science Press: MD p. 1-8.
- [4] Bayer, J., Muthig, D., Göpfert, B., *The Library System Product Line- A Kobra Case Study*, http://www.iese.fraunhofer.de/pdf_files/iese-024_01.pdf
- [5] Bosch, J., *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. 2000, New York, New York: Addison-Wesley.
- [6] Chidamber, S. and C. Kemerer, *A Metrics Suite for Object-Oriented Design*. IEEE Transactions on Software Engineering, 1994. **20**(6): p. 476-493.
- [7] Davis, J.S. and R.J. LeBlanc, *A Study of the Applicability of Complexity Measures*. IEEE Transactions on Software Engineering, 1988. **14**(9): p. 1366-1371.
- [8] Dix, A.J., *Formal Methods for Interactive Systems*. 1991, London: Academic Press.
- [9] Don Batory, et al., *Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study*. ACM Transactions on Software Engineering and Methodology, 2002. **11**(2): p. 191-214.
- [10] Fenton, N.E. and M. Neil, *Bayesian Belief Nets: a Causal Model for Predicting Defect Rates and Resource Requirements*, in *Software Testing and Quality Engineering*. 2000 p. 48-53.
- [11] Fenton, N.E. and S.L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*. 2nd ed. 1997, London, U.K.: International Thomson Computer Press.
- [12] Gannon, J.D., E.E. Katz, and V. Basili, *Metrics for ADA Packages: An Initial Study*. Communications of the ACM, 1986. **29**(7): p. 616-623.
- [13] Garlan, D., R. Allen, and J. Ockerbloom. *Exploiting Style in Architectural Design Environments*. in *ACM SIGSOFT '94 Second Symposium on the Foundations of Software Engineering*. 1994. New Orleans, LA: ACM Press: p. 175-188.
- [14] Halstead, M.H., *Elements of Software Science*. 1977, New York: Elsevier.
- [15] Henry, S. and D. Kafura, *Software Metrics Based on Information Flow*. IEEE Transactions on Software Engineering, 1981. **7**(5): p. 510-518.
- [16] Hitz, M. and B. Montazeri. *Measuring Coupling and Cohesion in Object-Oriented Systems*. in *Proceedings of the International Symposium on Applied Corporate Computing*. 1995: p. 25-27.
- [17] Johansson, E. and M. Höst. *Tracking Degradation in Software Product Lines through Measurement of Design Rule Violations*. in *In Proceedings of 14th International Confer-*

ence in Software Engineering and Knowledge Engineering (SEKE). 2002. Ischia, Italy

- [18] Jones, C., *Programming Productivity*. 1986, New York: McGraw Hill.
- [19] Lorenz, M. and J. Kidd, *Object-Oriented Software Metrics*. 1994: Prentice-Hall.
- [20] Luckham, D.C. and J. Vera, *An Event-Based Architecture Definition Language*. IEEE Transactions on Software Engineering, 1995. **21**(9): p. 717-734.
- [21] Magee, J. and J. Kramer. *Dynamic Structure in Software Architectures*. in *ACM SIGSOFT '96 Fourth Symposium on the Foundations of Software Engineering*. 1996. San Francisco, CA: ACM SIGSOFT: p. 3-14.
- [22] McCabe, T.J., *A Complexity Measure*. IEEE Transactions on Software Engineering, 1976. **2**(4): p. 308-320.
- [23] Medvidovic, N., D.S. Rosenblum, and R.N. Taylor. *A Language and Environment for Architecture-Based Software Development and Evolution*. in *21st International Conference on Software Engineering*. 1999. Los Angeles, CA: IEEE Computer Society: p. 44-53.
- [24] Medvidovic, N. and R.N. Taylor, *A Classification and Comparison Framework for Software Architecture Description Languages*. IEEE Transactions on Software Engineering, 2000. **26**(1): p. 70-93.
- [25] Myers, G.J., *Composite/Structured Design*. 1978, New York: Van Nostrand Reinhold.
- [26] Pant, Y.R., Verner, J. M., *Software Quality and Productivity. Theory, Practice, Education and Training*. 1995, London: Chapman & Hall.
- [27] Perry, D.E. *Generic Descriptions for Product Line Architectures*. in *Second International Workshop on Development and Evolution of Software Architectures for Product Families*. 1998. Las Palmas de Gran Canaria, Spain
- [28] Shaw, M., et al., *Abstractions for Software Architecture and Tools to Support Them*. IEEE Transactions on Software Engineering, 1995. **21**(4): p. 314-335.
- [29] Shaw, M. and D. Garlan, eds. *Software Architecture: Perspectives on an Emerging Discipline*. 1996, Prentice-Hall.
- [30] Simon, F., F. Steinbrückner, and C. Lewerentz. *Metrics-Based Refactoring*. in *Proceedings of the European Conference on Software Maintenance and Reengineering*. 2001: IEEE Computer Society Press: p. 30-38.
- [31] Tracz, W., *DSSA (Domain-Specific Software Architecture) Pedagogical Example*, in *ACM SIGSOFT Software Engineering Notes*. 1995.
- [32] van der Hoek, A., et al. *Taming Architectural Evolution*. in *Sixth European Software Engineering Conference (ESEC) and the Ninth ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9)*. 2001. Vienna, Austria: p. 1-10.
- [33] van Ommering, R., et al., *The Koala Component Model for Consumer Electronics Software*. Computer, 2000. **33**(3): p. 78-85.
- [34] Wolf, A.L., L.A. Clarke, and J.C. Wileden, *The AdaPIC Toolset: Supporting Interface Control and Analysis Throughout the Software Development Process*. IEEE Transactions on Software Engineering, 1989. **SE-15**(3): p. 250-263.