# Measuring Product Line Architectures

Ebru Dincel[1], Nenad Medvidovic [1], and André van der Hoek [2]

[1] Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781 USA
{edincel, neno}@usc.edu

[2] Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3425 USA
andre@ics.uci.edu

**Abstract.** Software application families and their accompanying architectures (also referred to as product line architectures or PLAs) are a promising area in which the potential of software component reuse can be fully realized. Evolving such application families necessitates making informed architectural decisions. Among industry and research communities, it is recognized that software metrics can provide guidance during the making of such decisions. In this paper, we introduce metrics that are specifically geared to assess product line architectures and help in maintaining their quality.

## 1   Introduction

In recent years, the focus of the software engineering community has shifted from programming stand-alone applications to developing component-based application families. Various technical challenges exist in this domain, such as the need to represent family members, to express and capture commonality, variability, and incompleteness, as well as to incorporate domain knowledge while populating generic, reference architectures [1]. In addition to technical challenges, organizations face strategic, financial, and human factors challenges that make it difficult to initially adopt product line families within an organization.  However, if properly deployed, large-scale reuse results in numerous rewards including reduced costs and risks, higher reuse and predictability, better performance modeling, and more effective communication between stakeholders. Initial evidence is showing that these benefits far outweigh the initial cost, and many organizations are beginning to leverage product-line architectures (PLAs) as a basis for software component reuse.

In many areas of software engineering, the use of metrics has proven to be helpful in assessing particular situations. They help us learn from the past, evaluate the present, and sometimes even predict the future. Metrics provide condensed information about the current state of a system or process, track the progress towards goals, and provide measurements that form a basis for guiding stakeholders in comparative decision making. However, the current set of metrics as defined in the literature (e.g., cohesion [6], cyclomatic complexity [3], fan-in/fan-out [4], and depth and node density [5]) is very much focused on the object level and cannot be directly applied to product line architectures. Specifically, product line architectures exhibit several unique features that make direct use of the above metrics impossible, namely hierarchical composition, optionality, and variability. In essence, these metrics assume a static set of interfaces and are thus not equipped for supporting the diversity that is present in PLAs. For example, if we want to calculate the fan-in and fan-out of a particular component in a product family architecture, we would not know what to do with the PLA's optional or variant components to which the component in question is connected.

Our work focuses on PLA-level metrics that support the hierarchical, incomplete, and diverse nature of PLAs to guide architectural decisions during system evolution. In particular, we provide an incremental set of metrics that allow an architect to make informed decisions about the usage levels of architectural components, the cohesiveness of the components, and the validity of product family architectures. Although only in the beginning stages of our investigation into these metrics, we believe that they (and their future refinements) show promise for applicability in the product line architecture domain.

The remainder of the paper is organized as follows. We first provide a short overview of relevant concepts in product family architectures in Section 2. Then we introduce our proposed metrics in Section 3 and conclude in Section 4 with an outlook at our future work.

## 2  Overview of PLA Concepts

Certain properties are shared across PLAs. The first is *partiality*; PLAs must have partial representation to support the commonality of a product family, yet provide enough flexibility for family members to satisfy different requirements or accommodate future requirements arising from either internal organizational strategies or external market forces. The second property is *diversity*. Components in a PLA can be *mandatory* (they are part of a generic architecture), *optional* (their existence is not guaranteed, dependent on some context), and *variant* (different algorithmic implementation, different interfaces, or different platform dependencies). The final property *is compositionality*. For example, the architecture in Figure 1 is composed of three complex components (each of which is, in turn, hierarchically composed of primitive components). Complex components *CC1* and *CC3* are mandatory, whereas *CC2* is optional. Additionally, *CC1* comprises 3 primitive components: *PC12* is mandatory, *PC11* is variant, and *PC13* is optional.
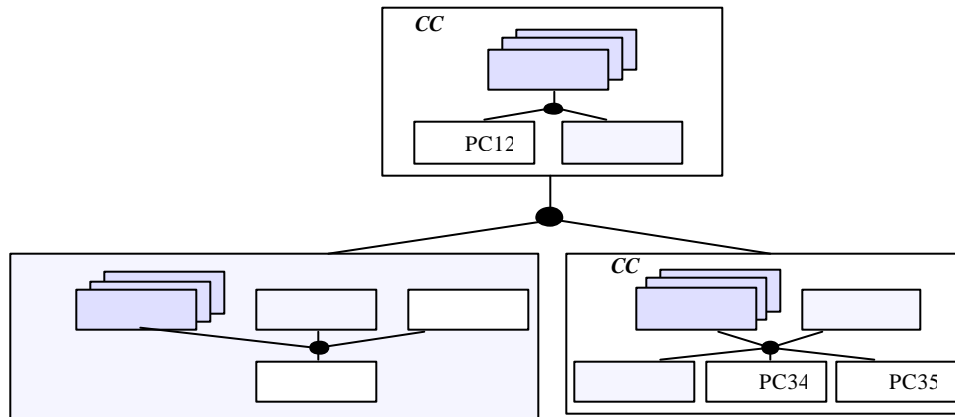
**Fig. 1.** Example product line architecture. Unshaded boxes represent mandatory components; lightly shaded boxes represent optional components; darkly shaded boxes represent variant components.

In order to make the analysis of complex, product line systems such as the one depicted in Figure 1 more efficient, specific techniques must be provided. One such technique that we have exploited extensively in developing PLA metrics is *type checking*. Type checking stipulates that a service (i.e., an operation with its accompanying interfaces) provided by one component will satisfy a service required by another component if and only if their interfaces and behaviors match as defined in [7].

To illustrate how architectural type checking works, let us assume that complex component *CC1* in Figure 1 is part of a logistics system for routing incoming cargo to a set of warehouses. Its constituent component *PC11* is a variant clock component that provides time measurement to the other components, *PC12* models delivery ports, while *PC13* is an optional component that models vehicles. Each component has a set of provided and a set of required services, denoted in Table 1 by P$i$ and R$i$, respectively. For brevity, we have omitted component service behavior specifications from Table 1; see [7] for an example of the complete service specification. When performing a type check of the specified architecture, the required services of *PC12* and *PC13* (*R1* and *R2*, respectively) are matched to the provided services of all components along their communication links. In this case, the only component along *PC12*'s and *PC13*'s communication links is *PC11* and an attempt is made to match *PC11*'s provided services with *PC12*'s and *PC13*'s required services. *PC11*'s *P1* provided service matches both the *R1* and *R2* required services.

**Table 1.** Component services.

| Component | Provided Interface Element | Required Interface Element |
|---|---|---|
| PC11 | P1: Tick () <br> P2: setClockSpeed (rate: Integer) | |
| PC12 | P3: newShipment <br> (port; PortID; shp: ShipmentType) <br> P4: unloadShipment <br> (port: PortID; shp: Integer) <br> P5: getDeliveryPorts() | R1: Tick () |
| PC13 | P6: addShipment <br> (veh: VehicleID; shp: ShipmentType) <br> P7: unloadShipment (veh: VehicleID) <br> P8: getVehicles (): \set VehicleType | R2: Tick () |

It should be noted that architectural type checking is not an "all or nothing" proposition; rather, architecture-level interoperability is a point on a spectrum in which the highest degree of interoperability is achieved when every service required by every component is provided by some other component(s) along its communication links. This issue is further discussed in Section 3 below. Type checking has provided a necessary, though not sufficient, basis for developing the PLA metrics discussed below.

## 3  Proposed PLA Metrics

Based upon a preliminary examination, we have defined a number of initial metrics that we believe will form the basis for more advanced metrics in the domain of product family architectures. Below, we discuss these metrics, their derivation, and their meaning.

### 3.1  Primitive Components

The basic building blocks of our metrics are the *Required Service Utilization (RSU)* and the *Provided Service Utilization (PSU)*. These two metrics are context-dependent: a given component will have different RSU and PSU measures depending on the architecture of which it is a part. The RSU is defined, per basic component, as the number of required

services that are "satisfied" by other basic components *within* a complex component. The PSU is defined, per basic component, as the number of provided services that are used by other basic components *within* a complex component. For example, in the architecture of Figure 1, the RSU and PSU of the component *PC11* are 0 and 0.5, respectively. Similarly, the RSU and PSU of the component *PC12* are 1 and 0, respectively. The RSU and PSU for the other components can be computed in a similar fashion.

Both the RSU and PSU have a well-defined meaning. The RSU defines a satisfaction rate: the closer the RSU is to 1, the more services that are required by a basic component are actually provided by the other basic components in an architecture. In fact, for a basic component that is fully contained within a complex component, ideally the RSU should be 1. The PSU, on the other hand, defines the utilization rate of the provided services of a basic component: the closer the PSU is to 1, the more functionality that is provided by a basic component is actually used by the other basic components. Note that, contrary to the RSU, full containment within a complex component does not necessarily lead to a PSU of 1: some services may be provided that are never used. The PSU, thus, can also be used in an inverse manner: the closer the PSU is to 0, the more "bloated" a basic component is. In such a case, a basic component carries with it a lot of extra functionality that makes it more heavy weight than required by the other basic components in the given architecture.

Note that the RSU and PSU are related to, but different from the concepts of fan-in and fan-out. Whereas fan-in and fan-out provide absolute numbers of "connections", our RSU and PSU define a satisfaction and utilization *rate*. This provides a slightly more useful metric since the context in which the connections are made is taken into account.

### 3.2 Complex Components

The hierarchical nature of a product family architecture complicates the nature of the RSU and PSU as we go up the complexity hierarchy: because different architectural styles and architecture description languages define different rules of service propagation from lower level components to higher-level components (some may prescribe that all provided and required services are propagated, others may prescribe that only "left-over" services are propagated, and yet others may prescribe that the provided and required services of higher level components are explicitly defined), the relationship between the provided and required services of a complex component and the provided and required services of its constituent (basic and complex) components is unclear. Nonetheless, the RSU and PSU of higher-level components also provide useful information to a system-level architect. We define the RSU and PSU measures for a complex component in exactly the same way as the RSU and PSU for a basic component, with the observation that the actual propagation of services from lower level components to higher-level components depends on the particular applicable rules. In our example, we only propagate "left-over" services: those that are not satisfied within the complex component. The meaning of the RSU and PSU for complex components remains the same: the RSU defines the satisfaction rate of a component and the PSU defines the utilization rate.

### 3.3 Average per Complex Component

The *average RSU* and *average PSU* per higher-level complex component are another set of useful measures. They are calculated by averaging the RSU and PSU of the components within a complex component. For example, the average RSU and PSU for component *CC1* are 0.67 and 0.125, respectively. These two metrics can be utilized to assess the cohesion of a complex component: the closer to 1 both the average RSU and average PSU are, the more self-contained, and thus cohesive, the component is.

The average RSU and PSU can also be used to classify the nature of a component. If the average RSU is high and the average PSU is low, the component is a *service* component at the next higher level since the component provides many services that are not used internally. If the average RSU is low and the average PSU is high, the component is a *driver* component since it requires many services that are not provided internally. If the average RSU is high and the average PSU is high, the component typically is a *transformational* component—services are translated to and from each other. Finally, if the average RSU is low and the average PSU is low, this typically is a *marginal* component, i.e., one that serves a very limited function in the system. Such a component should be closely examined for its usefulness and potential for absorption into another component.

### 3.4 Average per Product Family Architecture

The second and third complications of a product family architecture are its inherent abilities to capture optionality and variability. For our metrics, this poses a challenge: when a component is optional, the RSU and PSU of other components (and the optional component itself!) depend on whether the optional component is included in the architecture or not. As such, the RSU and PSU of a complex component or architecture have to be calculated per the above *after* the selection process of optional and variant components has taken place.

However, another useful metric is the average RSU and PSU per product family architecture. These metrics are calculated as follows: enumerate all possible configurations that a product family architecture may exhibit and calculate the average of the RSUs and PSUs of each of those configurations. In the ideal case, this average RSU is 1, indicating that all possible configurations are valid—valid meaning that all required services are provided by the components inside the configuration. In reality, however, the average RSU will be lower, indicating that some configurations are not valid. The lower the RSU, the more "spotty" a product family architecture is and the more attention has to be paid to the configuration process to ensure a proper configuration is selected.

The average PSU for a product family architecture serves a similar kind of role. In the extreme case the average is 1, indicating that all services that are provided by a product family architecture are actually used in each of the instances of that family. In reality, of course, the average is lower: since a product family architecture is build to provide a good

degree of flexibility, it cannot be expected that all services that are provided are used within each of the product family members. If the average is too low, however, it indicates a degree of separation within the product family architecture and perhaps a split into two or more separate product family architectures is required.

Of note is that not only the average is a useful metric, but also the span over the average RSU and average PSU of a product family architecture. The larger the range of average RSUs and PSUs, the more unbalanced a product family architecture is. Note that the span of average RSU and PSU values directly takes into account the existence of variant and optional components in product line architectures.

## 4  Conclusion

This paper has presented some preliminary results in applying metrics to product family architectures. Based upon two basic metrics that operate at the individual, basic component level, namely the Required Service Utilization and the Provided Service Utilization, we have defined additional metrics that are able to assess complex components and product family architectures as a whole. While our experience with the metrics is limited, we believe their close relationship to existing measures, such as cohesion, fan-in, and fan-out, is an indication of their applicability and relevance. Our immediate future work involves applying the metrics on an actual product family architecture to evaluate their applicability in a real-world setting. Additionally, we plan to refine and enhance our existing set of metrics while also introducing several new metrics that operate at the level of the individual services.

## 5  Acknowledgements

# References

1. A. van der Hoek, M. Rakic, R. Roshandel, and N. Medvidovic. Taming Architectural Evolution. *ESEC/FSE 2001*, Vienna, September 2001.
2. J. Poulin. *Measuring Software Reuse*. Addison Wesley, 1997.
3. T. J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2(4): 308-320, 1976.
4. S. Henry and D. Kafura. Software Structure Metrics Based on Information Flow. *IEEE Transactions on Software Engineering*, 7(5): 510-518, 1981.
5. M. Lorenz and J. Kidd. *Object-Oriented Software Metrics*. Prentice Hall, 1994.
6. B. Meyer. Reusability: The Case for Object-Oriented Design. *IEEE Software*, 4(2): 50-64, March 1987.
7. N. Medvidovic, D. S. Rosenblum, and R. N. Taylor. A Language and Environment for Architecture-Based Software Development and Evolution. *ICSE'99*, Los Angeles, CA, May 1999.