(Reference) Architecture = Components + Composition (+ Variation Points)?

Kung-Kiu Lau and Simone Di Cola
School of Computer Science
The University of Manchester
Manchester M13 9PL, United Kingdom
kung-kiu,dicolas@cs.manchester.ac.uk

ABSTRACT

The notions of architecture, component and composition are perceived differently in different communities. In order to discuss how component-based development can contribute to the definition and use of reference architecture in practice, in this position paper, we outline some fundamental characteristics of components and composition and posit their relevance to reference architecture.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures

Keywords

Software architecture; components; composition; reference architecture

1. ARCHITECTURE = COMPONENTS + COMPOSITION

Is it stating the obvious, to say that 'architecture = components + composition'? It would seem not. For one thing, different research communities have different notions of what architectures and components are, notwithstanding the generally accepted view of an architecture as a design blueprint and components as parts of some whole – a design or a system. Composition seems to be under the radar altogether.

For example, within the Software Architecture community [2] we understand that 'architecture' can include various kinds of artefacts (from guidelines to documents to resources), apart from design. Equally, components can take many forms. Composition becomes virtually invisible as a result.

Within the CBSE community [13], an architecture usually represents the structure of the software system under construction. However, composition is usually not a first-class entity in an architecture. Rather, software units are usually 'glued' together, albeit often using very sophisticated programming frameworks.

*The secretary disavows any knowledge of this author's actions.

CobRA'15, May 6, 2015, Montréal, QC, Canada.

http://dx.doi.org/10.1145/2755567.2755978.

2. COMPONENT MODEL ≡ COMPONENTS + COMPOSITION

In CBSE, the study of component models [12, 5, 9] has highlighted the role of composition. However, in theory and in practice, composition remains very underdeveloped.

A component model defines components and their composition. In other words, both components and composition mechanisms are first-class entities in a component model, which means they are co-equal top-level semantic elements. Composition mechanisms which are first-class entities can be defined as explicit (mathematical) operators.

Among current component models, there is a universally accepted view of a component. This is depicted in Fig. 1: a component is a software unit with required and provided services.

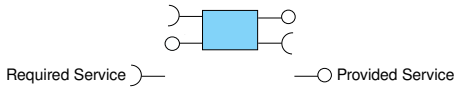

Figure 1: A generic component.

In current component models, components are first-class entities. Fig. 2 shows the three main types of components in these models: (a) objects (b) architectural units and (c) encapsulated components. Each of these types is a variation of the generic component: (a) an object's methods are its provided services, but an object has no visible required services (hence the blurring out in Fig. 2(a)); (b) an architectural unit has input ports as required services, and output ports as provided services; (c) an encapsulated component has only provided services but no required services.

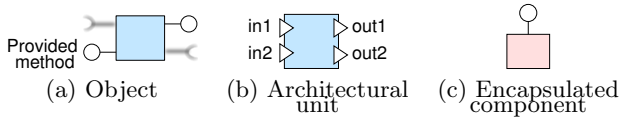

Figure 2: Types of components in current component models.

In contrast, there is no universal view of composition. In current component models, composition is not always de-

fined as first-class entities. Fig. 3 shows the types of composition in these models. In models where components are

Components	Provided services	Required services	Composition mechanism
Objects	Methods	——	Method call
Architectural units	Out-ports	In-ports	Port connection
Encapsulated components	Methods	*None*	Exogenous composition

Figure 3: Types of composition in current component models.

objects, composition is not a first-class entity: object 'compose' by method calls, which are hard-wired in the code of objects. In such models, architectures are often defined as class diagrams in which composition is defined as object aggregation or object composition, which are defined in terms of object life cycles, but not as explicit composition mechanisms.

In models where components are architectural units, composition is also not a first-class entity: architectural units 'compose' via port connections; consequently composition is defined directly for ports, but not directly for whole architectural units.

Method call and port connection are examples of message passing, direct and indirect respectively, as shown in Fig. 4, where U_1 and U_2 are generic software units. Direct message

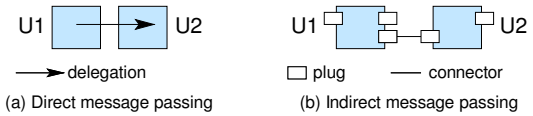

Figure 4: Composition by connection.

passing is also called delegation. Indirect message passing applies to units with 'plugs' (e.g. ports) and is defined by connectors that link the plugs. They are both examples of composition by connection, which is not a first-class entity when used in component models.

In models with encapsulated components, composition can be defined as a first-class entity. Here, without required services, components have to be coordinated by an external coordinator (Fig. 5), i.e. by exogenous composition, and thus in the model, coordination as exogenous composition is co-equal to components.

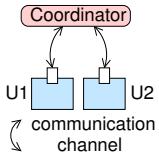

Figure 5: Composition by coordination.

3. ARCHITECTURE DESCRIPTION LANGUAGE = COMPONENT MODEL

Architecture description languages (ADLs) define architectures as architectural units (with ports) 'composed' by port connections. ADLs are thus component models where architectural units are components (Fig. 2). They are by far the most widely used component models in practice (certainly in CBSE). So it is pertinent to ask if they are the best component models to take us towards reference architectures.

One potential disadvantage of ADLs is that because composition by port connection is not a first-class entity, it is not possible to define explicit composition operators for whole architectural units. For any two architectural units A_1 and A_2, it is not possible to define a composition function $F(A_1, A_2)$. Rather F can only be defined in terms of the ports of A_1 and A_2. Furthermore, since an architectural unit can have an arbitrary number of ports, and (matching) ports between architectural units can be linked in arbitrary combinations, F can be defined only for specific architectural units. Any F defined for A_1 and A_2 (in terms of their ports) will be undefined for other A's.

On the other hand, one potential advantage of ADLs is that a composite architectural unit can have the same type as that of its constituent architectural units, even if composition is only defined at the level of ports. This kind of algebraic property enables hierarchical composition, which is very important for systematic construction as well as managing complexity. However, in this case, the property results from the fact that the ports of a composite architectural unit can be related to (e.g. delegated to, as in UML2.0 [15]) those of the constituent architectural units; but not from the use of a composition operator for architectural units. An algebraic composition operator for architectural units would be much more desirable still for hierarchical construction and managing complexity.

4. VARIATION POINTS = ?

Whatever ADL/component model we believe in or adopt, it should be able to accommodate variation points, if we want to use it to define reference architectures. For the purpose of this paper, we adopt the narrow view of a reference architecture as an architecture template for all possible products (in a domain). For a wider debate on terminology and semantics see [14].

Variation points are relationships between parent-child features in a feature model. So the question arises as to how to represent features and their relationships in a component model. In a component model, the only relationships are 'composition' relationships between components, so it would seem that we should represent features as components and variation points as composition of components.

In component models with objects as components, this means that objects would represent features, and method calls between objects would represent variation points. The only kind of 'architecture' that could be defined would be class diagrams in which objects are features and some of their associations are variation points. It is hard to see how such an 'architecture' can be used as a template.

Another potential problem with this approach is that representing features by objects may not be straightforward. This is because the mapping between features and objects may not be clear cut, let alone one-to-one. A feature may require several objects to implement it, and conversely, an object may implement parts of multiple features. Close coupling between objects may also hinder the task of mapping features to/from objects.

In ADLs, architectural units can be used to represent features, and port connections to represent variation points

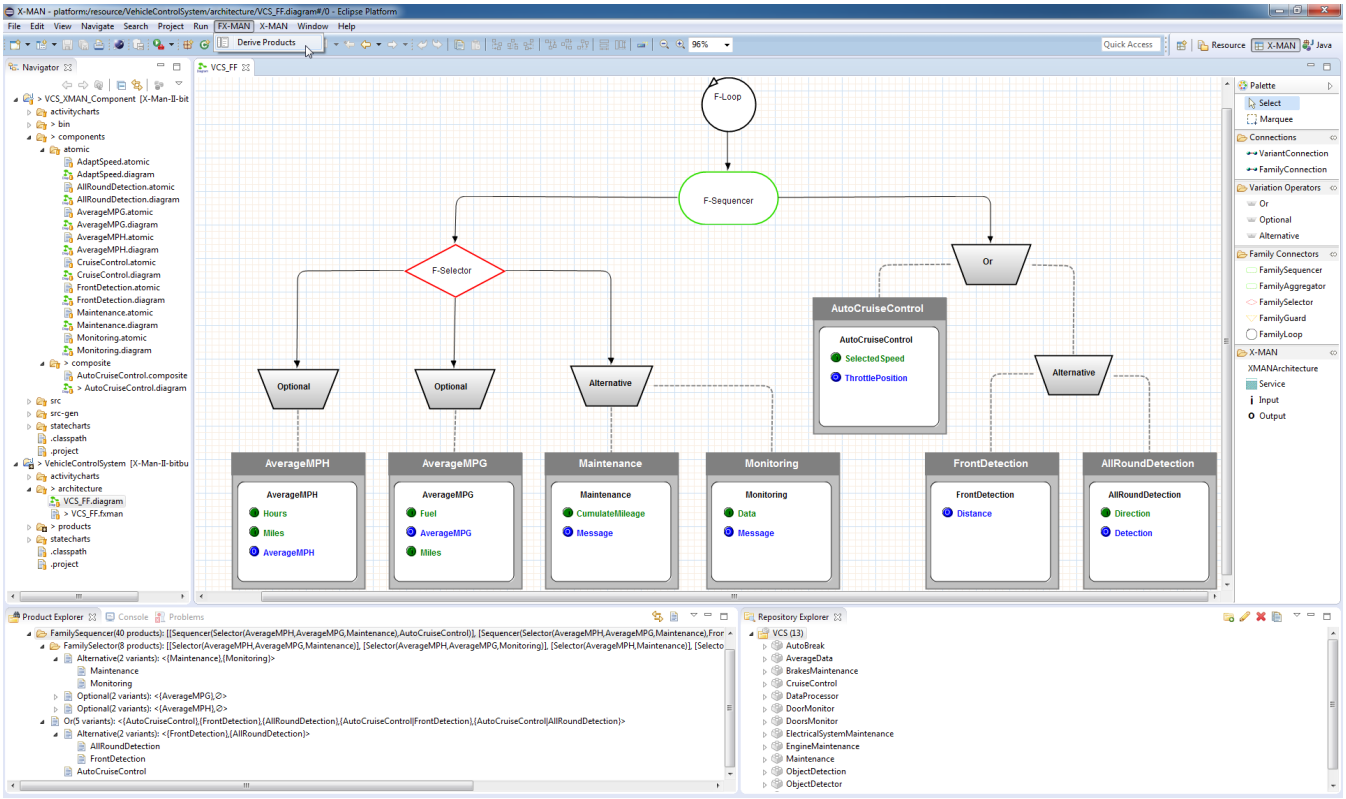

Figure 6: Is this a reference architecture for vehicle control systems?

(e.g. [3]). (In some ADLs, variation points can also be represented by special elements, e.g. *switch* in Koala [16].) However, because of the 'binary' nature of port connection, namely that it is either 'on' or 'off', architectural units and ports can only represent *optional* or *alternative* (exclusive or) variation points, but not *or* (inclusive or). Nevertheless, an ADL architecture with variation points can represent an architecture template. Whether not having *or* is a deal breaker is a moot point though. Curiously, in all the literature on FODA [10, 11] (the most widely adopted domain analysis technique) that we have come across, there is no example of *or*.

As in the case of component models with objects as components, representing features by architectural units may not be straightforward; neither is mapping features to/from architectural units. Architectural units are also closely coupled, albeit indirectly.

An alternative to defining variation points in component models is to deal with variation points indirectly. This is the realm of *variability management* [4]. Rather than defining variation points explicitly, some object-oriented variability management techniques construct products by following the variability specified in the feature model. For example, in *feature-oriented software product lines* [1], products are coded directly and selected features are added by programming techniques. One popular technique is *weaving* [6], which injects features as aspects into objects. Feature and variation points here are thus only represented as object-oriented (aspect-oriented) programming language constructs, namely aspects, join points and point cuts, and do not have their usual semantics. The question is therefore whether such an approach can provide a meaningful architecture template in the normal sense of domain analysis for all possible products.

5. REFERENCE ARCHITECTURE = COMPONENTS + COMPOSITION + VARIATION POINTS?

An architecture template defined by an ADL represents a parametric product whose parameters determine which features it contains. Such a product can be derived, or configured, by instantiating the parameters of the template. The template is thus the blueprint for individual products in the product family, to be configured one at a time. However, it is not a master architecture for the product family, that contains the architectures of all possible products. In a master architecture, all products are already present and therefore do not require configuring one at a time from a template.

We conjecture that to define a master architecture as a reference architecture, we need to define a component model in which variation points are also first-class entities, co-equal with components and composition.

To test this conjecture we have developed a component model (called FX-MAN because it is based on another component model called X-MAN [8]) with encapsulated components composed by exogenous composition operators, as well as explicit variation operators that can permute a set of architectures into variations defined by the variation points in the feature model.

Compared to objects and architectural units, mapping features to/from encapsulated components is relatively straight-

forward, since such components have no external dependencies, and are therefore very loosely coupled (via coordination).

Fig. 6 shows a master architecture (we conjecture) for vehicle control systems (VCS) [7] built using our tool for FX-MAN. This architecture mirrors the feature model for vehicle control systems shown in Fig. 7. It explicitly con-

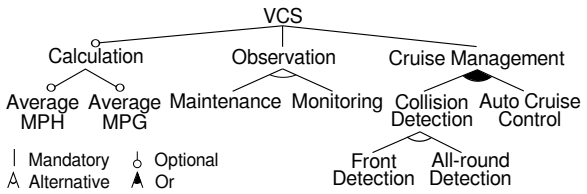

Figure 7: Feature model for vehicle control systems.

tains the architectures of all possible variants specified by the feature model, i.e. 40 products. This product family is shown in the product explorer view at the bottom-left of Fig. 6.

Is the architecture in Fig. 6 a reference architecture? Are we on the right track? We sincerely hope so, but we would appreciate constructive feedback, and above all, collaboration with reference architecture experts!

Acknowledgements

We wish to thank our colleagues Cuong Tran, Chen Qian and Rehman Arshad for their enthusiastic collaboration and insightful feedback.

6. REFERENCES

[1] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines*. Springer, 2013.

[2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. SEI Series in Software Engineering. Addison-Wesley, third edition, 2012.

[3] T.J. Brown, I.T.A. Spence, and P. Kilpatrick. A relational architecture description language for software families. In *Software Product-Family Engineering*, pages 282–295. Springer, 2004.

[4] L. Chen, M. Ali Babar, and N. Ali. Variability management in software product lines: a systematic review. In *Proceedings of the 13th International Software Product Line Conference*, pages 81–90. Carnegie Mellon University, 2009.

[5] I. Crnkovic, S. Sentilles, A. Vulgarakis, and M.R.V. Chaudron. A classification framework for software component models. *IEEE Transactions on Software Engineering*, 37(5):593–615, October 2011.

[6] P. Gilles, G. Vanwormhoudt, B. Morin, P. Lahire, O. Barais, and J.-M. Jézéquel. Weaving Variability into Domain Metamodels. *Software & Systems Modeling*, 11(3):361–383, July 2012.

[7] D. Hatley and I. Pirbhai. *Strategies for real-time system specification*. Addison-Wesley, 2013.

[8] N. He, D. Kroening, T. Wahl, K.-K. Lau, F. Taweel, C. Tran, P. Rümmer, and S. Sharma. Component-based design and verification in X-MAN. In *Proc. Embedded Real Time Software and Systems*, 2012.

[9] K.-K.Lau. Software component models: Past, present and future. In *Proceedings of the 17th International ACM SIGSOFT Symposium on Component-based Software Engineering*, pages 185–186. ACM, 2014.

[10] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, and A.S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie-Mellon University, 1990.

[11] K.C. Kang, J. Lee, and P. Donohoe. Feature-oriented product line engineering. *IEEE Software*, 19(4):58–65, 2002.

[12] K.-K. Lau and Z. Wang. Software component models. *IEEE Transactions on Software Engineering*, 33(10):709–724, October 2007.

[13] J. Maras, L. Lednicki, and I. Crnkovic. 15 years of CBSE Symposium – impact on the research community. In *Proceedings of the 15th International ACM SIGSOFT Symposium on Component-Based Software Engineering*, pages 61–70. ACM, 2012.

[14] E.Y. Nakagawa, A. Pablo Oliveira, and M. Becker. Reference architecture and product line architecture: a subtle but critical difference. In *Software Architecture*, pages 207–211. Springer, 2011.

[15] OMG. OMG Unified Modeling Language Specification, November 2007. `http://www.omg.org/cgi-bin/doc?formal/07-11-01.pdf`.

[16] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala component model for consumer electronics software. *IEEE Computer*, 33(3):78–85, 2000.