Towards Defining Families of Systems in IoT: Logical Architectures with Variation Points

Simone Di Cola(✉), Kung-Kiu Lau, Cuong Tran, and Chen Qian

School of Computer Science, The University of Manchester,
Manchester M13 9PL, UK
{dicolas,kung-kiu,ctran,cq}@manchester.ac.uk

Abstract. In system design, the distinction between a logical architecture at design level and the corresponding physical distributed architecture at implementation level is recognised as good practice. In this paper we show how we can define logical architectures in which variation points can be defined explicitly. Such architectures define families of systems, and should therefore be useful for defining such families in IoT.

Keywords: Software architecture · Product families · Component model · Variability

1 Introduction

The distinction between a *logical* architecture and its physical counterpart, a *physical* distributed architecture, is well-known in system design and is deemed good practice. A logical architecture can be regarded as a *design*, with the corresponding physical architecture as its *implementation*. For example, in [4] Broy describes different architecture levels for cars (Fig. 1): at design level he identifies a logical architecture which, at platform level, is mapped to its corresponding hardware architecture.

In the provisioning of Cloud services, the architecture of software and hardware components is a very challenging task [11]. Specifically, a software architecture S is a kind of logical architecture, since S is normally regarded as a design level artefact[1]. However, in our view, S often looks more like a physical architecture, particularly when the level of abstraction is low [3].

At a low abstraction level, architectural units (with ports) defined by ADLs (architecture description languages) [21] can appear to resemble chips (with pins). Consequently, an ADL architecture (containing architectural units connected together via their ports) is similar to a circuit boards (containing chips wired together via their pins). This physical resemblance can suggest that a software architecture is not a logical architecture, but a physical one. More importantly, this resemblance also seems to carry over into certain architectural properties of an ADL-defined software architecture.

[1] Notwithstanding Broy's view of software architectures as task level artefacts, as reflected by the automotive software standard AUTOSAR (www.autosar.org).

B. Mandler et al. (Eds.) IoT 360° 2015, Part I, LNICST 169, pp. 419–426, 2016.
DOI: 10.1007/978-3-319-47063-4_43

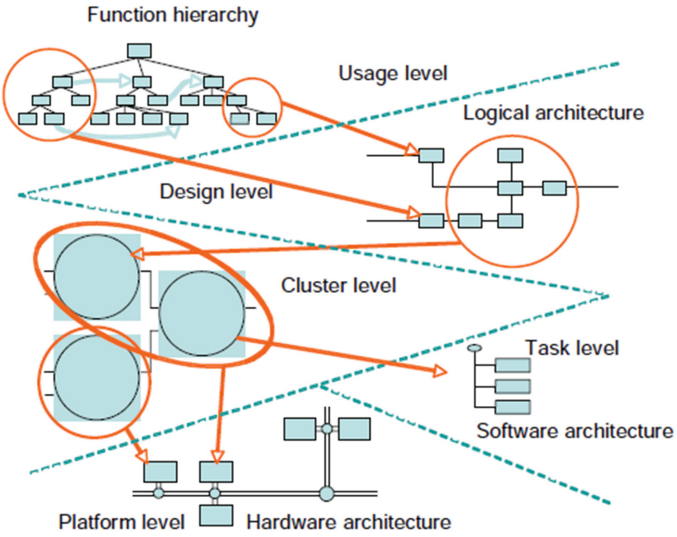

Fig. 1. Broy's architecture levels for cars [4].

The property we want to focus on is the ease of explicitly defining variation points, as in a feature model[2] [17] in family of systems [5,24] for IoT. On a circuit board, switches can direct flow in various directions; correspondingly, in some ADLs (e.g. Koala [28]) switches are used to guide (and configure) bindings between architectural units. However, switches do not fully define variation points in the sense of feature models, namely *optional*, *alternative* (exclusive 'or'), and *or* (inclusive 'or'). At best, switches can be used to create templates for generating different behaviours (code fragments) that correspond to *optional* or *alternative* features. Moreover, a feature model can also define more general dependencies between features as *composition rules* (e.g. feature A *requires*/*excludes* feature B), but switches cannot define such rules.

In this paper we briefly show how we define logical architectures in which we can explicitly define the full set of variation points; the latter are thus first-class citizens in our architectures. We can also handle composition rules.

2 Related Work

Our work in this paper is on the topic 'Variability and architecture description'.

A key observation in [12] is that "variability is often not explicitly described in software architectures". Our work (partially) addresses this issue: we show how we can define logical architectures with explicit variation points.

Existing approaches that also define explicit variation points include the ADLs Koala [28], xADL [9], and Mae [27]. In terms of variation points, the

[2] A hierarchical representation of a product family in terms of features.

difference between these approaches and ours is that they only define *optional* and *alternative* variation points, whereas we define all possible variation points.

In Koala, an *alternative* variation point is defined by a special construct called `switch`, which routes connections among component interfaces, according to input coming from a special component called `module`. A Koala architecture is a template, and a module is used to configure its instances. To generate a particular instance, the Koala compiler removes unconnected components.[3]

Koala also allows variation within a component via a `diversity interface` (a special kind of required port). However, a diversity interface only provides a general parameterisation mechanism, which allows any kinds of variations that are possible at component code level. So it does not define variation points at architecture level.

In xADL, architectures are modelled as instances of predefined XML schemas. At architectural level, an *optional* variation point is expressed by the `optional` tag. Like a Koala diversity interface, a `variant` tag defines variation within a component. Both tags are guarded by user-defined Boolean expressions, which must respect their semantics. For instance, a `variant` tag only specifies alternatives if the user defines guard expressions which are mutually exclusive.

As a predecessor of xADL, Mae also has a textual language to define architectures with variability. Architectural elements can be included, or excluded by evaluating the associated *name/value* pairs.

Other ADLs, like MontiArc^{HV} [14] and Plastic Partial Component [23], do not define variation points explicitly, but only place-holders for different realisations of a named but otherwise unspecified feature. Like in Koala, an architecture in these ADLs is a template that needs to be configured in order to derive its instances.

3 Defining Logical Architectures with Variation Points

Before we describe our approach, we need to be precise about what we mean by logical architectures. We follow the definition given in [25]:

> "The *logical architecture* is a breakdown of the functionality into interacting logical components. It represents the functional decomposition of a system into functional components, as well as the behaviors of these components at the logical level. The functional components provide the functionalities described in the requirements model."

A logical architecture is thus the logical view of a system architecture [18].

Clearly ADLs, or more generally, component models [19,20] can be used to define logical architectures. However, for reasons mentioned earlier, and judging by existing work, it seems that it is not straightforward to define variation points in ADL-defined software architectures. Consequently, we decided to use a component model [7] to define logical architectures, and in our model, we have defined variation points explicitly.

[3] Extensions to Koala for configuration definition and generation are provided by Koalish [2] and Kumbang [1].

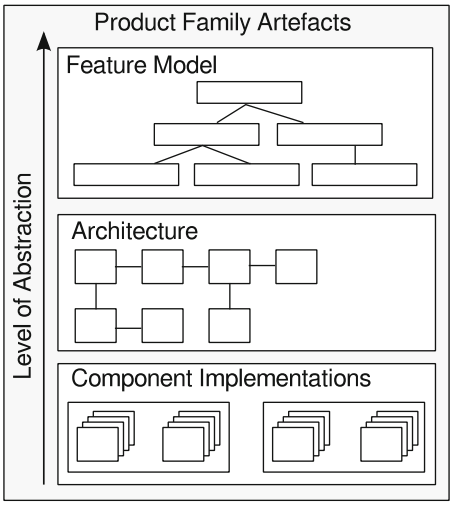

Fig. 2. Levels of abstraction in product family artefacts [26].

Our motivation is to define logical architectures for Cloud systems that will be as close a match as possible to feature models, rather than ADL-defined architectures, in the context of product family artefacts (Fig. 2). Supporting the development of family of services is very useful for the production of IoT (Internet of Things) oriented applications, where services are related to context based information [10]. However, this means we have to define logical architectures as trees, since feature models are trees.

Our approach is based on a component model (X-MAN [15]) useful to develop also Cloud systems [8], that constructs logical architectures as trees. In X-MAN, components can be atomic or composite, and architectures are built by hierarchically composing components using connectors that implement coordination mechanisms. Thus an architecture is a tree of coordinated components, both atomic and composite. In Fig. 3, `AverageMPH`, `AverageMPG`, `Maintenance`, `Monitoring`, `FrontDetection`, and `BackDetection` are atomic components, whereas `AutoCruiseControl` and `AutoBrakeBackDetection` are composite components. The insets show these composite components as trees.

However, X-MAN does not define variation points, and therefore it cannot define product families. Hence, we expanded it with variation operators and family connectors; together they realise the variability expressed by variation points in a feature model. Variation operators are applied to X-MAN architectures to generate variations which are tuples of X-MAN architectures. Family connectors are applied to these tuples to generate product families. Thus the expanded model (FX-MAN [7]) creates architectures with a full set of variation points. Such architectures are product families described by feature models.

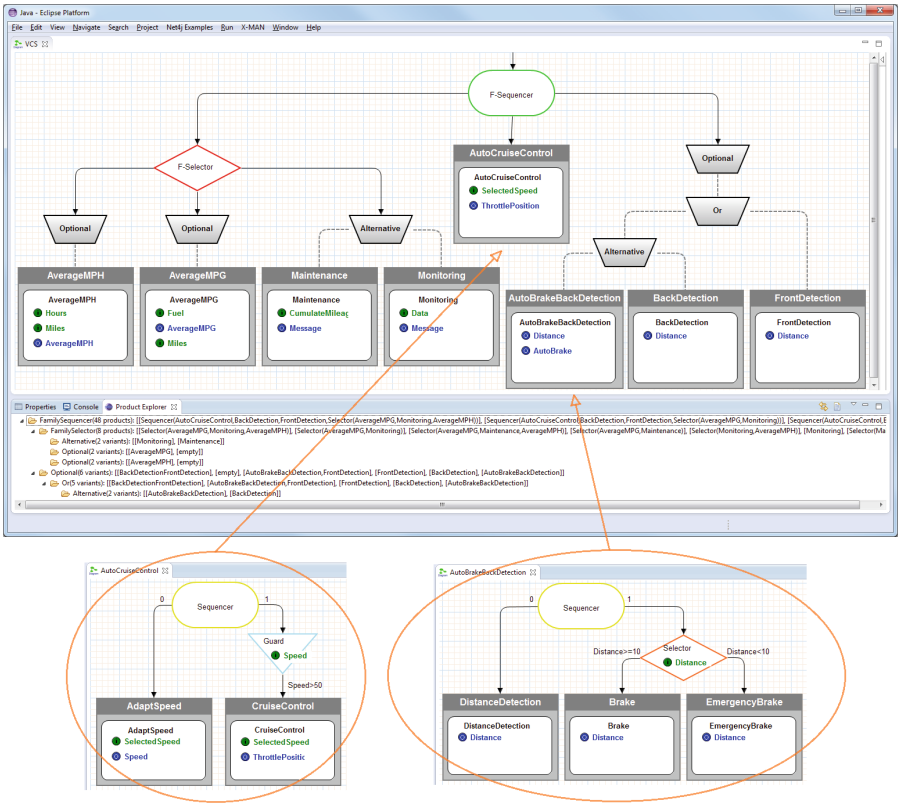

Fig. 3. Logical architecture for Vehicle Control Systems.

An example[4] of a logical architecture is shown in Fig. 3. It is the logical architecture of a product family of vehicle control systems (VCS), whose feature model is shown in Fig. 4.

As can be seen in Fig. 3, in a logical architecture the first-class citizens are: X-MAN architectures, variation operators and family connectors. X-MAN architectures appear at the bottom, and variation operators appear on top of these architectures. In Fig. 3 there are three *Optional*, two *Alternative* and one *Or* variation operators. Variation operators can be nested like variation points in feature models; in Fig. 3 an *Optional*, an *Alternative* and an *Or* variation operators are nested. Family connectors appear on top of variation operators, or X-MAN architectures which are mandatory; in Fig. 3 there are one *F-Selector*, and one *F-Sequencer* which connects to a mandatory X-MAN architecture `AutoCruiseControl`.

Clearly the logical architecture of VCS mirrors the tree structure of its feature model (Fig. 4). Indeed, the leaves of the feature model are implemented by X-MAN architectures. For instance, the AverageMPH feature is implemented by

[4] The example has been created using our FX-MAN Eclipse tool.

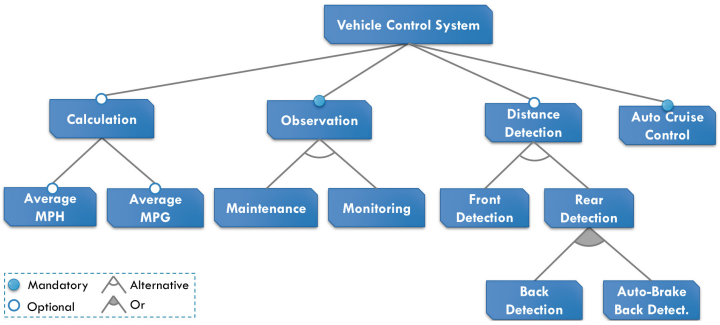

Fig. 4. Feature model for Vehicle Control Systems.

the `AverageMPH` component. Variations specified by variation points in feature models are generated by operators that take as input tuples of X-MAN architectures and return tuples of their variations. For example, an *Optional* variation operator applied to the `AverageMPH` component returns the tuple ⟨`AverageMPH`, ∅⟩. The product explorer view at the bottom of Fig. 3 shows all the variations in the VCS example. Finally, the product family defined by a feature model is constructed by family connectors that compose tuples of X-MAN architectures. In the example, the product family is constructed by the family connectors *F-Selector* and *F-Sequencer*. The product family contains 48 products, as can be seen in the product explorer view.

4 Discussion and Conclusion

Our logical Cloud architecture is executable: all the products in the product family it defines are composed from executable X-MAN components. This is in contrast to the general nature of a logical architecture as merely a logical representation (of the decomposition) of the function hierarchy (Fig. 1), i.e. a structure without behaviour. Executability means our logical architecture can realise not only the feature model (except for non-functional features) but also the functional model of the domain (which defines the behaviour of all possible products in the domain). We are currently examining suitable formulations of the functional model for facilitating the validation of its realisation.

Our logical architecture is a tree, so in terms of levels of abstraction for product family artefacts, it is closer to a feature model than an ADL-defined architecture (Fig. 2). This means that in practice the construction of a product family architecture, which is currently a difficult challenge [6,13], can be closely guided by the feature model. Moreover, since it can also realise the functional model, a logical architecture constructed this way can be a reference architecture for the domain, the construction of which is currently also a difficult challenge [22]. It will be interesting to investigate these issues further.

Although ADL-defined architectures can also serve as logic architectures, current ADLs do not define all possible variation points as first-class citizens.

For instance, Koala, xADL, and Mae do not define the *Or* (inclusive 'or') variation point at architecture level. However, some ADLs provide variation mechanisms at component code level. Such variations are internal to components and can be defined in arbitrary manners. This can be done in Koala via a `diversity interface` and in xADL via a `variant` tag. By contrast, our approach defines the full set of variation points as first-class citizens, with fixed semantics. Our experience provides some evidence that it is easier to define variation points using our logical architectures.

With variation points as first-class citizens our logical architecture explicitly contains all the members of a product family. This means that all the products can be extracted directly, rather than configured individually. Furthermore, composition rules can be realised by filters applied to the whole product family.

Returning to Fig. 1, for a chosen platform our logical architecture can be deployed to a physical architecture. In this regard, it maybe advantageous to transform our logical architecture into an equivalent ADL one, especially when the last resembles a hardware architecture, e.g. [16].

Finally, since our work has been done in the component-based development community, we would really appreciate any feedback from the Cloud architecture community.

References

1. Asikainen, T., Männistö, T., Soininen, T.: Kumbang: a domain ontology for modelling variability in software product families. Adv. Eng. Inf. **21**, 23–40 (2007)
2. Asikainen, T., Soininen, T., Xu, Y.: A koala-based approach for modelling and deploying configurable software product families. In: van der Linden, F.J. (ed.) PFE 2003. LNCS, vol. 3014, pp. 225–249. Springer, Heidelberg (2004)
3. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. SEI Series in Software Engineering, 3rd edn. Addison-Wesley, Boston (2012)
4. Broy, M.: Challenges in automotive software engineering. In: Leon, J., Osterweil, H., Rombach, D., Soffa, M.L. (edr.) 28th International Conference on Software Engineering, pp. 33–42. ACM (2006)
5. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley, Boston (2002)
6. Clements, P.: Biglever newsletter: from the ple frontline - paul's three surprises: part 3
7. Cola, S.D., Lau, K.-K., Tran, C., Qian, C., Arshad, R., Christou, V.: A component model for software product families. In: Paper submitted to the 18th International ACM Sigsoft Symposium on Component-Based Software Engineering (2015)
8. Cola, S., Tran, C., Lau, K.-K., Celesti, A., Fazio, M.: A heterogeneous approach for developing applications with FIWARE GEs. In: Dustdar, S., Leymann, F., Villari, M. (eds.) ESOCC 2015. LNCS, vol. 9306, pp. 65–79. Springer, Heidelberg (2015). doi:10.1007/978-3-319-24072-5_5
9. Dashofy, E.M., van der Hoek, A., Taylor, R.N.: A comprehensive approach for the development of modular software architecture description languages. ACM Trans. Softw. Eng. Methodol. (TOSEM) **14**, 199–245 (2005)
10. Fazio, M., Celesti, A., Puliafito, A., Villari, M.: An integrated system for advanced multi-risk management based on cloud for IoT. In: Re, G.L. (ed.) Advances onto the Internet of Things. AISC, vol. 260, pp. 253–269. Springer, Heidelberg (2014)

11. Fazio, M., Puliafito, A.: Cloud4sens: a cloud-based architecture for sensor controlling and monitoring. IEEE Commun. Mag. **53**(3), 41–47 (2015)
12. Galster, M., Avgeriou, P., Weyns, D., Männistö, T.: Variability in software architecture: current practice and challenges. SIGSOFT Softw. Eng. Notes **36**(5), 30–32 (2011)
13. Garlan, D.: Software architecture: a travelogue. In: Proceedings of the on Future of Software Engineering, FOSE, pp. 29–39. ACM, New York (2014)
14. Haber, A., Rendel, H., Rumpe, B., Schaefer, I., Van Der Linden, F.: Hierarchical variability modeling for software architectures. In: 15th International Software Product Line Conference (SPLC), pp. 150–159. IEEE (2011)
15. He, N., Kroening, D., Wahl, T., Lau, K.-K., Taweel, F., Tran, C., Rümmer, P., Sharma, S.: Component-based design and verification in X-MAN. In: Proceedings of Embedded Real Time Software and Systems (2012)
16. Tran, C., Saudrais, S., Lau, K.-K., Štěpán, P., Tchakaloff, B.: A holistic (component-based) approach to autosar designs. In: Proceedings of 39th EUROMICRO Conference on Software Engineering and Advanced Applications, pp. 203–207. IEEE (2013)
17. Kyo, C., Kang, J.L., Donohoe, P.: Feature-oriented product line engineering. IEEE Softw. **19**(4), 58–65 (2002)
18. Kruchten, P.: The Rational Unified Process: an Introduction. Addison-Wesley Professional, Boston (2004)
19. Lau, K.-K., Wang, Z.: Software component models. IEEE Trans. Softw. Eng. **33**(10), 709–724 (2007)
20. Lau, K.: Software component models: past, present and future. In: Proceedings of the 17th International ACM Sigsoft Symposium on Component-based Software Engineering, pp. 185–186. ACM (2014)
21. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Trans. Softw. Eng. **26**(1), 70–93 (2000)
22. Nakagawa, E.Y.: Reference architectures and variability: Current status and future perspectives. In: Proceedings of the WICSA/ECSA 2012 Companion Volume, WICSA/ECSA 2012, pp. 159–162. ACM, New York (2012)
23. Pérez, J., Díaz, J., Costa-Soria, C., Garbajosa, J.: Plastic partial components: a solution to support variability in architectural components. In: Joint Working IEEE/IFIP Conference on Software Architecture, & European Conference on Software Architecture, WICSA/ECSA, pp. 221–230. IEEE (2009)
24. Pohl, K., Böckle, G., Van Der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer, Berlin (2005)
25. Pretschner, A., Broy, M., Kruger, I.H., Stauner, T.: Software engineering for automotive systems: a roadmap. In: Future of Software Engineering, FOSE 2007, pp. 55–71. IEEE Computer Society, Washington (2007)
26. Sinnema, M., Deelstra, S., Nijhuis, J., Dannenberg, R.B.: COVAMOF: a framework for modeling variability in software product families. In: Nord, R.L. (ed.) SPLC 2004. LNCS, vol. 3154, pp. 197–213. Springer, Heidelberg (2004)
27. van der Hoek, A., Mikic-Rakic, M., Roshandel, R., Medvidovic, N.: Taming architectural evolution. In: Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-9, pp. 1–10. ACM, New York (2001)
28. van Ommering, R., van der Linden, F., Kramer, J., Magee, J.: The Koala component model for consumer electronics software. IEEE Computer (2000)