Towards A Core Language for Separate Variability Modeling

Alexandru F. Iosif-Lazăr[1], Ina Schaefer[2] and Andrzej Wąsowski[1]

[1] IT University of Copenhagen {afla|wasowski}@itu.dk [*]
[2] Technische Universität Braunschweig i.schaefer@tu-braunschweig.de

Introduction. Model-driven development of software product lines (SPL) [13, 3] exploits rich models of systems to represent the product line architecture, or *base model*, and variability models to describe and derive specific *variants*. Separate variability modeling involves two aspects: (i) describing the system's configuration space through a *feature model* (a set of features that characterize the products in the family); and (ii) describing the set of transformations applied to the base model by the activation of each feature, which we refer to as *variation points*. This architecture of variability models enables automatic *variant derivation* for any given legal configuration. For example, the HATS Abstract Behavioral Modeling Language [2] is actually a set of smaller languages that model each aspect separately and interact to provide the full model.

Feature models [7] (or alternatives such as decision models [12]) have been extensively researched and formalized and they are now widely used to represent SPL configuration spaces. However, each existing variability modeling language has redesigned the representation of transformations to the base models, while the automatic execution of these transformations has been implemented in an *ad hoc* manner.

Examples of general variability modeling approaches (that are independent of the language in which the base model is developed) include the *Orthogonal Variability Model* (OVM) [10], *Delta Modeling* [5, 11] and CVL [4]. OVM is a simple language and a methodology for superimposing variability over any software development artifact without interfering into its contents. Delta Modeling is a methodology that expands existing languages with statically executable variation points selected from a somewhat stable set. CVL is an industrial attempt to create a generic language that facilitates separate variability modeling for models specified in any MOF-based language [8]. It also demonstrates great flexibility through a wide range of variation points.

Our objectives are (i) to understand the execution semantics used by the aforementioned approaches and determine the core features required by separate variability modeling languages, and (ii) to provide the formal specification of a language that could be used in the development of a trustworthy product derivation tool.

We are motivated by the fact that trustworthy product derivation is essential to the development of safety critical embedded systems in domains such as automotive or industrial automation [1, 6]. Industrial standards such as IEC 61508 mandate the use of state of the art tools and quality assurance techniques. So far, the industry certifies individual products, or even avoids introducing any variability into safety critical parts of the systems[1]. Our goal is to facilitate the development of such systems and to enable usable certification strategies for product line tools.

[*]Supported by ARTEMIS JU under grant agreement n° 295397 and by Danish Agency for Science, Technology and Innovation

[1]Personal communication with partners in ARTEMIS projects.

A compact language for separate variability. We propose an abstract semantics of a core language for separate variability modeling as expressive and versatile as CVL. We will describe the language in a series of steps. First, we will formalize our representation of SPL base models. Second, we will introduce the fragment substitution variation point and show its syntax and the execution semantics for a set of variation points. Finally, we will give an overview of how the feature model drives the variant derivation.

Our **base models** are multigraphs of attribute-less untyped objects connected by directed links. Both objects and links are discrete entities with identity and we write $\mathbb{O}$ and $\mathbb{L}$ to denote the infinite universes of objects and links, respectively. We use the functions $\mathsf{src}, \mathsf{tgt} : \mathbb{L} \to \mathbb{O}$ to indicate the endpoints of each link.

A base model *Bm* is a pair $(BmObj, BmLnk)$ where $BmObj \in \mathbb{O}$ is a finite set of objects and $BmLnk \in \mathbb{L}$ is a finite set of links. We say that *Bm* is *closed under links* (or, simply, *closed*), meaning that for each link $l \in BmLnk$ its endpoints are contained in the model, so $\mathsf{src}\, l, \mathsf{tgt}\, l \in BmObj$. Similarly, any fragment f is a pair $(fObj, fLnk)$.

The **fragment substitution** in CVL is a very expressive variation point, as most of the other variation points can be easily reduced to it. However, we found its syntax unnecessarily complex. We simplify it such that a *fragment substitution fs* is a triple (p, r, Bdg) where p is a placement fragment, r is a replacement fragment and *Bdg* is a set of links called a *binding*. It is the *only* variation point used in our language. Its execution removes the placement from the model and uses the binding links to embed the replacement in its stead. Given a base model and a set of fragment substitutions *Fs*, we define well-formedness constraints that help both to facilitate formalization and to eliminate confusion about the effect of their execution.

The **execution semantics** of our fragment substitutions are captured by seven inference rules which test properties of each object and link and decide if they must be part of the variant. The execution simply iterates over all objects and links and copies those for which the rules apply, while skipping those for which no rule applies. For example, one of the rules copies objects from replacement fragments that do not occur in placement fragments in the same time:

$$\frac{(\_, r, \_) \in Fs \quad o \in r\,Obj \setminus \bigcup_{(p,\_,\_)\in Fs} p\,Obj}{o \in [\![Fs, Bm]\!]\,Obj} \ (\textsc{r-obj})$$

Another rule that copies links from replacement fragments also checks that the source and target of each link are also copied. This helps proving that by starting from a *closed* base model and executing a set of well-formed fragment substitutions we will obtain a *closed* variant model.

$$\frac{(\_, r, \_) \in Fs \quad l \in r\,Lnk \setminus \bigcup_{(p,\_,\_)\in Fs} p\,Lnk \quad \mathsf{src}\, l, \mathsf{tgt}\, l \notin \bigcup_{(p,\_,\_)\in Fs} p\,Obj}{l \in [\![Fs, Bm]\!]\,Lnk} \ (\textsc{r-lnk})$$

A **feature model** is a hierarchical structure of features that imposes dependencies on the variation points. A configuration is a set of features that are present in a variant. Each of these features determines the execution of a variation point. There can even be features that are activated multiple times so that the variation point is executed for each activation. We deal with this unavoidable complexity by proposing a flattening semantics which copies the variation points in a flat model on which we simply apply the fragment substitution execution semantics.

Implementation, confluence and translation validation. By providing a formally defined language we facilitate the implementation of a verifiable tool. A copying semantics can be implemented in declarative rule-based model transformation languages more easily and it is easier to argue about it using theorem provers.

The execution of our semantics is **confluent**. While the CVL specification suggests an implementation by in-place transformations (which makes the transformation order critical) our rules always produce the desired result independently of the order in which they are applied. This offers a great deal of flexibility to the SPL designer and paves the way to new ideas on how to implement variant derivation tools.

A verifiable correct implementation of a variability modeling language is hard to obtain. As an alternative, having a semantics formalized as inference rules enables the verification of the derivation through **translation validation** [9].

This approach to validation is independent of the actual implementation. What it does require is: (i) a common semantic framework for both the input and the output—which we provide by representing all models and fragments as graphs; (ii) a formalization of the notion of *correct execution*—which we provide in the form of inference rules and (iii) a proof method which allows to automatically verify that the output is correct.

Presentation. In the presentation we will discuss the main problem of variability modeling. We will introduce the formal semantics of our core variability language and give examples of graph manipulation via execution of fragment substitutions. We will also demonstrate the expressive power of fragment substitutions and how model variability specified in other languages can be rewritten using our syntax on a working example.

References

[1] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wasowski. A survey of variability modeling in industrial practice. In S. Gnesi, P. Collet, and K. Schmid, editors, *VaMoS*, page 7. ACM, 2013.

[2] D. Clarke, N. Diakov, R. Hähnle, E. B. Johnsen, I. Schaefer, J. Schäfer, R. Schlatte, and P. Y. H. Wong. Modeling Spatial and Temporal Variability with the HATS Abstract Behavioral Modeling Language. In M. Bernardo and V. Issarny, editors, *SFM*, volume 6659 of *Lecture Notes in Computer Science*, pages 417–457. Springer, 2011.

[3] P. Clements and L. M. Northrop. *Software Product Lines: Practices and Patterns.* Addison-Wesley, 2002.

[4] CVL Joint Submission Team. *Common Variability Language (CVL). OMG Revised Submission*, 2012.

[5] A. Haber, K. Hölldobler, C. Kolassa, M. Look, K. Müller, B. Rumpe, and I. Schaefer. Engineering Delta Modeling Languages. In *SPLC*, 2013.

[6] J. Hutchinson, M. Rouncefield, and J. Whittle. Model-driven engineering practices in industry. In R. N. Taylor, H. Gall, and N. Medvidovic, editors, *ICSE*, pages 633–642. ACM, 2011.

[7] 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, November 1990.

[8] Object Management Group. *Meta Object Facility (MOF) Core Specification Version 2.0*, 2006.

[9] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In B. Steffen, editor, *TACAS*, volume 1384 of *Lecture Notes in Computer Science*, pages 151–166. Springer, 1998.

[10] K. Pohl, G. Böckle, and F. J. v. d. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* Sprinter Verlag, 2005.

[11] I. Schaefer, L. Bettini, F. Damiani, and N. Tanzarella. Delta-oriented programming of software product lines. In *SPLC'10*, pages 77–91. Springer-Verlag, 2010.

[12] K. Schmid, R. Rabiser, and P. Grünbacher. A comparison of decision modeling approaches in product lines. In *VaMoS*, pages 119–126, 2011.

[13] T. Stahl and M. Voelter. *Model-Driven Software Development.* John Wiley & Sons, 2004.