Enumerative Variability in Software Product Families

Chen Qian
School of Computer Science
The University of Manchester
Manchester, United Kingdom
cq@cs.manchester.ac.uk

Kung-Kiu Lau
School of Computer Science
The University of Manchester
Manchester, United Kingdom
kung-kiu.lau@manchester.ac.uk

***Abstract*—In Software Product Line Engineering (SPLE), in the problem space, variability in a product family is specified in an *enumerative* manner (by a feature model), i.e. all valid variants are enumerated. However, in the solution space, current SPLE approaches use *parametric* variability (variability parameterised on features occurring in a single product variant) instead. In this paper, we take a closer look at enumerative variability, show how it can also be used in the solution space, and briefly discuss why it may be advantageous to do so.**

***Keywords*-Software product line**

I. Introduction

Software Product Line Engineering (SPLE) traditionally proceeds in two phases: (i) *domain* engineering and (ii) *application* engineering [1]. Domain engineering is concerned with the *problem space*: analysing/capturing domain requirements and then creating domain artefacts accordingly. Application engineering is concerned with the *solution space*: using domain artefacts from the domain engineering phase to construct products in the domain, i.e. the product family [2].

The central issue in SPLE is variability. In the problem space, variability in a product family is specified by a feature model, and in the solution space, product variants are created according to this variability. Variability defined by a feature model (in problem space) is *enumerative* in nature, as it includes all valid variants. In contrast, in solution space, current SPLE approaches use *parametric* variability, i.e. variability parameterised on features occurring in a single product variant, and configure code (from a code base) for one product variant at a time.

This divergence is due to the perception that enumerative variability is merely a model whereas parametric variability deals with implementation, i.e. real code. In this paper, we study enumerative variability more closely. We show how it can also be implemented in the solution space, and why this may bring certain advantages.

II. Related Work

A. Enumerative Variability in Problem Space

The standard way to define variability in the problem space is to use a feature model (or feature tree). Fig. 1 shows a simple example for a family of '`hello world`' systems. A feature model names all the available features and sub-features, and shows all the valid combinations thereof in the product family [3]. Features and their sub-features are depicted as parent-child relationships, and variability is defined by *variation points*: *mandatory*, *optional*, *alternative* (exclusive 'or'), *or* (inclusive 'or'); and may be constrained by cross-tree relationships, e.g. 'feature PLANET *requires* feature ATTRIBUTE' [4]. In Fig. 1, valid variants include: 'HELLO EARTH', 'HELLO BEAUTIFUL EARTH', 'HELLO WONDERFUL EARTH', 'HELLO BEAUTIFUL PLANET' and 'HELLO WONDERFUL PLANET', but not 'HELLO PLANET', due to the 'PLANET *requires* ATTRIBUTE' constraint.

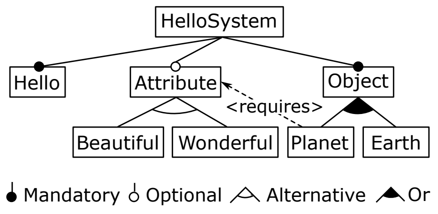


Fig. 1: Feature model for 'Hello World' product family.

A feature model enumerates all the valid combinations of features, i.e. all the valid product variants. In Fig. 1, there are a total of 7 valid variants: the five mentioned above, plus the following two specified by the *or* variation point: 'HELLO BEAUTIFUL PLANET EARTH' and 'HELLO WONDERFUL PLANET EARTH'. The variability defined by a feature model is thus enumerative variability. It provides a configuration model for all product variants to be constructed in the solution space.

However, a feature model does not say anything about behaviour, i.e. how the features are to be implemented. Enumerative variability defined by a feature model is thus by itself insufficient for constructing product variants in the solution space.

B. Parametric Variability in Solution Space

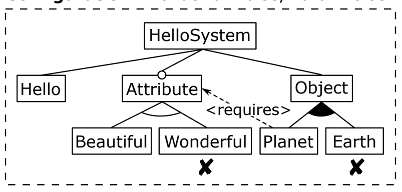
In solution space, SPLE approaches construct code for valid product variants specified by the feature model. The behaviour of the features (and hence the product variants) has to be implemented according to their functional requirements. The behaviour of features that interact with one another needs particular attention. This is known as *feature interaction*: "a feature interaction occurs in a system whose complete behavior does not satisfy the separate specifications of all its features" [5]. For example, in the feature model in Fig. 1, so far we have assumed that the features PLANET and EARTH do not interact, so that when they are combined we get the variants 'HELLO BEAUTIFUL PLANET EARTH' and 'HELLO WONDERFUL PLANET EARTH'. Suppose they do interact in such a way that when combined they no longer output either 'PLANET' or 'EARTH', but just 'WORLD' instead; then these two variants will become 'HELLO BEAUTIFUL WORLD' and 'HELLO WONDERFUL WORLD'.

The general approach to code construction is to start with a code base that consists of code for all the mandatory features (i.e. the base system), with extension points for excluding or including code fragments or modules that implement the variable features. Then for a chosen valid variant specified by the feature model, i.e. an instance of the feature model, called a *configuration* for this product, code is constructed from the code base by excluding code for unselected features, or including code for selected features. Approaches based on code exclusion and inclusion are said to adopt *negative variability* and *positive variability* respectively [6].

```
class HelloSystem {
  void print () {
     System.out.print("Hello");
     //#ifdef Beautiful
     System.out.print("␣Beautiful");
     //#elif Wonderful
     System.out.print("␣Wonderful");
     //#endif
     //#ifdef Planet
     System.out.print("␣Planet");
     //#endif
     //#ifdef Earth
     System.out.print("␣Earth");
     //#endif
  }
  static void main (String[] args) {
     new HelloSystem().print();
  }
}
```

(a) Code base (150% model)

Configuration: Wonderful=false, Earth=false



Variant: Hello Beautiful Planet

(b) Variant configuration

Fig. 2: Negative (parametric) variability in solution space.

Negative variability and positive variability are illustrated in Fig. 2 and Fig. 4 respectively for the 'hello world' family. Fig. 2a shows the code base for 'hello world'. It includes code fragments for all features, both mandatory and variable, i.e. more than will appear in any valid variant (for this reason this kind of code base is called a 150% model). Code fragments for variable features are placed at extension points annotated by boolean conditions which define feature selection (e.g. *#ifdef Beautiful*) and are to be excluded for non-selected features.

Fig. 2b shows a specific configuration, i.e. a valid variant specified by the feature model, which excludes the features WONDERFUL and EARTH; it is the variant: 'HELLO BEAUTIFUL PLANET'. The code for this variant is constructed from the code base in Fig. 2a by excluding code fragments for WONDERFUL and EARTH.

It should be noted that the code base in Fig. 2a assumes that there is no feature interaction. If, for example, PLANET and EARTH interact in the manner described above, i.e. to produce just WORLD, then the last part of the code base should be modified by the code fragment in Fig. 3, for any configuration in which both PLANET and EARTH are selected.

```
//#ifdef Planet && Earth
System.out.print("␣World");
//#elif Planet
System.out.print("␣Planet");
//#elif Earth
System.out.print("␣Earth");
//#endif
```

Fig. 3: Code fragment for Planet-Earth interaction.

Negative variability is adopted by *annotation-based* SPLE approaches [7], which are widely used in industry [8], with leading commercial tools such as pure::variants [9] and Gears [10]. The example in Fig. 2 is annotated by *C preprocessor* (cpp) [11].

SPLE approaches based on architectural description languages (ADL), e.g. Koala [12] and xADL 2.0 [13] use boolean connectors to compose components and to exclude unselected ones in a configuration. Therefore, these approaches also adopt negative variability.

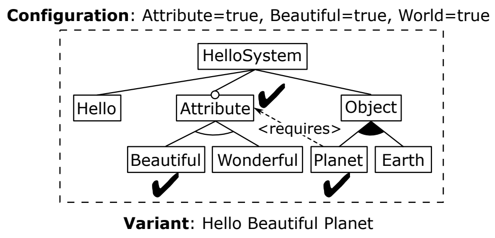
In positive variability, the code base is the base model consisting of code for mandatory features only. Fig. 4b shows an *Aspect-oriented Programming* (AOP) [14] example. The base model is just the HelloSystem class that prints 'HELLO'.

For the configuration in Fig. 4a, code fragments for the selected features BEAUTIFUL and PLANET are added to the base model by aspect weaving at explicitly defined extension points (**after**), as shown in Fig. 4b. The Beautiful class extends the HelloSystem class to give a class that prints 'HELLO BEAUTIFUL', and the Planet class extends this class further to give a class that prints 'HELLO BEAUTIFUL PLANET'.

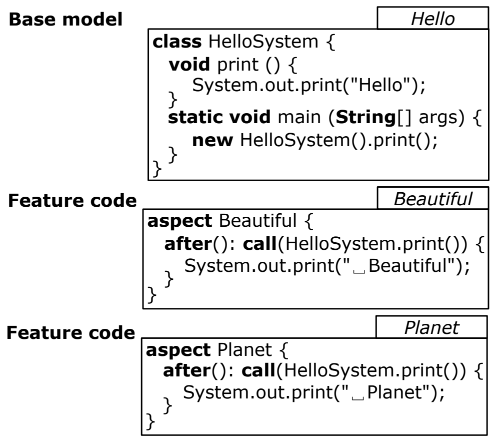
In case of feature interaction (e.g. between PLANET and EARTH as described above), a code fragment for the interaction feature (e.g. Planet+Earth), has to be created and used for any configuration in which all interacting features (e.g. PLANET) and EARTH, are selected; instead of code fragments for the original features.

AOP-based SPLE approaches include XWeave [15] and AFM [16]. Aspect weaving is a form of superimposition (merging code fragments). Other superimposition-based SPLE approaches can handle both positive and negative variability. These include *Delta-oriented Programming* [17] and *Feature-oriented Programming* [18].

In summary, in parametric variability, the feature model is instantiated to create one configuration at a time, and code is


Configuration: Attribute=true, Beautiful=true, World=true
HelloSystem
Hello
Attribute
Object
<requires>
Beautiful
Wonderful
Planet
Earth
Variant: Hello Beautiful Planet

(a) Variant configuration



Base model — Hello
```
class HelloSystem {
  void print () {
     System.out.print("Hello");
  }
  static void main (String[] args) {
     new HelloSystem().print();
  }
}
```

Feature code — Beautiful
```
aspect Beautiful {
  after(): call(HelloSystem.print()) {
     System.out.print("␣Beautiful");
  }
}
```

Feature code — Planet
```
aspect Planet {
  after(): call(HelloSystem.print()) {
     System.out.print("␣Planet");
  }
}
```

(b) Base model + code fragments for selected features

Fig. 4: Positive (parametric) variability in solution space.

constructed for this one product variant (in solution space). In the next section, we will show how we can use the enumerative variability defined by the feature model (in problem space) to construct *all* the products in a family at once, and not just one at a time (in solution space).

III. Enumerative Variability in Solution Space

In parametric variability, variation points in the feature model are not implemented. A variation point specifies multiple variants: 2 for *optional*(F), n for *alternative*$(F_1, \ldots, F_n)$, and $(2^n - 1)$ for *or*$(F_1, \ldots, F_n)$, where Fs denote features. In parametric variability, the programmer can configure and implement only one variant at a time. If we could implement variation points in their entirety, i.e. all the variants together, then we would be able to construct code for all the variants at once; that is, we would be able to adopt enumerative variability, as opposed to parametric variability, in solution space. In this section, we present an approach for doing so. As far as we know, this has not been done before, and in Section V we will discuss its pros and cons.

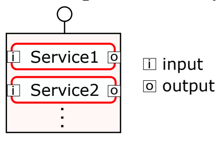
Our approach is to use a *component model* [19] that can be used to model and construct product families, rather than single products. We have defined and implemented such a component model, called FX-MAN [20], as a Model-driven Engineering tool [21] for developing families of component-based systems.[1] Here we will show how FX-MAN can be used to (model and) implement enumerative variability in solution space.

[1]A full account of FX-MAN is not necessary here.

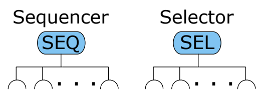
A. Features

In parametric variability approaches, features identified in the feature model (in problem space) are implemented as code fragments (in problem space). The mapping of features to code fragments is m-to-n, $m, n \geq 1$, in general, reflecting the fact that a feature may be implemented by multiple fragments whilst a code fragment may appear in the implementation of multiple features [22]. For example, annotative approaches like pure::variants usually use a 1-to-n, $n > 1$, mapping, whilst FOP and AOP use a 1-to-1 mapping. Obviously, a 1-to-n mapping is easier to manage than a m-to-n mapping, but not as easy to manage as a 1-to-1 mapping. Conversely a 1-to-1 mapping is not as easy to define as a 1-to-n or m-to-n mapping [23]. More crucially, any mapping must be accompanied by a strategy to deal with feature interaction [24] when features and their corresponding code fragments are combined.

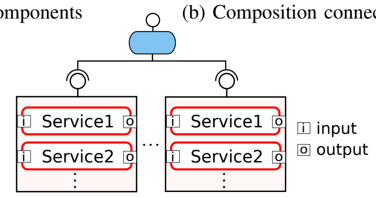
Using FX-MAN, code fragments can be implemented by either *components* or *services* provided by components. A component in FX-MAN provides multiple services (Fig. 5a), which are implemented by its methods. Mapping a feature



(a) Components



(b) Composition connectors



(c) Composite components

Fig. 5: FX-MAN: Components and composition connectors.

1-to-1 to a component would achieve a 1-to-1 mapping, but since a component contains n services, it would mean a 1-to-n mapping between features and services. We choose a 1-to-1 mapping. Obviously if all components have only one service each, then components and services coincide.

For the *Hello World* example, we can indeed map features to one-service components, as shown in Fig. 6.

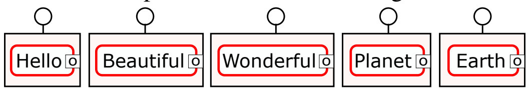


Fig. 6: *Hello World*: Feature-to-service mapping.

In FX-MAN, components can be composed into composites (Fig. 5c) by composition operators, defined as connectors (Fig. 5b). These operators initiate control and invoke services in components. They also coordinate control flow and service invocations between components. A sequencer defines *sequencing*: *SEQ*$(C_1(S_1), \ldots, C_n(S_n))$ calls service S_1 in component C_1, …, service S_n in component C_n sequentially. A selector defines *branching*: $SEL(C_1(S_1), \ldots, C_n(S_n))$ selects just one service to call, depending on a selection condition. Thus component composition connectors allow us to combine components and hence their (methods and) services. For example, the composite in Fig. 5c will combine the (methods and) services into another set of services (not shown here). For the purposes of this paper, we do not need to consider how the services of a composite are formed. The important thing is the fact that the composition operators are algebraic, i.e. composite components are the same type as their sub-components, and composition is therefore strictly hierarchical. The hierarchical composition enables us to model variation points, as well as the composition of product families.

B. Variation Points

Fig. 7 shows the different levels of composition in FX-MAN. At the lowest level of composition are (possibly composite)

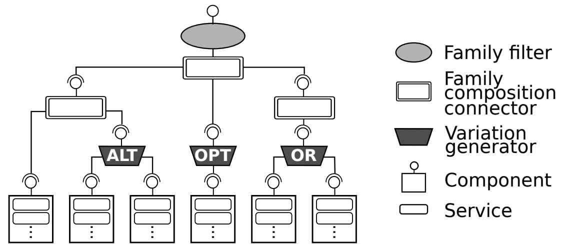


Fig. 7: FX-MAN: Levels of composition.

components. We can use these to implement (leaf) features in a feature model, as we have done in Fig. 6.

The next level of composition in FX-MAN offers *variation generators*. These operators take sets of components as input and produce permuted sets of components, i.e. variations thereof. Variation operators are therefore also algebraic and hierarchical. FX-MAN implements the full range of standard variations, viz. *optional*, *alternative* and *or*.

For the Hello World example, we can simply apply the variation operators for the variation points defined in its feature model, as shown in Fig. 8. The results of these variation

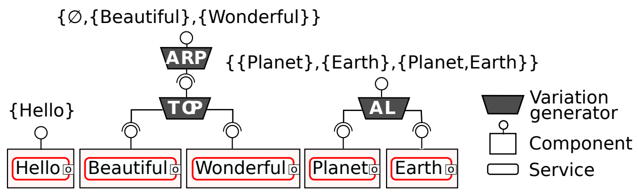


Fig. 8: Hello World: Variation points.

operators are of course: *optional*(*alternative*(BEAUTIFUL, WONDERFUL)), and *or*(PLANET, EARTH) respectively, as specified in the feature model. Precisely, they are the following sets: $\{\emptyset, \{\text{BEAUTIFUL}\}, \{\text{WONDERFUL}\}\}$ and $\{\{\text{PLANET}\}, \{\text{EARTH}\}, \{\text{PLANET}, \text{EARTH}\}\}$.

C. Family Composition

A set of components generated by a variation point is of course a family of products. So at the next level of composition in FX-MAN, family composition takes place, by means of family composition operators, also defined as connectors. These operators are defined in terms of the component composition operators, i.e. a family composition connector, e.g. *F-SEQ*, applies the corresponding component composition connector, e.g. *SEQ*, to corresponding components in the families. The result of a family composition is also a family, so this level of composition is also algebraic.

Which family composition operator to use is a design choice to satisfy the requirements. For the *Hello World* example, the simple family connector *F-SEQ* can meet the requirements, so we can apply it to the families in Fig. 8 to compose the whole family for the *Hello World* example. The whole family is shown in Fig. 9. The family filter on the top filters out

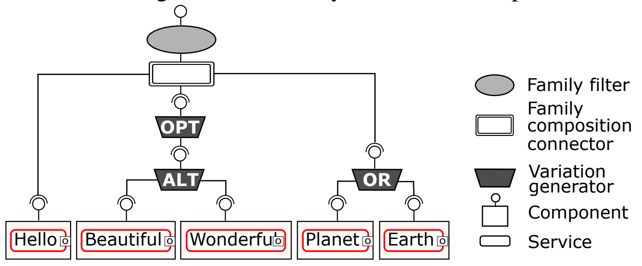


Fig. 9: Hello World: Product family.

invalid variants from the family, i.e it implements the cross-tree constraint, in this case 'PLANET *requires* ATTRIBUTE'.

D. Feature Interaction

In Fig. 9 we have assumed that there is no feature interaction. In case of feature interaction, we can still use FX-MAN to implement enumerative variability, but not in an elegant way. First, for every feature interaction, we have to create an optional component with a service mapped to the interaction feature. Then we need to add constraints to the family filter so that in any configuration in which interacting features are selected, the optional component is chosen instead of those for the original features.

So, if for example, PLANET and EARTH interact as described before, then the product family for '*Hello World*' is as shown in Fig. 10. The constraints that have to be added to the

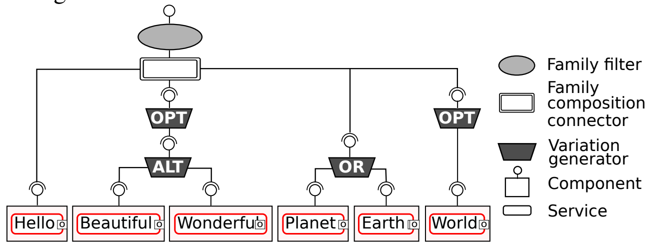


Fig. 10: Hello World: Product family with feature interaction.

family filter are: 'WORLD *excludes* PLANET, EARTH'; 'PLANET *excludes* WORLD'; 'EARTH *excludes* WORLD'

IV. IMPLEMENTATION EXAMPLE

In this section we show evidence of implementation of enumerative variability as discussed in the previous section.

The implementation is done using the FX-MAN tool. For legibility we will show cropped screenshots.

First, Fig. 11 shows the implementation of the '*Hello World*' family without feature interaction (Fig. 9). The small disc in

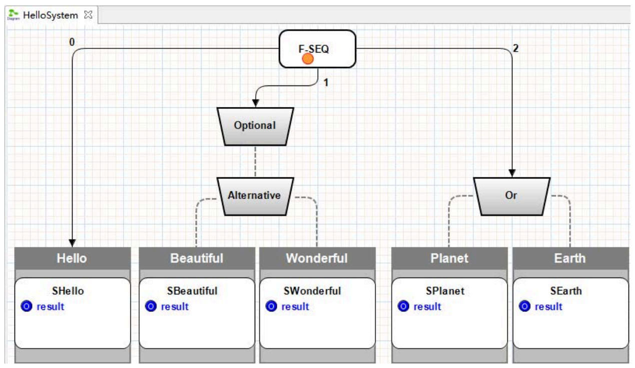
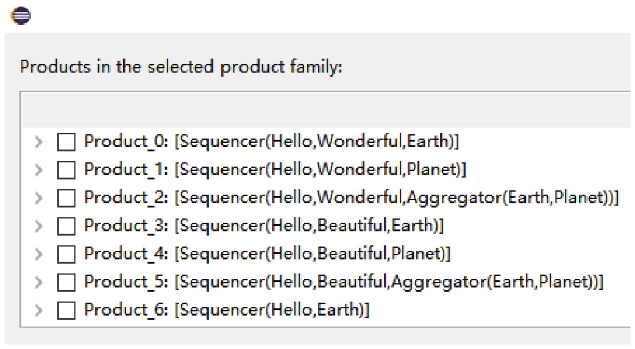


Fig. 11: Hello World without feature interaction.

the top-level family composition connector *SEQ* represents constraints in the family filter (first two constraints in Fig.15). Fig. 12 shows the products in this family listed in the Product Explorer of the FX-MAN tool. The *Aggregator* operator in



Products in the selected product family:

- Product_0: [Sequencer(Hello,Wonderful,Earth)]
- Product_1: [Sequencer(Hello,Wonderful,Planet)]
- Product_2: [Sequencer(Hello,Wonderful,Aggregator(Earth,Planet))]
- Product_3: [Sequencer(Hello,Beautiful,Earth)]
- Product_4: [Sequencer(Hello,Beautiful,Planet)]
- Product_5: [Sequencer(Hello,Beautiful,Aggregator(Earth,Planet))]
- Product_6: [Sequencer(Hello,Earth)]

Fig. 12: Product family in Fig. 11.

Product_2 and Product_5 is the set union operator ∪, resulting from an *or* variation point.

Next, Fig. 13 shows the implementation of the '*Hello World*' family with feature interaction (Fig. 10). The products in

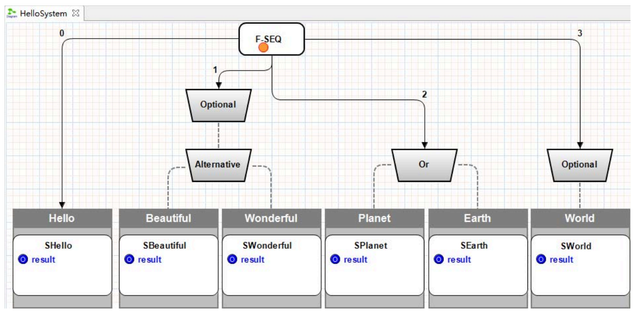
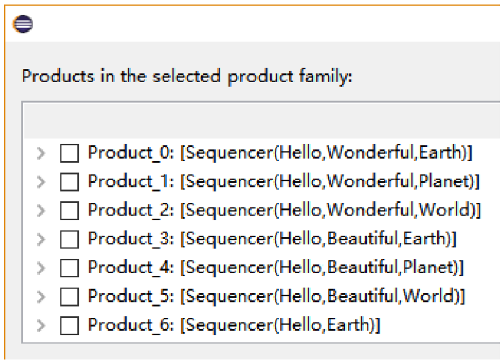


Fig. 13: Hello World with feature interaction.

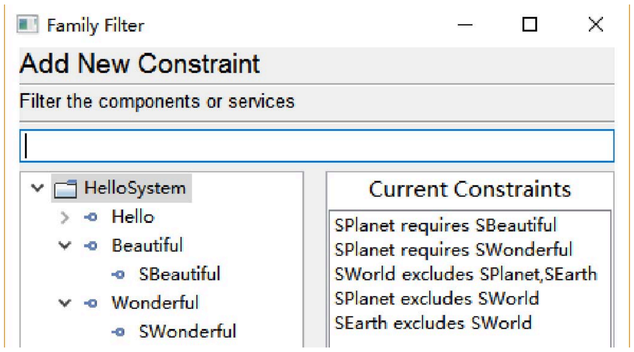
the family are shown in the Product Explorer in Fig. 14. The constraints added to the family filter for feature interaction can be seen in Fig. 15 (last three constraints).



Products in the selected product family:

- Product_0: [Sequencer(Hello,Wonderful,Earth)]
- Product_1: [Sequencer(Hello,Wonderful,Planet)]
- Product_2: [Sequencer(Hello,Wonderful,World)]
- Product_3: [Sequencer(Hello,Beautiful,Earth)]
- Product_4: [Sequencer(Hello,Beautiful,Planet)]
- Product_5: [Sequencer(Hello,Beautiful,World)]
- Product_6: [Sequencer(Hello,Earth)]

Fig. 14: Product family in Fig. 13.



SPlanet requires SBeautiful
SPlanet requires SWonderful
SWorld excludes SPlanet,SEarth
SPlanet excludes SWorld
SEarth excludes SWorld

Fig. 15: Hello World with feature interaction: Family filter.

V. Discussion and Conclusion

In this paper we have focussed on enumerative variability. Current SPLE techniques use enumerative variability only in problem space. We have shown that it is possible to use it in solution space too. Of course, we have only explained the idea and demonstrated its feasibility. It remains to analyse the merits and demerits of the idea and its realisation.

In principle, enumerative variability enables the products in a family to be constructed all at once. However, in practice, whether this can be realised effectively, and whether it will bring benefits, is still an open question. From a customer's point view, constructing all products at once may seem like overkill, since each customer usually wants only one product, and this is what current SPLE techniques construct (using parametric variability). From the perspective of these techniques, generating all the products in a large-scale family with a high degree of variability, is NP-hard, in terms of computation and memory costs. So, whether/how enumerative variability can help with current SPLE practice needs to be investigated.

One area where enumerative variability could potentially bring advantages is product line (i.e. product family) testing [25]. Family-based testing should be more efficient than product-based testing, since product-based testing inevitably performs inefficient, redundant computations, due to similarities and commonalities between the products [26]. However, hitherto research on family-based testing has been limited to discussion at the levels of specifications and models (see e.g. [27]–[29]), i.e. only in problem space. The simple reason is that no SPLE approach so far constructs the complete product family in solution space. So, currently only product-based testing (e.g. [30], [31]) is performed (in solution space).

With enumerative variability, family-based testing should become possible. Suitable testing techniques based on enumerative variability should be investigated. It will not at all be surprising if enumerartive variability enables many of the current product-based testing techniques to be adapted for efficient family-based testing.

Finally, in this paper we have used the FX-MAN component model to demonstrate the feasibility of using enumerative variability in solution space. However, FX-MAN is not ideal for this purpose. This is not surprising since FX-MAN was originally designed for families of general component-based systems. One inefficiency in FX-MAN is the way the family filter is defined and implemented: it could be improved so that the filtering can be done before or during a family composition, rather than only after the whole family has been constructed. Perhaps using a filter is just not efficient, and a better alternative should be found. In conclusion, we have used the current form of FX-MAN, to demonstrate the feasibility of implementing enumerative variability, but in future it is worth devising a better component model than FX-MAN for this purpose.

REFERENCES

[1] K. Pohl, G. Böckle, and F. J. van Der Linden, *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media, 2005.

[2] K. Czarnecki and U. Eisenecker, *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., 2000.

[3] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," Software Engineering Institute, Carnegie-Mellon University, Tech. Rep. CMU/SEI-90-TR-021, 1990.

[4] A. S. Karataş, H. Oğuztüzün, and A. Doğru, "Mapping extended feature models to constraint logic programming over finite domains," in *International Conference on Software Product Lines*. Springer, 2010, pp. 286–299.

[5] J. P. Gibson, "Feature requirements models: Understanding interactions." in *FIW*, 1997, pp. 46–60.

[6] I. Groher and M. Voelter, "Expressing feature-based variability in structural models," in *In Workshop on Managing Variability for Software Product Lines*. Citeseer, 2007.

[7] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh, "Form: A feature-oriented reuse method with domain-specific reference architectures," *Annals of Software Engineering*, vol. 5, no. 1, pp. 143–168, 1998.

[8] T. Berger, R. Rublack, D. Nair, J. Atlee, M. Becker, K. Czarnecki, and A. Wasowski, "A survey of variability modeling in industrial practice," in *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*. ACM, 2013, p. 7.

[9] D. Beuche, "Modeling and building software product lines with pure::variants," in *Proceedings of the 16th International Software Product Line Conference-Volume 2*. ACM, 2012, pp. 255–255.

[10] C. Krueger and P. Clements, "Systems and software product line engineering with biglever software gears," in *Proceedings of the 17th International Software Product Line Conference co-located workshops*. ACM, 2013, pp. 136–140.

[11] J. Liebig, S. Apel, C. Lengauer, C. Kastner, and M. Schulze, "An analysis of the variability in forty preprocessor-based software product lines," in *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, vol. 1. IEEE, 2010, pp. 105–114.

[12] T. Asikainen, T. Soininen, and T. Männistö, "A Koala-Based Approach for Modelling and Deploying Configurable Software Product Families," in *Software Product-Family Engineering*. Springer, 2004, pp. 225–249.

[13] E. M. Dashofy, A. v. d. Hoek, and R. N. Taylor, "A comprehensive approach for the development of modular software architecture description languages," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 14, pp. 199–245, 2005.

[14] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *Proceedings of ECOOP'97*. Springer, 1997, pp. 220–242.

[15] I. Groher and M. Voelter, "Xweave: models and aspects in concert," in *Proceedings of the 10th international workshop on Aspect-oriented modeling*. ACM, 2007, pp. 35–40.

[16] S. Apel, T. Leich, and G. Saake, "Aspectual feature modules," *Software Engineering, IEEE Transactions on*, vol. 34, no. 2, pp. 162–180, 2008.

[17] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella, "Delta-oriented programming of software product lines," in *Software Product Lines: Going Beyond*. Springer, 2010, pp. 77–91.

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

[19] K.-K. Lau and S. di Cola, *An Introduction to Component-based Software Development*. World Scientific, 2017.

[20] S. di Cola, C. Tran, K.-K. Lau, C. Qian, and M. Schulze, "A component model for defining software product families with explicit variation points," in *Proceedings of 19th International ACM SIGSOFT Symposium on Component-Based Software Engineering*. ACM, April 2016, pp. 79–84.

[21] S. di Cola, K.-K. Lau, C. Tran, and C. Qian, "An MDE tool for defining software product families with explicit variation points," in *Proceedings of the 19th International Conference on Software Product Line*. ACM, 2015, pp. 355–360.

[22] P. Sochos, M. Riebisch, and I. Philippow, "The feature-architecture mapping (FArM) method for feature-oriented development of software product lines," in *Engineering of Computer Based Systems, 2006. ECBS 2006. 13th Annual IEEE International Symposium and Workshop on*. IEEE, 2006, pp. 9–pp.

[23] P. Sochos, I. Philippow, and M. Riebisch, "Feature-oriented development of software product lines: mapping feature models to the architecture," in *Net. ObjectDays: International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World*. Springer, 2004, pp. 138–152.

[24] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec, "Feature interaction: a critical review and considered forecast," *Computer Networks*, vol. 41, no. 1, pp. 115–141, 2003.

[25] R. Kolb and D. Muthig, "Challenges in testing software product lines," in *Proceedings of the 7th Conference on Quality Engineering in Software Technology, CONQUEST*. Fraunhofer Publica, 2003, pp. 81–95.

[26] M. Jaring, R. L. Krikhaar, and J. Bosch, "Modeling variability and testability interaction in software product line engineering," in *Composition-Based Software Systems, 2008. ICCBSS 2008. Seventh International Conference on*. IEEE, 2008, pp. 120–129.

[27] L. Etxeberria and G. Sagardui, "Product-line architecture: New issues for evaluation," in *International Conference on Software Product Lines*. Springer, 2005, pp. 174–185.

[28] A. Bertolino and S. Gnesi, "Pluto: A test methodology for product families," in *International Workshop on Software Product-Family Engineering*. Springer, 2003, pp. 181–197.

[29] S. Kang, J. Lee, M. Kim, and W. Lee, "Towards a formal framework for product line test development," in *Computer and Information Technology, 2007. CIT 2007. 7th IEEE International Conference on*. IEEE, 2007, pp. 921–926.

[30] G. Perrouin, S. Sen, J. Klein, B. Baudry, and Y. Le Traon, "Automated and scalable t-wise test case generation strategies for software product lines," in *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*. IEEE, 2010, pp. 459–468.

[31] A. Hervieu, B. Baudry, and A. Gotlieb, "Pacogen: Automatic generation of pairwise test configurations from feature models," in *Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on*. IEEE, 2011, pp. 120–129.