Feature-Oriented Component-Based Development of Software Product Families: A Case Study

Chen Qian and Kung-Kiu Lau

School of Computer Science
The University of Manchester
Kilburn Building, Oxford Road, Manchester, United Kingdom, M13 9PL
Email: chen.qian,kung-kiu.lau@manchester.ac.uk

***Abstract*—*Feature-Oriented Software Development* (FOSD) is widely used in *Software Product Line Engineering* (SPLE). FOSD constructs product families by incremental feature implementations. In this paper, we introduce a feature-oriented component-based approach, which implements features as an encapsulated components for further family modelling. A case study of elevator systems is also presented to describe the use of our approach.**

***Keywords*–SPLE; FOSD; CBD; Enumerative variability.**

I. Introduction

Software Product Line Engineering (SPLE) traditionally proceeds in two phases: domain engineering and (ii) application engineering [1]. In the domain engineering phase, existing SPLE approaches (i) usually use a *feature model* to specify variability, and (ii) from these, identifies and implements domain artefacts, e.g., a code base [2]. In the application engineering phase, SPLE (i) creates product configurations, and (ii) assembles one product at a time from the domain artefacts based on its configuration [3].

The key abstraction of *Feature-Oriented Software Development* (FOSD) is a *feature*, which represents a logical unit of behaviour specified by a set of functional requirements [4]. FOSD aims at constructing software product families by incremental feature implementations [5]. Thus, variability can be traced from features directly to the domain artefacts, which promises the manageability and maintainability of product families.

In this paper, we present a feature-oriented approach to construct product families in a component-based manner, i.e., following a component model [6]. In [7], we elaborated the principles of our component model and discussed the feasibility of using it in SPLE. Hence, this paper discusses the concrete details. In Section II, we show the essential background knowledge and related work. In Section III, we present a case study to exemplify how to construct a software product family step-by-step from scratch by our approach and tool. Finally, in Section IV, we finish the paper by conclusion of our work and discussion of the future work.

II. Background and Related Work

The essence of FOSD is to model and implement variable domain artefacts, and each of them must be mapped onto a non-mandatory feature. In general, three main categories of variability mechanisms are adopted in FOSD approaches and tools [8]: (i) *annotative*, e.g., CIDE [2], FORM's macro language [9] (ii) *compositional*, e.g., AHEAD [10], FeatureC++ [11] and (iii) *transformational*, e.g., Δ-MontiArc [12], Delta-Oriented Programming (DOP) [13].

Annotative approaches usually build a single, superimposed model, namely 150% model, to represent all product variants. The variable features are implemented as code fragments with annotations, i.e., boolean feature expressions. Subsequently, in order to generate a product, code fragments corresponding to unselected features have to be removed according to the product configuration. By comparison, compositional and transformational approaches develop code fragments isolated from the base programs. Compositional approaches add the fragments correlating with the selected features to the base program for product generation. In regard to transformational approaches, fragments are not only added to the base model, but also modified the existing code under some circumstances.

There are two kinds of variability in current FOSD approaches, as known as negative and positive variability [14]. The former is adopted by annotative approaches, whereas the latter is used by compositional approaches. But both of them are used in transformational approaches. However, no matter negative or positive, we consider such a variability as a *parametric variability*, due to it is parameterised on the presence or absence of features in a single product. Thus, SPLE approaches using parametric variability can only generate one product at a time. On contrary, *enumerative variability* includes all valid variants directly [7]. For example, in the problem space, feature model defines enumerative variability, while a configuration model defines parametric variability. In this paper, our approach construct enumerative variability in the solution space, which results in a whole product family and therefore all products can be generated in one go.

Another area where our approach could bring advantages is feature mapping. The early work on SPLE, such as FODA [15], did not represent features explicitly, instead build n-to-m mappings between features and domain artefacts, which causes severe tangling and scattering in the code base eventually. Hence, the construction of product families are infeasible. FOSD have made a great progress by bringing a distinguishing property that aims at 1-to-n feature mappings. However, it becomes obvious that the ideal mapping is 1-to-1 [16], but it is difficult to be achieved in current FOSD approaches, mostly because of the cross-cutting concern of features. Our approach is capable to build 1-to-1 mappings between features

Copyright (c) IARIA, 2018. ISBN: 978-1-61208-668-2

and components. In this paper, we show how to deal with the cross-cutting problem in the case study.

Regarding to the outstanding maintainability and reusability, *component-based development* (CBD) is another paradigm that seems suitable for SPLE. But earlier researches certified that constructing product families only using CBD is barely feasible, due to features often do not align well with the decomposition imposed by component models [5]. Some researches [17], [18] try to integrate CBD and FOSD in order to obtain both their advantages in SPLE, but problems still occurs. In this paper, our approach adopts a state-of-the-art component model that partners FOSD very well.

III. A Case Study

We have developed a web-based graphical tool that implements our component model and constructs product families [19]. The graphical user interface (GUI) is realised using *HTML5* and *CSS3*, whereas the functionality is implemented using JavaScript. In particular, we adopt the latest edition of *ECMAScript* as *JavaScript* specification since its significant new syntax, including classes and modules, supports complex applications. Additionally, we import *jQuery*, the most widely deployed JavaScript library, to improve code quality and enhance system extensibility. For the purpose of user-friendliness, all building blocks, including constraints and interaction, can be easily added through buttons and dialogue boxes.

In this section, we use an example of the elevator product family, which originates from [20], developed by Feature-Oriented Programming (FOP) in *FeatureIDE* [21]. The elevator contains a control logic mode that can be either `Sabbath` or `FIFO`, and an optional feature called `Service`. In `Sabbath` mode, the elevator reaches all levels periodically without user input, whereas in `FIFO` mode, the elevator moves to specific floors in turns according to the requests. The `Service` is special, it allows authorised persons to send the elevator to the lowest floor. Notably, `Service` is a *cross-cutting feature*, as its implementation scatters across other features' (`Sabbath` and `FIFO`) implementations. Consequently, the behaviour of `Service` can be triggered at any time, and on any mode, during elevator running period.

We will implement the elevator family in our tool, by extending it with the 3 features one at a time. Notably, for the clarification, we omit data channels in the figures in this paper. The design and implementation process is identical to the original example. Therefore, we can evaluate our approach based on the scientific control.

A. Adding Feature "Sabbath" to the Elevator Product Line

We add `Sabbath` as an optional child feature of the root, as shown in Figure 1. Then we need to implement a component for it. It is worth noting the underlying component model is already described in [6] and [7].

According to the requirement analysis of `Sabbath`, we can identify 3 behaviours behind it. Figure 2 shows the *composite component* composed by several *atomic components* and *composition connectors*. For example, *MovingUp* controls the elevator to move one floor up and returns the next direction, while *MovingDown* makes the opposite move. Contrariwise, *Flooring* leaves the elevator in the current floor. The *selector*

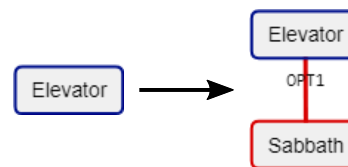

Figure 1. Adding `Sabbath` to the feature model.

SELs define branching depending on selection conditions, whereas the *sequencer SEQ* defines sequencing sequentially. As a result, the elevator changes direction when it reaches bottom floor and top floor.

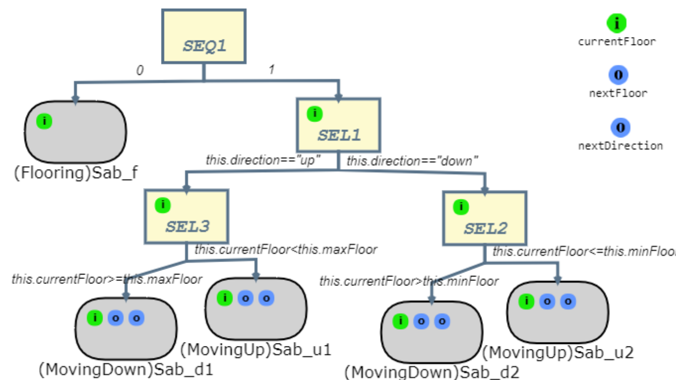

Figure 2. Component *Sabbath*.

After all, the elevator moves one floor up or down for each execution of `Sabbath`. In order to keep the elevator running, we only need to apply an *adaptor*, called *loop*, to repeat the control to this component (not discussed in detail here). Figure 3 shows the transition systems of the `Sabbath` component within the elevator product, which gives us a clear vision of the behaviour. So far no cross-cutting occurs, due to only one product exists.

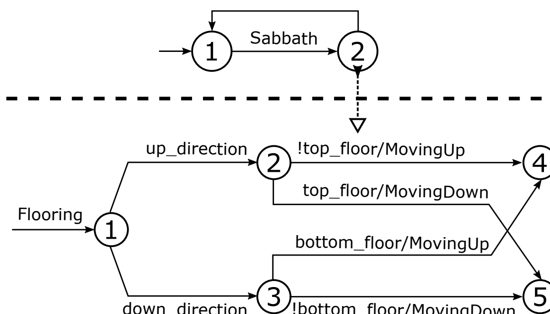

Figure 3. Transition systems of Elevator and *Sabbath*.

B. Adding Feature "Service" to the Elevator Product Line

Now, we add `Service` feature to the elevator product family. As this feature has no functional dependencies with `Sabbath`, we put it under the root. In addition, `Service` feature is not always required by every product, thus we set it optional. Figure 4 shows the change of the feature model.

We can reuse two components implemented before: *MovingDown* and *Flooring*. But in order to construct *Service* component, we need to implement another, namely *StopService*, which allows the authorised persons to deactivate the

Copyright (c) IARIA, 2018. ISBN: 978-1-61208-668-2

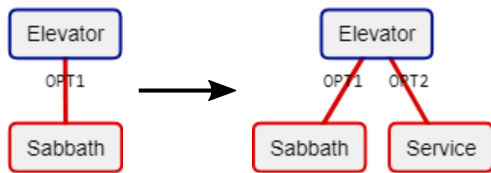

Figure 4. Adding `Service` to the feature model.

`Service` functionality after the elevator reaches the bottom floor. Figure 5 shows the construction of *Service* component.

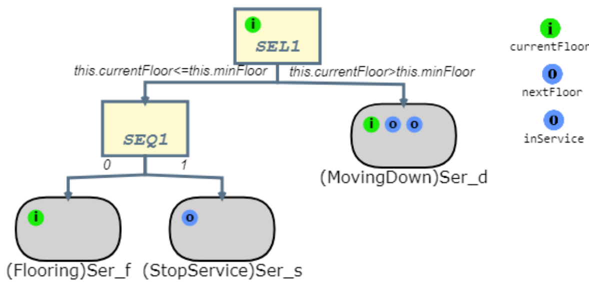

Figure 5. Component *Service*.

Next, we need to apply *variation generators* to the components, which are mapped onto the variation points specified in the feature model (Figure 4). A variation generator generates multiple variants: it takes (a set of) sets of components as input and produces (a set of) permuted sets of components, i.e., variants. We have implemented variation generators for the full range of standard variation points, viz. *optional*, *alternative* and *or* (respectively OPT, ALT and OR. Notably, the components are algebraic and hierarchical at this level, which means the variation generator can be nested.

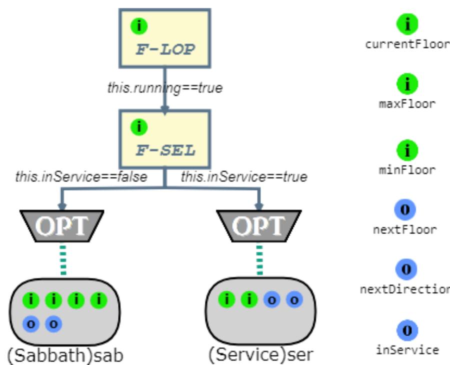

Figure 6. Elevator product family with *Service*.

At the next level of composition in our component model, family composition takes place, by means of family composition operators, also defined as connectors. A family composition operator is applied to multiple input component sets to yield a set of product variants, i.e., a (sub)family of products. These operators are defined in terms of the component composition operators: a family composition connector forms the Cartesian product of its input sets, and composes components in each element of the Cartesian product using the corresponding component composition connector. For example, in Figure 6, the family composition connector *F-SEL* applies the corresponding component composition connector *SEL* to components in each element of the Cartesian product, as well as the *F-LOP* indicates *LOP* (loop). The result of a family composition is thus also a family, so this level of composition is also algebraic.

The choice of family composition connectors is a design decision that only depends on the functional requirements, however it will not affect the total number of products in the (sub)families. Figure 6 shows the elevator family with two optional features. Derived from the family model in Figure 6, Figure 7 shows a *featured transition system* (FTS) [22] of the family, which depicts the cross-cutting feature `Service` takes part in family behaviour, e.g., 1-2-3-1 workflow.

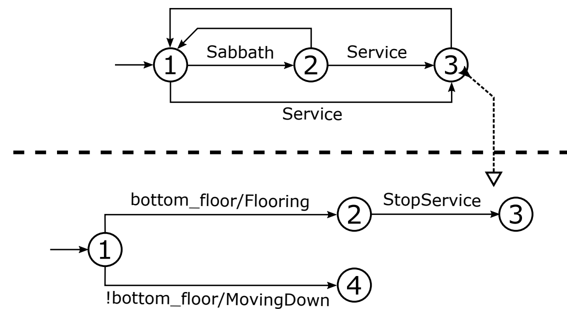

Figure 7. FTS of Elevator and *Service*.

At present, the elevator family generates 3 products, as shown in the *product explorer* in Figure 8. The product explorer enumerates all valid products in the form of variability resulting from each variation generator at any level of nesting. For each product, the user can examine its structure and built-in components, and hence compare this product with the corresponding variant derived from the feature model.

Elevator: 3 products

- Product 1: **LOOP**(**SEL**(*sab*, *ser*))
- Product 2: **LOOP**(*sab*)
- Product 3: **LOOP**(*ser*)

Figure 8. Product explorer (3 products).

C. Adding Feature "FIFO" to the Elevator Product Line

`FIFO` (first in, first out) is another control logic of the elevator. As the name suggests, if more than one floor requests exist, the oldest request is handled first, i.e., the elevator directly move to the required floor. By contrast with the feature `Service`, `FIFO` is an alternative to the already existing `Sabbath` mode. Therefore, we reform the feature model by adding features and modifying variation points, as Figure 9 shows.

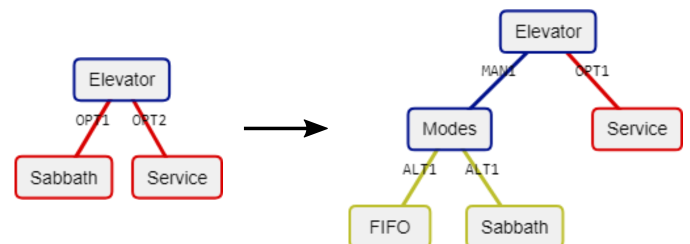

Figure 9. Adding `FIFO` to the feature model.

Likewise, we implement a component *FIFO* for feature `FIFO`. Figure 10 shows the architecture of the component,

Copyright (c) IARIA, 2018. ISBN: 978-1-61208-668-2

which is a composition of four sub-components. Two of them are newly prepared: *FloorQueue* and *RemoveRequest*. The former appends an incoming floor request to the end of the request queue, while the latter removes the first floor request from the queue.

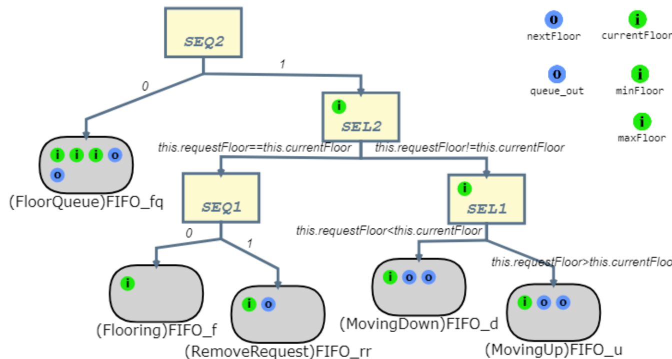

Figure 10. Component *FIFO*.

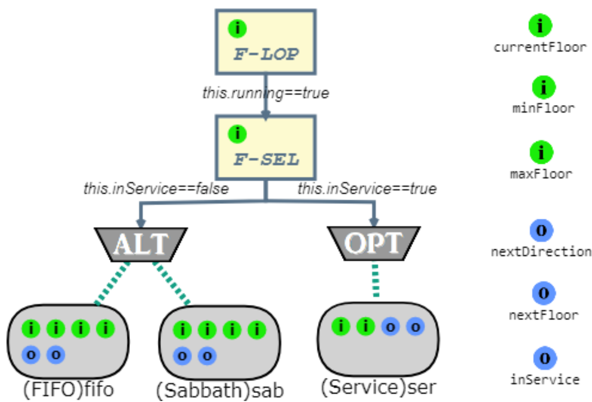

Figure 11. Elevator product family with *FIFO*.

After all leaf features are realised, now we can construct the new elevator product family. Since we have already presented *variation generators* and *family composition connectors* in Section III-B, the details of family construction are no longer described here. Figure 11 shows the latest elevator product family, which contains 4 valid products. The behaviour of the family is illustrated in Figure 12, in which we can see `Service` remains cross-cutting relationships with both `FIFO` and `Sabbath`. Finally, the products are enumerated in the product explorer in Figure 13.

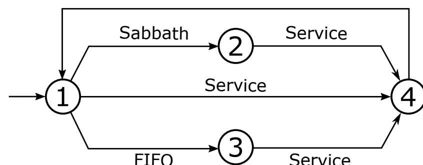

Figure 12. FTS of Elevator family.

D. Testing Elevator Product Line

Software product family testing comprises two related testing activities: domain testing and application testing. The former refers to domain engineering, whereas the latter refers to application engineering. Figure 14 depicts a V-model that describes the stages of domain engineering [23]. It involves 3 different testing types, each of which tests a product line at a specific level.

Elevator: 4 products

- Product 1: **LOOP**(**SEL**(*fifo*, *ser*))
- Product 2: **LOOP**(*fifo*)
- Product 3: **LOOP**(**SEL**(*sab*, *ser*))
- Product 4: **LOOP**(*sab*)

Figure 13. Product explorer (4 products).

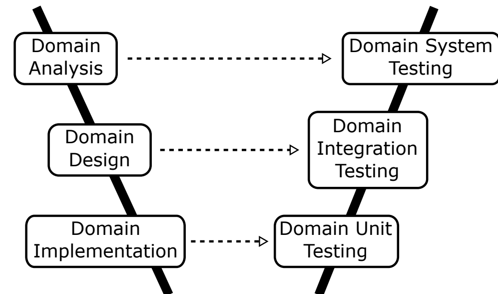

Figure 14. The V-Model for domain engineering.

Domain unit testing is easy to be operated for a feature-oriented architecture constructed in a component-based manner. In our approach, every component can be invoked via a *provided service*, which is a piece of behaviour (an input-output function) implemented by its methods. Therefore, the components, no matter atomic or composite, can be tested by the traditional techniques, e.g., structural testing. Our tool provides a workbench that all components can be executed directly. For example, we test component *MovingUp* within a simulation that controls an elevator to move from fifth floor to seventh floor. The result is demonstrated in Figure 15.

```
Going up...                VM197:8
Going up...                VM197:8
                           VM197:19
```

(index)	current	next	direction
0	5	6	"up"
1	6	7	"up"

Figure 15. Unit testing result of *MovingUp*.

Domain integration testing focuses on the testing of combinations of components. For most SPLE approaches, variability handling is a huge challenge in the integration testing phase, due to it heavily influences the components and their interactions [24]. Briefly, a variation point in the feature model may be modelled by multiple variation points scattered across a number of components, which results in too many component interactions for exhaustive testing. However, our approach may copes with the problem.

Firstly, the variability in family model is embodied by variation generators, which achieve a 1-to-1 mapping onto variation points in feature model, as well as the 1-to-1 feature mappings refers to the components. As we introduced earlier, the variability is *enumerative* in the final product family architecture. Notably, the product family architecture is isomorphic to the feature model. Thus, mismatched variability caused by invalid component interactions can be easily spotted in product

explorer. For example, in Figure 11, if we misplace an OR variation generator on the top of *FIFO* and *Sabbath* instead of ALT variation generator, then we can observe 5 products in the product explorer. That is obviously incorrect due to the feature model only gives 4 product variants in total.

Secondly, the component model adopted in our approach exposes a provided service, but no required service. Such components are known as *encapsulated components* [6]. There are no coupling between encapsulated components, i.e., they do not call others. Instead, they are invoked by composition connectors. Thereby, we do not need to worry about the faulty component compounds, i.e., the incorrect bound interfaces. Moreover, as we mentioned earlier, the level of family composition is algebraic, as it generates a set of encapsulated components, which can be tested directly.

In conclusion, the distinguished property of our component model makes domain integration testing convenient and crystal, especially for the software product families of non-trivial size. All we have to do is examining the 1-to-1 mappings between (i) (leaf) features and components, (ii) variation points and variation generators. The correctness of behaviours between components will be tested at the next level.

Domain system testing evaluate all products' compliance with their requirements. It becomes obvious that verifying the behaviour of each product individually is difficult due to an exponential number of combinations of assets is unmanageable [25]. Since by our approach, the control flows are clearly coordinated by exogenous connectors and the components are directly mapped onto features, every family model can derive a family-based functional model that describes the combined behaviour of an entire family, such as FTS [22] or *Featured Finite State Machine* (FFSM) [26]. In this section, we have generated FTS diagrams for different elevator families in Figure 7 and Figure 12. The FTS describes the overall behaviour of family, and hence the behaviours of every product within the family. For example, if we misplace a *F-SEQ* (family sequencer) instead of *F-SEL* (family selector) in the family model in Figure 11, then the derived FTS would not show the 1-4 workflow as in Figure 12. Therefore, we can detect the problems in family composition, due to the FTS does not describe the required behaviour.

Family-based functional model can help us to verify the behaviour of component interactions, but it is not straightforward to validate the execution results. However, we cannot order test executions for every single product. Thus, we should execute a test case in the whole product family, i.e., all configurations of the family, without actually generating a concrete product. In that case, a product line testing method, called *Variability-Aware Testing* [27], is perfect for our family model. The key step of variability-aware testing is to extract an abstract syntax tree (AST) with explicit variability. In our approach, the final family model has a tree structure with enumerative variability, so it provides a seamless migration to the testing model. The testing model of the elevator example is demonstrated in Figure 16 (not discussed in detail here because of the limited space).

E. Product Generation

In Section II, we have presented that our approach construct enumerative variability in the solution space. In other words, all valid products are defined during composition. Unlike conventional FOSD approaches, we can 'pick up' any number of products from the product explorer in one go, instead of one product at a time via configuration.

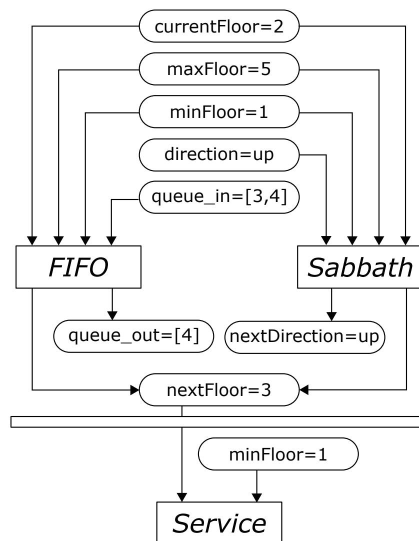

Figure 16. Variability-aware testing of Elevator.

Here, to exemplify, we choose product No.3 from the product explorer in Figure 13. Figure 17(a) shows the product architecture, which is executable, and Figure 17(b) expresses partial execution result.

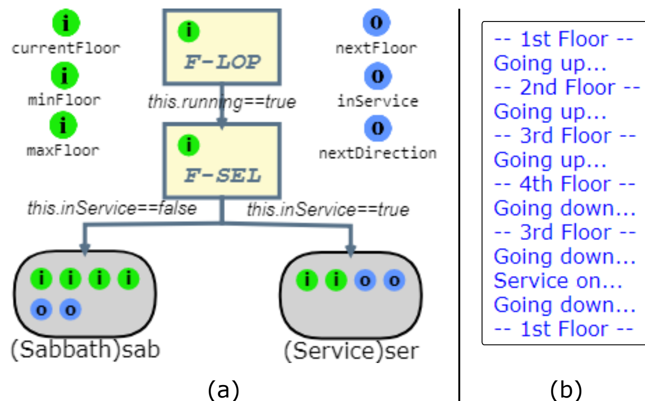

Figure 17. Product 3.

IV. Conclusion and Future Work

By comparison with the original elevator family implementation in [20], our implementation has many merits. For example, the original example is realised by FOP, in which every feature is mapped onto multiple code fragments scattering cross three classes. Each feature cannot be tested in isolation. Simply put, our approach provides better maintainability, manageability and testability because of the explicit 1-to-1 feature mappings. However, our family model only realises behaviour in the family. Contrariwise, in [20], FOP can define user interface for simulation, because as a low-level programming language, FOP can overwrite any code directly.

To provide step-by-step instructions of how to use our approach for family construction, we choose a small example for the simplicity. But our approach can be used for product families of non-trivial size. Besides the functions shown in

Copyright (c) IARIA, 2018. ISBN: 978-1-61208-668-2

Section III, our tool can deal with cross-tree constraints (e.g., '*require*', '*exclude*') among features, and among components. Our tool can also set cardinalities to narrow down massive families. Moreover, our tool can handle feature interaction problem, which becomes the most significant challenge in SPLE [28], by importing extra off-the-shelf components during composition. As a matter of fact, we have successfully constructed product families for industrial cases, i.e., consisting of dozens of features and hundreds of products. In future, we plan to present these results of our research.

According to [29], in the real world, many organisations adopt product families using three techniques: *proactive*, *reactive* and *extractive*. A proactive approach implies that a product family is modelled from scratch. In contrast, a reactive approach begins with a small, easy to handle product family, which can be incrementally extended with new features and artefacts. An extractive approach starts with a portfolio of existing products and gradually refactors them to construct a product family. At present, it is apparent that our work is proactive. However, recent researches [30], [31] in reverse engineering suggest that our work also has potential to support reactive and extractive techniques.

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] C. Kästner, S. Trujillo, and S. Apel, "Visualizing software product line variabilities in source code." in SPLC (2), 2008, pp. 303–312.

[3] K. Berg, J. Bishop, and D. Muthig, "Tracing software product line variability: From problem to solution space," 2005, pp. 182–191.

[4] J. Bosch, Design and use of software architectures: adopting and evolving a product-line approach. Pearson Education, 2000.

[5] S. Apel and C. Kästner, "An overview of feature-oriented software development." Journal of Object Technology, vol. 8, no. 5, 2009, pp. 49–84.

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

[7] C. Qian and K.-K. Lau, "Enumerative variability in software product families," in Computational Science and Computational Intelligence (CSCI), 2017 International Conference on. IEEE, 2017, pp. 957–962.

[8] A.-L. Lamprecht, S. Naujokat, and I. Schaefer, "Variability management beyond feature models," Computer, vol. 46, no. 11, 2013, pp. 48–54.

[9] K. C. Kang, J. Lee, and P. Donohoe, "Feature-Oriented Product Line Engineering," IEEE software, vol. 19, no. 4, 2002, pp. 58–65.

[10] D. Batory, J. Sarvela, and A. Rauschmayer, "Scaling step-wise refinement," IEEE Trans. Software Eng., vol. 30, no. 6, 2004, pp. 355–371.

[11] S. Apel, T. Leich, M. Rosenmüller, and G. Saake, "FeatureC++: on the symbiosis of feature-oriented and aspect-oriented programming," in Generative Programming and Component Engineering. Springer, 2005, pp. 125–140.

[12] A. Haber, T. Kutz, H. Rendel, B. Rumpe, and I. Schaefer, "Delta-oriented architectural variability using MontiCore," in Proceedings of the 5th European Conference on Software Architecture: Companion Volume. ACM, 2011, p. 6.

[13] 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.

[14] I. Schaefer, "Variability modelling for model-driven development of software product lines." VaMoS, vol. 10, 2010, pp. 85–92.

[15] 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.

[16] C. Kästner and S. Apel, "Feature-oriented software development," in Generative and Transformational Techniques in Software Engineering IV. Springer, 2013, pp. 346–382.

[17] W. Zhang, H. Mei, H. Zhao, and J. Yang, "Transformation from CIM to PIM: A feature-oriented component-based approach," in International Conference on Model Driven Engineering Languages and Systems. Springer, 2005, pp. 248–263.

[18] P. Trinidad, A. R. Cortés, J. Peña, and D. Benavides, "Mapping feature models onto component models to build dynamic software product lines." in SPLC (2), 2007, pp. 51–56.

[19] C. Qian, "Enumerative Variability Modelling Tool," http://www.cs.man.ac.uk/~qianc?EVMT, 2018, [Online; accessed 1-July-2018].

[20] J. Meinicke et al., "Developing an elevator with feature-oriented programming," in Mastering Software Variability with FeatureIDE. Springer, 2017, pp. 155–171.

[21] C. Kastner, T. Thum, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, and S. Apel, "FeatureIDE: A tool framework for feature-oriented software development," in Proceedings of 31st ICSE. IEEE, 2009, pp. 611–614.

[22] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin, "Model checking lots of systems: efficient verification of temporal properties in software product lines," in Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1. ACM, 2010, pp. 335–344.

[23] J. Lee, S. Kang, and D. Lee, "A survey on software product line testing," in Proceedings of the 16th International Software Product Line Conference-Volume 1. ACM, 2012, pp. 31–40.

[24] L. Jin-Hua, L. Qiong, and L. Jing, "The w-model for testing software product lines," in Computer Science and Computational Technology, 2008. ISCSCT'08. International Symposium on, vol. 1. IEEE, 2008, pp. 690–693.

[25] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake, "Analysis strategies for software product lines: A classification and survey," Software Engineering and Management 2015. Gesellschaft fr Informatik e.V., 2015, pp. 57–58.

[26] V. H. Fragal, A. Simao, and M. R. Mousavi, "Validated test models for software product lines: Featured finite state machines," in International Workshop on Formal Aspects of Component Software. Springer, 2016, pp. 210–227.

[27] C. Kästner et al., "Toward variability-aware testing," in Proceedings of the 4th International Workshop on Feature-Oriented Software Development. ACM, 2012, pp. 1–8.

[28] 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, 2003, pp. 115–141.

[29] C. Krueger, "Easing the transition to software mass customization," in Software Product-Family Engineering. Springer, 2002, pp. 282–293.

[30] R. Arshad and K.-K. Lau, "Extracting executable architecture from legacy code using static reverse engineering," in Proceedings of 12th International Conference on Software Engineering Advances. IARIA, 2017, pp. 55–59.

[31] R. Arshad and K.-K. Lau, "Reverse engineering encapsulated components from object-oriented legacy code," in Proceedings of The 30th International Conference on Software Engineering and Knowledge Engineering. KSI Research Inc., July 2018, pp. 572–577.

Copyright (c) IARIA, 2018. **ISBN: 978-1-61208-668-2**