An Enumerative Variability Modelling Tool for Constructing Whole Software Product Families

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*—Constructing a product family requires the formulation in problem space of a domain model (including a variability model) and its implementation in solution space. Current Software Product Line Engineering tools mostly aim to build an 'assembly line' for deriving one product at a time by assembling domain artefacts according to the variability model. Therefore, those tools support enumerative variability in problem space, but parametric variability in solution space. In this paper, we present a tool to model and implement enumerative variability in both spaces, and hence construct a whole product family in one go.**

Keywords–Enumerative Variability; Product Family Engineering; Web-based Tool.

I. INTRODUCTION

Current Software Product Line Engineering (SPLE) tools, e.g., pure::variants [1], AHEAD [2] and Clafer [3], construct a product family by using an 'assembly line' (product line). In the domain engineering phase, SPLE tools (i) construct a variability model in the problem space, and (ii) model and implement domain artefacts in the solution space, that can be used to assemble individual products. In the application engineering phase, SPLE tools assemble one product at a time from the domain artefacts in the solution space [4]. By contrast, we have defined an approach that constructs a whole product family in one go [5].

Existing SPLE tools usually define a feature model to specify variability in the domain engineering phase, and use a configuration model to specify a particular product variant in the application engineering phase. A feature model defines *enumerative variability*, as it includes all valid variants. A configuration model defines *parametric variability*, as it is parameterised on the presence/absence of features in a single product.

By contrast, we use enumerative variability in both the problem space and the solution space [5]. In Section II, we briefly introduce our product family engineering approach with the underlying component model. In Section III, we present a web-based tool that supports every step in our approach. In Section IV, we use an example to show how to construct a product family and derive products from the family by using our tool. Finally, in Section V, we finish our paper by conclusion of our work and discussion of the future work.

II. OVERVIEW OF OUR APPROACH

Starting from a feature model, we model and implement the enumerative variability defined by the feature model. Our approach is component-based, i.e., it follows a component model [6], [7]. We define a whole product family as a composition of variants of sets of components, as illustrated in Figure 1.

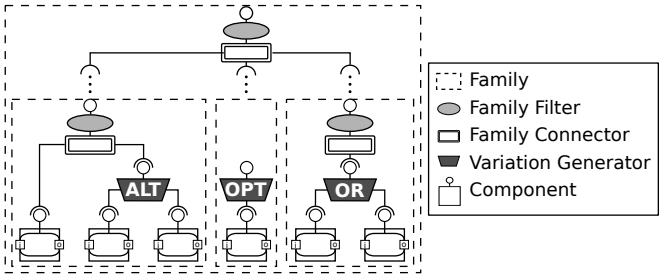


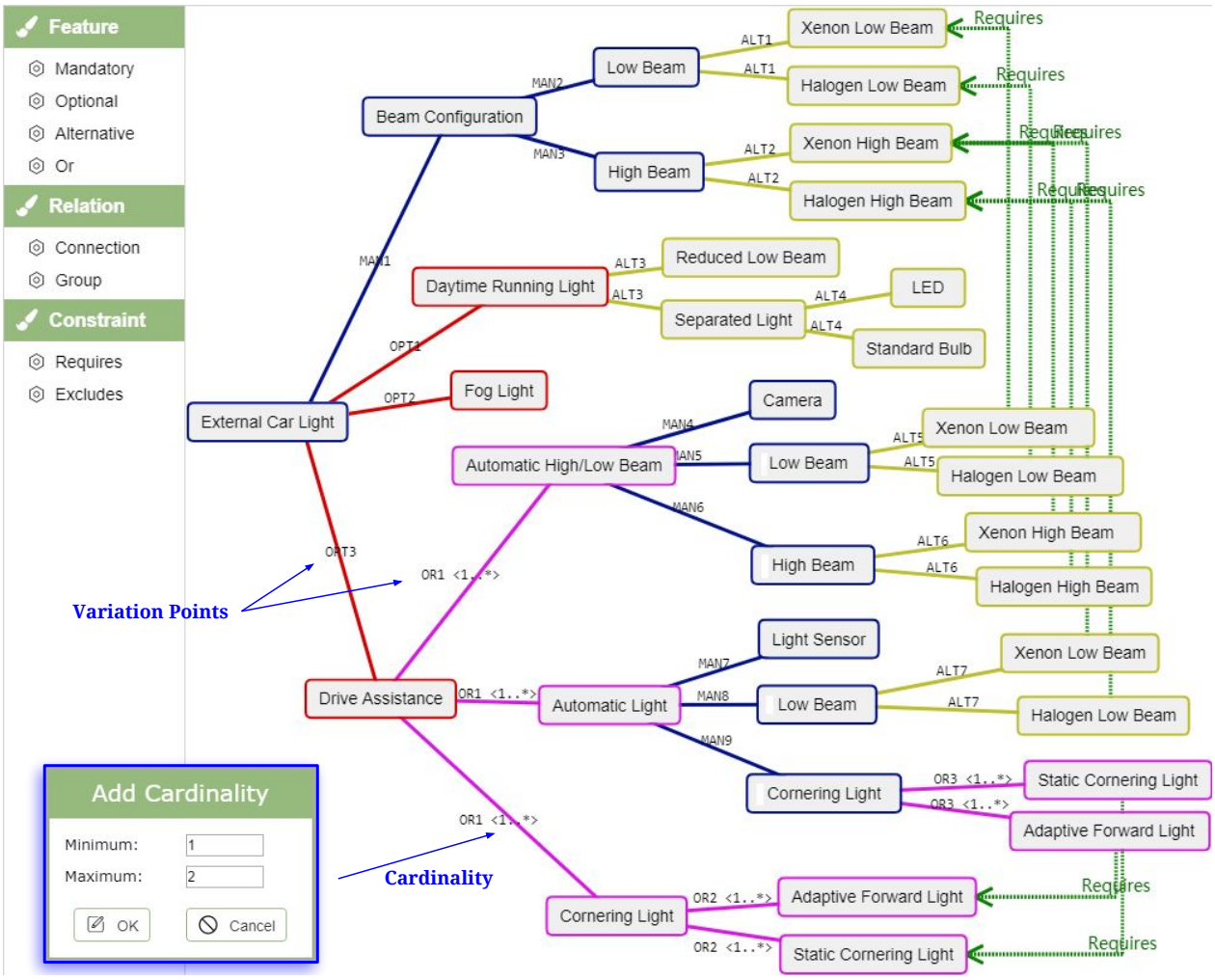
Figure 1. Component model: Levels of composition.

We proceed in three main stages, modelling and implementing (i) features, (ii) variation points, and (iii) product variants, respectively. Firstly, we construct components (atomic or composite) as the implementations of leaf features in the feature model. Notably, the abstract features aggregate behaviour corresponding to leaf features. A component is a software unit with a *provided service* (a lollipop in its interface) but no required services. Such a component is called an *encapsulated component* [7]. Then we apply *variation generators*, which model variation points in the feature model, viz. *optional*, *alternative* and *or* (respectively OPT, ALT and OR in Figure 1), and therefore generate sets of components as variations of the input set of components. So at the next level of composition, we apply *family composition connectors*. Each of them yields a set of product variants, i.e., a (sub)family of products, in the form of Cartesian product of its input sets, and composes components in each element of the Cartesian product using the corresponding component composition connector.

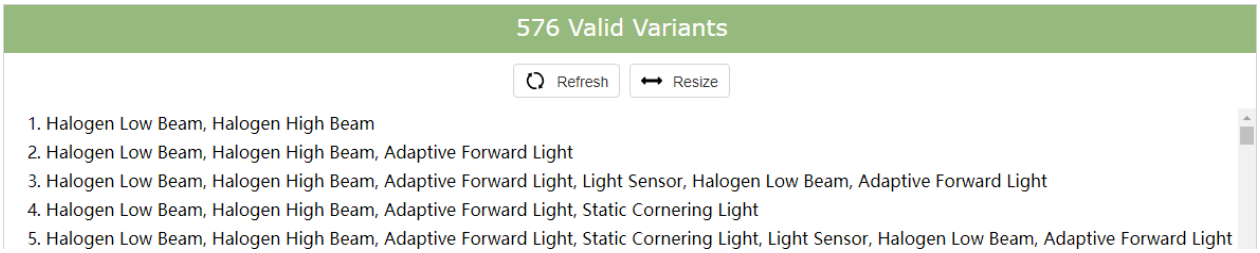
In our component model, the composition operators are algebraic, i.e., composite components are the same type as their sub-components, and composition is therefore strictly hierarchical. This important property enables us to model and implement the elements of a feature model (features, variation points, product variants) level by level.

Constraints in the feature model as well as feature interaction are dealt with by filters in family composition connectors. Invalid products are immediately removed from the Cartesian product of component sets produced by a family composition connector. We can also create new components for interacting features, and use them to replace original ones if feature

 ISBN: 978-1-61208-668-2



(a) Feature model.



(b) All 576 valid variants.

Figure 2. Canvas for constructing feature model.

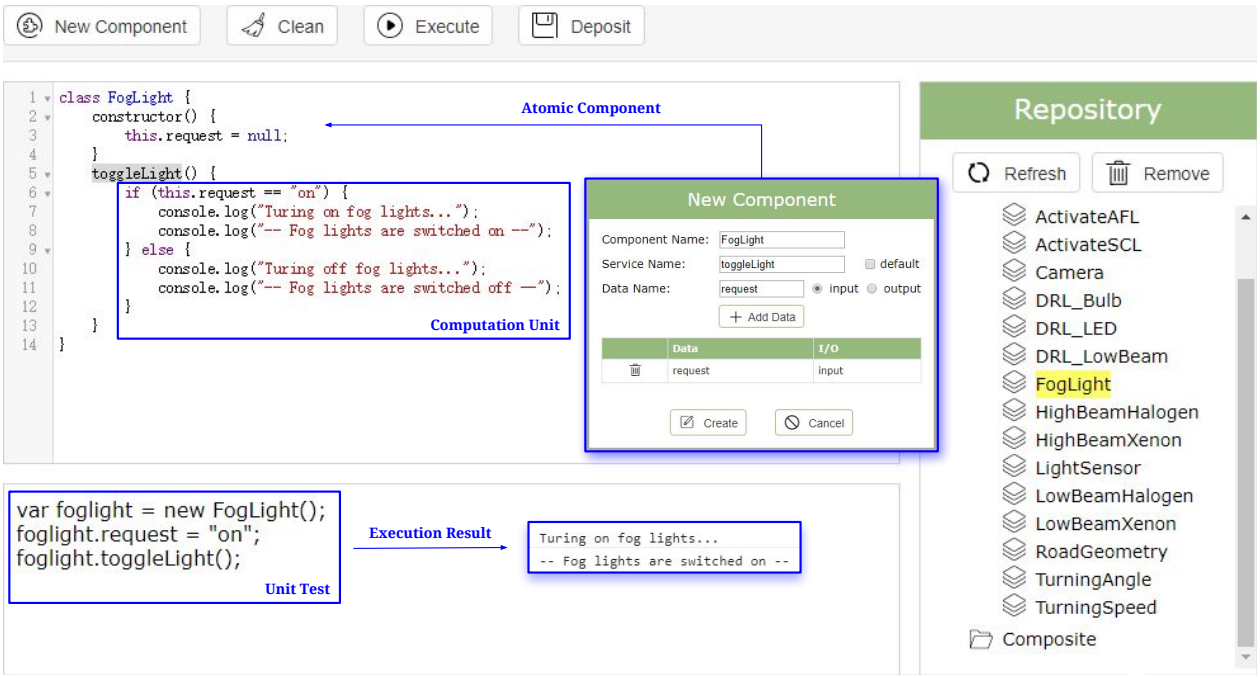
interactions occur in a product. The interaction rules are set up in the family filters, which are bound with every family composition connector, as shown in Figure 1.

III. The Tool

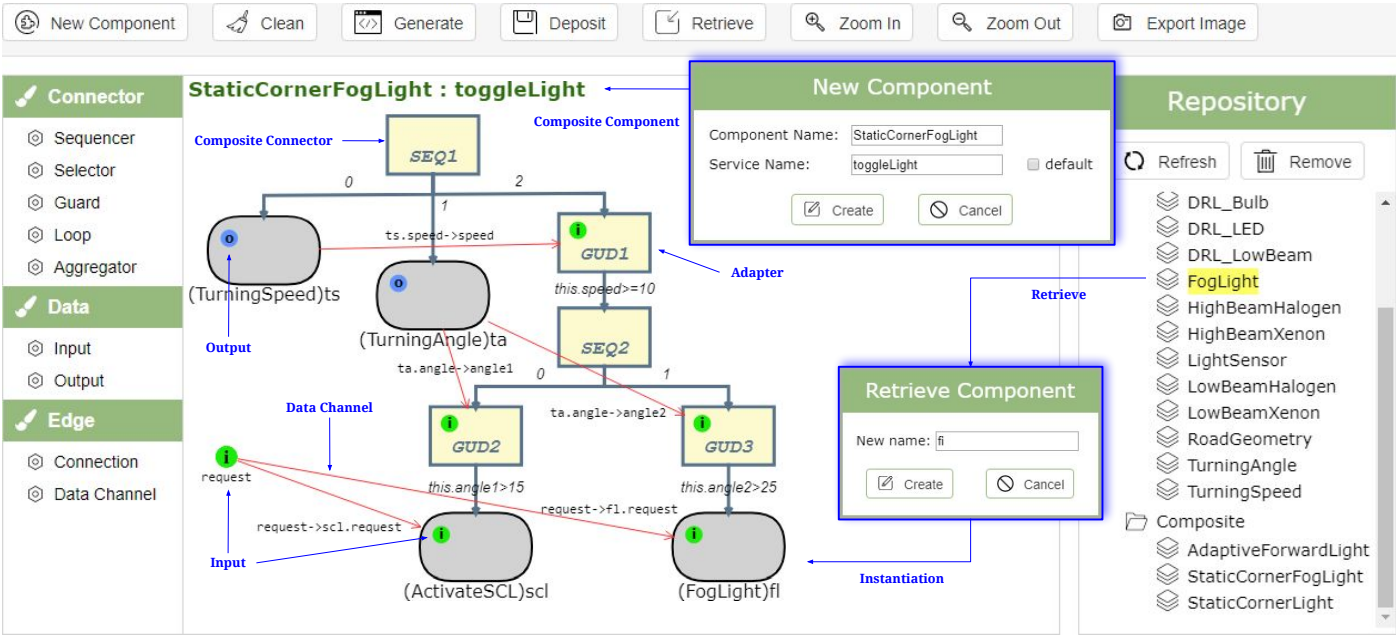
Our tool is a web-based graphical tool that implements our component model [8]. The GUI is realised using HTML5[9] and CSS3 [10], whereas the functionality is implemented using JavaScript [11]. In particular, we adopt the latest edition of ECMAScript as JavaScript specification since its significant new syntax, including classes and modules, supports complex applications. Additionally, we import jQuery [12], the most widely deployed JavaScript library, to improve code quality and enhance system extensibility. Our tool also offers a client-side repository called IndexedDB [13], which is a NoSQL database for massive amounts of structured data, as shown on the right side of Figure 3 (and Figure 4). For the purpose of user-friendliness, all building blocks, including constraints and interaction, can be easily added through buttons and dialogue boxes, as seen in Figures. 2-5.

The tool provides a workbench with functionalities that support the stages of our approach. Here, we describe these functionalities and illustrate them with the construction of a whole product family with 576 valid variants. The example is a family of *External Car Lights* (ECL) systems, which is adapted from an industrial example provided by *pure-systems*

 ISBN: 978-1-61208-668-2



(a) Atomic component.



(b) Composite component.

Figure 3. Canvases for constructing components for leaf features.

GmbH. The requirement of ECL family is demonstrated in Section IV. Figure 2(a) shows the feature model that contains all 576 valid product variants enumerated in Figure 2(b).

A. Component Construction

Figure 3 shows the canvases provided by our tool for constructing components for leaf features. After a component is created, it should be deposited in a repository, and therefore can be retrieved for further construction.

Figure 3(a) depicts the construction and deposition, of an atomic component, `FogLight`, which is the implementation of leaf feature FOG LIGHT. By simply clicking the 'New Component' button, we can define the component name, service name, input data and output data in a dialogue box, which automatically generates an implementation template for the computation unit. Additionally, we can immediately test the component as soon as the computation unit has been implemented, and examine the result through browser console.

Figure 3(b) illustrates the construction of a composite component, `StaticCornerFogLight`. According to requirements, it implements the feature interaction caused by STATIC CORNERING LIGHT and FOG LIGHT. A composite com-

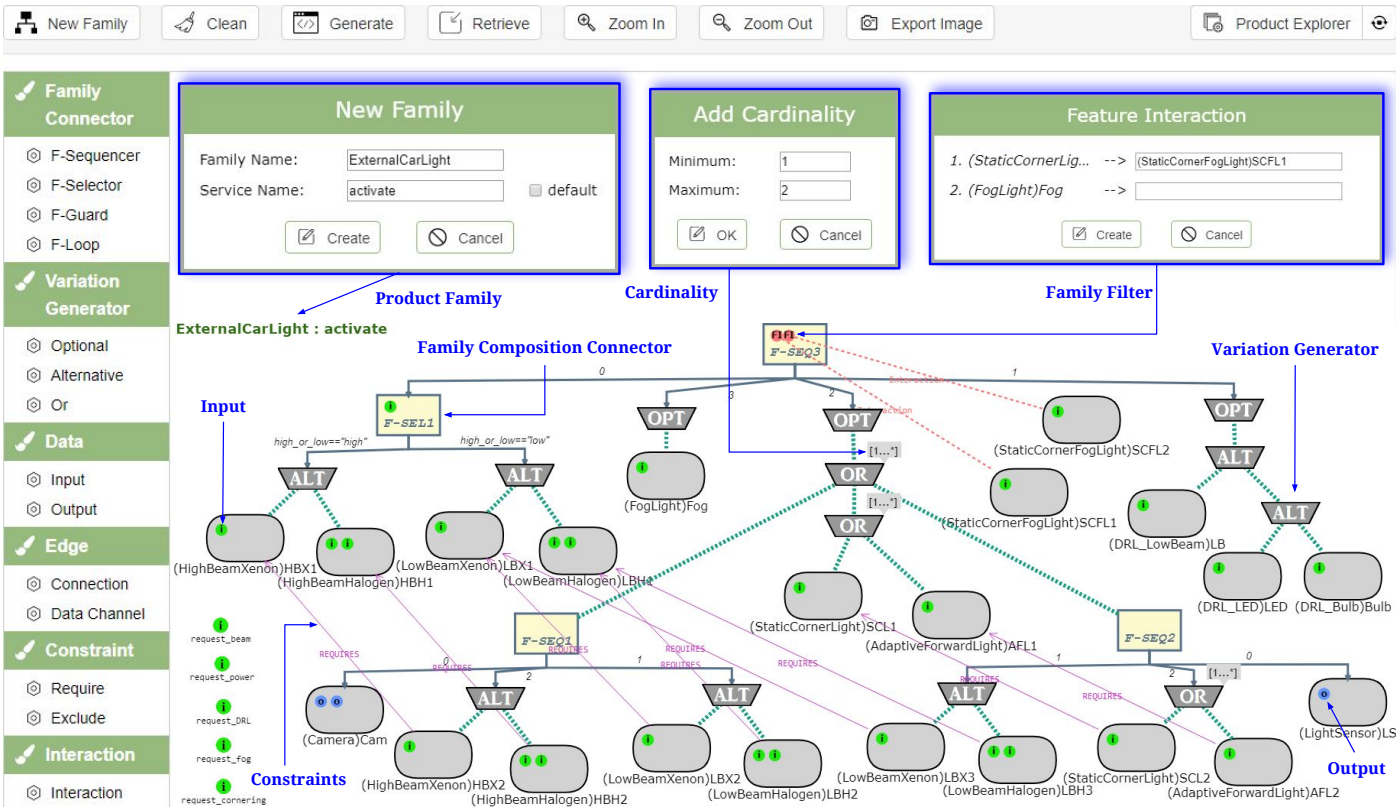


Figure 4. Canvas for constructing a whole product family from a feature model.

ponent is defined by its name and service, and constructed by composing components (retrieved from repository) via predefined composition connectors and adaptors (on the left of Figure 3(b)). Composition connectors include *Sequencer* and *Selector*, which provide sequencing and branching respectively. Adapters include *Guard* and *Loop*, which offer gating and looping respectively. Components can also be aggregated into a façade component, i.e., one that contains the aggregated components, by an *aggregator* connector *AGG*. Aggregates are essential for implementing the *or* variation point.

B. Variants and Family Construction

Figure 4 shows the canvas for constructing a whole product family from the feature model. It is worth noting that we omit the repository and data channels for clarification. A product family is defined by its name and service, and constructed by composing components (retrieved from repository) via predefined variation generators and family composition connectors (on the left of Figure 4). Constraints, derived from the constraints in the feature model (Figure 2(a)), as well as feature interaction, are defined as rules in a family composition connector filter. For example, if interacting features `StaticCornerLight` and `FogLight` are selected together, the former will be replaced by `StaticCornerFogLight` immediately.

C. Product Explorer

Figure 5 presents a useful feature of our tool, namely *Product Explorer*. It enumerates all valid products in the form of variability resulting from each variation generator at any level of nesting. For each product, by a simple click, the user can examine its structure and built-in components, and hence compare this product with the corresponding variant derived from the feature model. In this case, there are a total of 576 products in solution space, matching exactly the 576 variants enumerated in problem space in Figure 2(b). Figure 5 also shows the model structure of product No. 165.

Furthermore, any product can be executed and tested directly. A batch download link of all source code files is available.

IV. Demonstration Roadmap

In this section, we will present how to construct a software product family step-by-step from scratch by our approach and tool.

Step 1: Construct Feature Model

An ECL system can control headlights (including Low Beam lights and High Beam lights), Fog Lights and Daytime Running Light (including Reduced Low Beam lamp, LED and Standard Bulb). These lights can be switched on or off according to the driver's instructions. A beam is either Xenon or Halogen.

On the other hand, in some cases, an ECL system enables lights and signal devices by automatic detection. It provides a functionality called Driver Assistance, which supports Automatic Light, Automatic High/Low Beam and Cornering Light (including Static Cornering Light and Adaptive Forward Light).

Figure 2(a) shows the canvas for constructing the ECL feature model along with features, variation points and constraints. It defines an enumerative variability with a total of

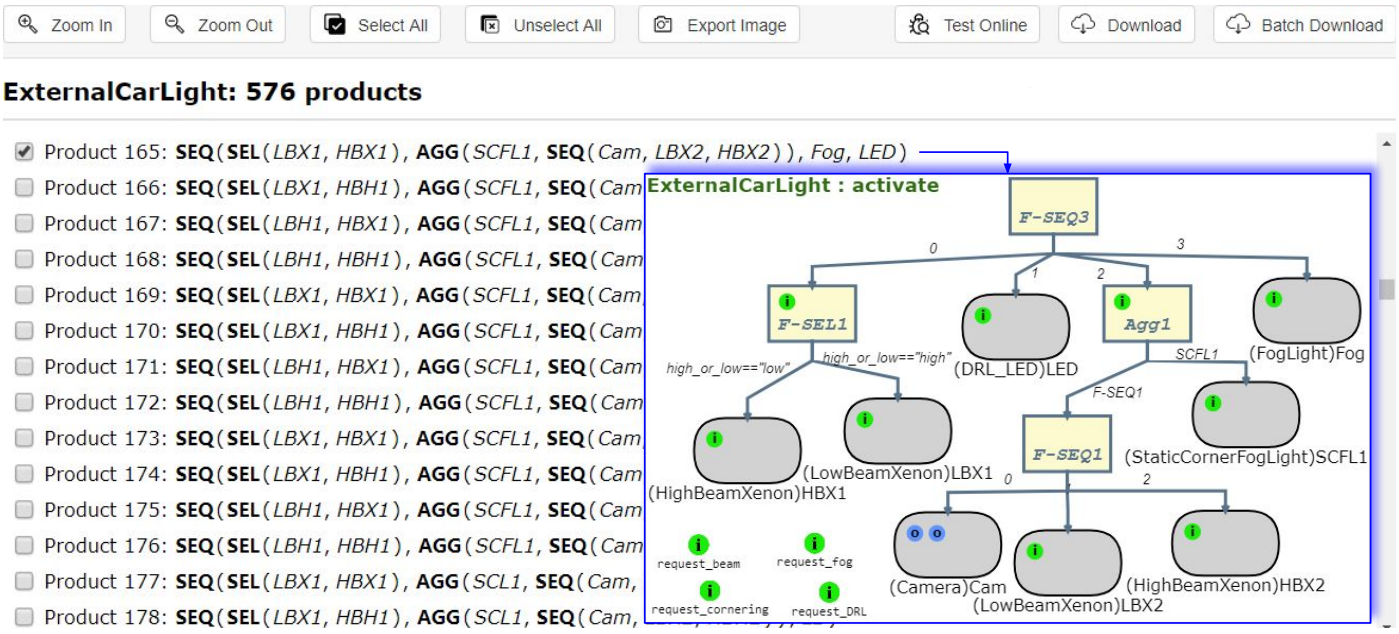


Figure 5. Product explorer and product No.165.

576 valid variants. Figure 2(b) shows all valid variants in terms of feature combinations.

Step 2: Construct Bottom Level Components

There are 20 leaf features in Figure 2(a), but some of them are reused. As a result, we construct 12 components (atomic or component) for them. In addition, we construct an optional component for feature interaction, as already stated in Section III-A. Once implemented, they are deposited for further construction. The repository in Figure 3 shows all the components, each of them can be tested directly, as illustrated in Figure 6.

Step 3: Apply Variation Generators

Now, we retrieve the pre-constructed components from repository. After instantiation, we have prepared 22 component instances for family construction, as shown in Figure 4. Then we apply the variation generators according to the feature model in Figure 2(a). For example, the OPT applied to *Fog* yields the set {$\emptyset$, {*Fog*}}; the ALT applied to *LED* and *Bulb* and gives the set {{*LED*}, {*Bulb*}}; and the OR applied to *AFL1* and *SCL1* generates the set {{*AFL1*}, {*SCL1*}, {*AFL1*, *SCL1*}}.

Step 4: Apply Family Composition Connectors

This step is to compose the variations into sub-families of the ECL family using family composition connectors. The choice of family composition connectors is a design decision, however it will not affect the total number of products in the (sub)family. For instance, in Figure 4, a family sequencer called *F-SEQ1* composes a sub-family, which is the implementation of abstract feature AUTOMATIC HIGH/LOW BEAM. The whole family is constructed when all sub-families have been constructed and composed. However, in order to filter out the invalid products, we need to add constraints between component instances, which are mapped onto the constraints we defined in Figure 2(a).

Step 4: Establish Interaction Rules

In Figure 4, we show a family filter for setting out interaction rules. It has been exemplified in Section III. The interaction will be displayed in the nearest family composition connector that composes the components involved. We add necessary data channels to define data flows. The final whole product family is shown in Figure 4.

Step 5: Test Products

All the products in the family are fully formed and executable, so they can be directly tested. Figure 6 shows the generated source code, testing code and result of product 165. This concludes the demonstration.

V. CONCLUSION AND FUTURE WORK

Our tool is a complete re-implementation of an earlier version presented in [14]. We have re-defined the underlying component model, added new capabilities including feature interaction, and used a different technology stack.

Compared to current SPLE tools, which use parametric variability to configure one product at a time in solution space, our tool offers a new possibility of constructing the whole family in one go, by using enumerative variability in solution space. Although, from a customer's point of view, constructing all products at once may seem like overkill, our approach/tool can be adopted by current SPLE techniques in application engineering, since our approach can provide all the necessary domain artefacts. For example, for the ECL family, our tool can generate an annotative code base using pure::variants notations. This code base can be correctly used in pure::variants for application engineering (Figure 7).

Finally, our tool will facilitate product line testing [15]. So far, our tool only provides a workbench for domain unit testing, i.e., testing components and products. However, it does not support domain integration testing and domain system testing [16]. Therefore, the further development of our tool includes (1) automatic comparison of variability specified in

 ISBN: 978-1-61208-668-2

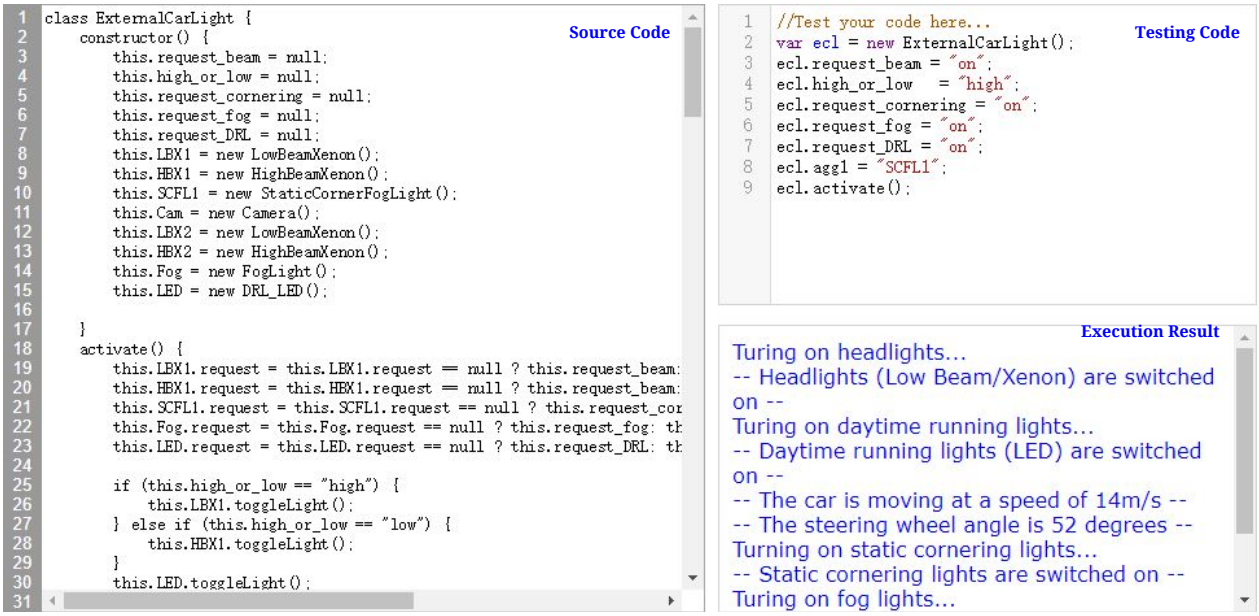


Figure 6. Product/Component testing.

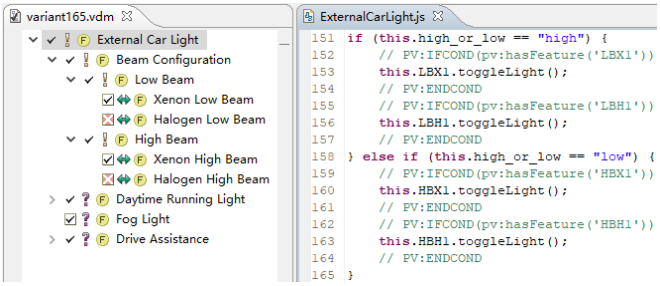


Figure 7. Generating annotative code base for pure::variants.

the problem space (Figure 2(b)) and implemented in the solution space (Figure 5), (2) the validity of every data channel, (3) generation of featured statecharts to examine behaviour at family level [17], [18].

REFERENCES

[1] 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.

[2] D. Batory, "A tutorial on feature oriented programming and the ahead tool suite," Generative and Transformational Techniques in Software Engineering, 2006, pp. 3–35.

[3] M. Antkiewicz, K. Bąk, A. Murashkin, R. Olaechea, J. H. J. Liang, and K. Czarnecki, "Clafer tools for product line engineering," in Proceedings of the 17th International Software Product Line Conference co-located workshops. ACM, 2013, pp. 130–135.

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

[5] 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.

[6] K.-K. Lau and Z. Wang, "Software component models," IEEE Transactions on Software Engineering, vol. 33, no. 10, October 2007, pp. 709–724.

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

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

[9] G. Anthes, "HTML5 leads a web revolution," Communications of the ACM, vol. 55, no. 7, 2012, pp. 16–17.

[10] T. Celik and F. Rivoal, "CSS basic user interface module level 3 (CSS3 UI)," 2012.

[11] D. Flanagan, JavaScript: the definitive guide. O'Reilly Media, Inc., 2006.

[12] D. S. McFarland, JavaScript & jQuery: the missing manual. O'Reilly Media, Inc., 2011.

[13] S. Kimak and J. Ellman, "The role of html5 indexeddb, the past, present and future," in Internet Technology and Secured Transactions (ICITST), 2015 10th International Conference for. IEEE, 2015, pp. 379–383.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

 ISBN: 978-1-61208-668-2