An MDE Tool for Defining Software Product Families with Explicit Variation Points*

Simone Di Cola, Kung-Kiu Lau, Cuong Tran, and Chen Qian
School of Computer Science
The University of Manchester
Oxford Road, M13 9PL, United Kingdom
dicolas,kung-kiu,ctran,cq@cs.man.ac.uk

ABSTRACT

Current software product line engineering tools mainly focus on variability in the problem space, and create product families by linking variability models to artefacts in the solution space. In this paper, we present a tool that can be used to define software architectures with explicit variation points, and hence product families, directly in the solution space.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.11 [**Software Engineering**]: Software Architectures

Keywords

PLE tools, architecture variability, component-based software development

1. INTRODUCTION

Product line engineering tools, e.g. pure::variants,[1] Gears,[2] and COVAMOF [8], mainly focus on modelling variability in the *problem space*, and create product families by linking variability models to artfacts such as a code base in the *solution space* [7].

By contrast, software architecture based approaches [1] create products directly in the solution space. However most of these approaches do not define variability explicitly [4], making it difficult to relate solutions to the problem space. In this paper we present a tool that can be used to define software architectures with explicit variation points, and hence product families. We briefly explain our approach and demonstrate the tool on an example.

2. TOOL OVERVIEW

Our tool supports the construction of a product family using a feature-oriented approach, as illustrated by Figure 1. Our approach consists of three main stages: **(i) build components** (atomic or composite) to model the leaf features (F_4-F_9) in the feature model, and compose them to model parent features wherever appropriate (F_6 and F_7 are composed into F_2); **(ii) apply variation operators** to the set of components built in (i), according to the variation points in the feature model ($\texttt{OPT}(F_4)$, $\texttt{OPT}(F_5)$, $\texttt{ALT}(F_8, F_9)$), and repeat this for nested variation points (2 times for F_1); **(iii) apply family connectors** to component sets from (i), (ii) and (iii) to produce one family (F_0).

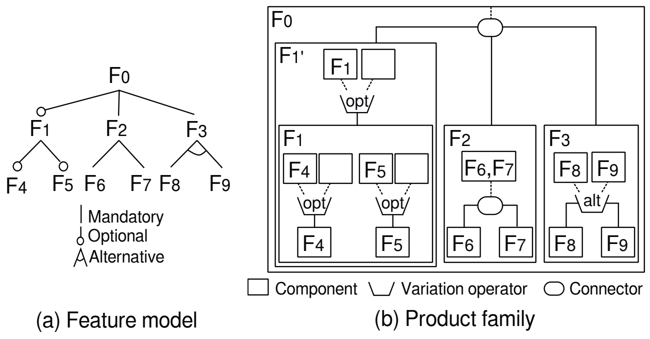


Figure 1: Our approach.

For each stage, the tool provides a canvas as a design space, as well as a palette of pre-defined design blocks. The tool also performs continuous validation. Errors appear as cross markers attached to erroneous entities and detailed in the *Problems* view. Figures 2, 3 and 5 illustrate our workbench for the three stages.

3. TOOL IMPLEMENTATION

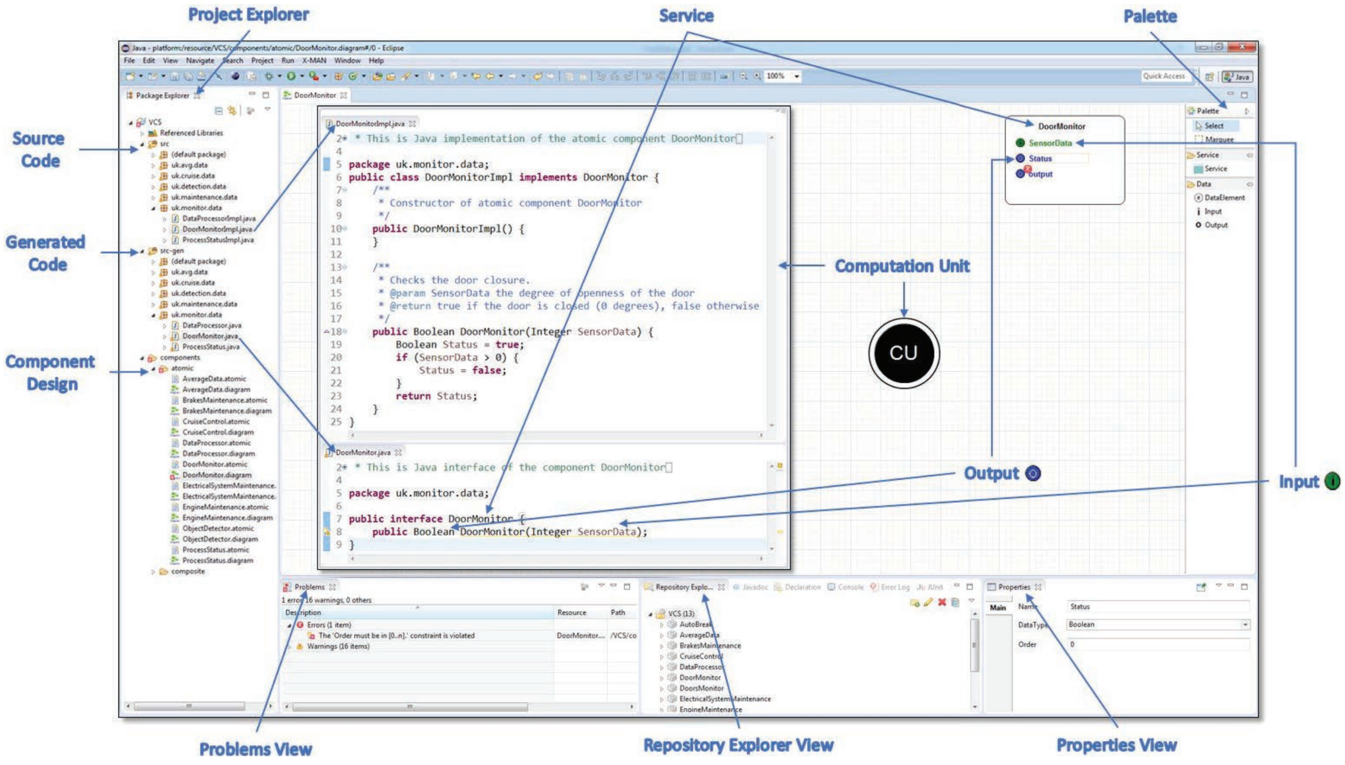
Our tool is implemented using a robust stack of model-driven technologies like Eclipse Modelling Framework [9], Graphiti,[3] and CDO.[4]

Our approach is based on a component model called FX-MAN, which is in turn based on the X-MAN component

*Research leading to these results has received funding from the EU ARTEMIS Joint Undertaking under grant agreement no. 621429 (project EMC2) and from the Technology Transfer Board (TSB) on behalf of the Department for Business, Innovation & Skills, UK.
[1]http://www.pure-systems.com/
[2]http://www.biglever.com/

[3]https://eclipse.org/graphiti/
[4]https://eclipse.org/cdo/

© 2015 ACM. ISBN 978-1-4503-2138-9.
http://dx.doi.org/10.1145/2791060.2791090



```java
 * This is Java implementation of the atomic component DoorMonitor

package uk.monitor.data;
public class DoorMonitorImpl implements DoorMonitor {
    /**
     * Constructor of atomic component DoorMonitor
     */
    public DoorMonitorImpl() {
    }

    /**
     * Checks the door closure.
     * @param SensorData the degree of openness of the door
     * @return true if the door is closed (0 degrees), false otherwise
     */
    public Boolean DoorMonitor(Integer SensorData) {
        Boolean Status = true;
        if (SensorData > 0) {
            Status = false;
        }
        return Status;
    }
}
```

```java
 * This is Java interface of the component DoorMonitor

package uk.monitor.data;

public interface DoorMonitor {
    public Boolean DoorMonitor(Integer SensorData);
}
```

```
1 error, 16 warnings, 0 others
Description                                                 Resource        Path
Errors (1 item)
  The 'Order must be in [0..n].' constraint is violated     DoorMonitor...  /VCS/co
Warnings (16 items)
```

```
Name      Status
DataType  Boolean
Order     0
```

Figure 2: Eclipse workbench for atomic component development.

model [6] and tool. X-MAN architectures have no variability, so in FX-MAN we addded: (i) *variation operators* that generate variants of a set of X-MAN architectures, which we call an *X-MAN set*; and (ii) *family connectors* that compose multiple X-MAN sets into a product family.

3.1 Component Development

X-MAN components can be atomic and composite. An *atomic* component is a unit of computation. Its *computation unit* (CU) contains the implementation of the component. An atomic component has a number of *provided services* implemented by the CU. Figure 2 depicts the design of an atomic component. To define a service we drag onto the canvas a service from the palette and then add inputs and outputs; then we can generate an implementation template for the CU and implement the specified service. Continuous validation will ensure that the final component is well-formed.

Components are deposited in a standalone or collaborative repository. In our tool, a repository is displayed in the *Repository Explorer* view, at the bottom of Figures 2 and 3.

A *composite* component is constructed by composing pre-constructed components (from the repository) by means of pre-defined composition connectors and adapters. These are (exogenous) control structures that coordinate the execution of their composed components. Our composition connectors are *Sequencer* and *Selector*. They provide *sequencing* and *branching* respectively. Adapters are *Guard* and *Loop*, allowing *gating* and *looping* respectively. A composite component, just as an atomic one, provides services; they result from the coordination of the services provided by its sub-components. In addition to composition connectors, X-MAN also defines an *Aggregator* connector, which aggregates in a new composite component the services exposed by its sub-components. An aggregated component effectively provides a *façade* to the aggregated services.

Figure 3 illustrates the design of a composite component called `AutoCruiseControl`. Two components, `AdaptSpeed` and `CruiseControl`, are retrieved from the repository and composed using a *Sequencer* and a *Guard*. Sequencing order, and gating conditions are specified as labels on coordination connections. The composition results in the *AutoCruiseControl* provided service.

In addition to control, data flow can also be specified in the design via data channels. Data can flow 'horizontally' between components and 'vertically' to the composite component services.

The result from component development is a repository of X-MAN components, which are also architectures themselves.

3.2 Variability

To model variability, in FX-MAN, we have defined three explicit variation operators: *Or*, *Alternative* and *Opt*. They define the standard 'inclusive or', 'exclusive or', and 'optional' variation points in feature models. Variation operators can be applied to X-MAN sets to generate variations. They can be nested within one another, as in feature models.

In our tool, variation operators can be dragged from the palette (Figure 5), and connected to component (in X-MAN sets) or to other variation operators. Figure 5 shows two *Alternative* and two *Optional* operators connected to four

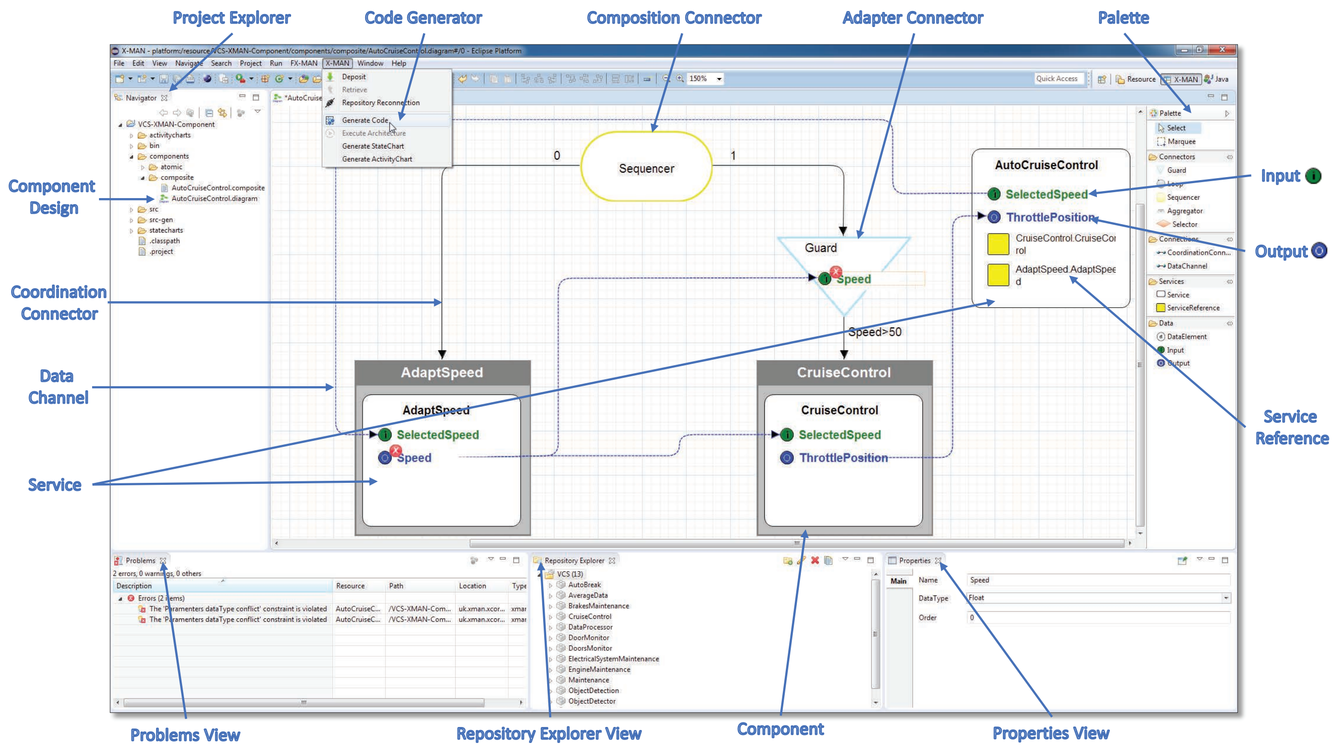


Figure 3: Eclipse workbench for composite component development.

X-MAN component.[5] It also shows an *Or* operator nested within an *Alternative* operator.

Variation generation results in sets of X-MAN architectures.

3.3 Composing Architectures into a Family

Family connectors compose X-MAN sets into a *product family*, which is an architecture containing the architectures of all the members. A product family can be adapted by a family adapter.

Family connectors and adapters are implemented in a palette in our tool as shown in Figure 5. To apply a family connector we drag it onto the canvas and make connections from it to X-MAN components (in X-MAN sets) or variation operators. In Figure 5, the *F-Selector* composes three sets of variations produced by the two *Optional* and the *Alternative* variation operators into a single architecture. An *F-Sequencer* composes the previous architecture with another set of variations created by the variation operator *Or* to yield a larger architecture. Finally, a *F-Loop* is connected at the top.

The constructed product family can be explored via the *Product Explorer* view (bottom, Figure 5). Architectures of individual members can be directly extracted and executed.

4. EXAMPLE

Consider vehicle control systems (VCS), adapted from [5], with the feature model in Figure 4.

First, we develop seven X-MAN atomic components (corresponding to seven leaf features): `AverageMPH`, `AverageMPG`, `Maintenance`, `Monitoring`, `FrontDetection`, `AllRoundDetection`, `AdaptSpeed` and `CruiseControl`. They are stored in the repository. The last two are then retrieved and composed into the composite component `AutoCruiseControl` in Figure 3.

[5]The family connectors and adapters specify coordination and adaptation applied to all variations.

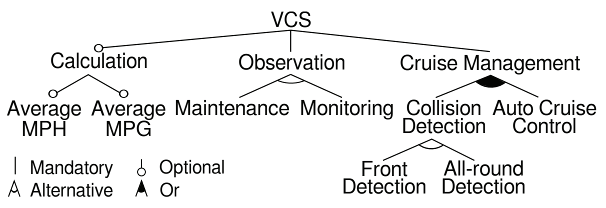
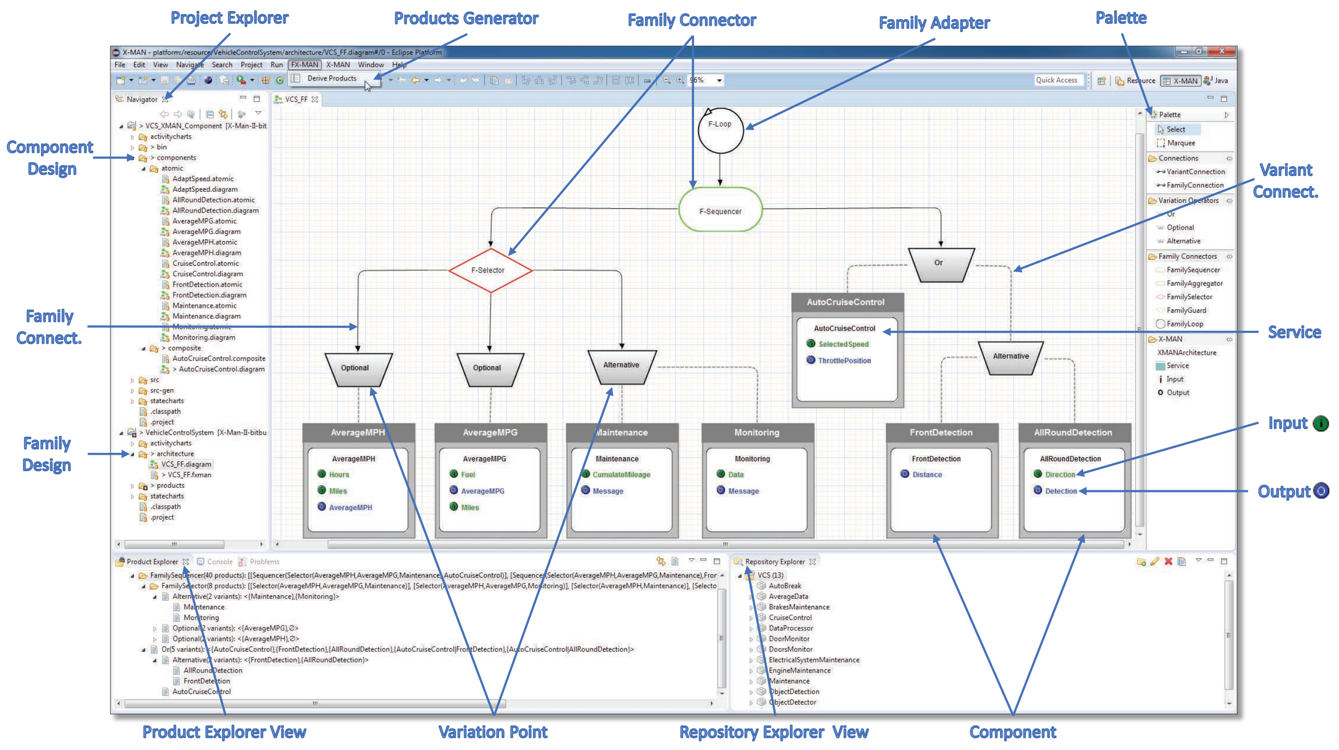


Figure 4: Feature model of VCS.

Second, we apply variation operators defined in the feature model to the X-MAN components that have been constructed to implement the leaf features. To this end, we retrieve the components and apply the specified variation operators to them. In Figure 5, we apply *Optional* to `AverageMPH` and `AverageMPG`; *Alternative* to `Maintenance` and `Monitoring`, and `FrontDetection` and `AllRoundDetection`; and *Or* to the latter and `AutoCruiseControl`.

Third, we compose all the above variations into all the possible products specified by the feature model. We use *F-Sequencer*, *F-Selector* and *F-Loop*, also shown in Figure 5. The resulting architecture contains a total of 40 products, which can be inspected, and extracted using the *Product Explorer* view in Fig. 5.

5. DISCUSSION AND CONCLUSION



Product Explorer

FamilySequencer(40 products): [[Sequencer(Selector(AverageMPH,AverageMPG,Maintenance),AutoCruiseControl)], [Sequencer(Selector(AverageMPH,AverageMPG,Maintenance),Fror
FamilySelector(8 products): [[Selector(AverageMPH,AverageMPG,Maintenance)], [Selector(AverageMPH,AverageMPG,Monitoring)], [Selector(AverageMPH,Maintenance)], [Selecto
Alternative(2 variants): <{Maintenance},{Monitoring}>
Maintenance
Monitoring
Optional(2 variants): <{AverageMPG},∅>
Optional(2 variants): <{AverageMPH},∅>
Or(5 variants): <{AutoCruiseControl},{FrontDetection},{AllRoundDetection},{AutoCruiseControl|FrontDetection},{AutoCruiseControl|AllRoundDetection}>
Alternative(2 variants): <{FrontDetection},{AllRoundDetection}>
AllRoundDetection
FrontDetection
AutoCruiseControl

Figure 5: Eclipse workbench for constructing architecture with variability.

Current product line engineering tools are mostly problem space based (see [2] for a survey). For example, in pure::variants, a family model is defined to establish links between features in the feature model (*problem space*) and available assets, i.e. code and documentation (*solution space*). By contrast, our approach is solution space based. By having a full set of explicit variation points (including the 'inclusive or') in our architectures, we can map our product families to the problem space. This is in contrast to most software architecture based approaches, which are also solution space based but do not define architectures with a full set of explicit variation points.

However, currently our tool lacks an interface to the problem space, i.e. it does not support the end user in the process of features (or product) selection. In this regard, we intend to integrate our tool with pure::variants. This will allow us to extract specific products, thus tackling potential scalability problems when dealing with large product families.

At the moment, our tool also does not handle constraints between features in the problem space; rather, we define them as filters on product families in the solution space. We plan to investigate with a real example how efficient these filters are, and whether/how they could deal with constraints in the problem space.

Furthermore, our approach only deals with structural variability, but not parametric variability [3] . Future work will therefore include an investigation of the latter.

Finally our tool is available at http://xmantoolset.ddns.net.

Acknowledgement

We wish to thank Michael Schulze for his insightful comments and constructive feedbacks.

6. REFERENCES

[1] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. SEI Series in Soft. Eng. Addison-Wesley, 2012.
[2] T. Berger, R. Rublack, D. Nair, et al. A survey of variability modeling in industrial practice. In *Proc. of VaMoS*, page 7. ACM, 2013.
[3] Jeffrey B Dahmus, Javier P Gonzalez-Zugasti, and Kevin N Otto. Modular product architecture. *Design studies*, 22:409–424, 2001.
[4] M. Galster, P. Avgeriou, D. Weyns, et al. Variability in software architecture: current practice and challenges. *ACM SIGSOFT Soft. Eng. Notes*, 36(5):30–32, 2011.
[5] D. Hatley and I. Pirbhai. *Strategies for real-time system specification*. Addison-Wesley, 2013.
[6] N. He, D. Kroening, T. Wahl, et al. Component-based design and verification in X-MAN. In *Proc. of ERTSS*, 2012.
[7] M. Schulze and R. Hellebrand. Variability exchange format a generic exchange format for variability data. In *Proc. of SE-WS*, volume 1337. CEUR-WS.org, 2015.
[8] M. Sinnema, O. De Graaf, and J. Bosch. Tool support for COVAMOF. In *Work. on Soft. Var. Manag. for Prod. Der.*, 2004.
[9] D. Steinberg, F. Budinsky, E. Merks, et al. *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

APPENDIX

A. DEMONSTRATION ROADMAP

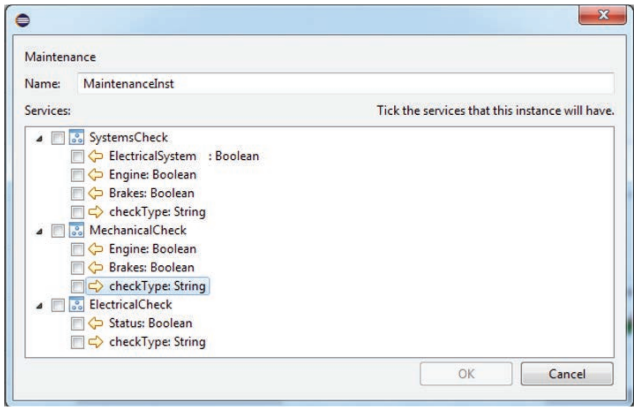
The tool demonstration begins with a quick overview of the theoretical foundations for our tool, and is followed by a demonstration of the steps described in Section 3, illustrated by the VCS example in Section 4. Each step is illustrated through small working examples.

Step 1: Construct X-MAN components

The first step is to construct X-MAN components, atomic or composite, that implement the leaf features in the feature model. Once implemented, they are deposited in a shared repository (accessible by the participants). There are seven leaf features, so we will construct seven X-MAN components: `AverageMPH`, `AverageMPG`, `Maintenance`, `Monitoring`, `FrontDetection`, `AllRoundDetection`, and `AutoCruiseControl`. As already stated in Section 4, all the components are atomic, except `AutoCruiseControl`, which is a composite of two atomic components `AdaptSpeed` and `CruiseControl`. We demonstrate how the tool supports their implementation, and how they can be composed into the `AutoCruiseControl` component. Moreover, we demonstrate how the tool allows us to generate code for the `AutoCruiseControl` component, and how to verify its behaviour via JUnit tests.

Step 2: Apply variation operators

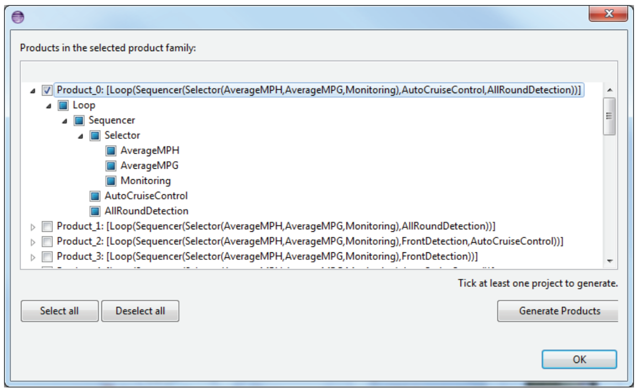
The second step is to apply variation operators defined in the feature model to the constructed X-MAN components. To this end, we retrieve all the seven components from CDO using the dialogue in Fig. 6, and apply the variation operators according to the feature model in Fig. 4. The *Optional* operators applied to `AverageMPH` and `AverageMPG` yield the tuple $F1 = \langle$`{AverageMPH}`$,\emptyset\rangle$ and $F2 = \langle$`{AverageMPG}`$,\emptyset\rangle$ respectively. The *Alternative* operator applied to `Maintenance` and `Monitoring` gives the set $F3 = \langle$`{Maintenance},{Monitoring}`$\rangle$. The *Or* operator applied to the X-MAN set consisting of `AutoCruiseControl` and the X-MAN set resulting from applying the *Alternative* operator to `FrontDetection` and `AllRoundDetection` yields the X-MAN set of 5 products: $F4 = \langle$`{AutoCruiseControl, AllRoundDetection}, {AllRoundDetection}, {FrontDetection, AutoCruiseControl}, {FrontDetection}, {AutoCruiseControl}`$\rangle$.


Maintenance
Name: MaintenanceInst
Services: Tick the services that this instance will have.

```
▲ ☐ SystemsCheck
    ☐ ElectricalSystem   : Boolean
    ☐ Engine: Boolean
    ☐ Brakes: Boolean
    ☐ checkType: String
▲ ☐ MechanicalCheck
    ☐ Engine: Boolean
    ☐ Brakes: Boolean
    ☐ checkType: String
▲ ☐ ElectricalCheck
    ☐ Status: Boolean
    ☐ checkType: String
```

Figure 6: Dialogue for retrieving components.

By means of the *Product Explorer* view, we demonstrate how each variation operator realises the variability just described, and how they can be nested in order to create complex variations of X-MAN sets.


Products in the selected product family:

```
▲ ☑ Product_0: [Loop(Sequencer(Selector(AverageMPH,AverageMPG,Monitoring),AutoCruiseControl,AllRoundDetection))]
    ▲ ■ Loop
        ▲ ■ Sequencer
            ▲ ■ Selector
                  ■ AverageMPH
                  ■ AverageMPG
                  ■ Monitoring
              ■ AutoCruiseControl
              ■ AllRoundDetection
▷ ☐ Product_1: [Loop(Sequencer(Selector(AverageMPH,AverageMPG,Monitoring),AllRoundDetection))]
▷ ☐ Product_2: [Loop(Sequencer(Selector(AverageMPH,AverageMPG,Monitoring),FrontDetection,AutoCruiseControl))]
▷ ☐ Product_3: [Loop(Sequencer(Selector(AverageMPH,AverageMPG,Monitoring),FrontDetection))]
```

Tick at least one project to generate.

Figure 7: Product extraction interface.

Step 3: Construct the product family

After generating variations, a tuple of X-MAN sets are composed via family connectors into a product family. We demonstrate how to create the 40 products defined by the VCS feature model. We choose to compose $F1$, $F2$, $F3$ into $F5$ with the family connector *F-Selector* because we want to allow a driver to choose any subset of the features: *AverageMPH*, *AvergageMPG*, *Maintenance* and *Monitoring*. Then we choose to compose $F5$ and $F4$ with *F-sequencer* to combine the driver's choice with the *Cruise Management* feature.

Step 4: Extract family members

All the 40 products can be extracted using the product extraction dialogue box in Fig. 7. The (partial) result is depicted in Fig. 8

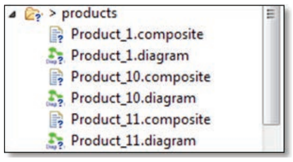

```
▲ > products
    Product_1.composite
    Product_1.diagram
    Product_10.composite
    Product_10.diagram
    Product_11.composite
    Product_11.diagram
```

Figure 8: Extracted product in project explorer.

We extract product No. 4 (Fig. 9), which is a premium version of VCS, has five components: `AverageMPH`, `AverageMPG`, `Monitoring`, `AutoCruiseControl`, and `AllRoundDetection`. The premium VCS is capable of displaying the result calculated by `AverageMPH`, `AverageMPG`, or `Monitoring` (chosen by the user). Then `AutoCruiseControl` is invoked with the aim of maintaining the speed (selected by the user). Finally, the system shows the distance from the vehicle to the nearest obstacle in a specified direction.

Step 5: Test family members

Because all the products in the family are fully formed and executable, we check their behaviour with JUnit tests. This is an advantage in practical development. We demonstrate this by testing the behaviour of the extracted product No. 4 (Fig. 10). This concludes the demonstration.

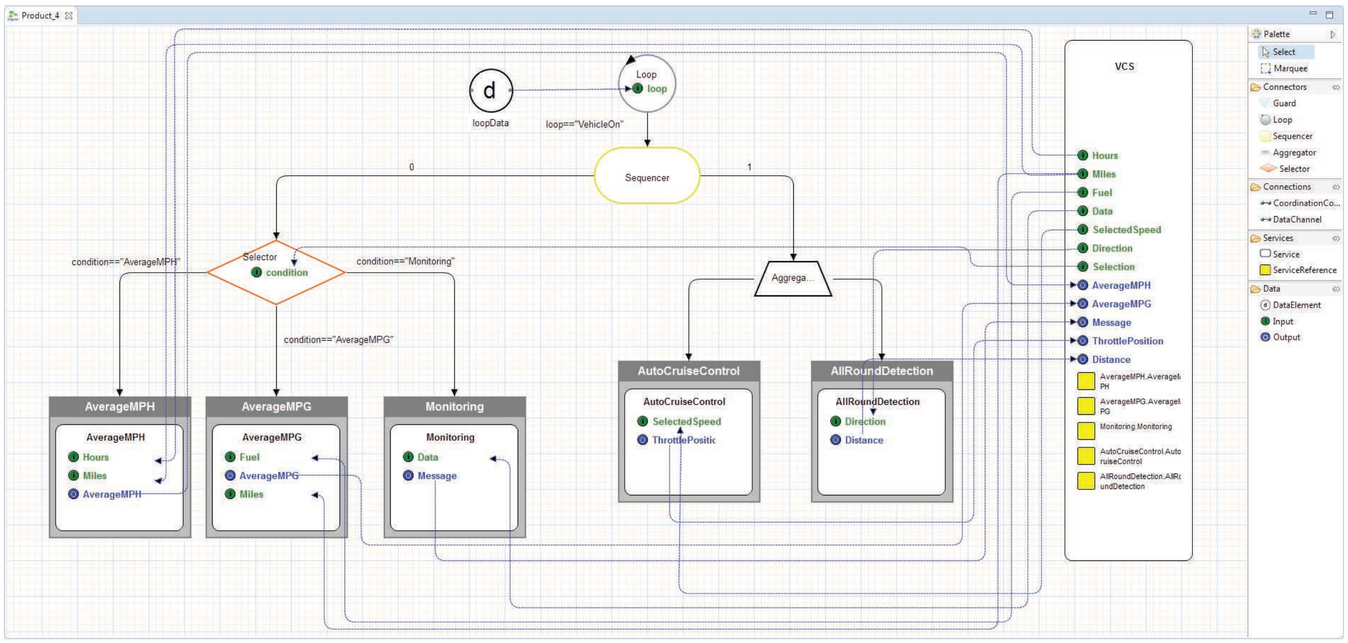
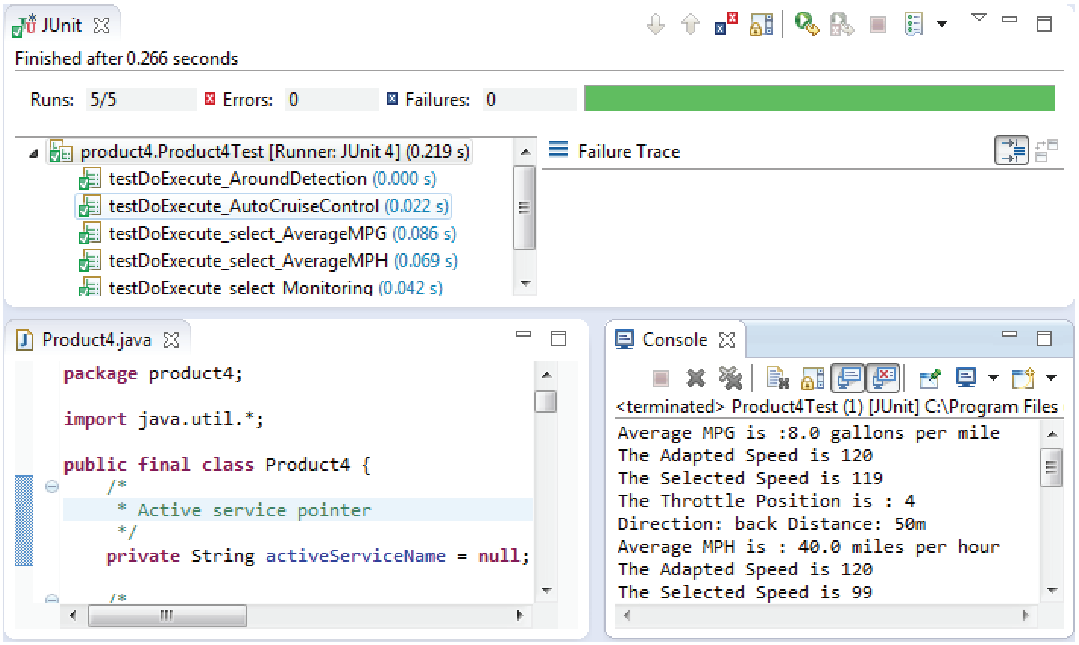


Figure 9: Product No.4



```
Finished after 0.266 seconds

Runs: 5/5    Errors: 0    Failures: 0

product4.Product4Test [Runner: JUnit 4] (0.219 s)
    testDoExecute_AroundDetection (0.000 s)
    testDoExecute_AutoCruiseControl (0.022 s)
    testDoExecute_select_AverageMPG (0.086 s)
    testDoExecute_select_AverageMPH (0.069 s)
    testDoExecute_select_Monitoring (0.042 s)
```

```java
package product4;

import java.util.*;

public final class Product4 {
    /*
     * Active service pointer
     */
    private String activeServiceName = null;

    /*
```

```
<terminated> Product4Test (1) [JUnit] C:\Program Files
Average MPG is :8.0 gallons per mile
The Adapted Speed is 120
The Selected Speed is 119
The Throttle Position is : 4
Direction: back Distance: 50m
Average MPH is : 40.0 miles per hour
The Adapted Speed is 120
The Selected Speed is 99
```

Figure 10: Product No.4 tests result.