Advancing Traffic Efficiency and Safety through Software Technology phase 2 (ATESST2)

Report type: D3.1 Appendix A3.3 (was I3.3.2)
Report name: EAST-ADL Update Suggestions for Variability Management and Reuse
Dissemination level: PU
Status: Final
Version number: 1.1
Date of preparation: 2010-05-28
Authors

Editor
Mark-Oliver Reiser (TUB)

E-mail
Mark-Oliver.Reiser@tu-berlin.de

Authors

Andreas Abele (CONTI)
David Servat (CEA)
Helko Glathe (CAR)
Matthias Weber (CAR)
Henrik Lönn (VTEC)
Ramin Tavakoli Kolagari (VTEC)
Hans Blom (VTEC)
Rolf Johannson (MGH)

The Consortium

Volvo Technology Corporation (S)  VW/Carmeq (D)  Centro Ricerche Fiat (I)
Continental Automotive (D)  Delphi/Mecel (S)
Mentor Graphics Hungary (H)  CEA LIST (F)
Kungliga Tekniska Högskolan (S)  Technische Universität Berlin (D)  University of Hull (GB)
## Revision chart and history log

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>2009-11-09</td>
<td>Initial version, because former content of I3.3.2 will be moved to I3.3.1, as agreed in order to harmonize the I3.x.2 deliverables.</td>
</tr>
<tr>
<td>0.2</td>
<td>2010-04-19</td>
<td>Added variability example.</td>
</tr>
<tr>
<td>0.3</td>
<td>2010-04-21</td>
<td>Improved effect attributes in variability example.</td>
</tr>
<tr>
<td>0.4</td>
<td>2010-04-22</td>
<td>Added chapter on defining configuration decisions.</td>
</tr>
<tr>
<td>0.5</td>
<td>2010-04-23</td>
<td>Added feature model diagrams in variability example (Section 8).</td>
</tr>
<tr>
<td>1.0</td>
<td>2010-05-28</td>
<td>Final corrections.</td>
</tr>
<tr>
<td>1.1</td>
<td>2010-05-28</td>
<td>Adaptations to be used as appendix for D3.1</td>
</tr>
</tbody>
</table>
List of abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FAA</td>
<td>Functional Analysis Architecture</td>
</tr>
<tr>
<td>FDA</td>
<td>Functional Design Architecture</td>
</tr>
<tr>
<td>HDA</td>
<td>Hardware Design Architecture</td>
</tr>
<tr>
<td>DSL</td>
<td>Domain Specific Language</td>
</tr>
<tr>
<td>CDM</td>
<td>Configuration Decision Model</td>
</tr>
<tr>
<td>FM</td>
<td>Feature Model</td>
</tr>
<tr>
<td>CD</td>
<td>Configuration Decision</td>
</tr>
</tbody>
</table>
Table of contents

Advancing Traffic Efficiency and Safety through Software Technology phase 2 (ATESST2).......................... 1
Authors .......................................................................................................................... 2
Revision chart and history log .................................................................................. 3
List of abbreviations ................................................................................................... 4
Table of contents ........................................................................................................ 5
List of Figures ............................................................................................................. 7
1 Introduction ............................................................................................................. 8
2 Domain Model ....................................................................................................... 9
  2.1 Vehicle Level vs. Artifact Level .......................................................................... 9
  2.2 Overview of Variability Modeling Concepts in EAST-ADL .............................. 10
  2.3 Feature Modeling ............................................................................................... 11
  2.4 Configuration Decision Models (Configuration Links) .................................. 13
  2.5 Multi-Level Feature Models ............................................................................. 14
  2.6 Concepts for Variability Management on Artifact Level ............................... 17
  2.7 Commonalities and differences on different Artifact Levels ......................... 19
3 Defining Configuration Decisions ....................................................................... 21
  3.1 The Source and Target Feature Models of a ConfigurationDecisionModel .... 22
  3.2 Referring to a Feature of a Source or Target Feature Model ........................ 23
  3.3 Testing for Source Configurations in the Criterion Attribute ....................... 24
  3.4 Effectuating Target Configurations in the Effect Attribute ............................ 24
  3.5 Selecting and Deselecting Optional FunctionPrototypes ............................... 25
  3.6 Summary .......................................................................................................... 25
4 Methodology ......................................................................................................... 26
  4.1 The Role of Feature Modeling on VFM Level ..................................................... 26
  4.2 Methodology .................................................................................................. 28
5 Plugins and Tooling ............................................................................................. 29
6 CVM Framework Usage ....................................................................................... 30
7 Extended Example ............................................................................................... 40
  7.1 Vehicle Level ................................................................................................... 41
    7.1.1 Feature model Customer_FM: ................................................................ 41
    7.1.2 Core Technical Feature model: .............................................................. 42
    7.1.3 Configuration decision model CustomerCDM: ....................................... 43
    7.1.4 Core configuration decision model (from core technical FM to artifact-level): .... 44
7.2 FDA........................................................................................................................................... 45
  7.2.1 FunctionParts.......................................................................................................................... 45
  7.2.2 Ports......................................................................................................................................... 45
7.3 CruiseControl................................................................................................................................ 45
  7.3.1 FunctionParts.......................................................................................................................... 46
  7.3.2 Ports......................................................................................................................................... 46
7.4 EntertainmentSystem......................................................................................................................... 46
  7.4.1 FunctionParts.......................................................................................................................... 46
  7.4.2 Ports......................................................................................................................................... 47
7.5 RainSensor.................................................................................................................................... 47
  7.5.1 FunctionParts.......................................................................................................................... 47
  7.5.2 Ports......................................................................................................................................... 47
  7.5.3 Variability................................................................................................................................ 47
7.6 WiperController.............................................................................................................................. 48
  7.6.1 FunctionParts.......................................................................................................................... 48
  7.6.2 Ports......................................................................................................................................... 48
  7.6.3 Variability................................................................................................................................ 48
7.7 WiperMotor .................................................................................................................................. 49
  7.7.1 FunctionParts.......................................................................................................................... 50
  7.7.2 Ports......................................................................................................................................... 50
  7.7.3 Variability................................................................................................................................ 50
7.8 WiperSwitch ................................................................................................................................. 50
  7.8.1 FunctionParts.......................................................................................................................... 50
  7.8.2 Ports......................................................................................................................................... 51
7.9 WiperSystem ................................................................................................................................. 52
  7.9.1 FunctionParts.......................................................................................................................... 52
  7.9.2 Ports......................................................................................................................................... 52
  7.9.3 Variability................................................................................................................................ 53
8 Conclusions ...................................................................................................................................... 55
9 References........................................................................................................................................ 56

## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>Comparison of variability on vehicle level and artifact level.</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Configuration link.</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Constituents of a configuration decision.</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Product (sub)lines.</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Reference model and referring models.</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Two ways of adding variability to an EAST-ADL artifact (an FDA in this case).</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Example of a configuration link.</td>
</tr>
</tbody>
</table>

*(Figures of Sections 6 and 7 not shown in the above table.)*
1 Introduction

This deliverable provides a detailed overview of variability management in EAST-ADL. In contrast to the domain model documentation, the discussions here are less technical and more focused on the fundamental principles behind the variability concepts in the language. Also the initial motivation that led to these concepts will be discussed. Another important aim was to highlight the relations between the individual, variability-related domain model entities and how they fit together to form a comprehensive, consistent variability management approach.

Special attention has been paid to not let this deliverable become too abstract. Therefore, two chapters with a strong application focus were added: Section 6 provides an overview of the usage of the variability-related tooling in a tutorial style and Section 7 provides an extended example, presenting a fictitious Car product line with complex variability (mainly focusing on the wiper system) and shows how this variability is modeled and managed on different abstraction levels of EAST-ADL.

The authors hope that the reader will find this deliverable a good complement to the more technical definitions provided in the domain model.

Enjoy!
Domain Model

This chapter provides an overview of those parts of the EAST-ADL domain model which are related to variability management and product line engineering. In addition it is explained, how those parts of the EAST-ADL domain model are affected by variability management.

2.1 Vehicle Level vs. Artifact Level

In order to understand variability management in EAST-ADL it is essential to distinguish two levels of concern – the vehicle level and the artifact level – and to clearly understand the purpose and characteristics of variability management on each of these two levels.

Variability management begins on the Vehicle Feature Level, where model range features and variability are defined. At this point, the purpose of variability management is to provide a highly abstract overview of the variability in the complete system together with dependencies between these variabilities. A “variability” in this sense is a certain aspect of the complete system that changes from one variant of the complete system to another. “Abstract” here means that, for an individual variability, the idea is not to specify how the system varies with respect to this variability but only that the system shows such variability. For example, the front wiper may or may not have an automatic start. At vehicle level, the impact of this variability on the design is not defined; only the fact that such variability exists is defined. The language entity for representing such an abstract variability is called “feature”. In the example, we might thus represent the variability of the wiper by introducing an optional feature named „RainControlledWiping“. This is subsequently refined during analysis and design.

Of course, the fact that the system varies always comprises to some extent information on how it varies (e.g. the mere fact that it has a rain sensor supplied); the point of variability modeling on the vehicle level however is to keep this additional information as limited as possible while still being able to contemplate on the variability (e.g. dependencies with other variabilities).

Usually this vehicle level variability is highly technical and contains many details not intended for end-customers. Therefore, it is preferable to have a separate view on this technically oriented variability for end-customer configuration, in which the technical details of variability can be presented differently, partly hidden and diversely packaged, for example according to marketing considerations. Therefore, at least two viewpoints will usually be distinguished on vehicle level: (1) end-customer variability and (2) technical variability.

While the details of how variability is actually realized in the system are largely suppressed at the vehicle level, they are the focus of attention when managing variability in other areas of the development process (cf. Figure 1). In fact, specific variability may lead to modifications in any development artifact, such as requirements specifications and functional models. Here, describing that a specific variability occurs is not sufficient; it is necessary to describe how each variability affects and modifies the corresponding artifact.
Variability on the vehicle level:
- Very abstract; no design/implementation details.
- Distinction of customer vs. technical perspective.
- Modeling means: only Feature Modeling.

Variability on the artifact level:
- Variability of the actual requirements, design, etc.
- Only technical perspective.
- Modeling means: Feature Modeling + Variation Points inside FAA/FDA/...-Diagrams

Figure 1. Comparison of variability on vehicle level and artifact level.

2.2 Overview of Variability Modeling Concepts in EAST-ADL

Variability modeling in EAST-ADL is based on three major techniques: feature modeling, configuration decision modeling and multi-level feature management.

The purpose of feature modeling is to define the commonalities and variabilities of the product variants within the scope of a product line. Their main purpose is to provide a basis for configuration: for every feature model a feature configuration can be defined, i.e. a selection and deselection of its features. This configuration can be complete, if the decision of selection or deselection was taken for all features, or partial, in case not all decisions have been taken yet. This way, every feature model defines a set of valid feature configurations, also called a configuration space.

Feature models are normally used on a high level of abstraction, as described above for vehicle level variability. However, in EAST-ADL, they are also applied on analysis and design levels and acquire a much more concrete meaning there. A comparison is given in section 2.7, “Commonalities and differences on different Artifact Levels”.

Configuration decision modeling, on the other hand, is aimed at defining configuration, i.e. the binding of variability. The configuration of a feature model $F_t$ is defined in terms of the configuration of another feature model $F_s$. A configuration decision model can thus be seen as a link from $F_s$ to $F_t$ that allows us to derive a configuration of $F_t$ from any given configuration of $F_s$. Therefore, $F_s$ is called the source feature model and $F_t$ is called the target feature model.

Finally, multi-level feature models are a means to strategically manage two or more separate, independent product lines. With this instrument at hand, not all variants of the complete system need to be managed within a single, extremely complex global product line. It is, instead, possible to subdivide the product line into smaller, subordinate product lines (called product sublines) without losing the possibility to manage them from a global perspective.

In addition to these three major techniques, artifact-level variability heavily relies on the use of another concept called variable elements, i.e. elements within the artifacts that can be marked optional and are therefore not present in all system variants. For example, ADLFunctionPrototypes can be marked optional in this way, which means that the containing ADLFunctionType does not contain this prototype in all system variants. In addition, ADLFunctionTypes can be provided with a feature model that exposes the variability within this function, the so-called internal variability. Such a feature model is called the public feature model of the ADLFunctionType and can be seen as an extension to this component's interface dedicated to variability. Finally, an ADLFunctionType can be supplied with its own configuration decision model, called internal binding, which defines how to configure contained lower-level functions, depending on the configuration of the container function’s public feature model.
While feature modeling is applicable to all areas of the EAST-ADL in which variability management takes place, the other two modeling means are only valid to some areas. This is shown in Figure 1, where the applicable modeling means are given on the left.

### 2.3 Feature Modeling

Feature modeling is the core concept for variability management in EAST-ADL. Feature models are a well-established instrument for modeling variability and have been investigated by the research community for several years. Their basic principles and constructs are quite straightforward and can be explained in a few minutes. At the same time, they are very flexible and can be employed in a variety of ways. However, this simplicity and flexibility also yields an important danger: It is easy to misuse them and to have, within a single team or company, conflicting perceptions of how to employ feature modeling and what role they play within the overall development process. Another quality of features that adds to this difficulty is their high level of abstraction. For these reasons, it is necessary to not only provide a technique for feature modeling but also to describe precisely how to use it.

The basic entity of feature modeling is, of course, the feature. Precise definitions for the term feature vary to some extent, but in the context of the EAST-ADL the following definition is being used:

“A feature is a characteristic or trait that an individual product variant within a product line may or may not have.”

The definition refers to a product line and product variants. What are these? When discussing variability, you always – at least implicitly – refer to a set of several similar but still different objects: Variability only occurs when comparing several objects that share many common characteristics (otherwise they cannot be compared at all) but at the same time also show substantial differences (otherwise there is no variability). Each of these similar but still different objects can then be seen as a product variant, while the set of all objects or product variants is referred to as the product line.

As simple as this definition may seem, several very important observations have to be noted here.

- The product line and the product variants a feature model refers to may correspond to the complete system, but this need not be the case. For example, a feature model may show the variability of vehicles within the vehicle range of an automotive manufacturer, but it is also possible that a feature model only shows the variability of a single, subordinate software function (such as calculation of optimal wiper speed based on the output value of the rain sensor). In EAST-ADL, the first example would be realized as the feature model on VFM level, while the second example would be realized as the public function feature model of an «ADLFunctionType».

- The purpose of a feature model is only to name and list the differences between product variants. It is not intended to describe in detail how the product variants differ from one another.

- A feature is either present in a product variant or is not; this means during product configuration it can be selected or deselected (in case it is (de)selected the feature’s variability is bound).

- A feature does not necessarily correspond to a subsystem or component (features may correspond to a subsystem, e.g. “climate control” or “rain sensor”, but this need not be the case, e.g. for a feature such as “low energy consumption”).
• A feature can reflect cross-cutting variability that affects many different subsystems or subcomponents of the product line and product variants the feature model refers to; therefore, a feature model can be structured completely orthogonally to system design.

• A feature need not be customer-visible, e.g. power consumption.

• A feature is not necessarily a functional requirement. A feature is not necessarily a “function” in the sense of function-oriented development.

• A feature can reflect a dedicated viewpoint extracted from the system or subsystem considering experience and optimization on reuse with dependence on associated constraints from this perceptive, that for sure should be expressed as a requirement.

The fact that a feature is defined to be simply a (variable) “characteristic” or “trait” of some (variable) system forces decision for the high level of abstraction and the great flexibility of feature modeling. Virtually every attribute of a system that can be variable can be such a characteristic or trait.

A feature model can be seen as a list of features, i.e. a list of characteristics or traits in which the product variants differ from one another or are common to all product variants. The fact that also such features are listed that are common to all variants, deserves a closer look. At first sight it is not necessary to list such features because no configuration will take place with respect to them (they are always present). However, feature models are usually not only used for configuration, but also as an overview of a system’s core characteristics and traits, and in that context of course, also the invariant characteristics and traits are of interest. In this second use of feature models, the features are interpreted as coarse grained requirements and thus feature modeling can be seen as an early step in requirements analysis. In EAST-ADL this second use of feature modeling is particularly important to be directed from a unique point at VFM level. On artifact level however it is not as relevant, even though it may be appropriate in some case to show invariant features in an ADLFunctionType’s public function feature model at artifact level, and describe variation by adequate mechanism at the artifact level for example.

Each feature is attributed with a cardinality of the form [<lower limit> ... <upper limit>]. This cardinality specifies how often the corresponding feature can be selected in a single system. For example, the cardinality [0..1] means that the feature may be selected once or not at all, in other words it may be selected (1) or deselected (0). A cardinality of [1] means that the feature must be selected once, which means the feature is mandatory. An upper limit greater than 1 would mean that the corresponding feature may be selected more than once.

In addition, features are structured in tree form. This means that a feature $f_c$ may be specified to be a child feature of another feature $f_p$, which is then referred to as the parent feature. This means that during configuration, $f_c$ may only be present in the product if its parent $f_p$ is present; whether, in this case, $f_c$ is always present or is optional depends on the cardinality of $f_c$ as indicated above. Consequently, also features with a cardinality of [1] are not always present in the product if one of their predecessors has a cardinality with a lower limit of 0.

Finally, child features may be grouped in feature groups. Each such feature group has a cardinality that specifies how often the grouped child features may be selected. A cardinality of [1] means that exactly one of the grouped children must be selected, which means that they are exclusive (XOR). A cardinality of [0..1] means that one may be selected (optional XOR). Cardinality [1..*] means that at least one must be selected (OR).

Additional constraints and dependencies may be specified with feature links. This way, it can be specified that one feature “needs” another feature or that two features mutually “exclude” each other.

Many further details on feature modeling can be found in [1].
2.4 Configuration Decision Models (Configuration Links)

As described in the previous section, a feature model names and lists the characteristics and traits that are common to all product variants or only present in some of them. In contrast, configuration decision modeling is aimed at defining configurations: The configuration of a feature model $FM_2$ – i.e. the selection and de-selection of its features – is defined in terms of the configuration of another feature model $FM_1$. A configuration decision model can thus be seen as a link from $FM_1$ to $FM_2$ that allows deriving a configuration of $FM_2$ from a given configuration of $FM_1$. $FM_1$ is called the source feature model or configuring feature model while $FM_2$ is referred to as the target feature model or the configured feature model.

![Configuration Link](image)

**Figure 2. Configuration link.**

A configuration decision model is simply a set of configuration decisions. Each configuration decision reflects a single configuration consideration that affects and adds to the overall configuration of $FM_2$. Together, all configuration decisions define the configuration of $FM_2$. It is important to note that this configuration need not be a complete configuration, i.e. the decision whether to select or deselect optional features may remain unanswered for some features in $FM_2$. In fact, it is an important attribute of the concept that configuration decision models can also be used to partially configure a target feature model. In that case, several configuration decision models can be used sequentially to fully configure the feature model. The partial configuration between feature models should not be confused with the multi-level approach (see Section 2.5).

A configuration decision has the following structure:

![Configuration Decision](image)

**Figure 3. Constituents of a configuration decision.**

- **An criterion.** This is a logical expression that refers to the configuration of $FM_1$ and specifies the condition under which the corresponding configuration decision will take effect. For example, “$F_3$ is selected but not $F_4$” or “All US cars except A-Class” could be valid criteria, depending on the contents of $FM_1$.
- **A set of included features.** These features of $FM_2$ will be included (or selected) in case the corresponding configuration decision’s criterion evaluates to true. When a feature is included/selected in that way, all its predecessors can be implicitly selected. In fact, this is more complex than just denoting a set of features, because it must be possible to set
values of parameterized features and create instances of cloned features; these details are thoroughly covered in [1].

- A set of excluded features. These features of $\text{FM}_2(!)$ will be excluded (or deselected) in case the corresponding configuration decision’s criterion evaluates to true. When a feature is excluded/deselected in that way, all its successors can be implicitly deselected.

- A rationale. This textual string describes the reason for this configuration decision in natural language. This is used when evolving the configuration decision model to see whether a certain configuration decision is still valid or not.

- A person in charge. The person that took the decision represented by this configuration decision. This is used when conflicts between configuration decisions need to be resolved or when evolving the configuration decision model to find out who is responsible for deciding on how to adapt this particular configuration decision.

Additional entities are used to organize the configuration decisions within a configuration decision model. Configuration decision folders can contain configuration decisions and other configuration decision folders and thereby modularize configuration decisions that belong together, for example because they reflect configuration considerations that share a common background or relate to certain parts of the feature model being configured ($\text{FM}_2$ in the above examples).

Again, apart from the basic technical concept, it is a separate question of how to apply configuration decision modeling. What is the precise role of the source model and that of the target model within the overall development context? This will be investigated further below.

### 2.5 Multi-Level Feature Models

EAST-ADL provides a special technical concept to support management of the feature models on the VFM level called multi-level feature models. In this section this concept is described. Please note that in ATESST2 this technique was generalized to also support multi-level management of other parts of EAST-ADL models, not only feature models (e.g. requirements); however, since this deliverable is devoted to variability, we only cover multi-level management of feature models here.

When development of a new model range (e.g. “the new E-Class model to be introduced in 2007”) is initiated, work is based – in nearly all cases – on predecessor models. The only exception to this rule is the introduction of a completely new model range – but even then at least some of the subsystems will be initially taken from existing products. From a strictly technical point of view, development in the automotive domain can thus be seen as a process of amending predecessor model ranges to optimize them (e.g. to reduce fuel consumption), to meet new requirements or to add innovative functionality. This process is repeated over and over again. It is important to note that what happens here is more than just a versioning of information: the information about the previous models cannot be disposed of because these vehicles will still be in use for many years after production ends. And – even more importantly – the old model range will be developed further, not only for bug fixing but also because some subcomponents may no longer be available as spare parts and therefore have to be replaced by alternative components, which may require changes to the surrounding hardware/software system. Thus, the information on the new model range does not replace that on the old one but rather constitutes a new branch in development.

In addition, a car manufacturer’s model range is usually made up of several model ranges (e.g. A-Class, C-Class, E-Class), which are in turn often further subdivided in some way (e.g. E240, E320, E320 CDI with diesel engine). Also, there often is a breakdown into passenger cars, freight vehicles, other commercial vehicles and possibly further categories. Moreover, today’s large corporate groups usually consist of large subgroups or brands (e.g. Mercedes-Benz, Chrysler, Smart). This situation is shown in Figure 4. All these models, lines, series etc. are hierarchically related to each other: for example, all models of a particular model range share certain
characteristics, just as all model ranges of a vehicle series do. Sometimes model ranges are even further subdivided, leading to additional levels. In summary, this means there are several, hierarchically structured “contexts” in which development takes place. For these contexts, we use the term product subline in the following. Each shaded box in Figure 4 can thus be seen as such a product subline. In many cases, they are reflected in a car manufacturer’s organizational structure as divisions, departments or other organizational units or as projects. This gives them a significance that goes beyond methodological considerations.

Figure 4. Product (sub)lines.

When applying standard product line concepts – especially feature modeling – to this situation, one would have to choose between two basic approaches:

- Organizing each leaf product subline as a separate product family/line of its own; these sublines would then be represented by one feature model each.
- Organizing all product sublines in one global product family; the sublines would then become partial configurations of a single global feature model.

For automotive development, neither possibility is truly satisfactory. In the case of the first approach, this is immediately obvious: if the product sublines are treated as separate, independent product families/lines, there is no way to document and manage the commonalities and differences between them, which means that the benefits of product family-oriented software development are not available on the level above the individual (leaf) sublines. The shortcomings of the second approach are not as obvious. Theoretically, it would be possible to have one global feature model because all the company’s brands, lines and models are merely additional forms of variation of an imaginary company-global variable car. But such a monolithic feature model would be enormously complex. First of all, even for a single car model, well above a thousand technical features are necessary to distinguish which functionality is mandatory and which is optional in principle – regardless of final marketing decisions. Second, consider the large number of models offered by today’s automotive manufacturers such as General Motors, Renault or Volkswagen. And the number of different models is only the tip of the iceberg, because each model is offered in many different markets with differing legislation, in many different styles (e.g. Classic, Elegance, Sport), and so on. Thus, a monolithic feature model would be unmanageable, especially when it comes to adding new product sublines, implementing and managing changes in the tree’s structure over a long period of time or – most importantly – making local amendments, i.e. amendments that are valid for only a single or only a few product sublines. There are several reasons why such local amendments are so important:
• sometimes a product subline requires amendments that are so far-reaching and so special that they would unnecessarily complicate the feature models of other sublines a great deal (e.g. if the structure of the climate control’s subfeatures differed fundamentally from A- to S-Class)

• amendments may be needed to solve urgent problems that do not exist in other product sublines or to solve problems that will be treated differently in future models (stop-gap solutions)

• innovations devised in the context of a single product subline may need to be introduced locally at first in case the discussion and coordination with the departments responsible for the other product sublines would take too long or is not immediately necessary and can therefore be postponed (local innovations).

• Perception of subsystem reuse from other product lines (could be from supplier) gives an additional diversity that could inherit from solution development, from rationale definition or extra configuration that need to be controlled and inserted to final view.

For these reasons, a special procedure for modeling variability in complex organizational contexts is proposed here. It can be seen as a compromise between one global feature model and many independent models. In particular, it is possible to harmonize the feature models of individual sublines over a longer period of time, step by step, and move only those features to a higher-level product subline that are already harmonized and consolidated.

A reference feature model is a normal feature model, except that there is some other feature model – called the referring feature model – that refers to it as its reference model. In Figure 5, “Series Cluster” is the reference model for the referring models “Series A” and “Series B”. A feature model can only refer to one reference feature model at the most, but several feature models can refer to the same reference model. The reference feature model can itself refer to a third feature model as its reference model, e.g. if “Series Cluster” referred to a company-global reference model called “Company Cluster”. Furthermore, a feature in the referring feature model can point to a feature in the reference feature model as its reference feature. It is then called a referring feature. In the example, this is only shown for “Series A” and omitted for “Series B”, so as not to clutter the presentation too much. The concept of a reference model now allows for a hierarchically structured tree of reference feature models that reflect the situation of hierarchical product sublines described above. Each node in the tree shown in Figure 4 would then be represented by a feature model that is both a reference model (for the lower-level models) and a referring model (with respect to the higher levels), except for the root which is only a reference model, and the leaf models which are referring models only. Such a tree of reference/referring feature models is called a multi-level feature tree.
The reference model now serves as a template and guideline for the referring model by defining default features together with their default properties and by defining which deviations from these defaults are allowed. Thus, the reference model becomes a means to strategically drive the content of the referring model without inhibiting deviations where this is necessary. It is important to note that reference feature models are no longer used as a basis for product configuration specification (as described for the core technical feature model in the previous section). Instead, they affect product configuration only indirectly by influencing the content of the referring models, which are the actual basis for product configuration specification within their respective sublines as described in the previous section.

What deviations from the reference model are allowed in a referring model is defined in the reference model by way of deviation attributes that can be associated with each reference feature. An example of such a deviation attribute is “allowChangeName”. If set to ‘NO’, the name of the corresponding feature may not be changed in referring models, if set to ‘YES’, the name may be freely changed and if set to ‘APPEND’, only a suffix may be added to the name.

### 2.6 Concepts for Variability Management on Artifact Level

As described above, the nature of variability management on artifact level is quite different from that on the vehicle level. It is not only concerned with defining what characteristics are variable but also how a particular development artifact varies.

For example, consider a cruise control system. Its main controller unit varies depending on whether the cruise control is adaptive or not (in the adaptive case, not the throttle is kept constant but the vehicle’s speed, by dynamically adapting the throttle to various properties of the environment, such as going uphill vs. downhill etc.). We not only have to define that there is a
variation (i.e. constant vs. adaptive cruise control) but also how each variant looks in detail. In other words, in artifact-level variability management the development artifacts are themselves being made variable.

From the perspective of EAST-ADL, there are two fundamentally different techniques of introducing variability in an FAA, FDA, or HDA artifact:

1. The FAA, FDA, etc. structure is itself declared as variable. This means that within the internal structure of a FunctionType one of the FunctionPrototypes or Connectors becomes optional, for example.

2. An elementary FunctionType (i.e. one that does not contain FunctionPrototypes) is provided with its own public feature model.

Figure 6 shows an example for both cases. The rain sensor prototype “rs” and the rear wiper motor prototype “rear” have been declared optional. Hereby, the actual structure of the FDA becomes variable, because in some cases we have the rain sensor inside the wiper system and in other cases we don’t (correspondingly for the rear wiper motor). Clearly this is an example of technique no. 1 above.

But there are in this picture also examples of the second technique of introducing variability. As indicated by the green “F” symbols on RainSensor, WiperController and the two WiperMotors, these FunctionTypes are supplied with their own public feature model, even though they are elementary, i.e. do not contain sub-functions. Let’s consider what this means in the case of the WiperController: it is declared variable and its variability is defined in its public feature model (not shown here, see chapter 7.9.3.1 for details); if its behavior is defined by hand-written C source code, for example, then later the configuration of this feature model can be provided to the wiper controller’s C code (maybe in the form of C preprocessor macros). The important point here is that the public feature model of WiperController is our window to an otherwise not accessible variability, because this variability is defined outside the EAST-ADL FDA model (in the C source code, in this case). Therefore, as far as EAST-ADL is concerned, the public feature model of WiperController is actually introducing its variability into the system model in a coherent, uniform way, which is precisely the idea of technique no. 2 in the above enumeration.

The above example from Figure 6 is taken from the extended example presented in detail in Chapter 7, so much more detail on the aforementioned variability can be found there.
Let us come back to technique no.1 for a moment, i.e. introducing variability into an FAA, FDA, HDA model by making the model itself variable. In principle, there are two traditional ways of defining such variations in a software development artifact:

- Variation Point / Variant approach: explicitly modeled variation points, each supplied with several variants,
- Merge & Optionality approach: directly merging all variants into the artifact and then marking them optional (plus dependencies).

Whenever an artifact contains a variation, i.e. a certain part of this artifact is variable in the sense that different content can occur at this point in the artifact, we can either explicitly mark this part of the artifact as being variable with a dedicated modeling element (a so-called variation point) and define the different forms the artifact can have at this location (the variants) or we can simply define the alternative content side by side directly within the artifact and mark them optional (plus defining dependencies between the optional content). In the first case, resolving variability then means to select a single variant at each variation point and in the second case it means ruling out all optional content which is not applicable.

In the old version of EAST-ADL both of these approaches were introduced for expressing variations in artifacts. In most cases the variation point mechanism had been used, but sometimes also the merge & optionality approach was applied. In the new version of EAST-ADL, the merge & optionality approach is now consistently applied for all cases of defining artifact variations. A further discussion of this and a detailed comparison can be found in the ATESST2 deliverable I3.3.1 in Section 3.1.

### 2.7 Commonalities and differences on different Artifact Levels

The following table shows which of the variability-related domain model elements and conceptions are applicable to the various abstraction layers.

<table>
<thead>
<tr>
<th>FeatureModel</th>
<th>FAA</th>
<th>FDA</th>
<th>HDA</th>
<th>IA</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Feature</th>
<th>FAA</th>
<th>FDA</th>
<th>HDA</th>
<th>IA</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Feature (without configuration decision)</th>
<th>FAA</th>
<th>FDA</th>
<th>HDA</th>
<th>IA</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FeatureGroup</th>
<th>FAA</th>
<th>FDA</th>
<th>HDA</th>
<th>IA</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FeatureLinks</th>
<th>FAA</th>
<th>FDA</th>
<th>HDA</th>
<th>IA</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reference FeatureModel</th>
<th>FAA</th>
<th>FDA</th>
<th>HDA</th>
<th>IA</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Referring FeatureModel</th>
<th>FAA</th>
<th>FDA</th>
<th>HDA</th>
<th>IA</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Deviation attributes</th>
<th>FAA</th>
<th>FDA</th>
<th>HDA</th>
<th>IA</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration Decision Model</th>
<th>FAA</th>
<th>FDA</th>
<th>HDA</th>
<th>IA</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Configuration Decision</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>------------------------</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>selection (through CD)</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>deselection (through CD)</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>partial configuration decision</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>
3 Defining Configuration Decisions

One of the most important concepts in EAST-ADL variability management are configuration links. As explained above, such a configuration link defines how to configure one or more target feature model depending on a given configuration of one or more source feature models. The red arrow from left to right in Figure 7 illustrates this. For example, in the configuration link we could define that whenever feature “EMEA” is selected on the source side, feature “Radar” is to be deselected on the target side, maybe because local legislation prohibits the use of radar-controlled adaptive cruise controls.

![Figure 7. Example of a configuration link.](image)

The conceptual notion of a configuration link is realized in EAST-ADL by the domain model element ConfigurationDecisionModel (CDM in the following). A single such CDM is made up of any number of so-called ConfigurationDecisions (CD in the following). Each CD represents a single, atomic rule on how to configure the target feature model depending on a given source feature model. As illustrated in Figure 7 above, an example of such a rule might be “Use Radar in USA.” or, more technically, “If feature ‘USA’ is selected in the source feature model, then select feature ‘Radar’ in the target feature model”. Taken together, all the ConfigurationDecisions, i.e. individual configuration rules, in a ConfigurationDecisionModel then define how to configure the entire target feature model depending on a given configuration of the source feature model.

Alternatively, you can think of a configuration link and a ConfigurationDecisionModel as a transformation of FeatureConfigurations (i.e. configuration of feature models): with the definition of a ConfigurationDecisionModel you can transform any given configuration of the source feature model into a configuration of the target feature model.

A configuration decision has the following core attributes:

- a criterion: a logical constraint on the configuration of the source(!) feature model,
- an effect: a specification of one or more configuration steps to be performed on the target(!) feature model if the criterion evaluates to true.

In the above example of Radar to be used in the USA, the criterion would be “USA selected” and the effect would be “select Radar”. Note carefully how the criterion refers to the source feature model only whereas the effect refers to the target feature model only. Note further that the criterion only reads configurations (or tests if configurations are present) whereas the effect actively performs configurations.
The basic syntax for referring to a feature of a source or target feature model within a configuration
decision’s criterion and effect attributes is defined by the Variability Specification Language (VSL)
as detailed in the CVM technical report [1].

In the remainder of this chapter we will now investigate in more detail how to define a
ConfigurationDecision’s criterion and effect attribute. We will organize this discussion according to
the following questions:

1. What feature models can I refer to in a CD?
2. How do I refer to a particular feature model in a CD?
3. How do I check for a source configuration in the criterion attribute of a CD?
4. How do I effectuate a target configuration in the effect attribute of a CD?
5. Special case: How do I effectuate the selection or deselection of an optional
   FunctionPrototype?

The following sections deal with each of these questions in sequence.

Along the way we will often distinguish two fundamental use cases of ConfigurationDecisionModels
in EAST-ADL: (a) on the vehicle level (used there to link a product feature model to the core
technical feature model) and (b) as the internal binding of a FunctionType (there used to define
how the FunctionType’s internal variability is bound depending on a given configuration of the
FunctionType’s public feature model).

3.1 The Source and Target Feature Models of a ConfigurationDecisionModel

As indicated on the previous page, each ConfigurationDecisionModel has source and target
feature models. But not all ConfigurationDecisionModels have them defined explicitly, sometimes
they are deduced implicitly from the model. In any case the following rules apply:

• *Within the criterion attribute all source feature models can be referenced (and only these).*
• *Within the effect attribute all target feature models can be referenced (and only these).*

So, this leads us to the next question of which source and target feature models a Configuration-
DecisionModel has.

**Case (a):**

In case of a ConfigurationDecisionModel on vehicle level, the source and target feature models are
defined explicitly. The source feature models are always one or more product feature models, the
target feature model is either the core technical feature model or one or more other product feature
model. No feature model may ever appear both as a source and target feature model of the same
ConfigurationDecisionModel. But it is of course legal and desired that the target feature model of
one ConfigurationDecisionModel may appear as source feature model of another, thus realizing
chains of configuration links.

**Case (b):**

In case of a ConfigurationDecisionModel of a FunctionType (then called *internal binding*) all source
and target feature models are defined implicitly and are deduced from the containment hierarchy of
the respective FunctionType. Given a FunctionType FT, the ConfigurationDecisionModel use as
its internal binding has FT’s public feature model as its single source feature model and the public
feature models of all directly and indirectly contained sub-FunctionTypes as its target feature
model.

As an example consider the WiperSystem from Figure 6 above. Its ConfigurationDecisionModel
has the public feature model of WiperSystem as its sole source feature model and the public
feature models of sub-functions RainSensor, WiperController and WiperMotor as target feature models. WiperMotor is particularly interesting, because WiperSystem contains two FunctionPrototypes for it. In such a case, the public feature model of the FunctionType, i.e. WiperMotor, appears twice as target feature model and when defining configuration, the ConfigurationDecision can distinguish between the two. More on this in the next section.

### 3.2 Referring to a Feature of a Source or Target Feature Model

We have seen above that in the criterion attribute of a ConfigurationDecision we refer to the source feature model(s) of the parent ConfigurationDecisionModel and in the effect attribute we refer to the target feature models. But what we actually want is to refer to individual features of these feature models. The following basic syntax applies:

\[
\text{<FeatureModelName>} \# \text{<FeatureName>}
\]

Before the hash character the feature model is specified, after the hash a particular feature of this feature model is denoted.

In case of a ConfigurationDecisionModel on vehicle level, simply the name of the feature model is used to denote it. In case of an internal binding (i.e. a ConfigurationDecisionModel of a FunctionType) a dot-separated path of FunctionPrototype names can be used to denote the public feature model of a particular lower-level FunctionPrototype.

The feature specification following the hash simply names a particular feature. If there is more than one feature with the same name in the feature model, then again a dot-separated path of feature names can be used to uniquely identify a particular feature. For example if a feature model FM contains two features “Wiper” and “CruiseControl” each having an optional child feature named “Advanced”, it would not be enough to just state “FM#Advanced” but we had to either state “FM#Wiper.Advanced” or “FM#CruiseControl.Advanced”. This path of parent features can be of arbitrary length.

As a rule, it is legal to leave out everything that is not required to uniquely identify a feature. If among all target feature models only one contains a feature named “XYZ”, then we do not have to explicitly specify the target feature model and can simply state “XYZ” (also omitting the hash in this case).

As an example for the syntax and semantics of these references to features, assume there are two FeatureModels called FMa and FMb and they both contain the Features Wiper and ClimateControl. In FMa (but not in FMb !), Wiper and ClimateControl are both refined into the child features Simple and Advanced. In addition, the wiper in FMa has a RainSensor. To denote the RainSensor in FMa you can state:

FMa#Wiper.RainSensor

or simply write:

RainSensor

This is sufficient here, because the name of Feature RainSensor is unique within FMa and within all FeatureModels referenced by the ConfigurationDecisionModel. In contrast, to denote the advanced version of the climate control in FMa you can specify:

FMa#ClimateControl.Advanced

or simply:

ClimateControl.Advanced

but merely stating "Advanced" would not suffice because there are two features with that name. Finally, to denote the wiper of feature model FMb you write:
### 3.3 Testing for Source Configurations in the Criterion Attribute

We can test if a feature is selected or deselected in a given configuration of a source feature model by following the feature reference as defined in the previous section by either \([+]\) (for testing if it is selected) or \([-]\) (for testing if it is deselected).

Two or more such tests can be combined with the standard logical operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>or (\lor)</td>
<td>evaluates to true iff both operands evaluate to true</td>
</tr>
<tr>
<td>and (\land)</td>
<td>evaluates to true iff at least one operand evaluates to true</td>
</tr>
<tr>
<td>true</td>
<td>evaluates to true</td>
</tr>
<tr>
<td>false</td>
<td>evaluates to false</td>
</tr>
</tbody>
</table>

In a single criterion attribute, these can be combined arbitrarily with or without brackets as in ordinary logical constraints.

The default is \([+]\), so if a feature reference as defined in the previous section appears without any suffix, then this will test for a selection.

For a more complete definition of a ConfigurationDecision's criterion please refer to the CVM technical report [1].

### 3.4 Effectuating Target Configurations in the Effect Attribute

Similar as feature references were suffixed with \([+]\) or \([-]\) in the previous section in order to test for a particular source configuration, we can suffix them in the effect attribute to perform a particular target-side configuration (which will of course only happen if the ConfigurationDecision’s criterion evaluates to true, as explained above).

We can use the following suffixes:

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>([0])</td>
<td>Sets the feature’s cardinality to ([0]), thus excluding it from the target configuration.</td>
</tr>
<tr>
<td>([1])</td>
<td>Sets the feature’s cardinality to ([1]), thus including it in the target configuration; note that actual selection of this feature still depends on configuration of its predecessors.</td>
</tr>
<tr>
<td>([-])</td>
<td>Sets the feature’s cardinality to ([0]), thus effectively deselecting the feature.</td>
</tr>
<tr>
<td>([+])</td>
<td>Sets the feature’s cardinality and the cardinality of all its predecessor features to ([1]), thus effectively selecting the feature.</td>
</tr>
<tr>
<td>(&lt;iname&gt;)</td>
<td>Creates an instance of a cloned feature; (&lt;iname&gt;) will be the name of the newly created feature instance.</td>
</tr>
<tr>
<td>(&lt;value&gt;)</td>
<td>Sets the value of a parameterized feature to (&lt;value&gt;).</td>
</tr>
</tbody>
</table>
Several of such individual configurations can be included in a single effect attribute as a comma-separated list.

Configuring a cloned feature does not mean selecting or deselecting it but instead instances of the cloned feature are created. Each such instance is provided with a name, which thus becomes a part of the configuration (not the feature model). If several instances are created for a single cloned feature, then the name is used to identify these instances. For example, consider a cloned feature “Wiper” with cardinality [0..*]. A first configuration decision might create an instance called “front” and a second might create another named “rear”; a third configuration decision creating or otherwise referring to an instance called “front” will denote the same instance as the first configuration decision. The name space for these instance names is a particular feature configuration.

3.5 Selecting and Deselecting Optional FunctionPrototypes

Similar as for selecting and deselecting features, you can also define the selection or deselection of an optional FunctionPrototype in the effect attribute (only for ConfigurationDecisionModels of FunctionTypes; not on vehicle level). Simply refer to the function prototype with a dot-separated path and append either [+] or [-].

Assuming we had a FunctionType “Car” containing a prototype “ws” typed by FunctionType “WiperSystem”. The WiperSystem contains an optional prototype “rear” of type “WiperMotor” (this is taken from the extended example in Section 7). Then we could select the rear wiper motor from the effect attribute of a configuration decision in the internal binding of “Car” with

```plaintext
ws.rear[+]
```

For more samples see the extended example below in Section 7.

3.6 Summary

More details on the definition of ConfigurationDecisions and a multitude of further examples can be found in the CVM technical report [1].
Methodology

While the preceding chapter focused on language concepts in the EAST-ADL domain model, this chapter now introduces the corresponding methodology. In other terms, the previous chapter introduced you to certain technical concepts in EAST-ADL whereas this chapter tells you how to apply them. Again, the scope is confined to variability management and product line engineering.

4.1 The Role of Feature Modeling on VFM Level

The variability management on VFM level has a global perspective, i.e. the entire vehicle as the complete system is investigated here. This basic role of the VFM has an impact on how feature modeling is applied here which will be explained in this section.

In principle, the VFM consists of a set of $n$ feature models that each provides a certain viewpoint on the vehicle’s overall variability together with several configuration decision models that relate these feature models with respect to their configuration. There is no strictly fixed number of feature models you will have and what viewpoints they will represent, but there are several typical cases you will encounter in most or at least in many project contexts. The following three sections describe three such typical uses of a feature model on vehicle level and the role they play within the overall product line infrastructure.

Core Technical Feature Model

This is the most important variability model of the entire product line infrastructure. It captures the variability of the entire vehicle on an abstract level. It is used as the basis of all configuration specifications and for configuration of the complete system. Here is an example of its role for configuration specification: if a requirement is valid only for some variants, the information when the requirement is valid is expressed in terms of a selection and deselection of features of the core technical feature model (e.g.: “requirement $R_x$ is valid only if features ‘AdvancedWiper’ and ‘RainSensor’ are selected” with “AdvancedWiper” and “RainSensor” being features of the core technical feature model). Note that this will be achieved in most cases indirectly through intermediate feature models instead of a direct reference to the core technical feature model. But in the end all configuration specifications are – at least indirectly – based on the core technical feature model.

Several important facts have to be kept in mind when working on the core technical feature model:

- Despite its technical viewpoint, the structuring of the core technical feature model is not supposed to reflect hardware or software design. Instead the structure should be optimized for simplicity, understandability, adaptability over time and suitability for integration of the variability in very different subsystems and subdomains such as engine control and telematics. The aim is to provide a coherent view on the variability of the entire vehicle from a global perspective.\(^1\)

"Structure" of a feature model in this sense means how the features are subdivided and hierarchically organized: Whether a certain variable aspect is reflected by a single feature or whether it is split up in two features; whether a certain feature should be introduced as a

\(^1\) Note that sometimes it may make sense to follow the design/implementation structure in the core technical feature model. This is particularly true where a rough structure is very much consolidated and has a great relevance to understand the complete system. In principle, however, the core technical feature model remains independent of hardware / software design.
child of some other feature or whether it should rather be a sibling of it; whether a certain variability should be reflected by a feature corresponding to a subsystem / component or rather to a more abstract characteristic (e.g. “RainSensor” vs. “AdaptiveWiper”).

- The core technical feature model is not supposed to be used as a basis for function-oriented development, i.e. functions are the main modeling entities rather than hardware units (as it is often the case in industrial practice). Function orientation is supported by the EAST-ADL by way of the FAA.

- Apart from being a basis for configuration specification, the core technical feature model is also used as a specification of coarse grained requirements and therefore is at the same time an artifact of requirements engineering

Overall, the core technical feature model is the heart of the entire variability management. It is also a good way to grasp the complete system’s overall characteristics.

**Feature Model for End-Customer Configuration**

As noted above, the core technical feature model is not suited for end-customer configuration. The variability in the core technical feature model has to be specifically packaged and partly hidden before being presented to an end-customer. This is the purpose of the feature model for end-customer configuration, or end-customer feature model. It can contain features that reflect certain aggregations of options such as “Elegance”, “Sportive” or “Classic”. These can then be further subdivided in order to allow for a fine-tuning of these packages. Also model ranges such as “A-Class”, “C-Class” or “E-Class” may appear in the end-customer feature model.

A configuration decision model is then used to map this customer view on the variability to the core technical feature model. In other words, it is defined how to configure the core technical feature model for certain configurations carried out by the end-customer. In the end, we can derive a configuration of the core technical feature model from any given end-customer configuration.

**Feature Model for Non-Customer Configuration**

This feature model plays a similar role as that for end-customer configuration. However, while the end-customer feature model is used to isolate and restructure variability resolved by the end-customer, this model is used to present variability that is invisible to the customer and hence not resolved by him. The simplest example for this is the market for which the vehicle is produced, e.g. EMEA, NAFTA, APAC or more specifically Europe, USA, Japan and so on. But there are other important reasons why a configuration may be independent from the customer. For example, if the rain sensor can be supplied by three different hardware suppliers and each rain sensor variant generates a different output value, this leads to variability in the wiper control system that interprets the rain sensor value. The binding of this variability, i.e. the choice which variant of the wiper control system to use, depends on the rain sensor installed and this may in turn depend on the plant that the vehicle is assembled in or on which supplier can currently offer the rain sensor at the best price. Hence, this variability needs to be resolved independently from the end-customer. So, in this case we see that variability is handled on different abstraction layers, but essentially is managed on the VFM.

---

2 Still, requirements may be connected to features.
4.2 Methodology

An extensive methodology for EAST-ADL was devised in WP5 that also covers variability management. With respect to variability, it was mainly investigated how variability is introduced in a step-by-step manner within the FAA, FDA or HDA.

For example, variability can be added to an existing FunctionType, which means additional variants of this FunctionType suddenly become available to higher-level functions that contain this FunctionType (directly or indirectly through other FunctionTypes). The engineers responsible for the higher-level function must then decide to either invariably bind this newly introduced variability (in its internal binding) or publish this variability further up the containment hierarchy in some form.

In addition to this, several work tasks were identified to deal with a product line’s variability on the vehicle level, for example to decide which variations should be covered by the overall product line strategy and thus provided by the artifact level, i.e. the variably defined FAA, FDA, and HAD. This is called scoping of the product line.

Instead of here copying the related documentation from the WP5 deliverables, the reader is kindly asked to refer to the EAST-ADL methodology description as provided by WP5.
5 Plugins and Tooling

This chapter provides an overview of the variability-related plugins used or developed in ATESSST2. The following chapter then describes, from a wider angle, how they fit together and how this forms an ensemble that can be used for variability and product line management.

The tool support for variability in EAST-ADL was extensively refined in Q1 and Q2 of 2009 according to the review of EAST-ADL variability concepts in WT3.3 in Q4 of 2008. From Q3 of 2009 on, the main focus was on consolidation, improvement of usability and addition of new ideas that resulted from experiments and evaluation efforts in ATESSST2. The extensive study of the HAVEit demonstrator is particularly noteworthy in this respect and led to numerous improvements and corrections of the tools. At time of writing the current release of this tool support is 0.6.1.

Compared to the status at the end of ATESSST1, the tool was renamed from “IoVM” to “CVM Framework”, which stands for “Compositional Variability Management Framework”, highlighting the novel approach to handling variability in composition hierarchies of FunctionTypes, as employed in ATESSST2 and EAST-ADL.

This tool support for variability is provided in two major parts:

1. **CVM Variability Management Framework**: 
   core CVM with Model Editor, textual VSL Editor, etc.

2. **EAST-ADL Bridge for CVM Organizer**: 
   EAST-ADL related functionality

3. **EPM Function Editor**

No. 1 is a stand-alone version of CVM covering the fundamental variability management concepts of EAST-ADL but without relying on the rest of EAST-ADL. No. 2 then integrates CVM into the EAST-ADL language and framework, using the EAST-ADL profile from CEA as a basis. This also allows to use CVM together with Papyrus. No. 3 is an experimental domain specific editor for editing the core of EAST-ADL elements, esp. FunctionTypes and their internal structure. It was implemented to experiment with various editing related aspects of variability management in EAST-ADL, for example: how can the editing of a FunctionType’s public feature model and internal binding be best supported with editing and visualization functionality in a DSL editor.

The plugin is available on the ATESSST2 SVN in folder /WP3/3.3/CVM/

A short movie demonstrating the plugin and its use together with Papyrus is available on the ATESSST2 SVN in folder /WP3/3.3/Misc (also some screenshots can be found there).

Further information on the plugin is available on the web-site [http://www.cvm-framework.org/](http://www.cvm-framework.org/)
This chapter provides a quick overview of the core functionality of CVM without going into great detail. It also shows how the different parts of CVM - the model editor, diagram editor and the VSL editor - fit together.

**Prerequisites**
For this tutorial, you need an installation of Eclipse with CVM, either as a stand-alone RCP version or as an Eclipse plugin. For more information on installation please refer to the CVM manual, section "Installation".

1. **Creating a New File**

   We start by creating a new model file. Such model files have the extension ".cvm".

   **Plugin Version**
   1. In Eclipse, select from the main menu "File / New / Other ...".
   2. In the dialog that opens up, select file type "CVM Variability Model" from category "CVM"; then click "Next".
   3. Choose a container (i.e. an Eclipse project to contain our new file).
   4. Change the file name to "tutorial1".
   5. Select to create an empty file.
   6. Click "Finish".

   **RCP Version**
   1. In the RCP version, select from the main menu "File / New CVM File ...".
   2. Choose a folder and file name in the dialog and click "Save".
   3. Select to create an empty file.

   This opens up the CVM model editor showing our newly created file. As you can see, the file has already been populated with a single root element of type `VariabilityModel`. Such a variability model is the root container for all information managed by CVM.

   The default name for this is "New Variability Model". To change this name, double-click on the name of the element in the model editor and type "VM" and press Enter.

2. **Creating a Simple Feature Model**

   To add content to our variability model we will now create some *Features*. Features cannot be contained directly in a variability model; instead we first have to create a `FeatureModel`.

   1. Right-click the variability model element, now called "VM", and select "New Child / Feature Model" from the context menu.
   2. Expand the variability model element (by clicking on the "+" symbol).
   3. As above, rename the newly created feature model by double-clicking on its name. Call it "FM1".

   Now, your editor window should look similar to this:

   ![Feature Model Editor](image)

   In this feature model, we now define the variability of a wiper in a fictitious car product line:
Example 1: The wiper comes in two core variants: a simple and an advanced form. In the second case, more sophisticated interval modes are provided. In addition, the wiper may provide an automatic mode in which wiping is controlled depending on the current rain intensity. All our cars have such a wiper.

To create the features for this example, you have several options:

- Right-click the parent feature (or feature model) and select "New Child / Feature" from the context menu.
- Select a feature (or feature model) and press ALT+C to create a child feature or press ALT+S to create a new sibling feature.
- Select a feature (or feature model) and click the button on the CVM tool bar.

Obviously the simple and advanced forms of our wiper are alternative, so we need a feature group of cardinality [1]. To create such a feature group you again have several options:

- As for features above you can use the "New Child" context menu action, use a keyboard short-cut (ALT+G for create group, in this case), or the tool bar.
- Alternatively, you can select two or more sibling features which are not grouped yet and select "Group Features" from the context menu. This will create a new feature group and add the selected features in a single step.

Finally, to set cardinalities use the context menu and to edit the feature's names and descriptions double-click on them in the model editor.

The complete feature model for the above example then looks like this:

![Feature model diagram](image)

Note that we have set feature "Wiper" and the feature group to cardinality [1].

3. Feature Modeling in the Diagram Editor

To illustrate configuration links later in this tutorial, we want to add a second feature model. We could do this as before for "FM1", but let's use the diagram editor for this.

This is what we want to model:

Example 2: In contrast to the rather end-customer oriented variability specification in feature model "FM1" above, we now want to specify our wiper's variability on a more technical level. It may have different forms of wiping modes and may have a rain sensor. A continuous wiping mode is always present. In addition, an arbitrary number of interval modes can be chosen during configuration. Finally there is the option of a flexible interval mode, where the driver can adapt the duration of the wiping interval by rotating a special button.

First, we need to create a variability diagram. So:

1. Right-click the root variability model and select "New Child / Variability Diagram" from the context menu.
2. Rename the new diagram to "VD".
3. To now open the diagram editor, please right-click our diagram "VD" and select "Open Diagram" from the context menu. This opens a new editor window showing an empty diagram.

4. Make sure to expand the tool palette on the right side of this window; you might have to click on the little arrow on the right side.

If everything went smooth you will see something like this:

Creating a feature model here in this diagram is pretty straightforward:

1. In the palette, click on the tool for creating feature model (called "Feature Model", located in the "Elements" tool box).
3. The name is already selected for direct editing. Type "FM2" and press Enter here.
4. Enlarge the feature model's visual to make room for contained features, by clicking and dragging the lower-right corner of it. (NOTE: In the future, you can create and resize an element in one go by clicking and dragging in step 2 above.)

In the same way create 6 features and rename and arrange them as shown in the following screenshot (don't resize them, they will resize automatically if no size is set by the user).
Now add parent/child relations as follows:

1. Click on the "Parent / Child" tool in the palette.
2. Click on the parent feature (do not drag!).

In this notation, optional features are marked with a white circle and mandatory features are marked with a black circle (the usual FODA notation by Kang et al.). As you can see, all features are optional. Use the context menu by right-clicking the features in order to ...

- set the cardinality of feature "Continuous" to [1], i.e. mandatory and ...
- set the cardinality of feature "Interval" to [0..*], i.e. cloned. Note how this cardinality, which is not available in FODA feature modeling, is presented in the diagram.

Your editor window should now show something like this:

The diagram as presented in this screenshot could be saved to an image file. To do this, make sure nothing is currently selected in the diagram and then right-click the diagram's background.
Select "Save to Image ..." from the context menu and choose a file name and image format in the dialog.

We won't use the diagram editor for the remainder of this tutorial, so you can close it now.

4. Parameterized Features and Feature Links

Before proceeding, we first refine our feature model "FM2" a bit further, to make the discussions later in this tutorial, esp. on configuration links, more interesting.

The feature "Interval" with cardinality [0.."] represents an interval mode with a certain, fixed duration. However, we cannot define the actual duration of such an interval yet. To achieve this, make "Interval" a parameterized feature of type Float:

1. Select feature "Interval".

2. If the properties view of Eclipse is NOT visible already:
   Right-click feature "Interval" (in the model editor!) and select "Show properties view" from the context menu.

3. In the properties view choose tab "Basic" and set a check-mark on option "Parameterized Feature" (you might have to scroll down to see this option).

4. Select "Float" as the type (still in the properties view).
   The parameter section in the properties view for feature "Interval" should now look like this:

   ![Parameterized Feature Properties](image.png)

   As you can see, we could also define a minimum and maximum value here or provide a default value. But we do not want this in our case.

5. Add an informative description to feature "Interval" by double-clicking the corresponding cell in the model editor.

At this point you should have this:

![Feature Model](image.png)

Let us now add a final detail to our feature model: a dependency. For this purpose, we assume that feature "RainSensor" may not be combined with the "Flexible" interval mode (for some technical reason). Such a dependency or constraint is realized in CVM by way of a FeatureLink:

1. Select feature "RainSensor".
2. **If** the properties view of Eclipse is **NOT** visible right now:  
Right-click feature "RainSensor" (in the model editor!) and select "Show properties view" from the context menu.

3. In the properties view choose tab "Links" --> the properties view now shows a list of feature links to/from feature "RainSensor" (should be empty for now).

4. Click on button "Create ..." (in the properties view).

5. In the dialog that shows up: select the target feature for the new feature link, i.e. "Flexible" in our case.

6. Click "Ok"  
--> the new feature link should now have appeared in the list.

7. Change the type of the newly created link to "excludes" and set a check-mark in column "<->" to make it bidirectional (both can be edited directly in the table).

Your properties view should list a single feature link now, as shown here:

![Feature Link Example](example.png)

5. **Configuring Feature Models**

So let us configure this feature model now:

1. Right-click feature model "FM2" in the model editor and select "Configure ..." from the context menu.

2. Edit the configuration in the configuration dialog that shows up (see screenshot below).

3. If you click "Ok" on leaving the dialog, then a new feature configuration will be created that captures the configuration edited in the dialog.
If you right-click an existing feature configuration instead of a feature model and select "Configure ..." from the context menu, then you can edit this existing configuration instead of creating a new one.

### 6. Configuration Links

Several feature models can be linked with respect to their configuration by way of a so-called *configuration decision model*. This means with the information captured in a configuration decision model it is possible to derive configurations of one feature model, called *target feature model*, from valid configurations of another, called *source feature model*. Such an arrangement of source and target feature models linked together by a configuration decision model is also called a *configuration link*.

The actual elements for defining how to configure the target feature model depending on the configuration of the source feature model are called *configuration decisions*. As the name suggests, a configuration decision model is made-up of many *configuration decisions*.

To illustrate this, let us create a simple configuration decision model:

1. In the model editor, right-click the root variability model "VM" and select "New Child / Config Decision Model" from the context menu.
2. Double click the newly created element's name and enter "CDM" as the new name.
3. We now have to define the source and target feature models linked by our new configuration decision model: Right-click element "CDM" and select "Edit Source & Target Feature Models" from the context menu.
4. In the dialog that opens, select "FM1" from the tree view; then click button "Add as Source"; in the "Set Name" dialog that opens up, confirm by clicking button "Ok" without entering a name.
5. Similarly, select "FM2" from the tree view and click button "Add as Target" and again confirm with "Ok".

Now you should see something like:
Note that we could provide local names for the source and target feature models that reflect their precise role within the configuration link we are currently defining. In our simple example, however, we do not need this. (Remark: with such explicit naming we can even use the same feature model more than once as source or target, for example to create two distinct target configurations for the same feature model.)

6. Click button "Ok" in the dialog to get back to the main model editor.

We have now created a configuration decision model called "CDM" with "FM1" as source and "FM2" as target feature model. The next step is to actually define how to configure "FM2" depending on a given configuration of "FM1":

1. Right-click "CDM" and select "New Child / Config Decision" from the context menu. This creates a new configuration decision, i.e. a conditional rule how to configure "FM2". For the next steps, please make sure you are in the model editor's grid view by clicking the 'Grid' tab on the lower side of the editor window.

2. In the row of the new element, double-click the cell in the first table column and enter "Advanced OR Automatic". This is an expression in propositional logic on the configuration of "FM1" that states when this rule shall trigger, i.e. have an effect on the configuration of "FM2".

3. In the row of the new element, double-click the cell in the table column called "Included Features" and enter "Flexible". This is the configuration decision's effect in the form of a list of features to be selected whenever the condition in column one (cf. step 2 above) evaluates to true. In this case we just state that feature "Flexible" is to be selected.

4. Repeat steps 1 through 3 to obtain five configuration decisions with the following information:

<table>
<thead>
<tr>
<th>Criterion (step 2)</th>
<th>Effect (step 3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1 true</td>
<td>Interval$short, short:Interval=1</td>
</tr>
<tr>
<td>#2 true</td>
<td>Interval$long, long:Interval=4</td>
</tr>
<tr>
<td>#3 Advanced</td>
<td>Interval$medium, medium:Interval=2.5</td>
</tr>
<tr>
<td>#4 Advanced OR Automatic</td>
<td>Flexible</td>
</tr>
</tbody>
</table>
5. The criterion "true" is used to define defaults that are always applicable. The effects of configuration decisions #1 to #3 are special in that they not merely select an optional feature but create instances of cloned features and assign values to them.

Now, your editor window should look something like this:

With these configuration decision we have defined how to configure "FM2" depending on a given configuration of "FM1": we always have two default intervals "short" and "long"; the advanced version of the wiper gets an additional medium interval; the automatic variant, represented by feature "Automatic" in "FM1" gets a rain sensor; finally, the flexible interval denoted by feature "Flexible" in "FM2" will be selected both for the advanced version of the wiper and the automatic variant (and the case of an automatic advanced wiper, of course).

To try out interactively this newly defined configuration link you use a dedicated view of CVM called "Configuration Preview":

1. If the configuration preview is not open yet, select "Window / Show View / Other ..." from the main Eclipse menu. In the dialog select "Configuration Preview" and click "Ok".
2. Place this view below the main editor window.
3. Select configuration decision model "CDM" in the editor. (Note: do not select an individual configuration decision because this will put the configuration preview in editing mode.)

The preview now shows the source feature model "FM1" on the left and the target feature model "FM2" on the right:

You can edit the left side and the right side will automatically be updated with the configuration derived from the one you edited on the left. The above screen-shot shows the configuration of "FM2" for the Simple variant of the wiper with "Automatic" feature: no special medium interval but "Flexible" and "RainSensor" are selected. A yellow highlighting the effects of the configuration link took place. Play around with the configuration on the left and verify that the configuration on the right is updated correctly through the configuration link.

CVM provides much more support for editing configuration decisions and for automatically deriving target configurations from given source configurations. Furthermore, it is possible to define configuration graphs, i.e. complex networks of configuration in which the nodes represent feature...
models and the edges are realized by configuration decision models. However, this detail is beyond the scope of this overview tutorial.

7. Textual Variability Specification

Finally, let us examine how our sample variability model would look like in the textual variability specification language VSL.

For this, we can now close the model editor. So,

### Plugin Version

1. Close the CVM model editor.
2. In the view called "Navigator" or "Package Explorer" find the ".cvm"-file we have created above.
3. Right-click it and select "Convert to .vsl" from the context menu, -- > a new file with the same name but extension ".vsl" will be created.
4. Now double-click this new ".vsl" file, -- > the VSL editor will open.

### RCP Version

1. **NOTE:** In the RCP version, the context menu action for converting from ".cvm" to ".vsl" files is not available yet. Therefore you need to create a new ".vsl" file with a sample model for now, so... Select from the main menu "File / New VSL File ...".
2. Choose a folder and file name in the dialog and click "Save".
3. Select to create a file with a sample model.

This is what you should see now:

![VSL Editor](tutorial.vsl)

**NOTE:** The code in this screenshot was simplified for illustration purposes.

In the editor window that opened up, you can see the VSL code of our small example from above. We won't go into detail of the syntax of VSL here, but you can experiment with the syntax in the editor. Whenever you save the file, the syntax will be checked and errors will be highlighted.

8. Summary

We have seen how to create feature models in the model and diagram editor and how to configure them and store the configurations. We have also examined how to link two feature models with respect to their configuration by way of configuration links. Finally, the alternative textual representation using the variability specification language was briefly presented.
7 Extended Example

The model files of this example are available on the ATESST2 SVN under folder
/Examples/EAST_Examples_Variability/

This example presents a fictitious car product line to illustrate the EAST-ADL.1 variability modeling concepts.

The example demonstrates several important use cases of defining variability within a system model's design architecture and of propagating this variability up to higher containment levels in the FDA and then to the vehicle level. On the vehicle level, the main variability definition is the core technical feature model, which defines the product line’s variability on a global level for the complete system - but still from a technical perspective. In addition, another feature model called Customer_FM is defined to provide an end-customer-oriented view on the technical variability of the core technical feature models. Configuration decision models define how to map the variability from this customer-oriented feature model to the core technical feature model and from that to the FDA (called artifact level).

The best entry point for exploring this example is the function WiperSystem. Start with finding out what variability is defined for the contents of the wiper system (i.e. subfunctions of the wiper system), then see how this variability is partially hidden from the outside and presented by way of the public feature model of function WiperSystem; finally, investigate how this variability is propagated further up and its binding is defined on higher levels.

Contents

FDA
CruiseControl
EntertainmentSystem
RainSensor
WiperController
WiperMotor
WiperSwitch
WiperSystem
7.1 Vehicle Level

This is the EAST-ADL vehicle level where the fundamental functionality and characteristics of the complete system are defined. The diagram also shows the architectures defined, in this example only a functional design architecture is provided.

7.1.1 Feature model Customer_FM:

A feature model that provides an orthogonal view on the core technical feature model. It is specifically tailored to the end-customer view.

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>NorthAmerica [0..1]</td>
<td>Parent: Country</td>
</tr>
<tr>
<td>Europe [0..1]</td>
<td>Parent: Country</td>
</tr>
<tr>
<td>Model [1]</td>
<td></td>
</tr>
<tr>
<td>A-Class [0..1]</td>
<td>Parent: Model</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>RealWiper</td>
<td></td>
</tr>
<tr>
<td>Classic</td>
<td></td>
</tr>
<tr>
<td>Luxury</td>
<td></td>
</tr>
</tbody>
</table>
### Core Technical Feature model:

The core technical feature model of this sample car product line.

<table>
<thead>
<tr>
<th>RearWiper [0..1]</th>
<th>Parent: A-Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>C-Class [0..1]</td>
<td>Parent: Model</td>
</tr>
<tr>
<td>Classic [0..1]</td>
<td>Parent: C-Class</td>
</tr>
<tr>
<td>Luxury [0..1]</td>
<td>Parent: C-Class</td>
</tr>
</tbody>
</table>

#### 7.1.2 Core Technical Feature model:

The body electronics system.

|-----------------------------|-------------------------------|
| Cruise Control [0..1]       | Parent: Body Electronics System  
A cruise control that automatically controls the vehicle's throttle (i.e. keeping it constant) and optionally adapts the throttle to various characteristics of the environment (uphill vs. downhill, distance to front vehicle, etc.). |
| Wiper [1]                   | Parent: Body Electronics System  
A wind-shield and optionally a rear window wiper. |
| Simple [0..1]               | Parent: Wiper  
Simple form of the wiper system (fewer wiping modes, no flexible wiping mode). |
| Advanced [0..1]             | Parent: Wiper  
An advanced form of the wiper system (more wiping modes, etc.). |
### 7.1.3 Configuration decision model CustomerCDM:

This configuration decision model maps the customer-oriented view on the system's variability as defined in Customer_FM to the core technical feature model.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Parent</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RainControlled [0..1]</td>
<td>Parent: Wiper</td>
<td>When selected, the wiper system has a rain sensor and automatically controls the wiper(s) when it is raining.</td>
</tr>
<tr>
<td>RearWiper [0..1]</td>
<td>Parent: Wiper</td>
<td>When selected, an additional wiper will be provided for the rear window.</td>
</tr>
<tr>
<td>Roof [1]</td>
<td>Parent: Body Electronics System</td>
<td></td>
</tr>
<tr>
<td>Retractable [0..1]</td>
<td>Parent: Roof</td>
<td>A sunroof.</td>
</tr>
</tbody>
</table>

The A-Class model always has the simple form of wiper and never such special equipment as rain-controlled wiping or a retractable roof. Rear wiper and cruise control are optional (as defined by other configuration decisions).

The A-Class has a rear wiper as optional special equipment.

The A-Class has a cruise control only on the US and Canadian market.

The C-Class model always has the advanced form of wiper and a rear wiper. Other special equipment depends on the model range Classic vs. Luxury (as defined by other configuration decisions).

The Classic model of the C-Class has rain-controlled wiping but no sunroof.

The Classic model of the C-Class has cruise control only for the US and Canadian market.

The Luxury model of the C-Class has all the special equipment, i.e. cruise control, rain-controlled wiping, and a sunroof.
7.1.4 Core configuration decision model (from core technical FM to artifact-level):

This configuration decision model maps the variability defined in the core technical feature model to the artifact-level variability as defined in the FDA.

<table>
<thead>
<tr>
<th>Simple</th>
<th>AdvancedIntervalModes[-], RainControlled[-]</th>
<th>Never has rain-aware wiping.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advanced</td>
<td>FlexibleMode, AdvancedIntervalModes</td>
<td>Rain-aware wiping is optional here.</td>
</tr>
<tr>
<td>RainControlled</td>
<td>RainControlled</td>
<td>Rain-aware wiping as special equipment.</td>
</tr>
<tr>
<td>Advanced and ! RainControlled</td>
<td>! RainControlled[-]</td>
<td></td>
</tr>
<tr>
<td>RearWiper</td>
<td>! RearWiper</td>
<td></td>
</tr>
<tr>
<td></td>
<td>--&gt; fda.ws.rear</td>
<td>A rear wiper.</td>
</tr>
<tr>
<td></td>
<td>--&gt; fda.ws.rear[-]</td>
<td></td>
</tr>
</tbody>
</table>
7.2 FDA

The top level of the functional design architecture of this sample car product line. In this simplified example, only the wiper system is modeled in detail. The cruise control and entertainment system are only shown for illustration purposes and to give an impression of what else might appear here.

7.2.1 FunctionParts

ws : WiperSystem
The wiper system.

es : EntertainmentSystem
The entertainment system.

cc : CruiseControl
The cruise control.

7.2.2 Ports

None.

7.3 CruiseControl

A cruise control system that automatically controls the vehicle’s throttle (i.e. at least keeping it constant) and optionally adapts the throttle to various characteristics of the environment (uphill vs. downhill, distance to front vehicle, etc.). Not further detailed in this example.
7.3.1 FunctionParts

None.

7.3.2 Ports

None.

7.4 EntertainmentSystem

The entertainment system, including audio, etc. Not further detailed in this example.

7.4.1 FunctionParts

None.
7.4.2 Ports

None.

7.5 RainSensor

A rain sensor component that measures the rain intensity and provides this as a float via its output port. The minimum rain intensity can be configured via feature "Intensity" in this function's public feature model.

7.5.1 FunctionParts

None.

7.5.2 Ports

| intensity |

Direction: OUT

[no description]

7.5.3 Variability

7.5.3.1 Public feature model of function RainSensor:

| Threshold [1] : Float | The minimum amount of rain at which the rain sensor will fire, i.e. signal that it rains. This is a float value between 0 and 1. |
7.6 WiperController

The core controller of the vehicle's wipers. The wiping modes can be configured very flexibly by way of this function's public feature model. The controller must also be configured to state whether a rain sensor is equipped or not (cf. feature "RainSensor" in the public feature model).

7.6.1 FunctionParts

None.

7.6.2 Ports

switchPos
  Direction: IN
  [no description]

rainIntensity
  Direction: IN
  [no description]

wipingSpeed
  Direction: OUT
  [no description]

7.6.3 Variability
7.6.3.1 Public feature model of function WiperController:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RainSensor [0..1]</td>
<td>When selected, the controller will support a rain sensor.</td>
</tr>
</tbody>
</table>
| Interval [0..*] : Float | Parent: Modes  
                      | An interval mode. The value specifies the duration of the interval in seconds. |
| Constant [1..*] : Float | Parent: Modes  
                     | A constant mode. The value specifies the wiping speed in wipes per minute.    |
| IntervalFlexible [0..1] | Parent: Modes  
                        | An interval mode that allows the driver to select the duration of the interval through the wiper switch. |

7.7 WiperMotor

An actuator for the actual wiping. This function can be used both for the front and rear wiper, but has to be configured appropriately for each of these cases (cf. this function's public feature model).
7.7.1 FunctionParts

None.

7.7.2 Ports

speed

Direction: IN

[no description]

7.7.3 Variability

7.7.3.1 Public feature model of function WiperMotor:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SpeedFactor</td>
<td>A modifier for this wiper's speed. The actual speed of the wiper is the requested speed multiplied by this feature's value.</td>
</tr>
<tr>
<td>AutoReturn</td>
<td>When selected, the wiper automatically returns to the home position when it is stopped. Otherwise it will simply stop at the current angle.</td>
</tr>
</tbody>
</table>

7.8 WiperSwitch

A switch for turning the wiper on and off and for selecting the available wiping modes.

7.8.1 FunctionParts

None.
7.8.2 Ports

switchPosition

Direction: OUT

[no description]
7.9 **WiperSystem**

The wiper system of the car, including the wind-shield wiper and an optional rear window wiper.

Note that most internal, lower-level variability inside the wiper system has been marked private and is therefore not directly accessible from the outside; instead, a public feature model is provided for the WiperSystem function that publishes this internal variability to the outside. An internal binding is provided that maps configurations of this public feature model to the internal variability.

An exception is the optionality of the rear wiper motor: it is directly published to and therefore configurable from the outside of the wiper system (note the green marker on function prototype „rear“ compared to the red marker on the other prototypes).

![WiperSystem Diagram](image)

### 7.9.1 FunctionParts

**sw : WiperSwitch**

With this switch the wiper(s) can be turned on/off and the wiping modes can be selected.

**rs : RainSensor**

The sensor for measuring the rain intensity. This function prototype is optional (as indicated by its dashed border in the WiperSystem diagram).

**ctrl : WiperController**

The core controller of the wiper system. A single controller is used even in the case of having an additional rear wiper.

**front : WiperMotor**

The motor for the wind-shield wiper.

**rear : WiperMotor**

The motor for the wiper on the rear window. Note that this function prototype is optional.

### 7.9.2 Ports

None.
### 7.9.3 Variability

#### 7.9.3.1 Public feature model of function WiperSystem:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RainControlled [0..1]</td>
<td>When selected, the wiper system has a rain sensor and automatically controls the wiper(s) when it is raining.</td>
</tr>
<tr>
<td>AdvancedIntervalModes [0..1]</td>
<td>When selected, two additional interval modes are available.</td>
</tr>
<tr>
<td>FlexibleMode [0..1]</td>
<td>Parent: AdvancedIntervalModes&lt;br&gt;When selected, one of the two additional modes is a flexible interval mode which allows the driver to control the duration of the interval from the wiper switch.</td>
</tr>
</tbody>
</table>

#### 7.9.3.2 Internal binding of function WiperSystem:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>Modes that are always available: 1 interval mode and 3 contant modes.</td>
</tr>
<tr>
<td>AdvancedIntervalModes</td>
<td>The first of the two advanced intervals.</td>
</tr>
<tr>
<td>FlexibleMode</td>
<td>The second of the two advanced intervals in case flexible.</td>
</tr>
<tr>
<td>( AdvancedIntervalModes &amp; ! ( FlexibleMode ) )</td>
<td>The second of the two advanced intervals in case NOT flexible.</td>
</tr>
<tr>
<td>RainControlled</td>
<td>In case rain controlled wiping is selected, then we have to select the optional FunctionPrototype “rs” AND configure the wiper controller accordingly.</td>
</tr>
<tr>
<td>TRUE</td>
<td>front#AutoReturn[0]</td>
</tr>
<tr>
<td>TRUE</td>
<td>rear#AutoReturn[1]</td>
</tr>
<tr>
<td>TRUE</td>
<td>front#SpeedFactor=1, rear#SpeedFactor=0.8277</td>
</tr>
<tr>
<td>TRUE</td>
<td>Threshold=0.4772</td>
</tr>
<tr>
<td>FALSE</td>
<td>rear</td>
</tr>
</tbody>
</table>
8 Conclusions

The variability management concepts implemented in EAST-ADL provide several innovative techniques for dealing with variability management.

This applies in particular to the dedicated support for variability within composition hierarchies (i.e. within the functional analysis architecture, the functional design architecture and the hardware design architecture) as realized by the public feature model and internal binding of FunctionTypes. The main benefit of this *compositional variability management* is that this allows a seamless transition from a purely technical viewpoint to a global, management-oriented viewpoint. Also, some challenges of the customer-manufacturer relationship as well as the reuse of subsystems can be addressed with this concept.

Then, by also employing the EAST-ADL support for product feature models on vehicle level, customer-oriented feature models can be defined. This extends the transition of viewpoint from technical to global / management-oriented on to the marketing-oriented viewpoint of the end-customer perspective.

Some of these conceptions were already available in EAST-ADL at the end of ATESSST1. But with the additions and improvements from ATESSST2, we are now confident to say that the variability management concepts in the new version of EAST-ADL constitute a comprehensive, self-contained approach to variability management in automotive systems engineering and reuse which is applicable in real-world projects as evaluated in particular by the ATESSST2 HAVEit demonstrator.
<table>
<thead>
<tr>
<th>References</th>
</tr>
</thead>
</table>