Exemplar of Automotive Architecture with Variability

Kacper Bak, Marko Novakovic, Leonardo Passos
Generative Software Development Lab
University of Waterloo
Waterloo, Canada
{kbak, mnovakov, lpassos}@gsd.uwaterloo.ca

***Abstract*—This work presents an exemplar of automotive architecture with variability. We extracted it from the actual documentation of Adaptive Cruise Control subsystem. Adaptive Cruise Control is a system responsible for maintaining driver selected speed or driver selected headway. We modeled architecture of the system in two modern architecture description languages: AADL and SysML. We also modeled architectural variabilities by applying the upcoming standard of Common Variability Language by OMG. The work shows how to introduce variabilities into existing architectures and presents our experience with modern architecture description languages.**

***Keywords*-software architecture, ADL, variability, CVL**

I. Introduction

Cars are assembled from physical components, such as engine, brakes, or lights. Modern cars, however, are software-intensive systems whose functionality is mostly implemented as programs. Software provides information processing functions, such as navigation or driving assistance. Adaptive Cruise Control (ACC) is an example of such a system. It controls engine and brakes so that the car automatically maintains selected speed and keeps safe distance to other cars. Software applications can make the ride safer and more enjoyable.

Car manufacturers have been using Software Product Lines (SPLs) to deliver car-specific software from a shared code and model base. The premise of SPLs is to automate the process of code generation, thus making it cheaper, faster, and more reliable. Software families are often described in terms of features (user-relevant characteristics) and architecture (represented as components and connectors). Product line architecture has a high degree of variability, so that it can be tailored to specific requirements. For example, a configurable ACC can lower the speed if the vehicle is too close to another car, or can simply generate a warning on the display. Design of a variable architecture is a challenging problem in itself, as it requires domain-specific knowledge to determine common and variable parts.

Engineers have been using models to develop new cars. They have mathematical models of engines, car aerodynamics or vibrations. The same could not be said about software engineers who traditionally modeled their solutions directly in the source code. Nowadays, many companies turn to model-based software development and in particular to Model Driven Architecture (MDA) [1] to provide abstractions over the code. The interesting question is: how to represent architecture's variability? A well known method are preprocessor directives (e.g. IFDEF in C) that switch on and off blocks of code. This approach, however, is error-prone, low-level, and hard to validate. Another approach is to introduce variability into models by annotating them with presence conditions [2]. Recently, OMG has started working on Common Variability Language (CVL)[3] to introduce variability into existing models without modifying them. Similarly to other OMG standards (e.g. UML2 [4]), CVL is likely to become a widely-used industry notation.

In this paper we present an exemplar of automotive architecture with variability. We extracted the exemplar from the actual documentation of the Adaptive Cruise Control subsystem provided by a car manufacturer. Before and during the process of extraction we faced several questions:

- What languages to use for modeling architecture?
- How to introduce variability into the architecture?
- What are the advantages and drawbacks of different approaches to variability modeling?

To answer these questions we looked into modern architecture description languages and picked AADL and SysML as promising notations. We then applied CVL to model architectural variabilities and to couple variability and architectural models. Due to the lack of good tool support from SysML and AADL, we designed a domain specific language so as to facilitate the integration with CVL. Our work brings the following contributions:

1) To the best of our knowledge ACC is the first exemplar of the actual automotive system with built-in variability. The SPL community can use it to evaluate their languages and tools.
2) We modeled various aspects of ACC in two modern architecture description languages: AADL and SysML. We describe our experience with both languages and corresponding tools. Designers of both languages can use our insights to improve their notations.
3) We used CVL to model variabilities and provided a short evaluation of the draft of proposed standard. This work gives an early feedback to CVL designers. It also shows how to apply CVL to existing models.

The paper is organized as follows. We introduce Adaptive Cruise Control subsystem in Sect. II. We describe its AADL model in Sect. III and SysML model in Sect. IV. We discuss ACC variabilities and different approaches to variability in

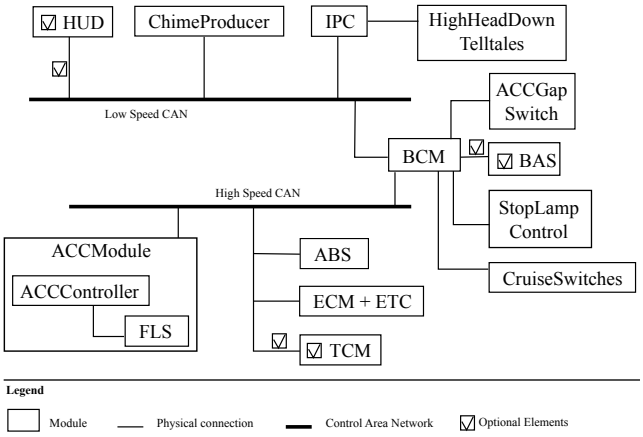


Figure 1. Physical connections diagram

Sect. V. We conclude in Sect. VII, after having described our prototype CVL application to ACC in Sect. VI.

II. Case Study: Adaptive Cruise Control

Adaptive Cruise Control (ACC) is a system responsible for maintaining driver selected speed or driver selected headway depending on the environment. It is an extension over Cruise Control system that only maintains driver selected speed, leaving the driver responsible for shifting gears and/or braking if the preceding vehicles drive slower than the host car. The driver of the host car with Adaptive Cruise Control system can set the speed and headway mode. If there are no preceding vehicles, the system maintains driver selected speed. As soon as preceding vehicles shows up, the system may apply braking, control throttle or shift gears to maintain driver selected headway.

There are many variabilities among different car manufactures or models. Those variabilities contribute to the way ACC controls the host vehicle. For example, in some cars it is impossible to apply automatic braking; other cars do not have automatic gear transmission. Driver selected speed can be displayed in miles or kilometers per hour, etc.

A. Architecture Extraction

We extracted main parts of the ACC architecture from the official documentation that we received from one of automotive manufacturers. Figure 1 shows ACC architecture with connections between ACC system and other vehicle systems. Here we present modules from the diagram.

A vehicle with installed ACC usually contains many other embedded systems. Those systems communicate with each other over a bus, such as Controller-Area Network (CAN). Depending on the required bus speed, devices use High Speed CAN (**HSCan**) or Low Speed CAN (**LSCan**). We modeled the core part of ACC as **ACC Module**. ACC Module contains **ACC Controller** and **Forward Looking Sensor** which is a sensor that scans the road and determines the speed of the preceding vehicles. There is also **Forward Collision Alert** that provides information-only functionalities (i.e. if there are preceding vehicles it takes no action other than displaying information to the driver). It is important to notice that some of those elements are optional, since some vehicle configurations might exclude them.

Transmission Control Module or **TCM** is on of such optional devices. It is used for automatic transmission so as to control car speed. If there is a need for braking, **ACC Module** sends appropriate signal to **ABS** Module which then applies the brakes to slow down the car. Vehicle speed can also be changed by increasing/decreasing throttle injection. This is done by **ECM** (Engine Control Module) and **ETC** (Electronic Throttle Control). All of those information/requests are represented as signals transmitted over the HSCan bus.

Driver-interfacing parts are modeled as *Driver Output Devices* and *Driver Input Devices*. Driver Output Devices are the top elements in the Fig. 1. **HUD** or Head Up Display is an optional device, and is used for projection on the front screen. **IPC** or Instrument Panel Cluster is a panel that is in front of the driver and contains other output devices, such as **HighHeadDown Telltales** (small screen next to speedometer). **Chime Producer** is a device that is used for playing sounds in vehicles.

Driver Input Devices read driver inputs for setting driver-selected headway or speed. **CruiseSwitches** are On, Off, Set, Resume switches, and are used for resuming or canceling ACC functionalities, or for setting driver selected speed. **ACCGapSwitch** is used for increasing or decreasing Headway. Choices and number of those switches vary among different car models. The **Body Control Module** (BCM) shields the input and output devices from the other systems in the car. Every access to these devices must first pass BCM, which then distributes received signal to the LSCan Bus. Signals from the input/output devices also pass by BCM, which then broadcast then into the HSCan.

There is another optional device: Brake Apply Sensing (**BAS**). it allows illumination of the stop lamps during automatic braking, with the help of **StopLampControl**.

III. Architecture in AADL

AADL stands for **A**rchitecture **A**nalysis & **D**esign **L**anguage [5]. The language aims at modeling embedded real-time computer systems. It was designed for automotive, avionics and aerospace systems, and contains domain-specific modeling constructions. Depending on modeler's preferences, AADL provides two notations:

1) *Textual notation*. AADL is developed to be an Architecture Description Language. The language has a simple and concise syntax that is easy to understand and use.
2) *Graphical notation*. Each of the main concepts of the language has a graphical representation. One needs proper tool support to benefit from using the graphical notation. Switching between graphical and textual representations is done without losing any information in the model.

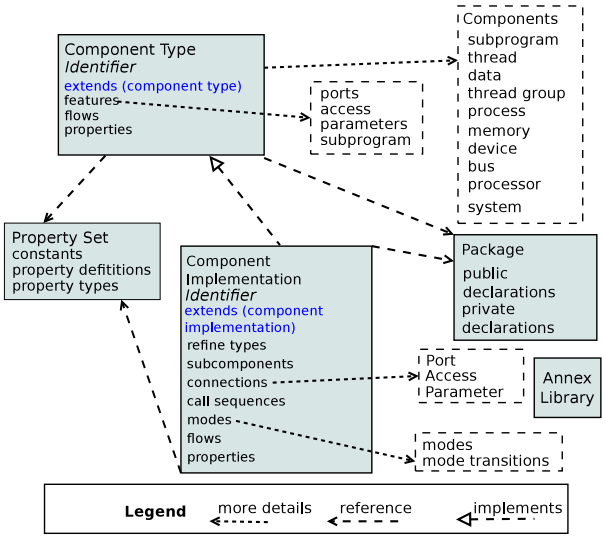


Figure 2. AADL Elements

The main element of AADL is called **Component**. It is a superclass of other AADL concepts. AADL models are essentially sets of components and connectors. The latter specify interactions among components. There are three main types of components in the language :

1) Application software
   - **thread**, **thread group**, **process**: similar to operating system's concepts with the same name and thread group components within a process
   - **data**: used for creating custom data types
   - **subprogram**: concepts such as call-return and calls-on methods Subprogram is used for modeling behavior
2) Execution platform (hardware)
   - **processor**: schedules and executes threads. Can be seen as a processor in PCs.
   - **memory**: stores code and data
   - **device**: represents an abstraction of any component that communicates with the external environment. Example: Forward Looking Sensor
   - **bus**: interconnects processors, memory, and devices. Example: LSCAN in Adaptive Cruise Control model
3) Composite
   - **system**: system is used for integration of other components into distinct units within the architecture. Example: ACC Module

Listed components are described in more details in Fig. 2. Components can have **features**. Features can be seen as data the component provides or requires. Features can contain more **in data port** and **out data port** definitions. There is also **port group** for aggregating input and output ports.

AADL distinguishes models and their implementations. Each model component can have multiple implementations. In implementation one can model the internals and details of a system. Implementation part in the model can be seen as another, more detailed view of architecture. For example, connections between components can only be defined when creating **system implementation**.

A. Adaptive Cruise Control Modeling

We applied AADL to model Adaptive Cruise Control system and to assess the language along that way. The process of modeling was iterative, i.e. we were learning the language and creating models at the same time. This approach allowed us to avoid bad practices early on and to refine the models over time.

To improve the quality of the models, we focused on modeling the structural aspect of the Adaptive Cruise Control system. AADL's minimalistic syntax was very convenient for structural modeling. We started with representing external hardware. We then connected this hardware to the ACC system. Next, we refined the internals of hardware and software elements. In addition, we connected hardware with software and modeled signals exchanged in the system by means of the High Speed CAN and Low Speed CAN.

```
system implementation car_system.i
  subcomponents
    acc_module_subsystem:
    system acc_module::acc_module;
    ...
    head_up_disp_device_subsystem:
      device head_up_disp_device;
    ...
 connections
    bcm_to_acc_connection : port group
    bcm_device_subsystem.
     bcm_to_acc_data_output
       → acc_module_subsystem.
         bcm_to_acc_data_input;
    ...
end car_system.i;
```

Figure 3. Part of the ACC model in AADL

Figure 3 shows an extract from the ACC model. It is a part of AADL system implementation of the *Car System*. The Car System contains all the hardware and software that is communicates with ACC. There are components (under the **subcomponents** section) and a list of **connections**. Examples of subcomponents include acc_module_subsystem and head_up_disp_device_subsystem. The former represents Adaptive Cruise Control core part - the logic providing ACC functionalities. The latter represents High Head Up Display, modeled as an AADL device. The display shows information about the current state of ACC to the driver, e.g. driver selected speed and headway. Furthermore, there is also one connection visible in Fig. 3. It contains description of the group of signals coming from Body Control Module bcm_device_subsystem to the ACC Module acc_module_subsystem.

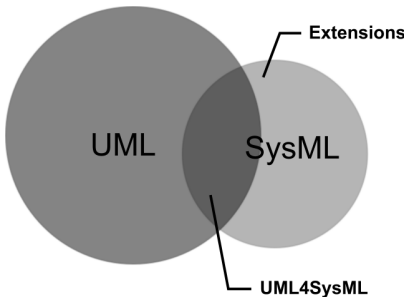


Figure 4. The relationship between SysML and UML modeling languages.

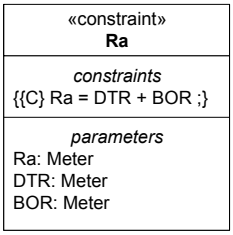


Figure 6. A range constraint (Ra) defined over the the ForwardCollisionAlert block.

B. Pros

Syntax: AADL has a very simple syntax that is suitable for modeling the structural part of Adaptive Cruise Control.

Documentation: The official website[1] has a link for downloading the Starter Kit, which is a well introduction to new users. It contains many AADL models that can be imported into the official IDE for AADL[2]. Also, documentation provided by the Starter Kit contains manuals explaining many aspects of AADL and presents interesting use cases.

C. Cons

Tool support: Tool support is not good enough. There is only one free IDE that is still in early phase of development. For example, graphical notation is lacking flows (connections), and the autocomplete option for textual representation is not helpful.

Syntax: Syntax for some parts of the language is hard to follow due to a list of multiple keywords; e.g. in definitions of *property sets*.

Behavior modeling: Behavior modeling is not supported in the original version of AADL. It is provided as an extension in the Behavior Annex. Although AADL can model behavior, there are no first-class constructions for expressing it.

IV. Architecture in SysML

SysML (***Sys**tem **M**odeling **L**anguage*) is a general purpose modeling language for system engineering that satisfies the requirements of the UML System Engineering Request for Proposal, as published by OMG [6].

SysML is based on a subset of UML, called UML4SysML, along with some extensions on its own (the SysML profile). The relationship between SysML and UML is shown in Fig. 4.

SysML defines nine diagrams that altogether allow one to model the behavior and the structure of a system. To limit scope, we modeled only the structural part of the ACC exemplar in SysML. For this task, all structural diagrams that the language provides were used so as to model different aspects of the system. Next, we describe each of them along with some examples. We assume some familiarity with UML modeling language and its vocabulary.

Package Diagram: groups elements that share a common responsibility, semantics, etc.

Block Definition Diagram: allows modeling of the entities (blocks) of a given domain. In system engineering, a block might represent either a physical or a logical element from a given domain. A block in SysML corresponds to class in UML. Figure 5 presents the core blocks in the block definition diagram for the ACCModule. The ACCController is an ECU - *Electronic Control Unit*, which is a dedicated hardware responsible for controlling a specific part of a car. Modern cars can have as many as 80 ECUs [7]. Each ECU runs an instance of a software; in this case the ACCController runs the ACCControllerSoftware. The ACCControllerSoftware may have a ForwardCollisionAlert, responsible for warning the driver of a imminent collision. The ACCController communicates directly with the ForwardLookingSensorController, which processes the environment and detects other vehicles within a given range. All communications of the ACCController to and from other modules of the car are performed by broadcasting signals through the HighSpeedCAN.

Internal Block Diagram: zooms into a given block. Generally, it is used for modeling the relationship of a given block to other blocks in terms of port connections. It is also used to define the set of constraints (generally seen as equations) that must be applied to a given block. Consider, for instance, the constraint Ra defined over the ForwardCollisionAlert block. It defines the range, measured in meters, in which the ACC must alert the driver of a possible collision. The Ra constraint uses two parameters, DTR (projected range of the vehicle ahead of the ACC vehicle) and BOR (projected range in which the vehicle will start braking), both defined as constraints. A constraint in SysML is coded in a specific programming language. In the presented diagram, the constraint was defined using the C programming language.

Parametric Diagram: shows how a given constraint can be calculated in terms of the block properties and how the result of one constraint calculation can be passed to either a parameter of another equation or stored as a property value. Properties in SysML resemble attributes in UML.

Parametric diagrams allow the simulation of the equa-

[1] http://www.aadl.info

[2] OSATE: http://www.aadl.info/aadl/currentsite/tool/osate.html

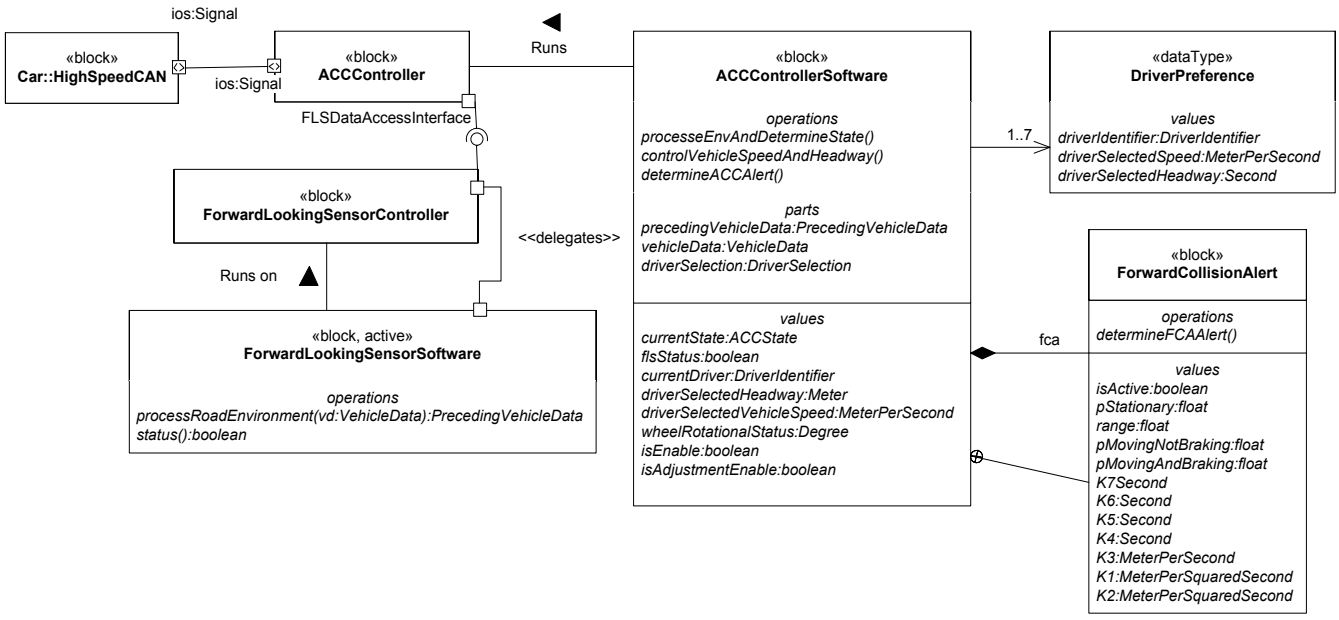


Figure 5. A partial view of the ACCModule block definition diagram.

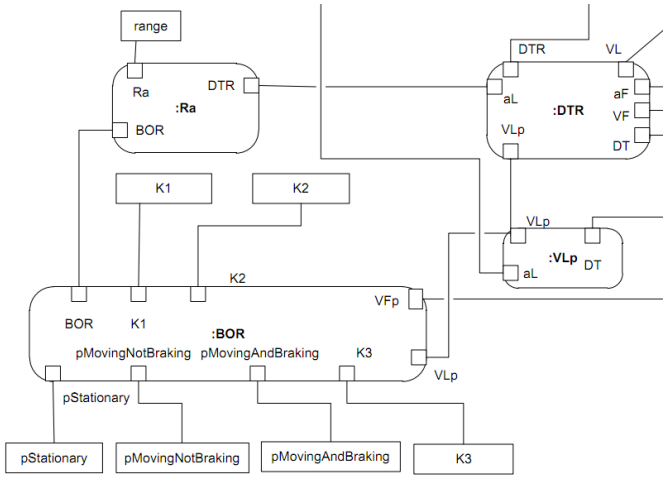


Figure 7. A fragment of the parametric diagram for the ACC.

tions associated with a given block if one configures the set of property values. Consider, for instance, a fragment of the parametric diagram defined for the ACC, as in Fig. 7. The Ra constraint outputs its value to the range property, defined in ForwardCollisionAlert. It uses the values produced by the BOR and DTR constraints with its corresponding parameters. Note that the direction of input/output values is implicit in parametric diagrams.

A. Pros

SysMŁwas expressive enough to model all aspects of the ACC system given the documentation provided by an automotive company. Although only structural modeling was used, SysML allows one to easily model behavior, which is not a straightforward task in AADL.

B. Cons

Our experience with SysML shows that constraint and parametric diagrams are very useful to allow one to simulate equations. However, parametric diagrams restrict modelers to bind equation parameters only to block properties. One cannot, for example, bind a parameter to an operation (method in UML) in order to receive a value from it. We believe that this is too restrictive.

Another drawback from SysML is the lack of good supporting tools. We evaluated the following set of tools when experimenting with SysML: Topcased [3], Papyrus [4], MagicDraw [5] and Microsoft Visio [6]. Papyrus and Topcased, although promising tools, are not yet mature; usability proved to be an issue and we found many compromising bugs while using them.

MagicDraw supports SysML by means of a plugin [7]. Together, they provide simulation and validation of models. However, due to the commercial characteristic of the plugin, which is not freely available for academic purposes, we did not further explore it.

Microsoft Visio: in terms of the quality of the produced diagrams and usability, Visio proved to be the best tool. By means of a third-party stencil [8], Visio supports all elements defined by the current SysML specification [6]. However, it produces raw diagrams, from which it is impossible to perform any kind of consistency checking, nor simulation.

Another drawback of SysML comes from lack of documentation. Currently, there are only two main books ([8],

[3]http://www.topcased.org
[4]http://www.eclipse.org/modeling/mdt/papyrus
[5]http://www.magicdraw.com
[6]http://office.microsoft.com/en-ca/visio
[7]Available at: http://www.magicdraw.com/sysml
[8]Available at http://softwarestencils.com/sysml

[9]) about it besides its official specification [6].

V. Variabilities

The extracted ACC subsystem is deployed in a range of car models. Different cars serve for different purposes and are further customized to match client's needs. Although car models significantly differ from each other, they are built from shared components. Such a production process is more economical and arguably less error-prone than building new components from scratch for each car model. The process significantly impacts ACC subsystem's architecture by introducing variation points to support components reuse. For example, a configurable ACC can lower the speed if the vehicle is too close to another car, or can simply generate a warning on the display. Variation points are removed during the configuration stage by specializing them with chosen option. Selection of a particular set of options results in a single product.

A. ACC as a Software Product Line

Adaptive cruise control is an interesting example of configurable software-intensive system. First of all, it is a software product line, i.e. a family of related software assembled from shared assets, such as models or components. Furthermore, it is an embedded system where the distinction between software and hardware is rather blurry, since a piece of software can be easily turned into FPGA hardware. Thus, a change in ACC configuration affects software, hardware and systems architecture.

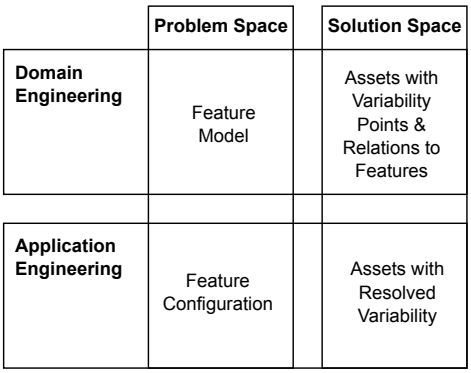


Figure 8. Idealized View of Software Product Lines

Software product lines are traditionally described in terms of *problem space* and *solution space*, as shown in Fig. 8. Problem space captures high-level requirements, usually in the form of feature models [10]. Feature models are tree-like structures that specify commonalities and variabilities within a software family. Solution space contains shared assets, such as models, or components and connectors. There is no one widely-accepted notation for solution space, as it depends on the domain. In our ACC exemplar, AADL and SysML models express architecture, but the complete solution space includes mathematical models of car physics. Mathematical models are used for simulation and code generation for target platform. Although this aspect is important in embedded systems, we excluded it due to time constraints.

Figure 8 also shows an orthogonal view of software product lines, i.e. *domain engineering* and *application engineering*. Domain engineering focuses on modeling software for a particular application area. It contains concepts and assets for expressing a range of products. For example, we scoped our course project to engineer the automotive domain. Application engineering, on the other hand, focuses on generating a single product from the domain. It contains configuration knowledge, which applied to domain assets, removes variability from them. Configuration knowledge is usually a set of binary decisions, or a set of (numerical, string) parameters.

B. ACC Variability Specification

The extracted ACC subsystem has two classes of variabilities: optional elements and calibrations. Optional elements are binary decisions that result in inclusion or exclusion of a particular asset. Exponential growth of binary decisions leads to hundreds of possible ACC configurations. Calibrations, on the other hand, are value assignments to variables. The ACC subsystem has tens of calibrations that can be fine-tuned within a certain interval. For example, the minimum speed to enable ACC must be between 30 and 50 km/h with resolution of 1 km/h. Value specifications increase the number of valid ACC configurations to the order of billions. Even though calibrations have more degrees of freedom than optional elements, they have lower impact on system's architecture. Therefore, we focused on modeling binary decisions.

Feature models naturally express binary decisions by means of *mandatory features*, *optional features* and *group cardinalities*. Figure 9 shows a feature model of the ACC exemplar. Mandatory features (marked with filled circles) must be present in all configurations; e.g. each ACC must have a forward looking Sensor to measure the speed of preceding car. Furthermore, ACC must be connected to Powertrain to be able to increase speed or request braking. Part of ACC is exposed to the user who can turn on/off the subsystem by Switches on the driving wheel and see status on the Display.

Optional features (marked with empty circles) may or may not be present in all products; some versions of ACC may issue forward collision Alert to the driver if the preceding car is to close. In other versions this feature may be unavailable and ACC will automatically reduce the speed. The Cancel button is another example of optional feature. Some car models might allow the driver to disengage ACC by pressing the button.

Group cardinalities impose restrictions on the number of selected children. In Fig. 9 there are only xor cardinalities (marked as empty arcs) that require exactly one subfeature to be selected from the group. For example, ACC must have either Radar or Lidar sensor. The Powertrain subsystem stores DriverIdentifiers and recognizes either Single or Multiple drivers to load their preferences. Most of the

groups include two elements, but it is not always the case. For example, there are three types of displays: projection on the front screen (HeadUp), a display placed between the driver and the passenger (HighHeadDown), or a small screen next to speedometer (Cluster).

C. Variability Implementation

One of the recurring questions in SPL community is: how to implement software variabilities in the source code? Answering the question alone is not enough, as there are several factors that must be taken into consideration: *code quality*, *maintainability*, *evolvability*, *scalability*, and *error-proneness*. Here we briefly describe several approaches and discuss their strengths and weaknesses.

1) Cloning: Cloning is a manual technique for deriving software products from shared assets. A programmer looks at requirements, selects relevant models and components from a shared pool and glues them together. This technique is so simple, that it does not require specialized knowledge or tool support, and can be applied to virtually any type of assets. Programmers have known code-clones for decades. It is one of the reasons why cloning is used in the industry for applications of moderate size. Other than that, cloning is error-prone, does not scale, and leads to different versions of initially single component.

2) Annotations: Annotations are probably the most popular means of introducing variability. They are usually implemented as presence conditions [2] or preprocessor directives (e.g. IFDEF in C). Presence conditions are logical formulas attached to model elements. An engine processes the model and excludes elements whose presence conditions evaluate to false. Preprocessor directives play similar role to presence conditions, but they annotate the source code. Preprocessor reads input file and conditionally includes parts of the code. A good and complex example of using preprocessor for variability realization is the Linux kernel [11]. Despite the ease of use, fine-granularity, and familiarity to programmers, preprocessor directives have been heavily criticized in academia for negative impact on code quality and maintainability [12], [13]. Recent work by Kästner [14] argues that preprocessors are still a reasonable solution but better tool support is needed.

3) Modularized Variability: Modularization and separation of concerns are two main ideas that tame software complexity. Modularized variability implements features in terms of closed modules, such as files, classes, packages. They are automatically composed by frameworks, mixing layers, and aspects. Modularized variability is preferable from the point of view of software evolvability or maintainability, but it also imposes more overhead to implement different variants. In the annotation approach, 1-2 lines of code are enough to conditionally include a chunk of code. In the compositional approach, the whole module must be written. Therefore, the level of granularity is much coarser than with annotations. Among practitioners compositional approach is not as popular as annotations. It requires appropriate tool support to establish mappings between problem and solution spaces. Tools for software product lines are still mostly prototypes.

D. ACC Assets with Variability

Selection of a single feature from feature model affects several assets from the architectural model. We investigated SysML and AADL to see how to implement model variabilities in these languages. SysML offers calibrations but provides no facilities for introducing structural variability. Without external technologies, the language is better suited for application engineering than for domain engineering. AADL, on the other hand, makes distinction between model and its implementation. Models express domain concepts, while implementations belong to application engineering. For a single model there can be multiple implementations. Implementations can contain signal value assignments to set calibrations. It is more interesting, however, to look into structural variabilities.

We start by mapping xor groups from Fig. 9 to elements of the AADL model. For example, forward looking Sensor is either Radar or Lidar-based. The device for forward looking sensor always exists in the model:

```
device forward_looking_sensor_device
    ...
end forward_looking_sensor_device;
```

Then there are two alternative implementations of the device:

```
device implementation
    forward_looking_sensor_device.radar_based
end forward_looking_sensor_device.radar_based;

device implementation
    forward_looking_sensor_device.lidar_based
end forward_looking_sensor_device.lidar_based;
```

In this way all alternative feature groups map to different implementations in the AADL model.

AADL cannot conditionally include model elements, so the only way to model optional elements is to provide empty implementations for excluded features. In practice it is a problem, because optional functionalities, such as Alert, impact many model elements (alert_commands_data_output, alert_commands_data, alert_commands_data_inv, determine_alert, etc.) and for each element there should be an empty implementation. AADL model would have to accommodate a union of all possible variations, which makes the model less readable and probably affects its maintainability. Another problem with AADL is that the user has no clear view of all variabilities. There is no single structure that would correspond to feature models. Our conclusion is that neither SysML nor AADL provide adequate support for introducing variability into models.

E. Common Variability Language

OMG is working on a new standard for modeling variability: Common Variability Language (CVL). The purpose of CVL is to introduce variability into existing models

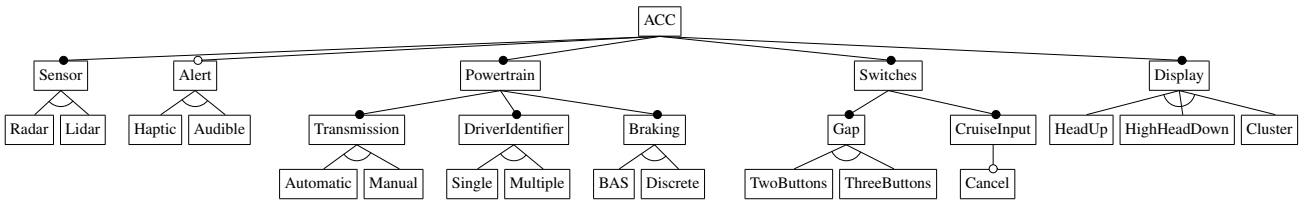


Figure 9. Feature Model of ACC Product Line

without modifying them. The language is in early stage of development. We wanted to understand what challenges implementers and users of the standard might have. Figure 10 presents the standard as proposed by Haugen [3]. It has a layered architecture with the following layers:

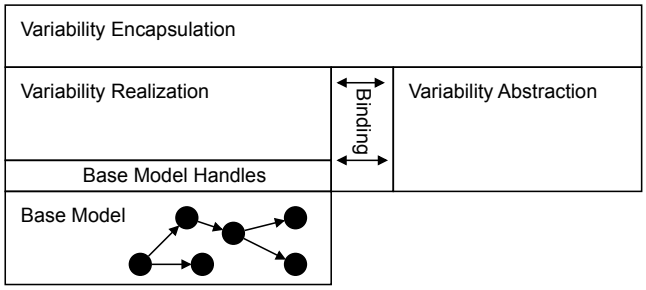


Figure 10. CVL Architecture

1) Base Model: contains the model to which CVL introduces variability. The standard assumes that the base model is a graph structure, composed of nodes and edges. CVL targets models whose meta-models comply with MOF [15].

2) Base Model Handles: reference nodes and edges from the base model. Base model handles provide an interface between CVL and external models.

3) Variability Realization: provides constructs for defining direct, concrete impact on the base model through definition of variation points. There are three types of variation points: *existence*, *substitution*, *value assignment*. Existence indicates optionality of nodes and edges. Substitution indicates that a model fragment may be substituted for another. Value assignment indicates that a value may be assigned for a particular element of base model.

4) Bindings: connect variability realization and abstraction. It is a *1-1* mapping that separates direct effect on base model from variability specification.

5) Variability Abstraction: provides constructs for specifying and resolving abstract variability. Variability specification has no direct effect on base model. Instead, it presents variability in terms of *choices* (binary decisions) and *variables*. Thus, it is very close to feature models.

6) Variability Encapsulation: builds on variability realization and abstraction layers and provides facilities for modularizing and aggregating variabilities.

Common Variability Language integrates the annotation and compositional approaches. On the one hand, it annotates base model elements with variation points. On the other hand, it separates base and variability models and stores annotations in its variability realization layer.

We found CVL relevant and well-suited to our project. First, we wanted to separate responsibilities so that members can work in parallel on base and variability models. Second, CVL naturally expresses feature models, such as the ACC feature model from Fig. 9, in the abstraction layer. Furthermore, SysML and AADL have their meta-models defined in MOF, so CVL can reference them by base model handles. Implementing variability realization was the only thing left to introduce variability to ACC architecture. It turned out to be more involving than we expected.

VI. Variable Architecture Modeling

CVL offers a very general approach for modeling variability, but it is unusable without proper tool support. To apply CVL to our AADL and SysML models we first turned to CVL Tool from SINTEF[9] which is a prototype implementation of the standard proposal. Unfortunately, we were unable to reproduce examples from the user's guide due to usability issues with the Eclipse plug-in. The documentation was unclear about realizing variability. We finally decided to implement both CVL and a domain specific language for automotive architecture in Meta-Programming System.

A. Meta-Programming System

Meta-Programming System is an open-source projectional workbench developed by JetBrains[10]. It follows the language-oriented programming paradigm [16] and is significantly different from common methods of writing source code. Projectional workbenches store all artifacts as models. Programmer operates directly on the abstract syntax tree instead of using a parser to recognize the input code. Projectional workbenches greatly simplify the development of new languages and allow arbitrarily mixing and matching different languages. For example, one could directly use SQL statements within Java programs. Projectional workbenches can show different views of one model, which improves code clarity and separation of concerns. For example, a single model could describe architecture and variabilities. Two different views would project these two aspects separately.

B. ACC Architecture in CArch

To address the problems of tool support in AADL and SysML and to overcome the limitation of SINTEF's prototype, we implemented a domain specific language called *CArch*. The design of CArch allowed us to:

[9]Available at: http://www.omgwiki.org/variability/doku.php?id=cvl_tool_from_sintef

[10]Available at: http://www.jetbrains.com/mps

- express the main concepts that we captured in SysML and AADL;
- ease integration with CVL, since CVL and CArch were both coded in a common environment – MPS. The use of MPS allows one to access the abstract syntax tree of any coded model so as to manipulate its structure, which is a requirement to implement CVL;
- create the language according to what is specific to ACC. For instance, in SysML there's no direct concept of hardware, software and system, i.e., one has to model them in terms of stereotype definition or rely on name convention. In contrast, CArch has software, hardware and system as first class elements. Additionally, CArch allows constraints definition over integers to limit their value range and resolution, something that is very recurrent in the domain of embedded systems.

CArch has the following language constructs:

- Package: as in SysML, it allows to group semantically related elements. A package can use services from other packages;
- Property: properties are similar to SysML block properties and UML class attributes. Properties in CArch define instances of types or containers. A property of an integer primitive type can be constraint in the range of values it might store, along with its resolution;
- Container: it is either a system, software or hardware. A container can only extend a container of the same kind. It contains properties, ports and connections between ports. Hardware and software can contain functionalities. System can contain hardware, software and data instances (properties), whereas hardware and software containers are allowed only data type properties. A System connects a set of hardware and software instances. A Container only exports its ports;
- Data types: extensible user defined composite types. Data types can be local, i.e., visible only inside the container in which its declaration occurs, or can be exported if declared inside a package;
- Functionality: a service that a software or hardware provides. Functionalities can contain data type properties and have input and output ports.

To illustrate part of the modeled exemplar, consider the fragment shown in Fig. 11. The system ACCPhysicalConnectionsSystem is defined inside the package ACCPhysicalConnections, which in turn uses the packages ACCModule, DriverOutput, DriverInputs and Common. These packages contain data types, software and hardware used in the definition of properties in ACCPhysicalConnectionsSystem. Next in the fragment, the system connects its properties by ports.

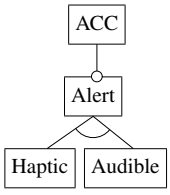
Compared to SysML and AADL, CArch allowed us to easily model the ACC system architecture because it was built according to a specific domain. CArch is still a work in progress. It currently does not support any behavior modeling or general constraints in the way SysML does.

```
package ACCPhysicalConnections
 use ACCModule
 use DriverOutput
 use DriverInputs
 use Common
 system ACCPhysicalConnectionsSystem
   accController is hardware ACCController
   chasisDevice is hardware ChasisDevice
   transmissionController is hardware TransmissionController
   hsCan is hardware HSCan
   lsCan is hardware LSCan
   onOff is hardware OnOffButton
   cancelButton is hardware CancelButton
   bodyControlModule is hardware BodyControlModule
   instrumentPanelClusterController is hardware
         InstrumentPanelClusterController
   connect onOff → OnOfStatusSignal_O to
         hsCan → Signal_I
   connect accController → ACCToBCMSignal_O
         to hsCan → ACCMToBCMSignal_I
   connect hsCan → ACCMToBCMSignal_O
         to bodyControlModule → ACCToBCMSignal_I
   connect bodyControlModule → BCMToACCSignal_O
         to hsCan → BCMToACCMSignal_I
   connect hsCan → BCMToACCMSignal_O
         to accController → BCMToACCSignal_I
   ...
```

Figure 11. Fragment of the modeled exemplar in our DSL - the CArch language.

C. ACC Variability in CVL

We modeled ACC variability by implementing essential elements from CVL meta-model and constructed MPS editors to provide concrete textual syntax. Figure 12a shows the forward collision Alert part of the ACC feature model from Fig. 9. Corresponding CVL model with mappings to the base model is depicted in Fig. 12b.



a) Feature Model

```
ACC
    1−1 Alert ?
        Haptic → HapticSeat
        Audible → Audible
```

b) CVL Model with Mappings

Figure 12. Alert Variability Models

There is a direct correspondence between feature models and our textual syntax. Feature hierarchy is established through code indentation, e.g. Alert is a subfeature of ACC. Furthermore, the question mark after feature name indicates that the feature is optional. Alert contains two subfeatures and precisely one of them must be selected (indicated by 1-1). The arrow following feature name in CVL model points to element from ACC base model (e.g. Haptic feature points to HapticSeat). The effect of selecting/excluding Haptic in feature model will be selection/exclusion of HapticSeat in the architectural model. In many cases single feature includes/excludes multiple base model elements. We handled

these cases by nesting mappings under the optional feature (not shown in Fig. 12). The full ACC variability model covers all features specified in the ACC feature model and defines mappings to elements from the architectural model. Figure 13 depicts a high-level part of it.

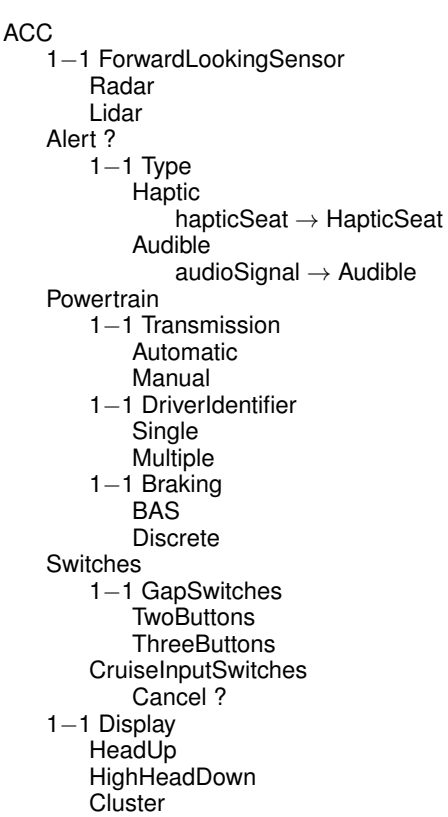


Figure 13. ACC Variability Model in CVL

We scoped CVL implementation to best fit our purpose. Common Variability Language applies to models whose meta-models conform to MOF. There is no MOF implementation for MPS, since MPS provides own elements for nodes (BaseConcept) and edges (LinkDeclaration). We used MPS primitives to reference ACC elements. We also restricted the CVL meta-model so that features (called VSpecs) explicitly form a hierarchical structure. These minor changes did not invalidate CVL applicability but simplified implementation in MPS.

Overall, our experience with CVL was positive, although initially it seemed to be another huge standard. The architectural layers are "thin" and usually involve only one element from each layer to connect abstraction layer with the base model. Probably the most complex is variability encapsulation layer, but we did not need to use it in the ACC exemplar. Separation of variability and base model allowed us to work independently on both parts. We found MPS very helpful in the integration phase of the two models. Model-centric MPS paradigm made it very easy to reference external elements. There is no doubt that CVL requires great tool support to make the standard usable in practice. We hope that future Eclipse implementations of CVL plug-in will be at least as usable as our MPS prototype.

VII. Conclusions

We extracted an exemplar of Adaptive Cruise Control from actual documentation. Our experience shows that AADL and SysML are fairly well-suited for modeling automotive systems, but a better tool support for these languages is required. None of these languages deals well with variability so we applied the draft of upcoming CVL standard for this purpose. Our prototype implementation in MPS showed that the language can be applied to existing models in a straightforward way. Creating mappings from the variability model to the base model is an additional cost that one has to pay for separating both concerns. We hope that good tool support will help practitioners in transforming their software into software product lines.

References

[1] J. D. Poole, "Model-driven architecture: Vision, standards and emerging technologies," in *ECOOP'01*.

[2] K. Czarnecki and M. Antkiewicz, "Mapping features to models: A template approach based on superimposed variants," in *GPCE'05*.

[3] O. Haugen, *CVL - Common Variability Language - a generic approach to variability for OMG standardization*, 2010.

[4] OMG, *OMG Unified Modeling Language*, 2009.

[5] J. J. H. Peter H. Feiler, David P. Gluch, *The Architecture Analysis & Design Language (AADL): An Introduction*, 2006.

[6] OMG, *OMG SysML v. 1.2*, 2010.

[7] C. Ebert and C. Jones, "Embedded software: Facts, figures, and future," *Computer*, vol. 42, pp. 42–52, April 2009.

[8] J. Holt and S. Perry, *SysML for Systems Engineering*. IET, 2007.

[9] T. Weilkiens, *Systems Engineering with SysML/UML: Modeling, Analysis, Design*. Morgan Kaufmann Publishers Inc., 2008.

[10] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," CMU, Tech. Rep. CMU/SEI-90-TR-21, 1990.

[11] S. She, R. Lotufo, A. Wasowski, and K. Czarnecki, "Variability model of the linux kernel," in *VaMoS'10*.

[12] J.-M. Favre, "The CPP paradox," in *EWSM'95*.

[13] M. D. Ernst, G. J. Badros, and D. Notkin, "An empirical analysis of C preprocessor use," *TSE*, vol. 28, no. 12, 2002.

[14] C. Kästner, "Virtual separation of concerns: Toward preprocessors 2.0," Ph.D. dissertation, University of Magdeburg, 2010.

[15] OMG, *Meta Object Facility (MOF) Core Specification*, 2006.

[16] S. Dmitriev, *Language Oriented Programming: The Next Programming Paradigm*, 2004.