# Automotive Electronic/Electric Architecture Modeling, Design Exploration and Optimization using Clafer

by

Alexandr Murashkin

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2014

## Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

**Abstract**

Modern car systems are getting more complex, so do car electronic/electric (E/E) architectures. E/E architecture of a car includes sensors, actuators, programmable ECUs and all the related communications. The complexity of E/E architectures increases dramatically: modern car have more than 100 of ECUs and communications spread over the entire vehicle. Therefore, the development of such architectures is a major challenge. Additional complexity comes from cross-cutting concerns, such as, variability and dependability.

To manage this complexity and reduce costs, architects and engineers apply model-based methods to automate analysis, perform simulation, and make key decisions before the actual implementation. To quantify the analysis results, engineers use quality attributes, such as, cost, power consumption, complexity, maintainability, and wire length. Introduction of quality attributes also allows engineers to perform architecture optimization.

In this work, we explore applicability of a modeling language called Clafer to address E/E architecture modeling, exploration and optimization problems. Clafer is a general-purpose domain-modeling language that comes with tools and solvers that are capable of performing consistency checking, design exploration and optimization. Clafer has minimalistic syntax, but its first-order logic based semantics is rich enough to represent a wide range of domains.

Our main contributions include: formulation of Clafer domain modeling principles with respect to architecture modeling, identification of Clafer design patterns in general, and demonstration of applicability of Clafer to architecture exploration and optimization. To evaluate our approach, we develop the Power Window case study. The Power Window system's E/E architecture is a rich representative of a car E/E architecture: it can be decomposed into subsystems, it can have smart or dumb sensor and actuators, and it requires wire or bus communications within and across subsystems. We consider the following topics: representing Power Window system's features and functions, automated hardware topology generation, and automated deployment of functions to hardware.

Our case case study concludes that Clafer is capable of representing all structural aspects of E/E architectures: from modeling a device to modeling an entire system. Clafer tools can facilitate automated deployment and hardware topology generation and perform architecture multi-objective optimization within a reasonable time. And finally, Clafer language features, such as arbitrary property nesting, result in clear, concise and lightweight structural models.

**Acknowledgements**

I would like to thank my supervisor, **Dr. Krzysztof Czarnecki** for accepting me to this program, giving directions, providing a lot of support, offering new opportunities and enlightening my interest in research.

I would like to express my acknowledgements to all labmates I had a chance to work closely with:

- **Ed Zulkoski** — for developing PythonSMT backend and his work Z3 translation which made possible my other research
- **Kacper Bąk** — for Clafer language semantics, Clafer Compiler and semantics discussions
- **Leonardo Passos** — for a lot of support and motivation during my studies
- **Jimmy Liang** — for developing an extremely efficient Clafer Choco-based backend, a lot of help with it and additional extensions made specifically for my research
- **Jesús Alejandro Padilla Gaeta** and **Pavel Valov** — for a great time at the internship, productive discussions and motivation
- **Jianmei Guo** — for transferring very deep knowledge and insights
- **Rafael Olaechea** — for developing ClaferMoo tool that motivated me at the beginning of my degree and influenced my entire research
- **Zinovy Diskin** — for Clafer language semantics discussions and general support
- **Zubair Akhtar** — for a great time working on architectural modeling and great summer school time

A special thanks to **Michał Antkiewicz** for collaboration, knowledge and technology transfer throughout my Master's degree completion

A special thanks to **Tom Fuhrman** and **Dr. Ramesh** from **General Motors Research & Development** for collaboration, support and domain knowledge transfer.

A special thanks to my lovely **Kristina Nazarova** for her big love, support and commitment.

A special thanks to my parents **Tatyana** and **Sergey** for their tolerance, love and support while being far away from me.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Modern car systems are very complex. Apart from the core systems like power train, transmission and braking, modern cars have many systems designed for safety, comfort, or entertainment. All these systems and subsystems with necessary communications define a car architecture.

Modern car architectures are electronic/electric (E/E) with an increasing role of software-controlled components [28], [18]. Programmable electronic control units (ECU), smart sensors and actuators and digital communication links like CAN bus or LIN bus are widely used. Yet, some components are purely electric and mechanically controlled for a variety of reasons and communicate to other components via discrete or analogue wires.

Heurung and Walz [22] state the "dramatical growth of electronic and electric systems in cars for the past decade". The overall electronic/electric architecture of a car is becoming more and more sophisticated: in modern cars, the number of ECUs is more than one hundred. Clearly, the development of such a complex architectures is a challenging task. It is time- and effort-consuming, costly in terms of money investment and has high safety requirements because of the potential to cause accidents leading to injuries or even fatalities.

To focus development efforts and reduce time-to-market, manufacturers design a universal E/E architecture to cover a superset of the functionality of various cars. However, designing such an architecture is even more challenging because of the variability concern. The design space becomes very big, meaning that there are many combinations of possible instances of such an architecture.

To manage this complexity and reduce costs and errors, architects and engineers apply model-based methods [22] to automate analysis, perform simulation, and make early

architecture decisions before the actual implementation (see related work in Section 2.4).

To quantify the analysis results, engineers use quality attributes. Most common examples include cost, power consumption, complexity, and maintainability. These quality attributes are hard to define and their definition depends on a use case. However, some specific metrics like latency or wire length are easy to define and measure. An important benefit of introducing quality attributes is an opportunity of optimization with respect to a single (i.e., cost) or multiple (i.e, dependability versus expandability) quality attributes. After optimization, the resulting set of optimal designs — or Pareto front — can be visualized and explored. The Section 2.3 elaborates on this topic.

Another aid of reducing complexity is standardization. EAST-ADL [12] is an architecture definition language (ADL) designed for the automotive industry. It covers activities from the high-level design — for instance, defining features and high-level analysis functions — down to the detailed design and implementation. We describe this standard in Section 2.1 and use it throughout our work.

To apply model-based methods, architects and engineers often use a modeling language (graphical or textual) to represent a part of a system or an architecture. After that, they are able to execute the tools on the created model. For example, a related research [24] introduces a language called AAOL to tackle software-to-hardware deployment and optimization problems.

The purpose of our work is exploring the applicability of an existing modeling language — namely, Clafer. Clafer [13] is a general-purpose structural modeling language. It is claimed to have minimalistic syntax, yet being expressive enough to represent various domains. In this work, we apply Clafer to address the E/E architecture modeling, exploration and optimization problems. The motivation behind using Clafer originates from its syntax that facilitates agile modeling and prototyping of problems and its tools that are capable of performing consistency checking and optimization.

Our main contributions include: formulation of Clafer domain modeling principles with respect to architecture modeling, identification of Clafer design patterns in general, and demonstration of applicability of Clafer in architecture exploration and optimization by solving function-to-hardware deployment and topology generation problems.

### 1.0.1 Thesis Organization

Chapter 2 reveals background information on the architecture modeling standard EAST-ADL, Clafer language and tools, multi-objective optimization, and the concept of a Pareto

front. Section 2.4 contains the related work in design exploration, optimization and analysis of E/E architectures and related areas, as well as tools and methods for design visualization and exploration.

Chapter 3 describes our approaches of modeling architectures in Clafer. Chapter 4 presents language-related patterns discovered while modeling these problems in Clafer. Chapter 5 illustrates high-level macro patterns for approaching complex problems.

Chapter 6 illustrates the Power Window case study as our evaluation part. Chapter 7 lists our conclusions and future work. Appendix A has source codes for our case study and experimental models.

# Chapter 2

# Background

## 2.1 EAST-ADL

EAST-ADL [12] [17] is an architecture description language (ADL) designed as an industry standard for automotive systems. EAST-ADL defines four levels of automotive architecture abstraction (Fig. 2.1). The top-most level — *Vehicle level* — contains a tree of vehicle features. These features may include Power Windows, Adaptive Cruise Control, Steering, Braking and others. Those, in turn, are decomposed into smaller features. For example, Power Windows can have an *Express Down* feature, meaning that the driver or a passenger can close a window completely by pressing a button on a switch once only without holding it. Similarly, modern cars support *Express Up* feature, or some cars do not have any express features at all.

The next level — *Analysis level* — defines abstractions which are just sufficient to make simulations, perform behavior analyses, and clarify interaction interfaces between the system and the plant. *Analysis functions* are behaviors with well defined input and output ports, while *functional devices* are abstractions of the actual hardware devices. For example, analysis functions for the Power Window subsystem may include *WinController*, *PinchProtection*, or *Remote Request Arbitration*. Functional devices of the same system, such as *Switch*, *Motor*, and *Current Sensor*, gather or supply signals directly from or to the environment. The analysis level avoids detailed function decomposition and implementation details, yet allows early analysis before the actual software and hardware exist.

At the *Design level*, architects and engineers decompose high-level analysis functions into low-level functions, define a hardware architecture and perform deployment of functions on hardware. This level takes implementation concerns into account: the type of

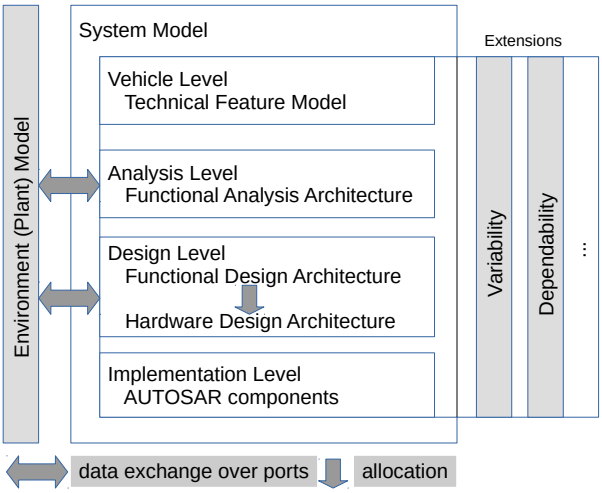



Figure 2.1: EAST-ADL Levels

hardware, interactions, performance, and fault-tolerance. Hardware architectures are defined in terms of ECUs, sensors, actuators, networking and wiring information. Functions are deployed on hardware, e.g., *WinController* can be located on a door module, a switch or the *BCM* — the body control module. *Current Sensor* can be implemented as an electronic circuit and can be allocated to a motor or a device that drives the motor (i.e., a switch or a door module). Function connectors and signals are mapped onto buses or wires. For example, the door module drives the motor by enabling power supply through a wire that is connected to the motor. All smart devices can communicate through a serial bus, such as LIN or CAN.

And finally, *Implementation level* defines system implementation using AUTOSAR components. AUTOSAR defines executable software components and their deployment onto hardware devices. It also provides standardized operating system services.

In our work, considering EAST-ADL has the following benefits. First, it defines clear terminology of the terms feature, function, system and others. Some of the terms such as feature and function are often used interchangeably in informal discussions, thus, bringing confusion and reducing clarity. EAST-ADL clearly states that a feature is a high-level, vehicle-level concept and defines what functionality is supported in a vehicle or a particular system. A function is a realization of functionality; a behavior that has inputs and/or outputs and performs some processing. Functions are decomposed into more fine-grained functions, eventually distinguishing functions designed for control or just input-output processing.

Second, EAST-ADL defines several levels of abstraction; therefore, it is easier to scope our work. We focus on the *Analysis level* and *Design level* while touching on the *Vehicle level* to define power window control features. However, we do not consider implementation/operational levels: these are outside our scope.

Third, EAST-ADL matches important use cases and concerns with the abstraction level they arise at. For example, at the design level engineers perform function deployment problem and are concerned about wiring cost. Therefore, if we propose a way to solve the deployment problem, our approach should also take wiring cost into account.

Finally, EAST-ADL does not restrict the use of concrete languages or tools. We can use any language or tool to perform the use cases suggested by a particular abstraction layer. In our work, we use Clafer language and tools.

## 2.2 Clafer Language and Tools

In this section, we introduce only basic features of Clafer language and general capabilities of Clafer tools. The details related to architectural modeling will follow in Sec. 3.

### 2.2.1 Language

Clafer (stands for *cla*ss, *fe*ature, *r*eference) is a general-purpose modeling language [13]. It is a structural modeling language designed to represent domains, meta-models, component and variability models.

Clafer unifies the concepts of classes, associations and properties, and defines a main concept — a *clafer*. Clafer specifications are built entirely from clafers, which can represent either properties, types, or references, depending on their nesting and syntactic modifiers.

Clafer is a textual language with minimalistic syntax. Figure 2.2a shows how a tree-like syntax of Clafer defines types (`Device` and `PWSubsystemHardware`) and containment hierarchy of concepts. Clafer allows for arbitrary property nesting; therefore, instantiation of devices is done in place, without defining separate types for door modules or switches.

An important feature of Clafer is variability support via cardinalities. First, Clafer has so-called clafer cardinalities to specify the possible number of instances: `Device 0..2` means there can be up to two devices in a set. Second, Clafer has group cardinality modifiers like `xor` — only one child is present, `or` – at least one child is present, and `mux` — at most one

```
abstract Device // defining a concept - abstract clafer
  smart ?  // optional concrete clafer: either smart or not

PWSubsystemHardware  // defining a concept - concrete clafer
  xor switch        // group cardinality xor -  only one switch can be present
    smartSwitch : Device      // an instance of a Device
     [smart]                  // asserting to be a smart device
    dumbSwitch : Device       // an instance of a Device
     [no smart]               // not smart

  smartComponents -> Device 0..2      // variable clafer cardinality: from 0 to 2
  dumbMotor : Device
    [no smart]
  doorModule : Device ? // optional door module
    [smart]
```

(a) Example Clafer model 1. Nesting denotes a containment hierarchy. Colons denote instantiation

```
abstract WireConnector
  src -> Device           // source is a reference to a device
  dest -> Device          // destination is a reference to a device
  [src.ref != dest.ref]   // cannot have source equals destination

  length ->> integer                        // wire length
  thickness ->> integer                     // wire thicnkess
  mass ->> integer = length * thickness     // wire mass
```

(b) Example Clafer model 2. References and integer attributes and constraints in Clafer

```
PWSubsystemHardware
  switch
    dumbSwitch
  smartComponents = doorModule
  dumbMotor
  doorModule
    smart
```

(c) Subsystem Instance

```
WireConnector$1
  src = dumbSwitch
  dest = doorModule
  length = 20
  thickness = 1
  mass = 20
```

(d) Wire Connector Instance 1

```
WireConnector$2
  src = doorModule
  dest = dumbMotor
  length = 20
  thickness = 7
  mass = 140
```

(e) Wire Connector Instance 2

Figure 2.2: Clafer: example models and instances. Shown containment, subtyping, inheritance, sets, integers and constraints

child is present. Optionality of clafers is denoted using question marks `?`. Optional clafers have clafer cardinalities `0..1`, thus they may or may not be present in model instances.

Clafer supports constraints written in first-order predicate logic. It has both existential and universal quantifiers, sets, and relations. Constraints can contain operations on sets and also on integers as on Fig. 2.2b, since Clafer supports integer types. Clafer has no support of real numbers yet, so techniques like scaling down and rounding to a nearest integer need to be used.

A specification written in Clafer language is called a *Clafer model*. An instance of a Clafer model is its full completion with all clafers fully instantiated and no variability left. Figure 2.2c shows a complete instance of `PWSubsystemHardware`: there is only one switch device and the door module is present. Figure 2.2d and Figure 2.2e illustrate two wire instances with all the values fully specified and computed.

Clafer has an implicit support for partial instances [15], i.e, a user can partially restrict the model with variability, thus still allowing some undecided variability. During the instance generation process, configurations become completed. Alternatively, a user can give a complete specification of a system with no variability left, meaning there will be only one possible instance. All these use cases are covered in details in Section 3.

### 2.2.2 Tools

Clafer comes with a set of tools [10] called *Clafer tools*. First, the tools include a compiler — Clafer Compiler. It translates Clafer textual models into various formats: a graphical and HTML representation, an XML-based intermediate representation, and reasoner-specific formats. Next, the tools include backends for automated reasoning use cases, such as, constraint checking, instance generation and completion, and multi-objective optimization. In this work, we use mainly the Choco-based backend [2] as it it the fastest one and works well with integers of a low scale. Clafer also has Alloy-based backend [3] and ClaferSMT backend [4].

Clafer tools reflect the combinatorial nature of Clafer. Given a specification (a model) written in Clafer language, Clafer backends generate all possible instances that satisfy the conditions, or prove that there are no instances in the given scope. In other words, a user can keep generating all the instances until the entire problem space is considered within the given scope. This nice feature of Clafer can be used for design generation. In this case, Clafer model can be treated as a configuration of the design, and the generated instances are concrete designs.

### 2.2.3 Prior Use

The semantics of Clafer is clearly defined in the thesis [13] and the papers focused on metamodeling [14] and partial instances [15]. Researchers demonstrated the use of Clafer for modeling and opitimization of product lines [27], [10], [31]. Moreover, Clafer models were used in example-driven modeling as both abstractions and examples [11]. Besides these, there was no prior work demonstrating the use of Clafer in various types of domains. In our work, we claim that the expressive power of Clafer is not restricted to representing feature models and simple examples, and researchers can facilitate the use of Clafer in industry such as automotive.

Moreover, from the user perspective, there is no systematic description of Clafer modeling process. There are stand-alone Clafer models for various academic purposes, however, there is no general description of modeling techniques, patterns and use cases of Clafer. In our work, we define a modeling methodology for architectural modeling, as well as discover patterns that may be reused across domains. This is another important contribution of our work.

## 2.3 Optimization

In our work, we consider optimization of electronic-electric architectures; therefore, in this section we introduce the notions of multi-objective optimization and a Pareto front. We demonstrate these concepts on a bus topology example, to make it closer to the domain of the paper.

### 2.3.1 Bus Topology Example

We consider four bus topologies to illustrate a multi-objective optimization problem. Figure 2.4 represents all four of them. *Flat bus* (Fig. 2.4a) — there is only one bus to which each component in a system is connected. Expandability is very straightforward: only one protocol is used, and adding another element to the bus is not a problem. Reliability is problematic, the same bus is often used by other systems. A failure of a component on a bus can cause the entire bus to become unavailable.

*Grid* defines dedicated buses and devices for each system or subsystem, and devices communicate to the required devices directly. Grid topology is considered to be the least expandable, since each subnetwork is tailored to support specific communication. However,

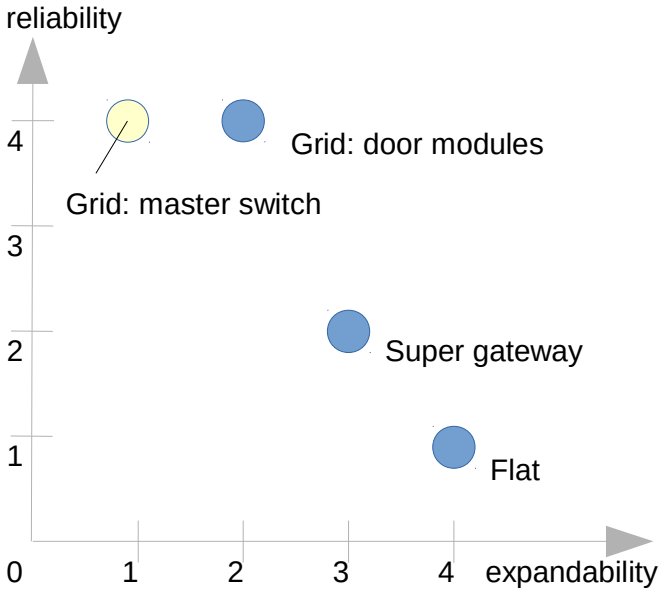
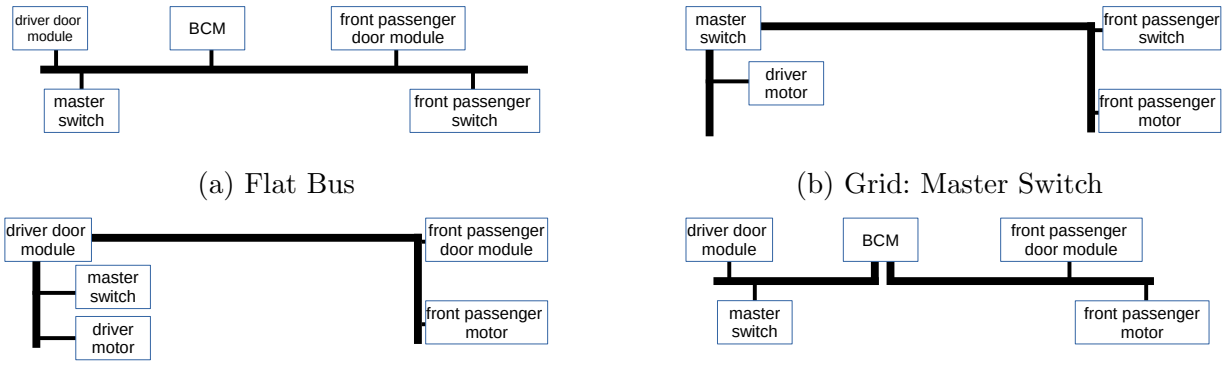



Figure 2.3: Bus topologies in terms of reliability and scalability. Blue circles — optimal instances — form a Pareto front, light-yellow circle — a sub-optimal instance and not in the Pareto front

reliability in general is good, since failure of a particular communication link does not brake communication link in other subsystems. We distinguish two types of Grid topologies for the Power Window subsystem: *Grid based on Master Switch* (Fig. 2.4b) and *Grid based on Door Module* (Fig. 2.4c). BCM — *Body Control Module*, the main electronic control unit that controls several systems in a car — may or may be not connected to any of the buses. The communication topology is quite self-contained, and BCM does not play a significant role.

And finally, we consider a *Super Gateway* topology (Fig. 2.4d). Two buses, one for the driver window, one for the front passenger window are connected together by the BCM. BCM is a gateway that can connect two buses of different communication protocols. Reliability is moderate, since the failure of one bus does not disable the other one. However, since there is a single gateway, its failure is likely have a significant negative impact.

For each of the four bus topologies, we can define two quality attributes: reliability and expandability. We assign values to be relative to each other: the highest value of expandability belongs to the most expandable alternative, while the least number — to the least expandable (Tab. 2.1). Once we specified the values, we can run optimization.




(a) Flat Bus

(b) Grid: Master Switch

(c) Grid: Driver Door Module

(d) Super Gateway

Figure 2.4: Bus topologies in terms of network structure. Buses are shown using a thick black solid line. BCM stands for Body Control Module

| | Flat | Grid: Master Switch | Grid: Door Module | Super Gateway |
|---|---|---|---|---|
| expandability | 4 | 1 | 2 | 3 |
| reliability | 1 | 4 | 4 | 2 |

Table 2.1: Bus topology quality values for expandability and reliability

### 2.3.2 Single-Objective Optimization

First, a safety engineer needs to optimize the system in terms of *reliability*. Therefore, he is interested in bus topologies that provide the highest possible reliability to the system. A bus topology is considered to be *optimal* with respect to a maximization objective if it has the highest possible value of the given objective. In our case, the most optimal bus configuration set — as perceived by the safety engineer — is the set of two topologies: *Grid: Master Switch* and *Grid: Door Module*. The problem faced by the safety engineer is a *Single-objective optimization* problem, since he or she considers only one objective and one quality attribute.

### 2.3.3 Multi-Objective Optimization

A system architect, in contrast to safety engineer, is interested in maximizing *expandability* of the system. However, it is not possible to improve expandability without sacrificing reliability: the more generic the bus is, the easier it is to add new devices, but the less

reliable and more error-prone the communication becomes. In this case, three bus configurations are the compromised options: *Super Gateway*, *Grid: door modules* and *Flat* are non-dominated with respect to the two quality objectives (Fig. 2.3). However, *Grid: master switch* is dominated by *Grid: door modules* because the latter one has better expandability. Therefore, this option is called *sub-optimal* as there is a better option with a better quality.

The problem faced by the system architect and the safety engineer represented above is a *Multi-objective optimization* problem. Two given objectives — to maximize expandability and to maximize reliability are the part of the problem formulation. The set of all optimal instances is called a *Pareto front* and consists of the three topologies described above. *Grid: master switch* is not in the Pareto front since it is sub-optional. Figure 2.3 illustrates the Pareto front for the bus multi-objective optimization example.

## 2.4 Related Work

### 2.4.1 Analysis and Optimization of E/E architectures

There are papers that focus on various stages of architecture design, as well as various specific concerns like cost, power consumption and safety.

Kugele and Pucea [24] introduce a domain-specific constraint and optimization language called AAOL (Automotive Architecture Optimization Language). The authors model a deployment problem and perform multi-objective optimization on a similar, but much smaller power window case study: two complete designs of the four-door power window system. Their AAOL language framework is designed in a modular way, so it can be concertized to accommodate specific solvers or tackle specific problems. In contrast, our work is focused on a general-purpose modeling environment.

Walla et. al. [34] demonstrates exploration of E/E architectures in terms of power consumption and allows a user to explore concrete designs and evaluate architecture specifications of automotive systems. It is possible to explore each ECU of an architecture variant individually, to view the distribution of power consumptions across all ECUs, as well as to catch its dynamics with time. The design exploration aspect is very related to our work, however, the use case of comparing various architecture variants to each other is not described in the paper. Also, we focus on the structural aspect only, as Clafer does not support simulation or behavioral analysis.

Florentz and Huhn [20] propose an architecture evaluation method and demonstrate it on a case study in an automotive industry - a power window system. Their case study

is less detailed and more high-level compared to ours, and their evaluation is conducted over two alternatives only: federated vs. centralized design, and two quality metric groups: cost and performance. Our approach is automated: quality computation and optimization is done by Clafer backends. Also, we deal with a more complex design space and consider other quality attributes (wiring length, wiring mass, and the number of smart devices).

Brandt et.al. [25] explore optimization of automotive E/E architecture in terms of cost. The authors present complex formulas of cost computation that distinguishes development and maintenance cost and also depends on the hardware and software components mapped to an ECU. Their work has a strong focus on the cost metric, in contrast to our approach that is more general and takes more metrics into account. Also, their paper does not describe exploration of alternative solutions, in comparison to our work.

Moritz at al [26] demonstrates the use of evolutionary algorithms to optimize E/E architectures in terms of cost and the defined metric called complexity (the average number of functions on ECUs). It does not consider design space exploration explicitly, so the focus of the paper is solely on optimization. In our work, we do consider design space exploration use cases. However, their complexity metric may be useful in our case study, since we currently do not restrict the number of components on ECUs.

Rupanov et. al. [30] consider an important activity of architectural design - safety analysis. The paper emphasizes the importance of architecture evaluation at early design stages and presents a methodology for building and evaluating architectures with respect to fault-tolerance and other quality metrics. Our case study can be extended with the safety aspect in the future.

### 2.4.2 Design Space and Pareto Front Visualization and Exploration

In our work, we use ClaferMooVisualizer tool [27] to represent design spaces and Pareto fronts. This tool is a web-based front-end that integrates all Clafer tools. It comes with four general-purpose visualizations. First, **Bubble Front Graph** — a multi-dimensional bubble chart that can visualize up to four dimensions simultaneously. Second, a tabular visualization called **Feature and Quality Matrix** that lists every property value of every optimal instance. Third, **Variant Comparer** that compares a chosen subset of optimal instances. And finally, there is a new powerful visualization called **Parallel Coordinates Chart** that represents each instance as a polyline connecting parallel objective axes in proper places. Unfortunately, the tools currently does not support any problem-specific visualizations: generating circuit diagrams may be very useful for our work.

Visualization of Pareto fronts is a well-known topic in research. There are various straight-forward methods, such as multi-dimensional scatter plots [32] and radar charts (can be found in Google Chart visualization libraries [7]), and also advanced techniques, such as Level Diagrams [16] and heatmaps [29]. These visualization techniques may be applicable in our context. However, we take advantage of Clafer Tools and built-in visualization methods, therefore, we do not use other tools or approaches.

There are several visualization suites and libraries. Examples include RAVE tool [9] that works with MatLab and has powerful visualization charts to be used in exploration. Google Charts [7] and Dojo libraries [5] are useful for web-based visualizations. META [8] is a very advanced tool suite that allows design exploration, optimization and simulation of cyber-physical system models, including architecture models. It is integrated with a tool suite by GA Tech ASDL [6], which allows design exploration by filtering by quality metrics, design ranking, visualization of metric values distribution. This tool set is likely to be applicable to E/E architectures as well.

# Chapter 3

# Principles of Creating Architecture Models in Clafer

Architecture models in Clafer are characterized by the following properties.

1. Clafers with cardinalities greater than one. This is the main distinction of architecture models from feature models, where cardinalities are typically at most one (`0..1`). However, architecture models can still have feature models for describing its configuration.

2. Many levels of containment and instantiation to accommodate system decomposition. Clearly, architectures are often about systems that are composed of subsystems or high-level analysis functions that are composed of lower-level functions.

3. Many *reference* clafers. References are required when modeling a wire source and destination or when deploying software to hardware.

4. Rich variability across the model. Variability appears in all aspects and in all levels: whether to have BCM or door modules or not, what types of switches to have, etc. Also, variability comes from alternative deployment rules and various communication topologies.

Clearly, modeling systems themselves is more challenging than just modeling their configurations. To model architectures, it is useful to have a methodology and guidelines to follow. Unfortunately, for Clafer, there are no such guidelines. So, in this work, we propose and describe such a methodology.

The following sections present various aspects of architecture models: from meta-modeling and domain modeling to multi-objective optimization. Our methodology description is supported by examples originating from case study domains or small examples from our past modeling experience, such as, service-to-machine deployment optimization.

## 3.1 Domain Modeling

To model a domain in Clafer, we start with a metamodel. A metamodel with the concepts Device, ECU, Function and various connectors is essential, and the way it is created will be reflected on the future modeling process.

### Basic Type Declaration

In Clafer, we use the `abstract` keyword to define types Figure 3.1. We distinguish dumb devices — purely electric with no software or control logic on it, and smart devices — that can have software on them and communicate via buses.

Boolean properties like smartness of a device can be specified using an optional clafer. Alternatively, we could define a `smartness` clafer with `xor` group cardinality to list the choices more explicitly. And finally, the third alternative is to define a reference clafer `smartness` and type it with the enumeration called `Smartness`.

Clafers defined with the `abstract` keyword are called *abstract* clafers, such as, `Device` type. Clafers defined without this keyword are *concrete* clafers: `smart`, `dumb` and `smartness` are concrete clafers. Typically, concrete clafers are instances of abstract clafers, or a universal base type `clafer` if the type is not specified.

```
abstract Device
  smart ?
```

(a) Alternative 1: optional clafer

```
abstract Device
  xor smartness
    dumb
    smart
```

(b) Alternative 2: clafers in a xor group

```
enum Smartness = smart | dumb

abstract Device
  smartness -> Smartness
```

(c) Alternative 3: use of an enumeration

Figure 3.1: Device type declaration in Clafer: dumb or smart devices

In terms of semantics, all the three ways are equivalent: a device can be either smart or dumb. However, there is a *reasoning performance* perspective, which denotes how fast and

efficient the compiled code will be when it comes to generating complete model instances and optimization. From this viewpoint, the first option is preferable, because it defines only one clafer for smartness, called `smart`. The second alternative defines three clafers: `smartness`, `smart` and `dumb`. The third alternative defines four: `smartness`, `Smartness`, `smart` and `dumb`, and also a reference that adds an overhead.

Now, we add more device types — for example, we may have *electronic* devices — the ones that have electronic circuits in them. Smart devices are electronic as well, since they do have such circuits. The devices that are not smart, yet have some electronics to support control logic are *electronic-only*.

```
abstract Device
  smart ?
  electronic ?
  [electronic => smart]
```

(a) Alternative 1: optional clafers with a constraint

```
abstract Device
  xor smartness
    dumb
    smart
    electronicOnly
```

(b) Alternative 2: clafers in a xor group

```
abstract Device
  smartness -> Smartness

enum Smartness = smart
   | dumb
   | electronicOnly
```

(c) Alternative 3: use of an enumeration

Figure 3.2: Device type declaration in Clafer: smart, dumb and electronic only devices

Figure 3.2 shows the updated alternatives to accommodate our changes. All three options appear to be well-extensible. However, the first option becomes less intuitive, and also requires constraints to allow the device to be either smart and electronic, or electronic, or none (dumb). The implication says that if a device is smart, then it is considered to be electronic in our domain. The best balance between performance and extensibility belongs to the second option: we can easily add device types if needed, but there is no additional overhead because of the references as in the third option.

### Inheritance with specialization

Now, we define subtypes of `Device`. Subtypes can be defined using inheritance syntax in Clafer: `Descendant : Ancestor`. For example, we define a subtypes of Device: ECUs, sensors or actuators. Figure 3.3 shows definition of a type `ECU` in alternative ways. First, subtypes are needed to for *specialization* — restricting the original type. For example, ECUs can only be smart, therefore, when inheriting from a Device, we force all the instances of the type ECU to be smart.

```
abstract ECU : Device
  [smart]
```

(a) For the Alternative 1

```
abstract ECU : Device
  [smartness.smart]

  // or just [smart]
```

(b) For the Alternative 2

```
abstract ECU : Device
  [smartness = smart]
```

(c) For the Alternative 3

Figure 3.3: Inheritance with specialization of Device in Clafer: ECU is a smart device

Note that for each of the alternatives, specialization is done in a different way. For the first alternative, we create a constraint and assert that the `smart` clafer have to be present. For the second alternative, since `smart` is nested under `smartness`, the most precise way of asserting smartness is writing `smartness.smart`. Clafer Compiler's name resolver, however, allows for specifying clafers by names without its full path. As soon as the clafer `smart` is unique in the namespace, we can simply write `[smart]`, and the resolver will automatically determine the clafer.

For the third alternative, since `smartness` is property of a type `Smartness`, we need to equate it to the proper enumeration value by saying `smartness = smart`. There is no short-hand for writing this statement.

### Inheritance with extension

Via inheritance, we can also extend the base clafers by adding new clafers. For example, we may want to add connectivity features: smart devices can be connected via LIN or CAN interfaces. Some devices can support both protocols at the same time, if they act like a gateway between various networks. Figure 3.4 shows alternative definitions of connectivity features.

Now our left-most alternative becomes bulky: we define two optional clafers named `connectivityLIN` and `connectivityCAN`. To make at least one of these present, we add an or-constraint (`||`). The second alternative, in contrast, is clearly organized. There is a clafer called `connectivity`, which acts like a group. The elements in the group — clafers `LIN` and `CAN` can be present either together or individually. The third alternative is to define an enumeration for the connectivity types: LIN, CAN or any.

From the performance reasoning point of view, the three alternatives are related similarly as in previous cases. The left-most alternative includes two clafers and one constraint.

```
abstract ECU : Device
  [smart]
  connectivityLIN ?
  connectivityCAN ?

  [connectivityLIN
    || connectivityCAN ]
```

(a) For the Alternative 1

```
abstract ECU : Device
  [smart]
  or connectivity
    LIN
    CAN
```

(b) For the Alternative 2

```
abstract ECU : Device
  [smart]
  connectivity -> Connectivity

enum Connectivity = LIN
  | CAN
  | any
```

(c) For the Alternative 3

Figure 3.4: Inheritance with extension of Device in Clafer. ECU has various types of connectivity, which are complementary, not exclusive

The middle alternative defines 3 clafers and also an implicit constraint for `or`. And the right-most alternative is implemented using 4 clafers and a reference. Set references (`->`) have an implicit uniqueness constraint as well.

### Containment hierarchy, instantiation and references

Now we model a composite system that contains or refers to various devices. To be more precise, we demonstrate modeling the hardware part of a Power Window subsystem. Figure 3.5a represents `PWSubsystemHW` clafer. It does not inherit any type, however, it includes instances of other types. Containment is modeled using just indentation: a tree-like syntax of Clafer is handy when modeling *part* relationships. Device instances called `switch` and `smartMotor` and `doorModule` are parts of the `PWSubsystemHW`. We also have a reference clafer called `bcm` to a body control module. In a way, the reference is also a part of the system, therefore, resides together with other devices.

Instances are defined using a colon notation (`:`) as well. Syntactically, the difference between instantiation and pure subtyping is the absence of the `abstract` keyword[1]. For consistency and clarity, we use the term *to define an instance* or *instantiate* when we use the colon notation without the `abstract` keyword. We use the term *subtyping* when we derive a type using the `abstract` keyword.

In our model, we define two instances of `Device`: a `switch` and a `smartMotor`. We can still use specialization and extension when instantiating: for example, we assert that the

[1]In Clafer semantics [13], however, subtyping by creating concrete clafers and instantiation are equivalent. This point is outside the paper scope, since we focus on the user perspective only

motor is not smart. Instances can also be optional. For instance, we may or may not have a door module in the hardware: therefore, we put a question mark (`?`) following `doorModule : ECU`.

```
abstract PWSubsystemHW
   switch : Device
   smartMotor : Device
    [smart]
   doorModule : ECU ?
   bcm -> ECU ?
```

(a) Containment hierarchy. Nesting denotes a containment hierarchy: switch, smartMotor, and doorModule are parts of a PWSubsystemHardware. The reference bcm is also a part of the containment hierarchy, but the ECU it points to is not a part of the subsystem. Colons denote instantiation: switch is an instance of Device. Question marks denote optional components. Arrows denote references.

```
abstract FPPWSubsystemHW : PWSubsystemHW
  masterSwitch -> Device
  switchRequestJunction : Junction ?

  [switch.smart => no
      switchRequestJunction]
```

(b) Inheritance (with extension) of a subsystem. Front passenger subsystem contains all the components from the PWSubsystemHW. It also adds a reference to the driver switch called masterSwitch, and switchRequestJunction to join requests from the driver and passenger doors by splicing wires. If the switch is smart, then should be no junction.

Figure 3.5: Containment hierarchy and inheritance of subsystems in Clafer

References are used to refer to a clafer that is not physically instantiated in place. References act as pointers in object-oriented languages: a reference clafer is actually a pointer to a clafer instantiated elsewhere. References are useful when representing components that are not the part of the containment hierarchy. For example, a body control module (BCM) is common for all subsystems and resides outside power window systems. However, power window subsystems often use BCM for various purposes: to deploy certain functions, to transfer signals from a driver window to passenger windows, etc.. In Clafer, references defined using single `->` and double `->>` arrows having different semantics[2]. We use single arrows `->` in most cases: modeling a reference to a device, function or any other component. We describe the use of references in Sec. 4.1.

[2]Single arrow denotes references that must point to different clafer instances (set semantics); double arrow does not have this restriction (bag semantics) [13]

### Configuration of nested instances

Next, we go one level up to the system level and combine instances of subsystems together into one complete hardware system. Figure 3.6 shows a definition of an abstract clafer that represents a system type: `PowerWindowSystemHardware`. The system is supposed to include and configure its subsystems. Therefore, we define several instances of subsystems: one is the basic one — for the driver, and three extended ones — for the three passengers. Moreover, we instantiate BCM at the system level, since it is common for all the subsystems.

Configuration is done via constraints. First, we assign the `bcm` references of each subsystem to point to the BCM instance we define at the system level. This is done via the constraint `[bcm.ref = BCM]`. Note that BCM is an optional clafer, therefore, it may not be instantiated at all. According to the equality constraints, if `BCM` is present, each subsystem will have a reference to the `BCM`; if there is no `BCM`, each of the subsystem will have no `bcm` reference.

Next, for each passenger window, we assign its reference to the master switch. The master switch is always the driver's switch, so we assign the reference `masterSwitch` of each subsystem to `driverSubsystemHardware.switch`. Now we have a complete subsystem that represents hardware components of the entire system of four windows.

```
abstract PowerWindowSystemHardware
  driverSubsystemHardware : PWSubsystemHW
    [bcm = BCM]
  frontPassengerSubsystemHardware : PPWSubsystemHW
    [bcm = BCM]
    [mainSwitch = driverSubsystemHardware.switch]
  rearLeftPassengerSubsystemHardware : PPWSubsystemHW
    [bcm = BCM]
    [mainSwitch = driverSubsystemHardware.switch]
  rearRightPassengerSubsystemHardware : PPWSubsystemHW
    [bcm = BCM]
    [mainSwitch = driverSubsystemHardware.switch]

  BCM : ECU ?
```

Figure 3.6: Combining subsystems into systems in Clafer

### Sets, set operations and quantifiers

Even though containment representation is handy, Clafer does not currently have a straightforward way of enumerating child clafers. To deal with this limitation, we use sets and store all the clafers we need explicitly using references.

First, we define a set called `localComponents`. This is a set of references to `Device` instances. Clafer cardinality `2..4` now determines the set cardinality: it is variable and contains from two to four instances of `Device`, depending on whether we have a door module, the BCM, both or none. BCM is considered local for each subsystem, since each subsystem can allocate software functions to BCM. The constraint `C1` in Figure 3.7 defines the elements of the set.

Next, we are interested in a set of smart components as well. We define another set with cardinality `0..4`. The lower bound is 0, since we may have no smart components at all, if all the switches and motors are dumb, and we do not have a BCM nor door module. The upper bound is 4, when all the local components are smart.

To populate the set of smart components, we create a query that may be stated as "from a set of local components, give me only those that are smart". The next two constraints `C2` and `C3` with quantifiers `all` and `no` represent such a query. Queries with quantifiers are extremely expensive in terms of reasoning performance, and in the following chapter's Section 4.5 we show a method to avoid quantifiers yet achieving the query semantics. However, for the demonstration purposes, we show a query with quantifiers. This query extracts only the smart components from the set of local components.

The constraint `C2` with the universal quantifier `all` states that each device `d` in a `localComponents` set will be included in the `localSmartComponents` set if and only if the device is smart. Clearly, the entire set of the local components is considered, and only smart devices can go to the "smart" set. The Clafer quantifier `all` is equivalent to the quantifier *for all*, or $\forall$.

The next constraint, `C3`, uses another universal quantifier `no` and states that it is impossible to have an element in the `localSmartComponents` set such that it is not in the set of the local components. This constraint is necessary, because the previous constraint (`C2`) does not make any restrictions on the devices that are not in the local component set, i.e, components of another window. Therefore, Clafer reasoners may put any smart component of another window into the set. The constraint `C3` denies such a case.

Next, we demonstrate an existential quantifier `some` using the constraint `C4`. The constraint says that `hasSmartComponent` clafer is equivalent to having at least one element in the `localSmartComponent` set. Thus, `some` is equivalent to *exists*, or $\exists$.

Another way of defining a `hasSmartComponent` flag is using set cardinality syntax: a hash sign `#` (constraint `C5`). By saying `#localSmartComponent`, we get the actual cardinality of the `localSmartComponent` set, which is fully decided with respect to the elements it has: it is already decided whether the `doorModule` and `bcm` are in the set. Obviously, `#smartComponents` is equivalent to $|smartComponents|$ in mathematics.

```
abstract PWSubsystemHW
  ...
  localComponents -> Device 2..4
  [localComponents = switch, motor, doorModule, bcm.ref]                // C1

  localSmartComponents -> Device 0..4
  [all d : localComponents | (d in localSmartComponents) <=> d.smart] // C2
  [no d : localSmartComponents | !(d in localComponents)]               // C3

  hasSmartComponent ?
  [hasSmartComponent <=> (some d : localSmartComponents)]               // C4

  hasSmartComponentAlt ?
  [hasSmartComponentAlt <=> (#localSmartComponents > 0)]                // C5

abstract PWSystemHardware
  ...
  allSmartComponents -> Device 0..13
  [allSmartComponents.ref =                                             // C6
      driverSubsystemHardware.localSmartComponents.ref ++
      frontPassengerHardware.localSmartComponents.ref ++
      leftRearPassengerHardware.localSmartComponents.ref ++
      rightRearPassengerHardware.localSmartComponents.ref ]
```

Figure 3.7: Sets and set operations in Clafer

And finally, at the system level, we demonstrate a set union operation ++ using the constraint `C6`. We create a set denoted as `allSmartComponents` and constrain it to be the union of the four sets of smart components for each of the foor windows. Note that we apply dereferencing a pointer operation (Sec. 4.1), so that the union operation is applied to the sets themselves, not their references. Also note that for `allSmartComponents` it is sufficient to have the upper cardinality of 13, because `bcm.ref` points to the same device;

therefore, this device will not be included twice in the same set.

### Modeling connectors

To be able to connect devices together, we need an abstraction of a connector. Figure 3.8a shows two definitions of connectors.

```
abstract WireConnector
  src -> Device
  dest -> Device
  [src.ref != dest.ref]
```

(a) Definition of a wire connector. Has references to source and destination devices

```
abstract WireMultiConnector
  connectedDevices -> Device 2..*
```

(b) Definition of a wire multi connector. Has references to source and destination devices

```
abstract WireConnector
  ...
  length ->> integer
  thickness ->> integer
  mass ->> integer
  [mass = length * thickness]

abstract DiscreteWireConnector : WireConnector
  [thickness = 1]

abstract PowerWireConnector : WireConnector
  [thickness = 7]
```

(c) Wire connector with integer properties and integer constraints. Mass is a function of a connector's length and thickness. Two subtypes of WireConnector with different thicknesses

```
abstract PWSubsystemHW
  ...
  comWire : PowerWireConnector
    [src.ref = switch]
    [dest.ref = motor]
    [no src.smart]

  linBus : WireMultiConnector ?
    [connectedDevices.ref =
       smartComonents.ref]
```

(d) Example of connectors: a binary connector that connects a switch to a motor and a LIN bus that connects all smart components

Figure 3.8: Modeling connectors in Clafer: single and multi-wire connector. Integer properties and example instances

The first one — `WireConnector` — is a binary connector that connect two devices only. It has references to the source `src` and the destination `dest` the wire connector joins. Connectors abstract away wires: each connector may be implemented using few wires (i.e., ground and power). We also specify a connector constraint that does not allow a device to

be connected to itself: it is achieved by the inequality of the dereferenced objects `src.ref` and `dest.ref`.

The second one — `WireMultiConnector` — is designed to connect several devices at the same time. There is a set of connected devices with the cardinality from two to star: `2..*`. In this case, we do not need a constraint that denies self-loops: clafer sets defined using a single arrow `->` automatically deny any repetitions[3]. Also, the multi-connector is not directed, because Clafer sets are not ordered.

### Quality attributes and integer types

We augment our wire connector definition with quality attributes, such as, length, thickness and mass (Fig. 3.8d). Integer quality attributes are defined using the bag reference notation (`->>`) [13]. Since Clafer does not support real numbers yet and floating point, we have to apply at least one of the following approaches:

1. Round real numbers to the nearest integer: 0.05 to 0, 2.5 to 3
2. Use absolute scaling: 0.05 to 5 and 2.5 to 250
3. Use relative scaling: take 0.05 as a unit, and since 2.5 is 50 times 0.05, it becomes 50

In Clafer, integer attributes do not have supplementary units. Therefore, we follow the same conventions throughout the model. We measure length in centimeters ($cm$) without any scaling. Thickness is relative: the base thickness is 1 and equals the cross-section area (in $mm^2$) of a discrete wire. The mass is a derived metric, and we do not care about density assuming its the same for all type of wires. During the instantiation process, the mass gets computed from the length and the thickness.

If any of the values is not specified, for example, we forgot to specify thickness, then backends will pick an arbitrary number and substitute it. Obviously, we avoid such a case and specify all the values.

## 3.2 Optimization in Clafer

Clafer has a support for both single-objective and multi-objective optimization. Optimization is done with respect to numeric quality attributes. The optimization workflow in Clafer

[3] As opposed to *bags* `->>` which allow repetitions

looks as follows. First, we model a problem domain and specify individual contributions to quality. Next, we define optimization objectives. And finally, we visualize the resulting set of optimal instances and make conclusions on it.

### 3.2.1 Modeling Problem Domain

For demonstration purpose, we model a bus topology example from a Sec. 2.3. This subsection describes how to represent this example in Clafer.

Figure 3.9a shows the formulation of the bus example in Clafer. The root abstract clafer `PWBusTopology` contains two integer attributes: `expandability` and `reliability`. All the bus topologies are connected using `xor` — meaning only one bus topology can be chosen. `Grid` topology types are also combined into a `xor` group, to force only of the types to be chosen.

Each choice imposes constraints on `expandability` and `reliability` quality attributes. We use the numbers from the Tab. 2.1 and create the corresponding constraints on the integer attributes. For instance, for the `flat` topology, `expandability` will be set to 4, and the `reliability` is set to 1. We follow a similar approach for all the four topology options.

### 3.2.2 Stating a Single-Objective Optimization Problem

To be able to optimize, we need to define an instance to optimize and specify optimization objectives. We do that on the last two lines of Figure 3.9a: we define a concrete clafer `mostReliableTopology`. And at the next line, we state an objective to maximize the reliability of the topology: `<< max mostReliableTopology.reliability >>`. The problem is fully specified now.

Now we run a Clafer optimization backend, and it generates the two instances illustrated on Figure 3.9b. These two instances match the ones from Sec. 2.3: only grid topologies are chosen, since they have the highest possible value of reliability.

### 3.2.3 Stating a Multi-Objective Optimization Problem

Specifying multi-objective optimization problem is easy: we just add more optimization objectives. So, we also need to state that we need to maximize expandability of the bus

```
abstract PWBusTopology
  expandability ->> integer
  reliability ->> integer
  xor type
    flat
      [expandability = 4]
      [reliability = 1]
    xor grid
      usingMasterSwitch
        [expandability = 1]
        [reliability = 4]
      usingDoorModule
        [expandability = 2]
        [reliability = 4]
    superGateway
      [expandability = 3]
      [reliability = 2]

mostReliableTopology : PWBusTopology
<<max mostReliableTopology.reliability >>
```

(a) Bus Topology model and optimization objectives

```
=== Instance 1 Begin ===

mostReliableTopology$1
  type
    grid
      usingDoorModule
  expandability = 2
  reliability = 4

--- Instance 1 End ---
=== Instance 2 Begin ===

mostReliableTopology$1
  type
    grid
      usingMasterSwitch
  expandability = 1
  reliability = 4

--- Instance 2 End ---
```

(b) Set of optimal instances

Figure 3.9: Single-objective optimization in Clafer

topology Figure 3.10a. We define an instance called `optimalBusTopology` and two optimization objectives: to maximize expandability and to maximize reliability.

Figure 3.10b, Figure 3.10c, and Figure 3.10d represent all the optimal instances produced by Clafer backends — three fully complete instances in Clafer textual representation. They match perfectly our three instances described in the introductory Sec. 2.3.

### 3.2.4 Visualization and Exploration of Optimal Variants

It is hard to make conclusions on Pareto fronts without a proper visualization. Therefore, Clafer comes with a visualization tool called ClaferMooVisualizer [27]. Figure 3.11 represents the tool with our multi-objective optimization example.

The tool uses a notion of a *variant*, which has the same meaning as an *optimal variant*

```
mostOptimalTopology : PWBusTopology
<<max mostOptimalTopology.reliability >>
<<max mostOptimalTopology.expandability >>
```

(a) Optimization Objectives

```
mostOptimalTopology$1
  type
    flat
  expandability = 4
  reliability = 1
```

(b) Optimal instance 1

```
mostOptimalTopology$2
  type
    superGateway
  expandability = 3
  reliability = 2
```

(c) Optimal instance 2

```
mostOptimalTopology$3
  type
    grid
      usingDoorModule
  expandability = 2
  reliability = 4
```

(d) Optimal instance 3

Figure 3.10: Multi-objective optimization in Clafer: the objectives and the Pareto front

or an *optimal instance*. There are four interacting visualizations in total: Bubble Front Graph, Feature and Quality Matrix, Variant Comparer and Parallel Coordinates Chart.

### Visualization

**Bubble Front Graph** (Figure 3.11, top left) is a bubble chart. Each bubble represents an optimal instance, or a variant. Bubbles are numbered, therefore, it is possible to refer to instances by a number. Bubbles are selectable: if we click on a variant, it becomes highlighted on all the rest visualizations. Bubbles are positioned with respect to two axes: in our example, the two axes represent values of expandability — the horizontal axis, and reliability — the vertical axis. The chart supports up to four dimensions: the third one is bubble color, and the fouth one is bubble size. Dimensions can be switched manually.

**Feature and Quality Matrix** (Figure 3.11, top right) is a matrix that represents variant clafers and their values per variant. Clafers that are present in a variant are marked with a green tick and clafers that are not present in the variant are marked with a red circle. If a clafer is not present, its entire subtree cannot be present. If there are quality attributes, their values are specified in the cell. For example, the reliability of the variant 1 equals to 4, as we can see from the last row of the matrix. Commonalities are grayed out automatically: the clafer `usingMasterSwitch` is not present in any optimal variant, therefore, it is grayed out.

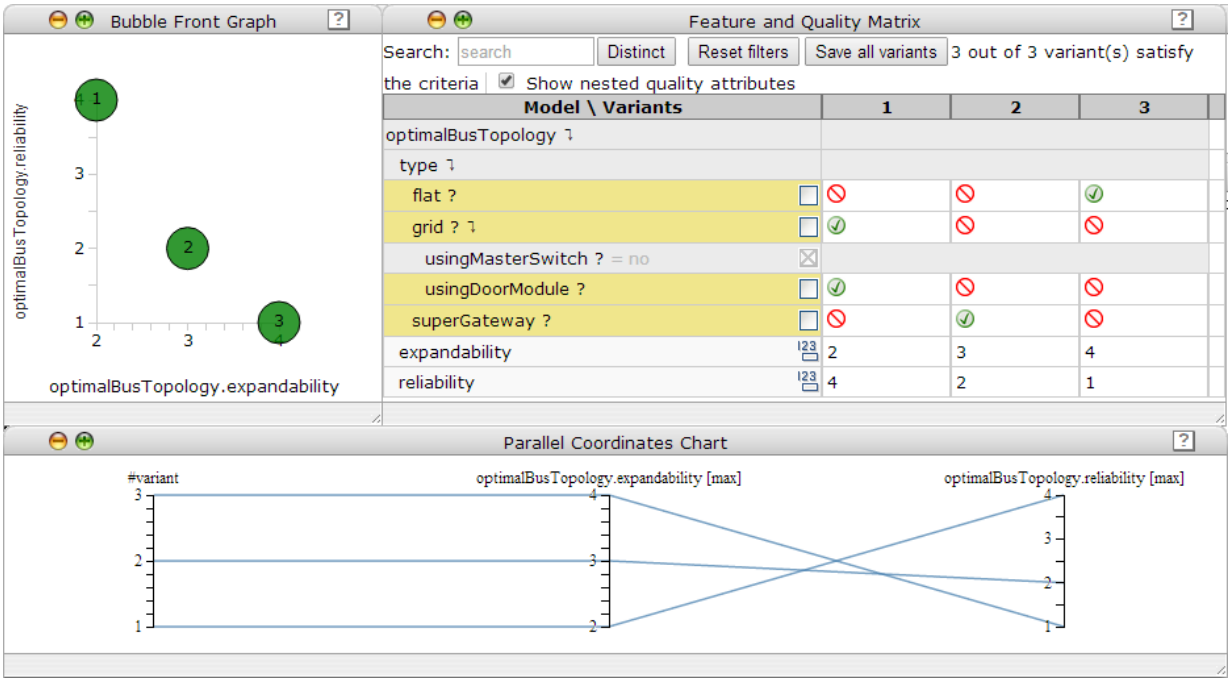



Figure 3.11: ClaferMooVisualizer with the bus example

**Parallel Coordinates Chart** (Figure 3.11, bottom) is a chart that represents each variant as solid line that goes through parallel axes. The three axes: a variant ID, expandability and reliability. The line crosses each axis in a point that corresponds to the actual variant ID or the value of the quality attribute. Each axis show a range of a given quality (or variant ID) and can be filtered to show only certain ranges of values. Axes can be rearranges manually.

**Variant Comparer** (Figure 3.12, left) is a matrix-like visualization that shows selected variants side-by-side and designed for comparison. The view is split into two parts: commonalities and differences, making it easier to focus on either.

### Exploration

Exploration use cases are listed in the related work [27]. There are two non-exclusive ways of exploration: filtering by quality values and filtering by features. We only show the first one.

For example, we are not interested in the worst expandability. Therefore, we mark the

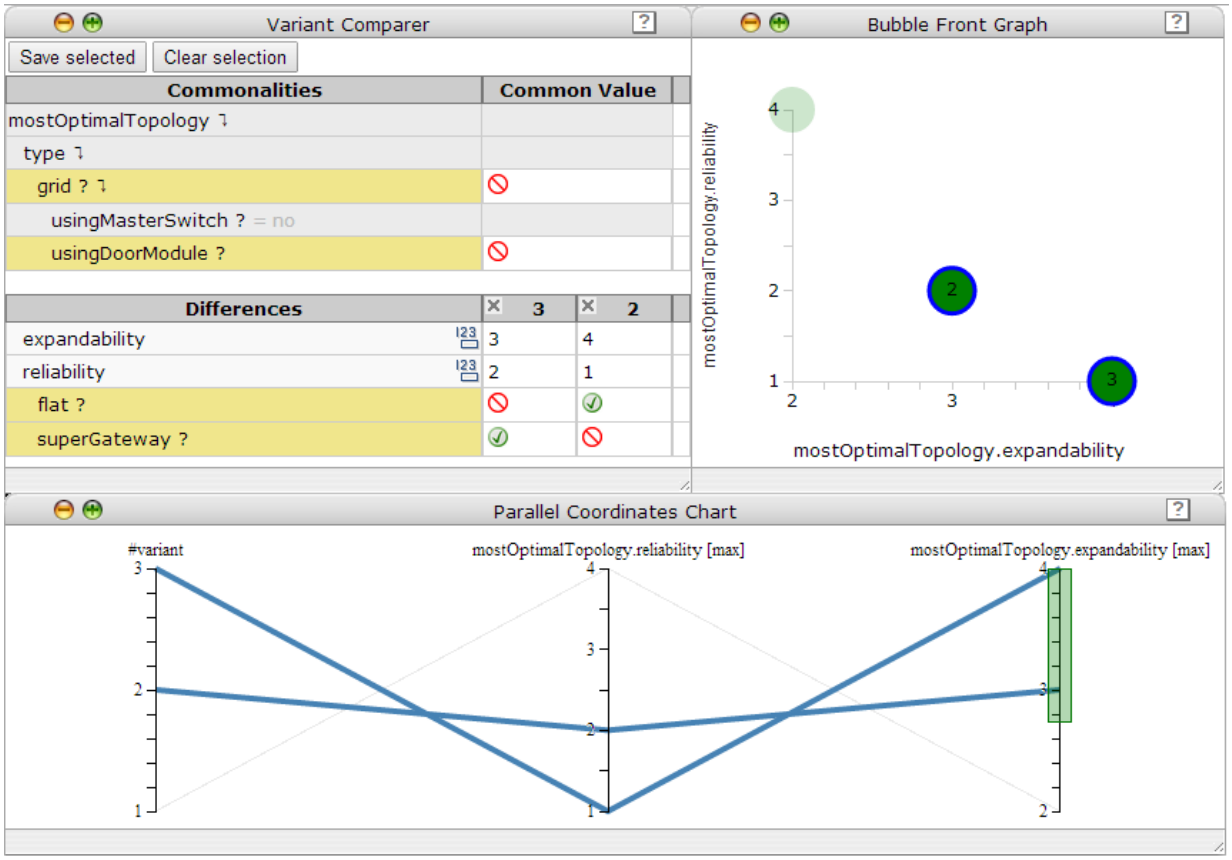



Figure 3.12: ClaferMooVisualizer with the bus example: Variant Comparer and filtering by quality

range from 3 to 4 using the Parallel Coordinates Chart's expandability axes. The views get filtered, and the variant with the expandability of 2 gets grayed out. Now, we can select the two remaining variants on the graph and use the Variant Comparer to compare them. Clearly, they are both non-grid topologies, therefore, the red circle in front of Grid is present in the Commonalities part. The Differences part lists the differences between the two variants: they have different quality and presense of `flat` and `superGateway` clafers. This concludes that the variant 3 is a *super gateway*, and the variant 2 is a *flat* bus topology.

This is a simple exploration scenario done for demonstration purposes. In our case study, however, exploration becomes essential: the instances are big and complicated, and it is not trivial to compare them and understand their position with relation to each other.

# Chapter 4

# Micro-Level Modeling Patterns and Advices

This chapter is designed for three purposes. First, for explanation of frequently used features of Clafer with clearly defined semantics, but not well-documented and problematic use, such as, *references*. Second, for solution to very common simple problems like typecasting, modeling one-to-one and many-to-one relationships in Clafer. And finally, the topics described here are building blocks for the case study.

For each subtopic, we propose potential extensions or fixes to Clafer language or tools that could facilitate better user experience and better reasoning performance.

## 4.1 Working with References

References in Clafer are used to represent UML-like unidirectional associations, such as, deployment association of a function to a device. References can be treated as pointers in languages like C: there is a pointer that points to a value stored in memory. In Clafer's context, a reference is a special type of clafer that points to another clafer instance defined elsewhere. For example, `deployedTo -> Device` is a reference that points to a single instance of a type `Device`.

Semantics of references is clearly described in [13]. In this section we describe references from the practical point of view, since working with references in Clafer is not always straightforward.

First, reference clafers often require **dereferencing** to access the actual value. By dereferencing we mean writing `.ref` in the end of the reference: `CurrentSensor.deployedTo.ref`. The code returns an instance of a `Device` the Current Sensor is deployed to.

Next, use of references is tricky when combining references and instances in the same expression, such as, comparing a reference to an instance. For such cases, we state the following rules derived from our experience and Compiler's implementation details.

1. When comparing two or more reference objects to each other, we have to use `.ref`. Omitting dereferencing results in a syntactically correct code, but semantically, the comparison is performed on references, which is usually not intended. The problem is similar to comparing pointers in languages like C++. Figure 4.1 depicts the problem. The incorrect version of the code does not use dereferencing. Therefore, the comparison is made on references, not on reference values. What makes this problem worse, comparing two references directly always results in false, since in the Clafer language design, any two clafers are disjoint unless one is in the inheritance hierarchy of another. Therefore, having such a constraint in the model will make the entire model to be unsatisfiable, and we have to use `.ref` to avoid such a mistake.

```
abstract Function
  ...
  deployedTo -> Device

WinController : Function
CurrentSensor : Function
```

(a) Problem context. DeployedTo is a reference to a Device. The modeler needs to state that the two functions — WinController and CurrentSensor — are deployed to the same device

```
[WinController.deployedTo
  = CurrentSensor.deployedTo]     //wrong

[WinController.deployedTo.ref
  = CurrentSensor.deployedTo.ref] //right
```

(b) Semantically incorrect (top) and correct (bottom) versions to specify that WinController and CurrentSensor have to be deployed on the same device. Both versions are correct from the syntactical viewpoint, though

Figure 4.1: Incorrect comparison of references: comparing without .ref

2. We do not use `.ref` in the middle of a join — a chain of clafers combined using a dot. Dereferencing is done automatically in this case by the name resolver. Specifying `.ref` in a middle of a join results in a syntax error. Figure 4.2 represents this case.

3. When equating or comparing a reference to a concrete non-reference clafer, `.ref` is not required, because it is implicitly generated by the compiler. Figure 4.3 illustrates

```
[WinController.deployedTo.ref.smart]
```

(a) Syntax error

```
[WinController.deployedTo.smart]
```

(b) Syntactically and semantically correct

Figure 4.2: Joining references

this case. In other words, having an expression with one reference operand and one non-reference operand does not require specifying `ref`. However, the types should match in any case.

4. In all other cases, we should use `.ref` since there is no guarantee it is always generated by the compiler properly. Moreover, the compiled code is more likely to be valid in terms of semantics.

```
BCM : ECU ?
[WinController.deployedTo = BCM]
```

(a) Without .ref — syntactically and semantically correct

```
BCM : ECU ?
[WinController.deployedTo.ref = BCM]
```

(b) With .ref — syntactically and semantically correct

Figure 4.3: Optional dereferencing when one operand is a reference, another one is an instance

#### Proposed corrections and extensions

Clafer Compiler should be more verbose when compiling references. A user has to be notified about obvious cases that cause contradictions, or the constraints that lead to such cases have to be forbidden.

## 4.2 Modeling One-to-One Relationships

When using references to model associations, there is a link from one object to another. However, the second object may not be aware of being referenced and may not be able to access the linked object. Moreover, an object may be referenced twice from different places and it is not forbidden by default.

For example, we have a one-to-one deployment problem. A function can be deployed on an ECU, however, we allow only one-to-one deployment in this section. We describe a many-to-one deployment deployment in a later section.

Figure 4.4a shows two concepts: `Function` and `ECU`. A function can be deployed on a single ECU. Each ECU can be deployed from a single function.

Figure 4.5 depicts the problem when deploying two different functions on the same ECU. Since in Clafer it is not forbidden, we receive an instance that has both `PinchDetection` and `WinController` deployed on the same ECU named `doorModule`.

### Method

To fix the problem described above, we introduce an inverse relation `deployedFrom`. Next, we add two constraints on the two relations. The semantics of the constraint `C1` is as follows. The function that the ECU is deployed from (referred by `this` keyword) has to be deployed to the ECU (`parent` refers to the instance of ECU). The semantics of the constraint `C2` is as follows. The ECU that the function is deployed to (referred by `this` keyword) has to be deployed from the function (`parent` refers to the instance of Function).

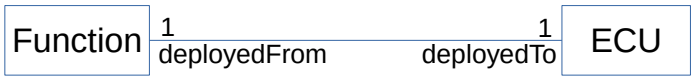
### Proposed corrections and extensions

Clafer may introduce a syntax for inverse relationships, so that a user does not have to write the inverse relationship constraints. Also, the Compiler could handle inverse relationships in a special way and thus generate a more optimal code.

## 4.3 Modeling Many-To-One Relationships

This pattern is applicable when we need to model a many-to-one (equivalently, one-to-many) relationship between two set of objects. For example, we have a deployment problem, where we deploy functions ECUs with the conditions that each function is deployed to a single hardware component, while an ECU can deploy several functions at the same time. So, we need a simple deployment association between `Function` and `ECU`. Figure 4.6a represents two association ends named `deployedTo` and `runs`, respectively.

In this pattern, we assume that the deployment can be done only to a single hardware component. If a distributed deployment is possible (i.e., we allocate the same function to




(a) UML representation

```
abstract ECU
  deployedFrom -> Function
    [this.deployedTo = parent]  // C1

abstract Function
  deployedTo -> ECU
    [this.deployedFrom = parent] //C2
```

(b) Clafer implementation

```
BCM : ECU
  deployedFrom = PinchDetection

doorModule : ECU
  deployedFrom = WinController

WinController: Function
  deployedTo = doorModule

PinchProtection: Function
  deployedTo = BCM
```

(c) Example instance in Clafer

Figure 4.4: One-to-one relationship pattern in Clafer: a UML diagram, a Clafer representation, and an example instance

more than one ECU for redundancy or function distribution), we need to model a many-to-many relationship. We can do this by extending the many-to-one relationship in a similar way as we extended one-to-one relationship to many-to-one.

### Method

Likewise in the previous pattern, we introduce an inverse relation `deployedFrom`. However, this time, the cardinality of the clafer is star (`*`), meaning there can be zero or more functions deployed to an ECU. And also, we add two constraints on the two relations. The semantics of the constraint `C1` is as follows. Every function the ECU is deployed from (referred by `this` keyword) has to be deployed to the same ECU (`parent` refers to the instance of `ECU`) as well. The semantics of the constraint `C2` is like in the previous example: the ECU that the function is deployed to (referred by `this` keyword) has to be deployed from the function (`parent` refers to the instance of `Function`).

```
abstract ECU

abstract Function
  deployedTo -> ECU
```

(a) Underconstrained, faulty implementation of the one-to-one pattern. In spite the cardinality of the reference deployedTo is one, there is no restriction constraining the number of functions deployed on the ECU

```
doorModule : ECU
BCM : ECU

WinController: Function
  deployedTo = doorModule

PinchDetection: Function
  deployedTo = doorModule
```

(b) Invalid, but syntactically correct instance that follows the faulty implementation. WinController and PinchDetection are deployed on the same ECU which contradicts our problem definition

Figure 4.5: Incorrect implementation of Many-to-One pattern with an invalid example

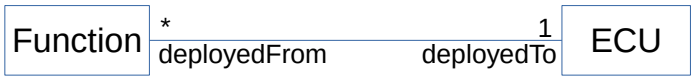
#### Proposed corrections and extensions

Inverse relationships can be marked, even in case of many-to-one relationships. The compiler can take advantage of handling inverse constraints and generate a more optimal code.

## 4.4 Typecasting

Typecasting is needed when a clafer is defined as an instance of one type, however, is used via reference typed with it's ancestor's type. In other words, we have a reference to a clafer, but the type of the reference is more generic than the type of the clafer it points to. And the goal is to access the subclafers of the more specific type via a more generic reference.

For example, we have a `Device` type, and `SmartDevice` type (Figure 4.7a). The latter is an extension of the former one. We have a switch reference pointing to a device. However, the switch in the end can be instantiated as a `Device` or a `SmartDevice`. In any case, the property `cpu` is inaccessible via the switch reference. The reason is that Device does not have this property: this property is introduced in `SmartDevice`. Therefore, it is impossible to get an access to cpu.




(a) UML representation

```
abstract ECU
  deployedFrom -> Function *
    [this.deployedTo = parent]    // C1

abstract Function
  deployedTo -> ECU
    [parent in this.deployedFrom]// C2
```

(b) Clafer implementation

```
doorModule : ECU
  deployedFrom = WinController,
     PinchDetection

WinController: Function
  deployedTo = doorModule

PinchProtection: Function
  deployedTo = doorModule
```

(c) Example instance in Clafer

Figure 4.6: Many-to-one relationship pattern in Clafer: a UML diagram, a Clafer representation, and an example instance

### Method

Figure 4.7b shows the pattern for this problem. We define an additional clafer called `switchAsSmartDevice` which gets eqruated to `switch.ref`.

### Side effects

Side effects include possible performance influence by the additional reference clafer named `switchAsSmartDevice`.

### Proposed corrections and extensions

First, Typecasting can be done very efficiently when using a proper instruction to the name resolver, so that it can resolve the properties that are not in the current type. The suggested syntax is as follows: `(SmartSwitch)switch.cpu`. For type-checking, there is already a support: `switch.ref in SmartSwitch` returns true if and only if `switch.ref` is an instance of `SmartSwitch`.

```
abstract Device

abstract SmartDevice : Device
  cpu ->> integer

switch -> Device

// cannot state switch.cpu!
```

(a) Context: SmartDevice is a subtype of Device. The type of switch is not known: it may be Device or SmartDevice. The cpu property of switch is inaccessible: Device does not have it

```
switchAsSmartDevice -> ECU ?
[switchAsSmartDevice.ref = switch.ref]

// now can access switchAsSmartDevice.cpu
```

(b) Proposed method of typecasting the switch. Define an optional reference switchAsSmartDevice that gets assigned if and only if the switch is of a type SmartDevice. Now the cpu property becomes accessible via switchAsSmartDevice

Figure 4.7: Typecasting in Clafer

Second, typecasting in case of inheritance can be implemented via redefinition. Clafer semantics [13] actually supports a notion of refinement (redefinition). If `switch` is a property of `System`, then if we create a subtype of a system called `SystemWithSmartDevice` we can redefine switch to be `switch -> SmartDevice`.

## 4.5 Queries

A notion of query we apply is similar to database "select" queries. By queries we assume extraction of relevant clafers from a set of clafers. An example query can be states as follows: "From the set of all devices, give me only those that are smart".

In Clafer, there is no native support for queries. However, they can be simulated by sets and constraints over the sets.

### Method 1. Quantifiers

First, we define a set called `localComponents`. It may contain from two up to four instances of `Device`, depending on whether we have a door module, the BCM, both or none. Constraint `C1` populates the set.

Next, we define a set of smart components `localSmartComponents`. This set stores the result of our query. The cardinality of this set is `0..4`. We may have no smart components at all, if all the switches and motors are dumb, and we do not have an ECU nor door module. The upper bound is 4, when all the local components are smart.

To populate the set of smart components, we create a query. The next two constraints `C2` and `C3` with quantifiers `all` and `no` represent such a query. The constraint `C2` with the universal quantifier `all` states that all devices `d` in a `localComponents` set will be included in the `localSmartComponents` set if and only if the device is smart. Clearly, the entire set of the local components is considered, and only smart devices can go to the "smart" set. The Clafer quantifier `all` is equivalent to the quantifier *for all*, or $\forall$.

The next constraint, `C3`, uses another universal quantifier `no` and states that it is impossible to have an element in the `localSmartComponents` set such that it is not in the set of the local components. This constraint is necessary, because the previous constraint (`C2`) does not make any restrictions on the devices that are not in the local component set, i.e, components of another window. Therefore, Clafer reasoners may put any smart component of another window into the set. The constraint `C3` denies such a case.

Queries are very expensive in terms of performance. First, there are two sets with variable cardinalities. Even though one is a subset of another, for the reasoners this fact is not clear. Moreover, there are two universal quantifiers that can cause blow-ups. In our work, we propose a method to avoid using quantifiers.

### Method 2. Instances

The idea of the method is to define several versions of clafers that get instantiated depending on the conditions. For example, a switch can be smart or not smart, and therefore, we define two instances: `smartSwitch` and `dumbSwitch`. They are nested under a clafer with a `xor` group cardinality, therefore, exactly one of the two will be instantiated. For the `smartSwitch`, we assert it is smart. For the `dumbSwitch`, we say it is not smart.

However, when having two instances, we still need a generic switch clafer that will be present no matter whether it is smart or not. To achieve this, we transform switch into a reference. Then, the reference is gets assigned to a `smartSwitch` or `dumbSwitch`, depending on what instance gets instantiated. It is achieved by the constraints `C1` and `C2`. In the constraint `C1`, parent refers to the `switch` clafer, and this refers to the `smartSwitch`. In the constraint `C2`, parent refers to the `switch` clafer, and this refers to the `dumbSwitch`.

```
abstract PWSubsystemHW
  ...
  localComponents -> Device 2..4
  [localComponents = switch, motor, doorModule, bcm.ref]              // C1

  localSmartComponents -> Device 0..4
  [all d : localComponents | (d in localSmartComponents) <=> d.smart]// C2
  [no d : localSmartComponents | !(d in localComponents)]             // C3
```

Figure 4.8: Query implementation in Clafer, method 1. Modeling queries using universal quantifiers

### Reasoning performance measurement of two methods

Since we have two alternatives, it is easier to quantify the instance generator's reasoning performance for each of the alternatives. The goal of the experiment is to compare the two modeling methods to each other in terms of solver's reasoning performance. We measure time it takes to the solver to get the first instance — this is what matters most for the user. The unit of measurement are seconds, with uncertainty of 1-2 seconds. We do not care about precise values. Moreover, out timeout is equal to 5 minutes: at the modeling stage, users do not want to wait a lot for getting the first instance. The setup was done on Intel (R) Core(TM) i7, 2.00 GHz, 8 GM RAM, x64. The model we tested the setup on is Appendix A.2, with varying number of `Subsystem` instances.

Figure 4.10 represents the performance test results for the two backend — Alloy-based backend called ClaferIG [3] and Choco-based backend called ClaferChocoIG [2]. For the Alloy-based backend, the method 1 performs faster than method 2. However, the backend is very slow, and starting at 6 subsystems, the performance drops down significantly.

For the ClaferChocoIG backend, the method 2 outperforms the method 1. Method 2 is significantly faster, moreover, it has much better scalability properties. In our opinion, the reason of good performance of the method 2 is absence of quantifiers, while method 1 does not scale well because of quantifiers. Since ClaferChocoIG is more scalable, we prefer to use this in our experiments. Therefore, method 2 gives a significant performance gain for our entire work.

```
abstract PWSubsystemHW
  xor switch -> Device
    smartSwitch : Device
      [parent = this]   // C1
      [smart]
    dumbSwitch : Device
      [parent = this]   // C2
      [no smart]
  doorModule : ECU ?
  ...
```

(a) Updated definitions of a switch

```
xor motor -> Device
  smartMotor : Device
    [parent = this]
    [smart]
  dumbMotor : Device
    [parent = this]
    [no smart]
bcm -> ECU ?
...
```

(b) Updated definitions of a motor

```
localComponents -> Device 2..4
[localComponents = switch.ref, motor.ref, doorModule, bcm.ref]
localSmartComponents -> Device 0..4
[localSmartComponents = smartSwitch, smartMotor, doorModule, bcm.ref]
```

Figure 4.9: Query implementation, method 2. Modeling queries using additional instances

### Side effects

Since for method 2, we have changed the clafer `switch` to be a reference, we need to dereference it every time we want to access the switch by writing `switch.ref`. On the other hand, we can always refer to the specific instances of `smartSwitch` and `dumbSwitch` without dereferencing.

### Proposed corrections and extensions

The proposed syntax is as follows:

```
localSmartComponents -> Device 0..4 {d : localComponents | d.smart}
```

which means that the `localSmartComponents` is a query over the `localComponents`. Arbitrary complex condition is specified after the vertical slash `|`.

In any case, the compiler and the solvers should take the semantics of a query into account and optimize the solution algorithms for dealing with queries faster.

| | Method 1: Quantifiers | Method 2: Instances |
|---|---|---|
| 3 subsystems | instant | instant |
| 4 subsystems | 5 sec | 55 sec |
| 5 subsystems | 2 min 0 sec | $> 5$ mins |
| 6 subsystems | $> 5$ mins | $> 5$ mins |
| 7 subsystems | $> 5$ mins | $> 5$ mins |
| 8 subsystems | $> 5$ mins | $> 5$ mins |

(a) Alloy-Based IG

| | Method 1: Quantifiers | Method 2: Instances |
|---|---|---|
| 3 subsystems | instant | instant |
| 4 subsystems | instant | instant |
| 5 subsystems | 4 sec | instant |
| 6 subsystems | 17 sec | instant |
| 7 subsystems | 1 min 30 sec | instant |
| 8 subsystems | $> 5$ mins | instant |

(b) Choco-Based IG

Figure 4.10: Reasoning performance of query implementation methods: measurement results. Shown time taken to generate the first instance

# Chapter 5

# Macro-Level Modeling Patterns

## 5.1 Bottom-Up Development with Modularization

Bottom-up development with modularization suggests decomposing a large problem into subproblems, modeling subproblems first, testing them, integrating, and then performing integration testing. This approach is inspired by bottom-up integration testing approach in Software Engineering [21]. In this work, we apply these ideas to the Clafer context. Also, by testing we mean a procedure of checking whether the model is satisfiable, and whether it generates valid instances within a reasonable time. In Clafer, application of this development approach is critical, and the reasons are explained in the following sections.

### 5.1.1 When to Apply

This pattern should be applied when modeling any complex model that can be decomposed. Typically, system architecture models are decomposable into hardware and functions, which, in turn, can be further decomposed, like on Figure 5.1. Cross-cutting concerns can also be modeled separately.

### 5.1.2 Rationale

Besides commonly known benefits of modularization, such as, model comprehension, separation of concerns, better maintainability, there are few very specific for Clafer. Native deficiencies of Clafer like debugging and scalability issues are critical and may become the most impeding factors in modeling using Clafer.

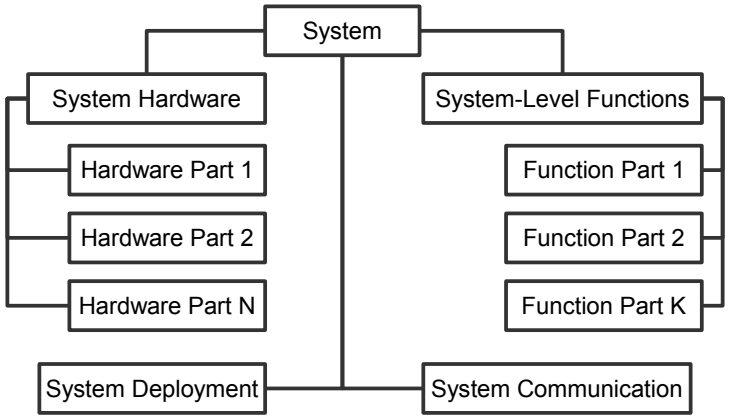



Figure 5.1: Bottom-up development with modularization pattern. Decompose system, model and test each part separately, and then integrate. Also, model cross-cutting concerns separately.

### Reason 1: Debugging Issues

Clafer models, unfortunately, are hard to debug. For example, a user creates a complex model of the Power Window system. For the sake of performance, the user does not perform modeling in a modular way. At the end, the final model is a solid bulk which cannot be easily decomposed. The model compiles perfectly, and therefore, it is syntactically correct. The user also expects the model to get instantiated. However, when trying to instantiate it, all the backends output that there is no instance found. Backends are capable of showing an *UNSAT Core* — a minimum set of constraints that causes the model to be unsatisfiable. However, it is often not the case: many constraints interact, and there are cross-dependencies across the entire model. Moreover, minimization of the UNSAT core is costly in terms of time, so for the large models it may not finish in a reasonable time.

So, a natural solution for the user is removing parts of the Clafer code to single out the cause of unsatisfiability. However, if the problem is not modularized, it is very hard to do. Eventually, the user ends up removing a lot of code, and then gradually adding constraints while the model remains satisfiable.

### Reason 2: Scalability Issues

Another problem for Clafer tools is scalability. Clafer uses exact solvers for optimization, and therefore, they have scalability problems. Moreover, Clafer Compiler may have perfor-

mance problems when compiling large models.

To define scalability problems, we introduce the notion of *a reasonable time* — a critical time period that a user can tolerate while waiting for instance generation or compilation. The reasonable time depends on the users and the user intention. However, at the modeling stage, a reasonable time may last for few minutes.

Scalability problems are identified in the following way.

- **Solver scalability**. The solver is running, however, within a reasonable time, neither an instance, nor the unsatisfiability message is shown. This means the solver is running and likely, tries to find solutions, however, the user does not know the result.
- **Compiler scalability**. The compilation is running, however, at some point the compiler seems to be hanging, and no output is generated within the reasonable time.

What makes the scalability problem worse is that it is very hard to predict exactly, what is the critical size for the models. A model with a thousand lines of Clafer code can work faster than the one with few hundred lines — it depends on the constraints, types of clafers used, etc. Sometimes adding an additional constraint causes a dramatic decrease or a dramatic improvement in performance.

#### Reason 3: Solver Implementation Issues

Clafer is relatively new project, and it is under constant development and improvement. The new versions 0.3.6.1 are mostly free from the major bugs. However, bugs or implementation issues can still be present. Therefore, a model can be slow to instantiate or mistakenly shown to be unsatisfiable not because of the model, but because of the solver bugs or inefficient implementations.

To be able to resolve bugs, singling out a root cause is important. Again, it is very hard to do if the model is built not in a modular way. Therefore, starting with a small model is really important.

### 5.1.3 How to Apply

Our methodology suggests the following:

1. **Decompose the modeling problem into parts**. If it is the Power Window system, decompose the system into each window subsystem and model communication between the subsystems separately. Moreover, decompose each window subsystem into functional and hardware architectures. Hardware can be decomposed further into wiring and devices, if required.

2. **Pick a small, simple subproblem and model it**. For example, pick a hardware topology of a driver power window subsystem. Model the parts, and compile the model. Also, do not create a big concrete model.

3. **Debug and test the subproblem**. Create a test case — example configuration for the driver window hardware. Run Clafer backends and test it.

4. **Model another small subproblem, and repeat this step**, until all the problems on this level are considered.

5. **Go one level up and model a bigger problem**. The complete system model include nested instantiations of the subsystems. Keep combining subproblems until you have the complete system modeled or faced performance problems.

6. **Model cross-cutting concerns separately**. Concerns like deployment, system communication or safety should not be scattered across the model. They can be tested separately and then included in the final model.

### 5.1.4 Side Effects

Well-structured in terms of modularization model may have some performance overhead because of inheritance, multiple instantiations and references. We do not look into the ways to minimize side effects. However, in our opinion, side effects are minimal compared to the benefits earned by this approach.

## 5.2 Collaboration

Collaboration is a process of interaction of several objects with the purpose to achieve a concrete goal. The idea is inspired by UML collaboration diagrams, however, in our context, the collaboration pattern has a different meaning. In Clafer, collaboration pattern is a creational pattern that facilitates:

1. Constraining several existing clafers—collaborators to make them achieve a concrete goal. For example, add deployment constraints in a way that every function is deployed on a hardware.

2. Creating new clafers to help the existing clafers—collaborators interact to achieve a concrete goal. For example, to make certain deployments possible, wire connectors have to be created.

We also define a notion of *collaboration environment* — a "board" the collaborating clafers act on. We call the clafers to be *collaborators*.

Figure 5.2 represents an example of collaboration. Four hardware components of a driver power window subsystem — collaborators — are put together to create wires and bus communication. To achieve the proper communication, potentially three mutually exclusive discrete wire connectors, four mutually exclusive power wire connectors, and four bus connections can be created, depending on conditions. The new connectors are not owned by any of the system devices: they belong to the collaboration environment.

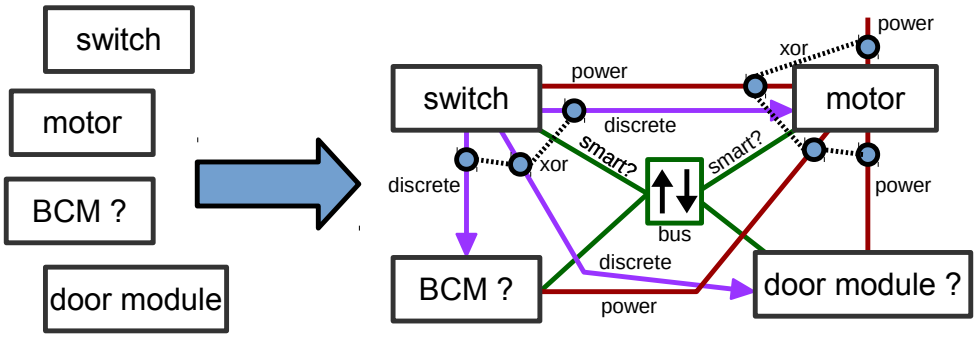



Figure 5.2: Collaboration pattern example. Deciding on connections across devices

### 5.2.1 When to Apply

This pattern can be applied to model cross-cutting concerns. Examples include: modeling communications across devices, modeling a deployment problem.

### 5.2.2 Rationale

Separation of concerns, avoiding scattering and ease of debugging are one of the primary reasons of using this pattern in Clafer.

**Reason 1: Avoiding Scattering**

Isolating and encapsulating collaboration in a separate place avoids scattering across the entire model. For example, deciding on deployment could be done in the definition of each analysis function. Thus, the properties of a function and its use could be done in the same place when the function is defined. However, we have many functions, and some deployment rules depend on the deployment decisions of other functions. Therefore, this creates complexity in the function definition.

**Reason 2: Separation of Concerns**

Isolating collaboration in a separate place allows for separation of concerns. Deployment, wiring, communication — all these are various concerns to be addressed.

**Reason 3: Easy Debugging**

Obviously, separation of concerns and encapsulating collaboration facilitates faster debugging.

### 5.2.3 How to Apply

Our methodology suggests the following:

1. **Decide on the goal**. The goal may be: to create wire connectors or to add deployment constraints.
2. **Create a collaboration environment clafer**. Create a separate clafer called `PowerWindowWiring` or `PowerWindowDeployment`
3. **Add collaborators**. Choose the required devices, ECUs, or functions. Add them to the collaboration environment clafer as references.
4. **Constrain collaborators**. Create deployment or wiring rules and add necessary constraints to the collaborators
5. **Create supplementary clafers**. Create necessary connectors or buses.

6. **Test the collaboration environment**. The ready environment can be tested separately.

7. **Include the collaboration environment into the main model**. We add an instance of `PowerWindowWiring` or `PowerWindowDeployment` to the root `PowerWindowSystem` clafer.

### 5.2.4 Side Effects

Collaboration objects have some performance overhead because of references. Also, when using reference, there is inconvenience in terms of working with references (i.e., writing `.ref` every time we need to access `switch`). We do not analyze or quantify side effects in this work.

# Chapter 6

# Power Window Control Case Study

## 6.1 Introduction

The Power Window (**PW**) system is a part of a car electronic-electric architecture that is responsible for Power Window operation (Figure 6.1). The most basic feature of the system is moving the window up or down depending on a switch button pressed.

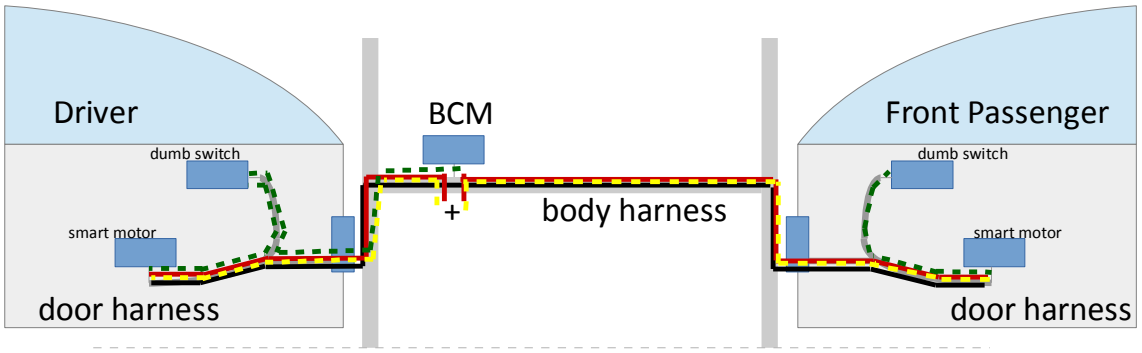



Figure 6.1: Power Window system E/E architecture overview: two-window configuration. Devices are blue horizontal rectangles. Power wire connector is a solid red line, device power connector is a dashed yellow line, discrete wire connector is a dashed green line. Power wire connectors go to the power supply fuse located in the body harness close to BCM — Body Control Module. Bus is a black solid line

### 6.1.1 Motivation

There are three reasons of choosing the Power Window system for the thesis case study. First, the PW system's E/E architecture is a rich representative of a car E/E architecture: it can be decomposed into subsystems, it has various configurations, and its hardware and functionality is variability-rich. Second, the PW system is a relatively small and self-contained system compared to, for example, the power-train system. Third, the PW system operation principles are easy to explain and understand without going too detailed on car control theory.

The complete power window system consists of two subsystems — for two-window configuration, or four and more — to support rear passengers. Subsystems are not completely isolated: the driver window's switch — called the master switch (MS) often can control other windows as well. Moreover, Body Control Module (BCM) is often involved in system operation. The subsystems communicate to each other via LIN (local area interconnection network) buses or discrete, power and analogue wires.

### 6.1.2 Challenges

In spite of relative simplicity of the power window subsystem, its E/E architecture is not straightforward. First, complexity comes from device variability. Switches can be smart or dumb, so do motors. Some configurations have dedicated *door modules* — separate ECUs. Moreover, certain configurations may also involve BCM.

Next, complexity comes from variability of connectors. Motor can be controlled via power wires, discrete wires or a LIN bus. Driver and front passenger window may communicate via a bus, while communication to the rear windows can be done via discrete wires. Analogue wires are also used when connecting a hall sensor to a device that drives the motor.

And finally, complexity comes from deployment variability. Control functions can be distributed across switches, motors, door modules and the BCM. Many deployments are theoretically possible.

### 6.1.3 Structure and Scope

The flow of the case study is built incrementally and consists of two parts.

1. Step-by-step modeling and design exploration of a single subsystem — Driver Window Subsystem
2. Extending the model by adding a front passenger window subsystem to make the complete PW System

For the first part, we perform and explain the following activities:

1. Creating a configuration model in terms of features (`expressUp`, `express`, `otherRemoteControl`) — Section 6.3.1
2. Modeling functional architecture in terms of analysis functions and functional devices, according to the EAST-ADL standard — Section 6.3.2
3. Modeling hardware topology and generate the proper one with accordance to the functional architecture — Section 6.3.3
4. Modeling possible deployments of functions onto the hardware — Section 6.3.4
5. Modeling wire connectors and buses corresponding to deployment rules — Section 6.3.4
6. Performing design exploration via specialization constraints — Section 6.6
7. Adding quality attributes and performing design exploration via multi-objective optimization — Section 6.7

For the second part, we demonstrate integration methods and a big illustrative example on optimization-based design exploration — Section 6.8. At the end of the case study, we make a summary and discuss limitations in Section 6.9.

## 6.2 Methodology

Our methodology is described in the previous chapters 3, 4 and 5.

We apply Macro-Level Patterns (Chapter 5) in the following way. *Bottom-up development* pattern (Sec. 5.1) suggests system-to-part decomposition and bottom-up development with testing. The approach is illustrated on Fig. 5.1. Thus, we start with a simple subsystem — a driver window subsystem.

When modeling cross-cutting concerns, such as, deloyment and wiring, we apply the *Colaboration* pattern as described in Sec. 5.2. However, we do not use this pattern excessively: we apply the pattern once for both deployment and wiring, since collaborators are the same, and therefore, there is no point introducing more side effects in terms of reasoning performance.

Throughout the modeling process, we refer to Chapter 3 for choosing appropriate modeling methods and building blocks. Also, we apply micro-level patterns 4 whenever it is necessary. For example, we use one-to-many modeling pattern (Sec. 4.2) to model function-to-hardware deployment.

We also use EAST-ADL levels to structure our modeling: we start with features ( *Vehicle Level*), then model functional architecture (*Analysis level*) and step down to the *Design level* to model hardware. These levels are orthogonal and complementary to our bottom-up development approach.

For optimization and instance generation, we use ClaferChocoIG solver [2] throughout the case study. For visualization and exploration, we use ClaferMooVisualizer [27] and ClaferConfigurator [10].

## 6.3 Modeling Features, Functions and Hardware

### 6.3.1 Vehicle Level: Features

According to EAST-ADL standard, we start with the *Vehicle level*. At the this level, we define features that are relevant to our systems and subsystems. Since we start with a subsystem, we list only the features specific to a subsystem.

First, each power window normally supports basic up and down features, we call id `BasicUpDown`. A human presses up or down button on his or her switch to make the window move up or down.

Next, some power windows support `express` features. It means a driver or a passenger can open or also close his or her window completely with one click on an express button. We model a feature with implementation in mind, therefore, therefore, we distinguish the feature `expressUp` as it makes our subsystem more complex. Here is the configuration model so far:

```
abstract PWSubsystemConfig
```

```
basicUpDown  // mandatory feature
express ?    // at least express down
  expressUp ?  // both express up and express down
```

So, the possible configurations will be: `express`, but no `expressUp` — means will have only express down. If we have both `express` and `expressUp` — we have both express up and express down. And finally, if we have neither of these, our subsystem does not support express features at all, only `BasicUpDown`. We cannot have `expressUp` without `express` because the former one is nested under the latter one, and can be present if the parent — `express` — is present.

### 6.3.2 Analysis Level: Functional Architecture

Based on our experience and observations of Power Window systems of various cars, we define a functional architecture for the system. We define it in terms of EAST-ADL concepts of *Analysis Function* and *Functional Device* and connectors across them (Sec. 2.1). Also, we introduce variability to the functional architecture, so that it can support various configurations. Figure 6.2 represents our functional architecture for the driver window subsystem.

*Switch* is a functional device that receives inputs from a driver. A driver presses one of the up and down buttons or express variations of those. We denote *Switch* functional device's output as *Request*.

Cars with a rooftop or remote operation (using a remote key) are able to adjust window position without switch operation. The request denoted by *Remote Request* is sent to the subsystem and then combined with the normal *Request*. We assume there is a specific function called *Remote Request Arbitrator* that does this combination. As a result, the *Arbitrated Request* goes further to a control function. As noted earlier, not all cars have remote operation. Therefore, the entire set — *Remote Request*, *Remote Request Arbitrator* and *Arbitrated Request* is optional its presence has to be configured. Now we realize that we need such a feature, thus we add it to the configuration feature model and name it `remoteOperation ?`, making it optional.

We name the main PW control function as *WinController*. This function is responsible for sending commands (*Command* signal) to the motor, while handling the extreme conditions when the window is already fully open or fully closed. To be able to verify these conditions, this function can measure the current running through the motor. If the current is too high, this means the motor is stuck, and two of the reasons of that are exactly

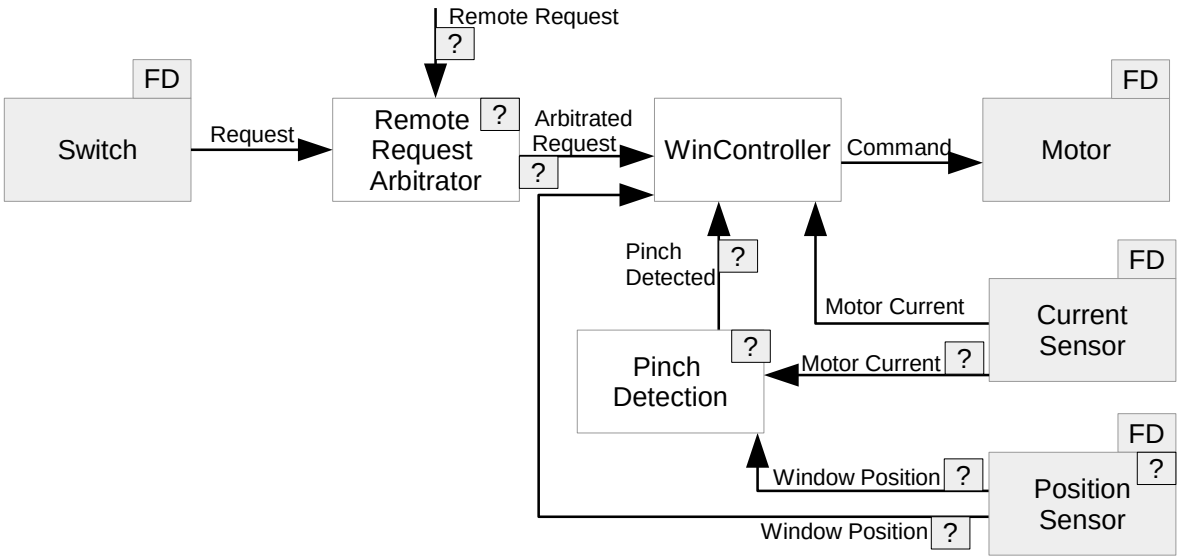



Figure 6.2: Complete PW driver subsystem's functional architecture. Hollow blocks represent analysis functions, shaded blocks with the FD label — functional devices. Optional analysis functions, functional devices and connectors are denoted using a question mark.

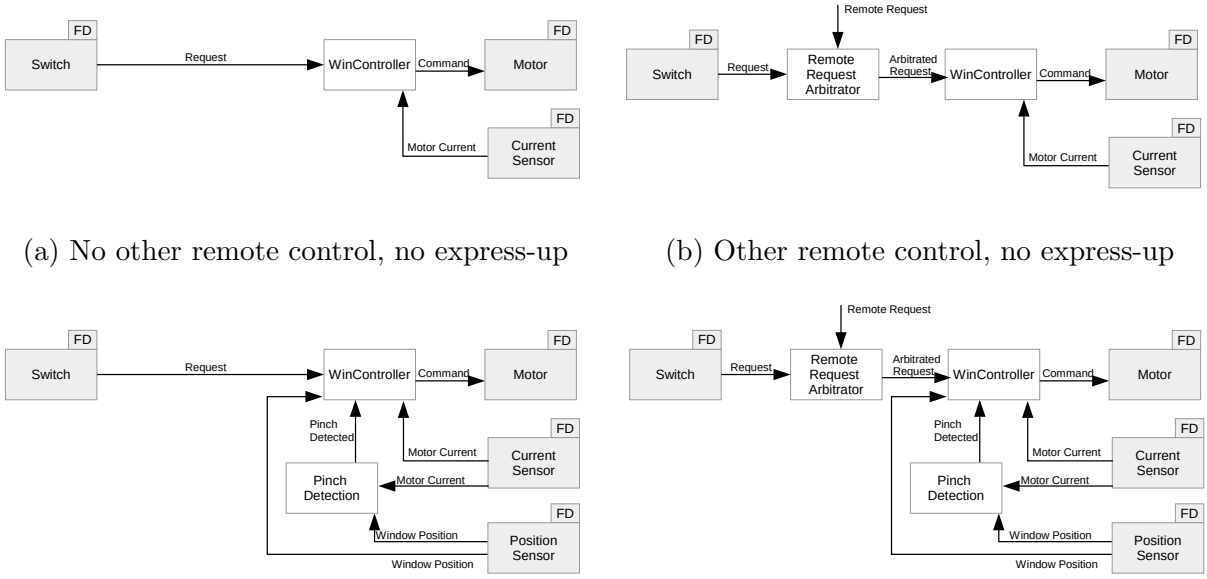
the extreme cases of the fully open and the fully closed window. So, there is a connection *Current* from a so-called *Current Sensor* to *WinController* which decides whether to send requests or not.

To clarify, in the final deployment, *Current Sensor* and *WinController* can be deployed onto the same device, so this connection may not be mapped to an actual wire. In case of an *express-down* feature support, *WinController* is supposed to send signals to the *Motor* functional device continuosly. At the final deployment step, *WinController* may be physically put on a smart motor, therefore, the motor may control itself and handle all the extreme cases and express requests. At the current abstraction level, however, the *Motor* is an abstraction and represents a functional device with a single function: moving the window up or down.

As noted earlier, some PW systems support the *express-up* feature. From the implementation point of view, the system becomes more complex. One of the reasons is a function called *Pinch Detection* that is supposed to stop the window from moving up in case it hits an obstacle, i.e., a human finger, to avoid injuries. This function, however, for security

reasons, should not stop the window from moving up if the window is being closed by a normal up button pressed by a human. Moreover, *Pinch Detection* function is supposed to be switched off in case the window is just about to be completely closed, so that it can be closed fully without falsely detected obstacle. Thus, the function accepts signals from another functional device that we denote as *Position Sensor* and sends a signal *Pinch Detected* to *WinController* which can stop the window from moving up. As in case of remote arbitrator, the entire set of functions and connectors related to express-up functionality is optional and marked with a question mark.

At this point, we have an analysis-level model of the PW system's driver subsystem. Depending on the configuration, optional functional devices and functions with corresponding connections may or may not be included in the concrete functional architecture. Figure 6.3 represents all four possible variants of the functional architecture.




(a) No other remote control, no express-up

(b) Other remote control, no express-up

(c) No other remote control and express-up

(d) Other remote control and express-up

Figure 6.3: Four possible variants of PW driver subsystem's functional architecture

The top two variants (Fig. 6.3a and Fig. 6.3b) do not support express-up, since they have neither pinch detection nor position sensor. They differ only by inclusion of a remote request arbitration feature. The bottom two variants (Fig. 6.3c and Fig. 6.3d) do support express-up and also differ by remote request arbitration functions.

This concludes the general description of PW driver subsystem's functional architecture.

Compared to the Driver subsystem, Passenger subsystems are slightly more complex. First, there is another arbitration component — *Switch Arbitrator* — that is supposed to arbitrate requests from the main switch and the passenger switch. In addition to that, rear window subsystems also have a child lock functionality which adds additional functions. In our work, we follow the bottom-up development pattern, therefore, we start modeling a simple (driver) subsystem first, test it, and then continue doing extensions.

### Basic types

First, to be able to model our concepts, we need to define types — a small metamodel. We define `AnalysisFunction` and `FunctionalDevice` concepts. Also, according to EAST-ADL specification, `FunctionalDevice` inherits `AnalysisFunction`. The semantic difference is that functional devices get and send information from or to the plant. However, we do not capture this distinction and use it for convenience.

```
abstract AnalysisFunction
abstract FunctionalDevice : AnalysisFunction
```

Next, we need an abstraction of a connector. We define a connector in a way we described in Sec. 3.1 and denote this concept as `AFConnector` ("analysis function connector"). Since functional devices inherit analysis functions, we can type both ends of a connector to be `AnalysisFucntion`:

```
abstract AFConnector
  src -> AnalysisFunction
  dest -> AnalysisFunction
```

The abstractions above combined with expressive power of Clafer are sufficient to model the entire functional architecture.

### Subsystem Functional Architecture

First, we define an abstract clafer called `PWSubsystemFunctionalArchitecture`. We follow the approach described in Sec. 3.1 and include all analysis functions and functional devices we see on the Figure 6.2 in the model:

```
abstract PWSubsystemFunctionalArchitecture
  WinController : AnalysisFunction
  Motor : FunctionalDevice
  Switch : FunctionalDevice
  CurrentSensor : FunctionalDevice

  PositionSensor : FunctionalDevice ?
  PinchDetection : AnalysisFunction?
  OtherRemoteArbitrator : AnalysisFunction ?
```

Next, we model connections by instantiating connectors. For example, we create a connection `conCommand` and specify that the source is `WinController` and the destination is `Motor` functional device. We decide to group certain connections (the ones related to pinch detection), so that the entire group inclusion or removal is easy. And also, we model two options for the connection with the source at `Switch`: in case we have a remote control, the signal goes to the remote arbitrator, otherwise it goes directly to `WinController`. The code below enumerates all the connectors we need:

```
  conCommand : AFConnector
    [src = WinController]
    [dest = Motor]
  conCurrent : AFConnector
    [src = CurrentSensor]
    [dest = WinController]

  conArbitratedRequestFromOtherRemoteArbitratorToWinController : AFConnector ?
    [src = OtherRemoteArbitrator]
    [dest = WinController]
  conRequestFromSwitchToOtherRemoteArbitrator : AFConnector ?
    [src = Switch]
    [dest = OtherRemoteArbitrator]
  conRequestFromSwitchToWinController : AFConnector ?
    [src = Switch]
    [dest = WinController]

  pinchDetectionConnections ?
    conPositionToWinController : AFConnector
      [src = PositionSensor]
```

```
    [dest = WinController]
  conPositionToPinchDetection : AFConnector
    [src = PositionSensor]
    [dest = PinchDetection]
  conCurrentToPinchDetection : AFConnector
    [src = CurrentSensor]
    [dest = PinchDetection]
  conPinchDetectionToWinController : AFConnector
    [src = PinchDetection]
    [dest = WinController]
```

And the final step is to add proper constraints with respect to the subsystem features. We add a reference to the configuration feature model and link it to the architecture model by using equivalence constraints in the following way:

```
config -> PWSubsystemConfig

[config.otherRemoteControl <=> OtherRemoteArbitrator]
[config.otherRemoteControl <=>
    conRequestFromSwitchToOtherRemoteArbitrator]
[config.otherRemoteControl <=>
    conArbitratedRequestFromOtherRemoteArbitratorToWinController]

[no config.otherRemoteControl <=> conRequestFromSwitchToWinController]
```

Thus, if we have `otherRemoteControl`, then we will have `OtherRemoteArbitrator` and the two connections. Otherwise, we have a direct connection from `Switch` to the `WinController`. Note that we use equivalence, since one directional implications are not enough to completely deny the absence of clafers.

And finally, we configure the pinch detection and position sensor functions and connectors in a similar way by connecting them to the `expressUp` feature. Now it becomes clear that grouping is useful, since we can enable or disable the entire group.

```
[config.express.expressUp <=> PinchDetection]
[config.express.expressUp <=> PositionSensor]
[config.express.expressUp <=> pinchDetectionConnections]
```

This makes our analysis model complete and configured with respect to the configuration feature model.

#### Testing

To be able to test, we move level up and create a clafer that will include instances of both `PWSubsystemFunctionalArchitecture` and `PWSubsystemConfig`. We call it `PWSubsytem` and its declaration and definition is done in the following way:

```
PowerWindowFunctionalArchitecture
  driverFAConfig : PWSubsystemConfig
  driverFA : PWSubsystemFunctionalArchitecture
    [config = driverFAConfig]
```

The code above is just enough to test our functional architecture. We run our code in any of the Clafer backends. The textual instances that get generated are very big, thus we do not include them. However, we get four instances in total, and they correspond exactly to the four functional architectures on Figure 6.3. The instances are generated instantaneously, and therefore, we do not have reasoning performance problems and are good to go.

### 6.3.3 Design Level: Hardware Topology

Now we move one level down according to the EAST-ADL specification and model a hardware topology. We also accommodate variability as we did for the functional architecture. We still follow the same bottom-up development approach: we start with a single subsystem only, and then combine them into a complete system hardware with testing. This subsection has many references to Chapter. 3, since we describe our methodology using examples from the hardware domain. Thus, we do not repeat the details in this subsection.

#### Basic types

First, we need to define basic types as described in Sec. 3.1. We prefer the alternative Fig. 3.2a to reduce the number of abstract clafers:

```
abstract Device
  electronic ?
  smart ?

  [smart => electronic]
```

And second, we define an ECU as a smart device:

```
abstract ECU : Device
  [smart]
```

This completes our hardware architecture so far. We model connectors in later sections.

### Defining a subsystem

Now, since we have the basic types to work with, we can define a subsystem. We already described hardware subsystem definition in Section 3.1 on a similar example.

Moreover, we need queries to represent a set of smart components. We apply the method two described in Sec. 4.5. We define two references — a `switch` and a `motor` both are of the type `Device`. Next, we define all the supporting instances: `smartSwitch`, `dumbSwitch`, `smartMotor`, and `dumbMotor`. A complete definition for the hardware subsystem is illustrated below:

```
abstract PWSubsystemHardware

  doorModule : ECU ?
  switch -> Device
  motor -> Device
  bcm -> ECU ?

  localComponents -> Device 0..4
  [localComponents = switch.ref, motor, bcm.ref, doorModule]
  localSmartComponents -> Device 0..4
  [localSmartComponents =  smartSwitch, smartMotor, bcm.ref, doorModule]

  smartSwitch : Device ?
    [smart]
    [switch = this]
  dumbSwitch : Device ?
    [no smart]
    [switch = this]
  [smartSwitch xor dumbSwitch]
```

```
  smartMotor : Device ?
    [smart]
    [motor = this]
  dumbMotor : Device ?
    [no smart]
    [motor = this]
  [smartMotor xor dumbMotor]
```

### Testing

We define another concrete clafer — `PWSystemHardware` — and instantiate our subsystem underneath. We also link the `bcm` reference to the actual BCM instantiated at this level:

```
PWSystemHardware
  driverSubsystemHardware : PWSubsystemHardware // instance for the
                                                // driver window
    [bcm = BCM]           // // configuration - setting the bcm reference

  BCM : ECU ?
```

Now we execute the model in one of the backends. There are 36 instances generated in total, meaning there are 36 possible designs for the driver window hardware, excluding connectors. We show only two instances below: the smallest instance and the largest one. The smallest instance looks as follows:

```
myPWSystemHardware
    driverSubsystemHardware
        switch = dumbSwitch
        motor = dumbMotor
        localComponents#0 = dumbSwitch
        localComponents#1 = dumbMotor
        dumbSwitch
        dumbMotor
```

There is no door module nor BCM. Both the switch and the motor components are dumb. There are two local components in total: switch and the motor. There are no smart components, so this is not shown.

The largest instance is very different. There is a door module and BCM, both are present. All components are both smart and electronic. There are all four smart components:

```
myPWSystemHardware
    driverSubsystemHardware
        doorModule
            electronic
            smart
        switch = smartSwitch
        motor = smartMotor
        bcm = BCM
        localComponents#0 = smartSwitch
        localComponents#1 = smartMotor
        localComponents#2 = doorModule
        localComponents#3 = BCM
        localSmartComponents#0 = smartSwitch
        localSmartComponents#1 = smartMotor
        localSmartComponents#2 = doorModule
        localSmartComponents#3 = BCM
        smartSwitch
            electronic
            smart
        smartMotor
            electronic
            smart
    BCM
        electronic
        smart
```

### 6.3.4 Design Level: Deployment and Wiring

First, we have to augment our metamodel with deployment capabilities. We use the *many-to-one* relationship pattern Sec. 4.2:

```
abstract AnalysisFunction
  deployedTo -> Device
```

```
    [parent in this.deployedFrom]

abstract Device
  ...
  deployedFrom -> AnalysisFunction *
    [this.deployedTo = parent]
```

To model deployment rules, we follow the collaboration pattern described in Sec. 5.2. So, we start with declaring the collaboration environtment called `PWSubsystemDeployment` and define two collaborators: in our case, the functional architecture `fa` collaborates with a hardware topology `ht`:

```
abstract PWSubsystemDeploymentAndWiring
  ht -> PWSubsystemHardware
  fa -> PWSubsystemFunctionalArchitecture
```

### Deployment rules

First, we deploy functional devices that have a clear destination. Clearly, the `Switch` functional device has to be deployed to `switch` hardware device, and `Motor` has to be placed on the `motor`:

```
  [fa.Switch.deployedTo.ref = ht.switch.ref] // to the local switch only
  [fa.Motor.deployedTo.ref  = ht.motor.ref ] // to the motor device only
```

To formalize the deployment rules, we analyze existing power window systems and extract common patterns. We also introduce new concepts that become useful to simplify deployment rules. For example, we add a concept of a *motor driver* — a device that supplies a load power to the motor. This abstraction is useful, because it simplifies reasoning on deployment of `CurrentSensor` and `WinController` functions. Since we do not know in advance what device drives the motor, we simply specify that this can be any of the local components. Moreover, if we have both BCM and a local door module, we would never assign the motor driver role to the BCM — it is suboptimal. Therefore, we have the following model:

```
  // Motor Driver
```

```
motorDriver -> Device
[motorDriver.ref in ht.localComponents.ref]
// MotorDriver can be any of {BCM, switch, motor, doorModule}

[(ht.doorModule && ht.bcm) => (motorDriver.ref != ht.bcm.ref)]
// if we have a door module and BCM, then the motor driver is not BCM
```

Next, we need to configure deployment in case we have a double-express functionality. If we do have `express.expressUp`, then the following conditions must be true. The `PositionSensor`, `PinchDetection` and `WinController` have to be deployed to smart components. Otherwise, it is impossible for other functions to work with `PinchDetection`, since the latter one has to be on a smart component. Moreover, if the position sensor is not deployed to the motor driver, it has to be deployed on a motor. All the constraints actually imply that the motor driver or a motor have to be smart:

```
// ExpressUp: Position Sensor, Pinch Detection, WinCotnroller
[fa.config.express.expressUp =>
  (
    (fa.PositionSensor.deployedTo.ref in ht.localComponents.ref) &&
    (fa.PinchDetection.deployedTo.ref in ht.localSmartComponents.ref) &&
    (fa.WinController.deployedTo.ref in ht.localSmartComponents.ref) &&
    ((fa.PositionSensor.deployedTo.ref != motorDriver.ref)
        => (fa.PositionSensor.deployedTo.ref = ht.motor.ref))
  )
]
```

Further, we have to state that `WinController` has to be deployed to an electronic component, if the window supports the `express` feature. The reason is that purely electric hardware cannot send repeated signals to the motor on its own. Therefore, the hardware has to be either smart (which implies it is `electronic`) or it has to include simple electronic circuits that support the required functionality (just electronic, not smart):

```
[fa.config.express => (fa.WinController.deployedTo.electronic)]
```

Now we consider deployment of `WinController` and `CurrentSensor` in more details. First, if the motor driver is smart, then `WinController` has to talk to it via a bus, meaning that it has to be deployed on a smart device. The current sensor is deployed to the motor driver, since the motor driver supplies power and therefore, can measure the current flowing through the motor:

```
[motorDriver.smart => (
    (fa.WinController.deployedTo.ref in ht.localSmartComponents.ref) &&
    (fa.CurrentSensor.deployedTo.ref = motorDriver.ref)
  )
]
```

The next case includes a completely dumb motor driver (meaning it's a dumb switch or a dumb motor, without any electronics). If that's the case, then two cases are possible. Either both `WinController` and `CurrentSensor` are deployed to the switch, or both are deployed to the motor:

```
[!motorDriver.electronic => (
// if the motor is driven by a dumb component (not even electronic)
  (
    (fa.WinController.deployedTo.ref = ht.motor.ref) &&
    (fa.CurrentSensor.deployedTo.ref = ht.motor.ref)
    // both WinController and CurrentSensor are on the motor (thermistor)
  )
  || // or
  (
    (fa.WinController.deployedTo.ref = ht.switch.ref) &&
    (fa.CurrentSensor.deployedTo.ref = ht.switch.ref)
    // both window controller and the current sensor are on the dumb switch
  )
)]
```

And the third case, if the motor is driven by an electronic but not smart device, then `WinController` can be deployed to any component (of course, it has to talk to the motor driver) and, the current sensor is deployed on the motor driver, since it can measure the current:

```
[(motorDriver.electronic && !motorDriver.smart) =>
  (fa.CurrentSensor.deployedTo.ref = motorDriver.ref) &&
  (fa.WinController.deployedTo.ref in ht.localComponents.ref)
]
```

And finally, we need to accommodate `OtherRemoteArbitrator`, if it's present. To make this rule short and simple, we say that if `WinController` is not on the motor, then the remote arbitrator has to go to the same place as `WinController`. In any case, the arbitrator has to be allocated on a smart component:

```
[fa.OtherRemoteArbitrator =>
  (fa.WinController.deployedTo.ref != ht.motor.ref =>
      (fa.OtherRemoteArbitrator.deployedTo.ref
             = fa.WinController.deployedTo.ref)
  )
  &&
  (fa.OtherRemoteArbitrator.deployedTo.ref in ht.localSmartComponents.ref)
]
```

This concludes our set of deployment rules for the PW driver subsystem. The next step is to generate wiring information.

### Connectors and Wiring Information

We can model wiring in the same collaboration environment, since deployment and wiring are closely related: if two functions have to communicate, then they should be either deployed to the same device, or there has to be a connection between the two devices the functions are deployed to. Moreover, we may add more deployment rules or impose additional constraints on the devices while modeling wiring.

In our approach, we consider bus communication separately and currently focus only on the wires: discrete, power and analog connectors. First, we need to extend our metamodel. We use the approach described in Sec. 3.1 to model the connectors:

```
abstract WireConnector
  src -> Device
  dest -> Device

abstract DiscreteWireConnector : WireConnector
abstract AnalogWireConnector : WireConnector
abstract PowerWireConnector : WireConnector
```

Now, we work in the collaboration environment named `PWSubsystemDeploymentAndWiring` and specify connectors. First, we need a power wire connector from the motor driver to the motor. However, if the motor drives itself, we obviously do not need such a connection[1], but rather assert that the motor is smart:

```
wireCommand : PowerWireConnector ?
  [src = motorDriver.ref]
  [dest = ht.motor.ref]

[(motorDriver.ref = ht.motor.ref) <=> ht.motor.smart]
// if the driver is not the motor, then we need a power wire for the command
// we are not making discrete connection to the motor
[ht.motor.smart <=> no wireCommand]
```

Next, consider a connection from `PositionSensor` to the motor driver. It is present only if we have `expressUp`. Moreover, since the sensor is deployed only to the motor driver or the motor, we need a physical connection only in the case the motor does not drive itself. Also, we realize that we need another constraint: if we support express-up, and the motor driver is a switch, then it has to be smart.

```
wirePosition : AnalogWireConnector ?
  [src = ht.motor.ref]
  [dest = motorDriver.ref]

[wirePosition <=>
  (fa.config.express.expressUp && (motorDriver.ref != ht.motor.ref) )]
// the wire is present if and only if we have express-up,
// and the motor driver is not on the motor
[(fa.config.express.expressUp && motorDriver.ref = ht.switch.ref)
  => ht.switch.smart]
```

The third connector we may need is a wire from the `switch` directly to the place where `WinController` resides. Obviously, we do not need any wire if `WinController` is on the switch. If `WinController` is not on the switch, then two cases are possible. If the switch drives the motor, then we do not need any wire, since the power wire is already there to connect the motor driver to the motor. If the switch does not drive the motor and it is not smart, we need a wire. If the switch is smart, we can still have a wire for redundancy.

[1] we model power supply from the fuse separately

```
[!fa.config.otherRemoteControl =>
  ((fa.WinController.deployedTo.ref = ht.switch.ref) => no wireRequestDirect ) &&
  ((fa.WinController.deployedTo.ref != ht.switch.ref ) =>
    (
      ((ht.switch.ref != motorDriver.ref && !ht.switch.smart)
           => wireRequestDirect ) &&
      ((ht.switch.ref = motorDriver.ref) => no wireRequestDirect )
    ))
]

wireRequestDirect : DiscreteWireConnector ?
  [src = ht.switch.ref]
  [dest.ref in ht.localSmartComponents.ref]
```

In case we have a remote control, `OtherRemoteArbitrator` has to be on a smart component, therefore, we need a wire only in case the switch is not smart.

```
[fa.config.otherRemoteControl => (
    (!ht.switch.smart) <=> wireRequestIndirect)
]

wireRequestIndirect : DiscreteWireConnector ?
  [src.ref = ht.switch.ref]
  [dest.ref in ht.localSmartComponents.ref]
// from switch to any other local smart component
// {BCM, smartMotor, doorModule}
```

All other connections are either modeled using buses — we model them in a separate section, or no connection required — i.e., functions are allocated to the same device.

## 6.4 Integration of Features, Functional Architecture, Deployment and Wiring

System integration is described in Section 3.1. We instantiate every entity we modeled previously — deployment and wiring, hardware and functional architecture, include them in `PowerWindowSystem` and configure:

```
PowerWindowSystem
  DeploymentAndWiring : PWSubsystemDeploymentAndWiring
  driverFAConfig : PWSubsystemConfig
  driverFA : PWSubsystemFunctionalArchitecture
    [config = driverFAConfig]
  driverHardware: PWSubsystemHardware
    [bcm = BCM]

  BCM : ECU ?
```

Now we have a complete driver subsystem with all the deployment rules and wiring.

## 6.5 Design Space Complexity

We can now generate possible PW driver subsystem topologies with deployment. We execute a backend — ClaferChocoIG — and it generates **876** possible designs. All of them satisfy our deployment rule specification. The reasons we get this many instances are as follows:

1. **Configuration space**. We have several features, including express features and other remote control. If we assert `otherRemoteControl` to be excluded by adding the constraint `[no driverFAConfig.otherRemoteControl]`, our design space decreases down to 502 instances.

2. **Hardware variability**. As we discovered while testing the hardware, there are 36 possible hardware designs determined by presence or absence of the door module or BCM, having smart or dumb components.

3. **Deployment variability**. In case of functions that are required to be deployed on a smart component and we have more than one smart component, a function can be deployed on any of these, since we assume smart-to-smart communication via buses across all the smart components. Therefore, in case we have both the smart switch, the door module, the BCM and the smart motor, and we have 2 functions to be deployed on a smart component, we have $2 * 4! = 2 * 4 * 3 * 2 = 48$ possible deployments of these functions.

Clearly, examination of every possible design even in case of having visualization tools is impractical. We see two ways of approaching the big design space problem.

1. **Adding conditions and constraints**. For example, we can ask the solver to produce only purely electric designs with the dumb devices only. Or, we can generate only designs that involve BCM and do not include additional door modules. We represent this approach in the next section Sec. 6.6.

2. **Optimization**. Optimization of deployments. Second, some deployments can be optimized. For example, if the functionality can be achieved using dumb devices, then there is no point of having expensive smart devices — a cost aspect. Moreover, we may want to maximize the utilization of the switch and motor: there is no point of sending signals from switch back to the BCM and then to the motor. And finally, we can optimize communication by minimizing the length and the mass of wiring. We explore all these approaches in the optimization Sec. 6.7.

## 6.6 Constraint-Based Design Space Exploration

### 6.6.1 Basic Electric Design Example

First, we start with a purely electric configuration. In Clafer, we can specify it as follows[2]:

```
PureElectric : PowerWindowSystem
  [driverHardware.dumbSwitch]
  [driverHardware.dumbMotor]
  [no BCM]
  [no driverFAConfig.express]
  [no driverFAConfig.otherRemoteControl]
```

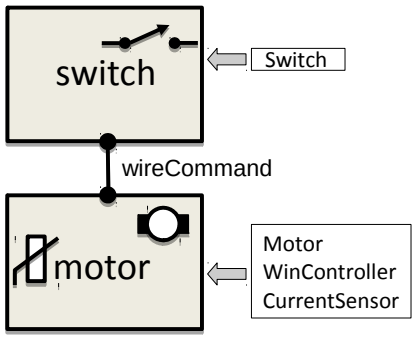
As a result, only two instances are generated. Both of them are pure electric.

Figure 6.4 illustrates the first variant. The analysis functions `WinController`, `Motor` and `CurrentSensor` are deployed to the `motor`. The current sensor can be implemented using a thermistor located on the motor. If the temperature increases (the current is too high), the thermistor activates and breaks the circuit. This functionality of stopping the motor belongs to the `WinController` function, therefore, it is located on the motor as well. With regards to the switch, it is purely dumb and supplies power to the motor via a power wire connector called `wireCommand`.

[2]Under current tools implementation, we have to convert the clafer PowerWindowSystem into an abstract to be able to instantiate it

```
dumbSwitch : Device
    deployedFrom = Switch
dumbMotor : Device
    deployedFrom#1 = WinController
    deployedFrom#2 = Motor
    deployedFrom#3 = CurrentSensor
wireCommand : Power Wire
    src = dumbSwitch
    dest = dumbMotor
```

(a) Textual Form



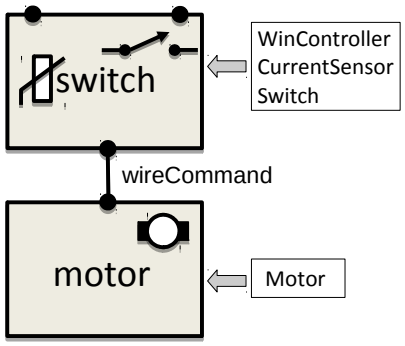

(b) Graphical Form

Figure 6.4: Basic electric variant 1

Figure 6.5 illustrates the second design variant. The switch also supplies power as in the previous variant. However, now the cut-off functionality is located on the switch. The switch supplies power, therefore, the current can be measured on the switch and trigger the cut-off. Thus, the three analysis functions — `Switch`, `CurrentSensor` and `WinController` are deployed to the `switch`. The motor is purely dumb without any other functionality.

```
dumbSwitch : Device
    deployedFrom#1 = Switch
    deployedFrom#2 = WinController
    deployedFrom#3 = CurrentSensor
dumbMotor : Device
    deployedFrom = Motor
wireCommand : Power Wire
    src = dumbSwitch
    dest = dumbMotor
```

(a) Textual Form




(b) Graphical Form

Figure 6.5: Basic electric variant 2

### 6.6.2 Contradiction Example

An engineer can mistakenly create a configuration that cannot be implemented. As in the previous examples, an engineer needs a purely electric configuration, but also needs to be able to control the windows remotely:

```
PureElectric : PowerWindowSystem
  [driverHardware.dumbSwitch]
  [driverHardware.dumbMotor]
  [no BCM]
  [no driverFAConfig.express]
  [driverFAConfig.otherRemoteControl] // is on
```

The following happens in the solution. `driverFAConfig.otherRemoteControl` enables the clafer `OtherRemoteArbitrator` to be present. There is a deployment rule that says that `OtherRemoteArbitrator` has to be placed on a smart device. Since there are no smart devices, Clafer backends output a contradiction message, and no instance gets generated. A similar effect may happen if we request an `expressUp` functionality while having dumb devices only. However, if we request for the *express-down* feature only, it can be implemented without smart devices, but then the tool will enforces a switch or a motor to be electronic, so that `WinController` can send signals continuously.

### 6.6.3 Complex Example with Smart Devices

The next configuration we try is as follows. There is `expressUp` and `otherRemoteControl` functionality. Both switch and motor are smart. There is no door module, but there is the BCM.

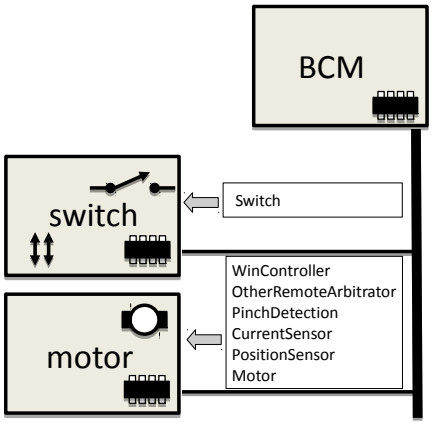
```
SmartSwitchAndMotorAndBCM : PowerWindowSystem
  [driverFAConfig.express.expressUp]
  [driverFAConfig.otherRemoteControl]
  [driverHardware.smartMotor]
  [driverHardware.smartSwitch]
  [no driverHardware.doorModule]

  [BCM]
```

Clafer tools generate **15** possible designs. All of them have the same hardware topology, and the motor drives itself in all the designs, because the motor is smart. Also, the `CurrentSensor` and the `PositionSensor` are deployed to the motor. The communication is implemented via a bus and not shown in the instance, since we model it separately.

The main variability comes from the deployment: `PinchDetection` can reside on any of the three smart devices (3 options), `WinController` can also reside on all three components, while `OtherRemoteArbitrator` is deployed on the motor without WinController (2 possibilities) or the device the `WinController` is deployed to (3 options). So, in total, we have $3 * (2 + 3) = 15$. Figure 6.6, Figure 6.7, and Figure 6.8 represents few generated deployments.

```
smartSwitch
    deployedFrom = Switch
smartMotor
    deployedFrom#1 = WinController
    deployedFrom#2 = Motor
    deployedFrom#3 = CurrentSensor
    deployedFrom#4 = PositionSensor
    deployedFrom#5 = PinchDetection
    deployedFrom#6 = OtherRemoteArbitrator
BCM
```

(a) Textual Form




(b) Graphical Form

Figure 6.6: Complex smart variant 1. Smart motor implements almost all the functionality

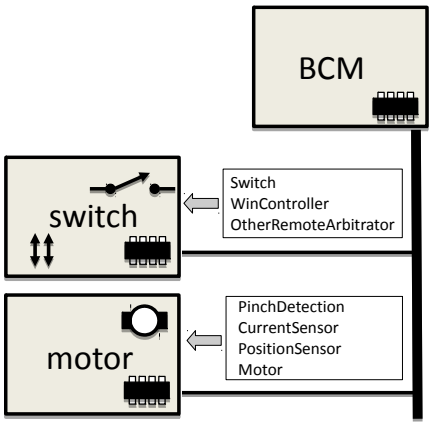
Clafer tools include a tool called *ClaferConfigurator*. It is a variant of *ClaferMooVisualizer* and also includes *Feature and Quality Matrix* visualization. We can use this tool to visualize and compare all the 15 instances. Unfortunately, the tool is too general, therefore it shows all the clafers even the ones we are not interested in. However, the tool hides all the commonalities and highlights the differences as illustrated by Fig. 6.9.

Unfortunately, ClaferConfigurator tool's tabular visualization has limitations. The tool is not designed to work well with reference clafers: it explicitly lists each set of functions in a cell, therefore, there is not enough space to show all of them. The tool, however, can be extended to support problem-specific custom visualizations.

We can create more configurations and explore them in a similar way. For example, other commonly used configurations include:

```
smartSwitch
    deployedFrom#1 = Switch
    deployedFrom#2 = WinController
    deployedFrom#3 = OtherRemoteArbitrator
smartMotor
    deployedFrom#1 = Motor
    deployedFrom#2 = CurrentSensor
    deployedFrom#3 = PositionSensor
    deployedFrom#4 = PinchDetection
BCM
```

(a) Textual Form




(b) Graphical Form

Figure 6.7: Complex smart variant 2. Load is balanced between the smart switch and smart motor

1. Having dumb switches, but smart motors.
2. Having smart switches, but dumb motors.
3. Having smart special door modules, but the switches and motors are all dumb.

All the possible options are supported by our model and can be found among the entire set of designs.

## 6.7 Optimization-Based Design Space Exploration

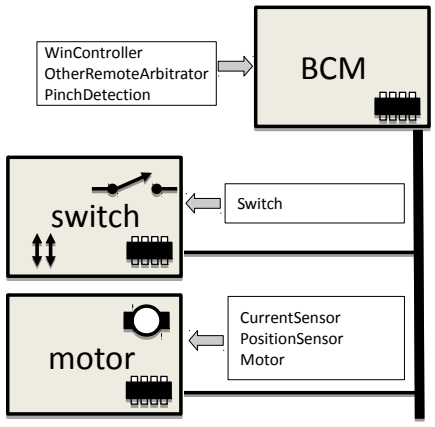
Optimization may give us two benefits:

1. Optimization reduces the design space by showing only the most optimal configurations.
2. Optimization allows for discovery of new designs not implemented previously.

In this section, we walk through all the steps of optimization: from defining the quality attributes to the visualization of optimal designs.

```
smartSwitch
    deployedFrom = Switch
smartMotor
    deployedFrom#1 = Motor
    deployedFrom#2 = CurrentSensor
    deployedFrom#3 = PositionSensor
    deployedFrom#4 = PinchDetection
BCM
    deployedFrom#1 = WinController
    deployedFrom#2 = OtherRemoteArbitrator
```

(a) Textual Form




(b) Graphical Form

Figure 6.8: Complex smart variant 3. BCM is involved and has some control functionality

### 6.7.1 Adding Quality Attributes

First, we need to define quality attributes. The following options are possible in our problem domain.

#### Number of smart components

As we saw in the complex example, the number of smart components is actually more than required to accommodate all the functionality. In some configurations, `BCM` has no functionality deployed nor communication implemented, therefore, it may not be needed in the context of the system. Moreover, all the functionality can be placed on the motor, therefore, there may be no necessity in having a smart switch, for example.

Another rationale is that smart components implement functions as a software. Therefore, it becomes more error-prone.

And finally, the number of conversions from discrete to digital inputs matter. It can be another quality attribute, however, it is related to the number of small components.

#### Wire length

Computing wire length has certain benefits. It is the best to have wires to be as short as possible, because of the following reasons. First, shorter links may result in less latency.

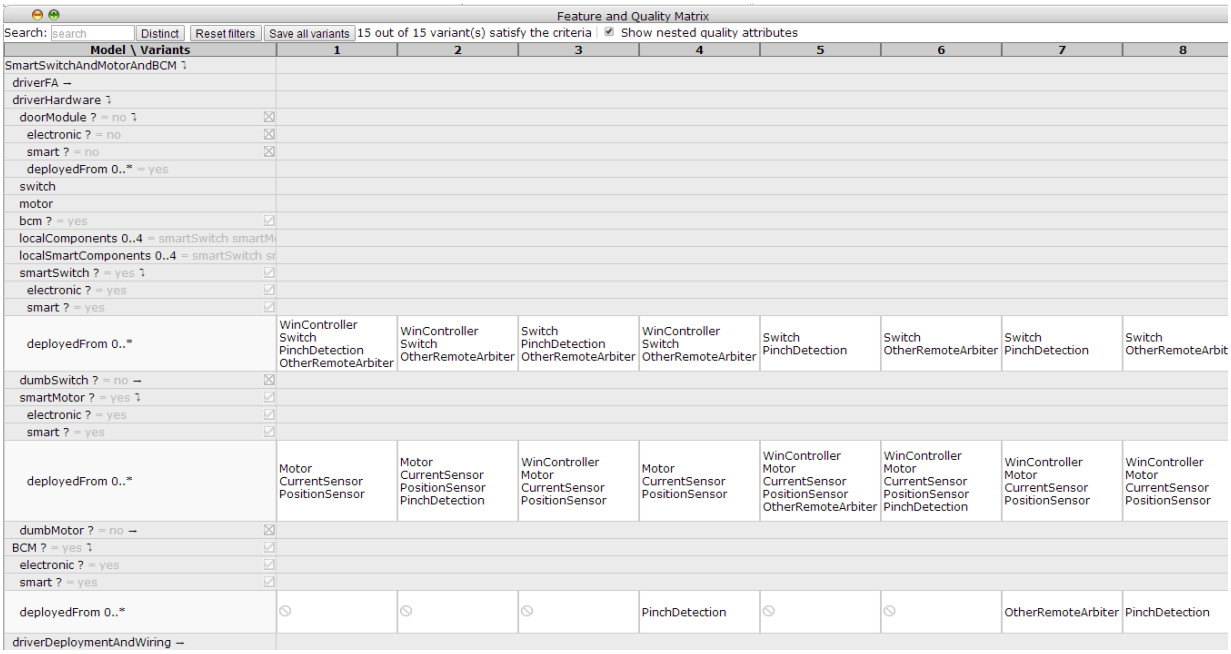



Figure 6.9: Clafer Configurator with the two basic electric examples

Having a wire from the main switch to the BCM and then back to the motor is likely to cause delays. Therefore, a link from the switch directly to the motor is more preferable. And second, shorter connections are less expensive, and the cost factor is very important in manufacturing such systems.

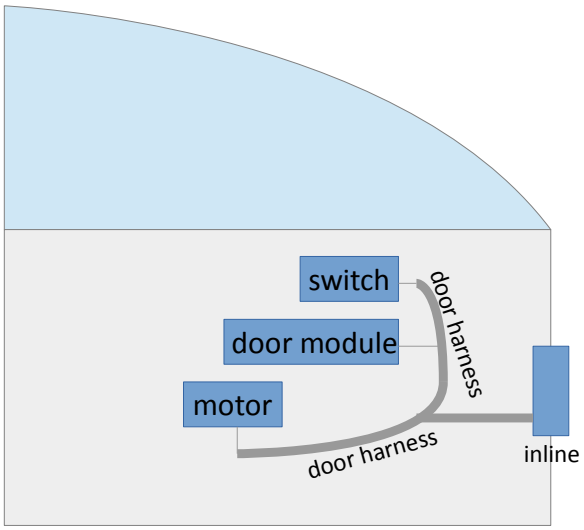
### Wire mass

We can define the mass of wire as wire length times wire thickness. Wire mass matters when we consider various types of wires. Load power wires are thicker than discrete wires, thus, they are more expensive. Wire mass metric can help dealing with this problem.

## 6.7.2 Modeling Optimization Problem

### Modeling Wire Length and Mass

We briefly explained modeling quality attributes in Sec. 3.1 on wire connector examples. We do the same in our case study: we define wire length, thickness and mass:




(a) Door Harness

```
switchToMotor = 40cm
switchToDoorModule = 20cm
motorToDoorModule = 30cm
inlineToSwitch = 45cm
inlineToMotor = 45cm
inlineToDoorModule = 35cm

inlineToSwitch = 25cm
inlineToMotor = 25cm
inlineToDoorModule = 15cm
```

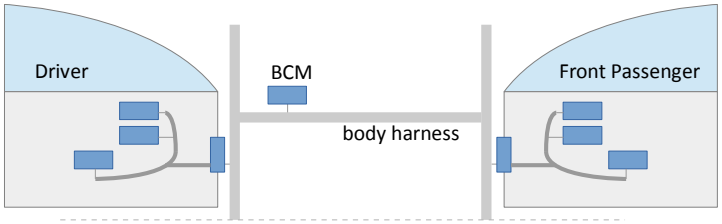
(b) Distance Data

Figure 6.10: Door harness and distances

```
abstract WireConnector
  ...
  length ->> integer                        // wire length
  thickness ->> integer                     // wire thickness
  mass ->> integer = length * thickness     // wire mass (thickness * length)

abstract DiscreteWireConnector : WireConnector       // inherits WireConnector
  [thickness = 1]     // take as a base thickness

abstract PowerWireConnector : WireConnector          // inherits WireConnector
  [thickness = 7]     // power wire is ~7 times thicker than a discrete one
```

Thickness is fixed for each type of wire. However, length is variable and depends on the wire harnesses in a car. In our example, we use the door harness topology represented on Figure 6.10 and body harness topology illustrated on Figure 6.11. Note that all distance data is encoded as constants: therefore, we can easily support other harness topologies.

Now, we walk through every wire connector we have and assign its length in accordance to harnesses.




(a) Body Harness

```
inlineDtoInlineFP = 170cm
inlineDtoBCM = 40cm
inlineFPtoBCM = 130cm
```

(b) Distance Data

Figure 6.11: Body harness and distances

First, the `wireCommand`. Its length depends on what the motor driver is. Therefore, we add an integer value for this distance and compute it using conditions.

```
abstract PWSubsystemDeploymentAndWiring
...
  distFromMotorToMotorDriver ->> integer
  [(motorDriver.ref = ht.switch.ref) =>
    (distFromMotorToMotorDriver = Dist.switchToMotor)]
  [(motorDriver.ref = ht.motor.ref) =>
    (distFromMotorToMotorDriver = 0)]
  [(motorDriver.ref = ht.doorModule) =>
    (distFromMotorToMotorDriver = Dist.motorToDoorModule)]
  [(motorDriver.ref = ht.bcm.ref) =>
    (distFromMotorToMotorDriver = Dist.inlineToMotor + ht.dist.inlineToBCM)]
```

Therefore, the `wireCommand` length becomes `distFromMotorToMotorDriver`:

```
  wireCommand : PowerWireConnector ?
    [src = motorDriver.ref]
    [dest = ht.motor.ref]
    [length = distFromMotorToMotorDriver]
```

Next, we consider `wirePosition`. It connects the motor to the motor driver. However, it is composed of two wires (the position sensor sends two signals), therefore, we multiply the length by two:

```
wirePosition : AnalogWireConnector ?
  [src = ht.motor.ref]
  [dest = motorDriver.ref]
  [length = 2 * distFromMotorToMotorDriver]
```

The lengths of `wireRequestDirect` and `wireRequestDirect` are more tricky. First, the number of wires vary for the switch with express command and the basic up-down switch. We handle it using the following statement:

```
numberOfDiscreteWiresFromSwitch ->> integer
[numberOfDiscreteWiresFromSwitch = if fa.config.express then 3 else 2]
```

Next, the wire length depends on the destination. Again, we include a set of implication statements to handle all possible cases:

```
wireRequestDirect : DiscreteWireConnector ?
  [src.ref = ht.switch.ref]
  [dest.ref in ht.localSmartComponents.ref]
  // from switch to any other local smart component
  // {BCM, smartMotor, doorModule}
  [(dest.ref = ht.motor.ref) =>
    (length = numberOfDiscreteWiresFromSwitch * Dist.switchToMotor)]
  [(dest.ref = ht.doorModule) =>
    (length = numberOfDiscreteWiresFromSwitch * Dist.switchToDoorModule)]
  [(dest.ref = ht.bcm.ref) =>
    (length = numberOfDiscreteWiresFromSwitch * (Dist.inlineToSwitch
                                                + ht.dist.inlineToBCM))]
```

For the `wireRequestIndirect`, the calculation is exactly the same. Therefore, we have all the four wires with distance calculation specified.

### Modeling Power Supply

Our optimization picture will not be complete unless we consider power supply and power wires. First, we list our assumptions:

1. A device that supplies a power to the motor needs to get this power from the fuse.

2. All electronic (including smart) devices need a device power, which is significantly less than the load power.

3. Device power wire has a similar thickness as discrete wires.

4. Power supply for both load and device power originates from a fuse located near the BCM.

5. We ignore *ground* for simplification assuming we can ground each device at its own location. Modeling ground is not difficult, though.

Taking all this into account, we can model power supply. We add a `powerFuse` as non-electronic, not smart device to the hardware topology. Then we start with a load power supply for the motor driver. We follow the same approach for the distance calculation as we did previously. Note that we add distances from a device to the door inline and the distance from the inline to the power fuse. Since power fuse is close to the BCM, we assume the distance between the BCM and the fuse is 0 (see Fig. 6.20 for an idea of power flow).

```
wireFromLoadPowerFuseToMotorDriver : PowerWireConnector
  [src.ref = ht.powerFuse]
  [dest.ref = motorDriver.ref]
  [(motorDriver.ref = ht.switch.ref) =>
    (length = ht.dist.inlineToPowerFuse + Dist.inlineToSwitch)]
  [(motorDriver.ref = ht.motor.ref) =>
    (length = ht.dist.inlineToPowerFuse + Dist.inlineToMotor)]
  [(motorDriver.ref = ht.doorModule) =>
    (length = ht.dist.inlineToPowerFuse + Dist.inlineToDoorModule)]
  [(motorDriver.ref = ht.bcm.ref) => (length = 0)]
```

Next, we model device power supply. We add an abstraction for the device power wire:

```
abstract DevicePowerWireConnector : WireConnector  // inherits WireConnector
  [thickness = 1] // device power wire has the same base thickness
```

Then, for each device, if and only if it is electronic (or smart), we add a device power supply. For example, for the door module looks as follows:

```
[ht.doorModule <=> wireFromDevicePowerFuseToDoorModule]
wireFromDevicePowerFuseToDoorModule: DevicePowerWireConnector ?
  [src.ref = ht.powerFuse]
  [dest.ref = ht.doorModule]
  [length = ht.dist.inlineToPowerFuse + Dist.inlineToDoorModule]
```

Length calculation is simple in this case. We do the same for the switch and the motor. For the BCM, we do not add a device power supply since it's already at the same place by our assumption.

### Modeling Buses

Modeling buses is a complicated part since bus may connect more than two devices. Moreover, there are various bus topologies and number of buses as described in the Section 2.3.1. Also we need to carefully calculate the bus length in each case.

Obviously, if the number of devices is less than two, we cannot have any bus. Otherwise we approach bus modeling in the following ways represented on Fig. 6.12.

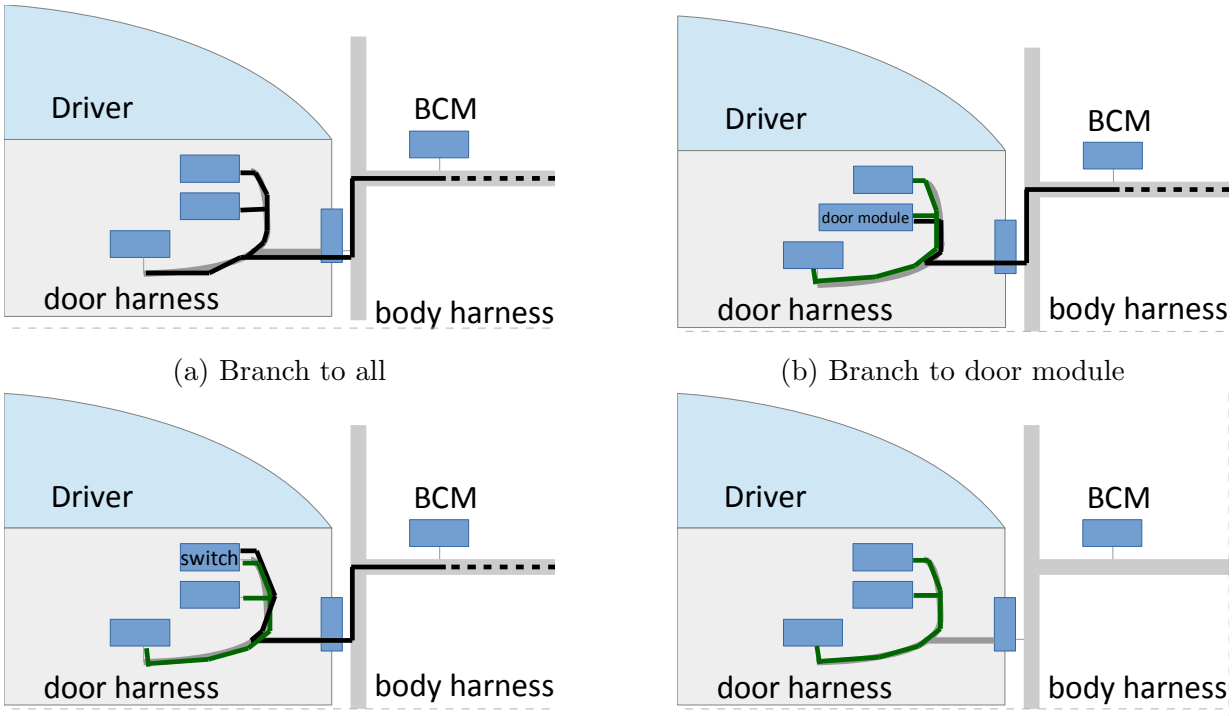
1. A single bus from the outside of subsystems (via BCM) connects each of the local smart devices — a *Flat* topology. We call it *branch to all smart components* (Fig. 6.12a).

2. The bus connects the BCM to the door module, and then the door module connects the rest local devices – applicable for both *Grid: Door Module* and *Super Gateway*. We call this option *branch to door module* (Fig. 6.12b).

3. The main switch is a hub that has a connection to other subsystems (and BCM if needed), and then the main switch has a local subnetwork connecting the rest local devices — a *Grid: Master Switch* topology. We name it *branch to switch* (Fig. 6.12c).

4. A single bus that connects each of the local devices — also *Flat* topology, but just for the door subsystem. We call this option *local bus only* (Fig. 6.12d).

For each case, we compute the bus length by adding length portions depending on presence or absence of a device. We abstract away wires from a device to the bus, but they are included in the length portions we add. Below is an example of length computation:




(a) Branch to all

(b) Branch to door module

(c) Branch to switch

(d) Local bus only

Figure 6.12: Bus topology implementation approach. Door branch bus is represented as a black solid line. Door local bus is a green solid line. A bus may go beyond BCM — a dashed black line — it does not matter in this implementation approach

```
xor busType ?
  branchToDoorModule
    ... (length computation)
  branchToSwitch
    ... (length computation)
  branchToAllSmartComponents
    [(no smartSwitch && no smartMotor && doorModule) => busLength =
        dist.inlineToBCM + Dist.inlineToDoorModule]
    [(no smartSwitch && smartMotor && no doorModule) => busLength =
        dist.inlineToBCM + Dist.inlineToMotor]
    [(no smartSwitch && smartMotor && doorModule) => busLength =
        dist.inlineToBCM + Dist.inlineToMotor + Dist.doorSpliceToDoorModule]
     ... (4 more cases)
```

```
  localBusOnly
    ... (length computation)
```

The approach above is relatively simple, yet extensible. First, it covers multiple bus topologies at once: Fig. 6.12b can be used for both *Super Gateway* and *Grid: Door Module*. Next, branch-based options are extensible: if the branch comes to a door harness, it does not matter where it comes from: it may connect BCM or go directly to the front passenger's smart switch.

### Adding Total Quality Attributes and Objectives

We recall our `PowerWindowSystem` clafer and add the total quality attributes.

```
PowerWindowSystem
  ...
  quality
    numberOfSmartComponents ->> integer
       = #driverHardware.localSmartComponents
    wireLength ->> integer = sum WireConnector.length
    wireMass ->> integer = sum WireConnector.mass
```

Clearly, we need to minimize all the objectives, so we write:

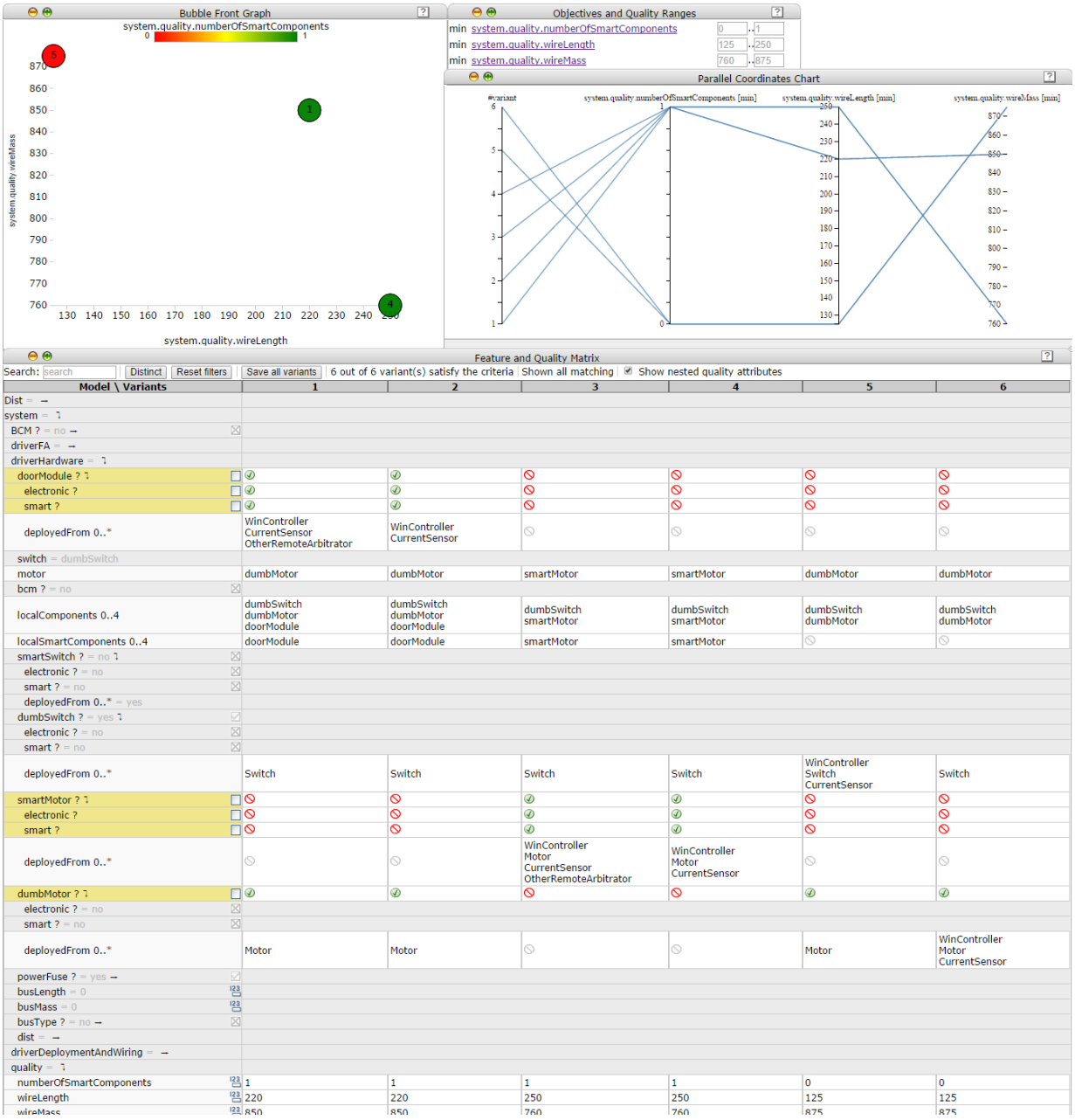
```
<<min PowerWindowSystem.quality.numberOfSmartComponents >>
<<min PowerWindowSystem.quality.wireLength >>
<<min PowerWindowSystem.quality.wireMass >>
```

## 6.7.3 Running Optimization

Now we execute our problem in the ClaferMooVisualizer tool described in Sec. 2.3. The tool generates 6 optimal designs, or variants. Figure 6.13 shows the tool screenshot with the **6** generated instances.

*BubbleFrontGraph* shows three clusters in terms of quality. Each cluster has two instances in it which have exactly the same quality, so they are shown on the graph merged. The number shown on the circle is an ID of a representative[3].

[3]the number on the circle should not be misinterpreted as the number of instnaces in the cluster



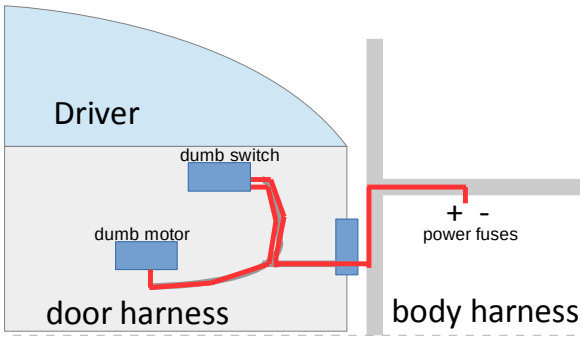

| Model \ Variants | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Dist = – | | | | | | |
| system = ↴ | | | | | | |
| BCM ? = no – | | | | | | |
| driverFA = – | | | | | | |
| driverHardware = ↴ | | | | | | |
| doorModule ? ↴ | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ |
| electronic ? | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ |
| smart ? | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ |
| deployedFrom 0..* | WinController<br>CurrentSensor<br>OtherRemoteArbitrator | WinController<br>CurrentSensor | | | | |
| switch = dumbSwitch | | | | | | |
| motor | dumbMotor | dumbMotor | smartMotor | smartMotor | dumbMotor | dumbMotor |
| bcm ? = no | | | | | | |
| localComponents 0..4 | dumbSwitch<br>dumbMotor<br>doorModule | dumbSwitch<br>dumbMotor<br>doorModule | dumbSwitch<br>smartMotor | dumbSwitch<br>smartMotor | dumbSwitch<br>dumbMotor | dumbSwitch<br>dumbMotor |
| localSmartComponents 0..4 | doorModule | doorModule | smartMotor | smartMotor | | |
| smartSwitch ? = no ↴ | | | | | | |
| electronic ? = no | | | | | | |
| smart ? = no | | | | | | |
| deployedFrom 0..* = yes | | | | | | |
| dumbSwitch ? = yes ↴ | | | | | | |
| electronic ? = no | | | | | | |
| smart ? = no | | | | | | |
| deployedFrom 0..* | Switch | Switch | Switch | Switch | WinController<br>Switch<br>CurrentSensor | Switch |
| smartMotor ? ↴ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ |
| electronic ? | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ |
| smart ? | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ |
| deployedFrom 0..* | | | WinController<br>Motor<br>CurrentSensor<br>OtherRemoteArbitrator | WinController<br>Motor<br>CurrentSensor | | |
| dumbMotor ? ↴ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ |
| electronic ? = no | | | | | | |
| smart ? = no | | | | | | |
| deployedFrom 0..* | Motor | Motor | | | Motor | WinController<br>Motor<br>CurrentSensor |
| powerFuse ? = yes – | | | | | | |
| busLength = 0 | | | | | | |
| busMass = 0 | | | | | | |
| busType ? = no – | | | | | | |
| dist = – | | | | | | |
| driverDeploymentAndWiring = – | | | | | | |
| quality = ↴ | | | | | | |
| numberOfSmartComponents | 1 | 1 | 1 | 1 | 0 | 0 |
| wireLength | 220 | 220 | 250 | 250 | 125 | 125 |
| wireMass | 850 | 850 | 760 | 760 | 875 | 875 |

Figure 6.13: ClaferMooVisualizer: PW driver subsystem optimization

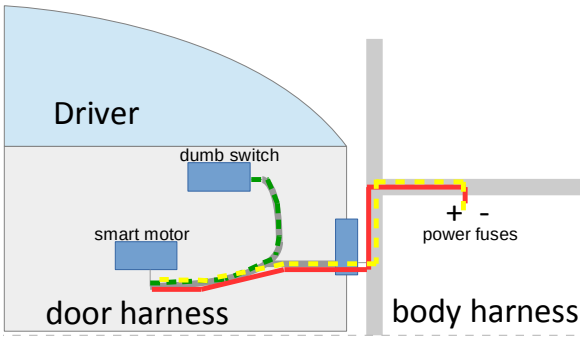
*Objectives* view and *Parallel Cordinates* chart reflect the ranges of quality: we can see that the number of smart components range from 0 to 1, the wire length takes only three values: 125, 220 and 250. Wire mass also takes three values ranging from 760 to 875.

*Feature and Quality Matrix* illustrates each design in terms of components and connectors. We collapsed non-interesting clafers or the ones that have redundant information: i.e., we do not show functional architecture clafers since we can see deployment by looking the hardware clafers.

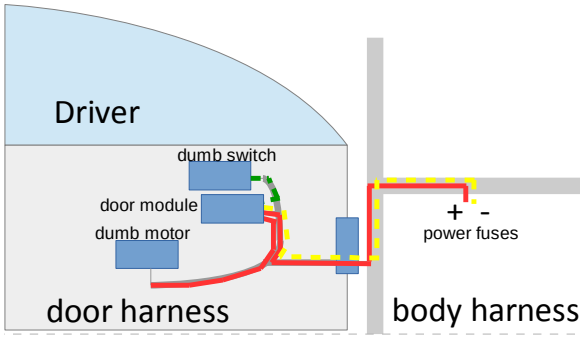
In spite of minimization of the number of smart components, the tool still produces 4 designs with a smart component. Now design exploration becomes very important: without doing exploration we do not have any insights on why this happens and why wiring length and the number of smart components are non-dominated in some cases.




(a) Design 1




(b) Design 2




(c) Design 3

Figure 6.14: Instance cluster representative designs for PW driver subsystem optimization. Load power wire connector is a solid red line, device power connector is a dashed yellow line, discrete wire connector is a dashed green line. Power wire connectors go to the power supply fuse located in the body harness

### 6.7.4 Performing Exploration

The next step is to understand what are the three clusters, we need to perform exploration. We use *Parallel Coordinates Chart* to brush an axis area we want to consider. The views get filtered, and it is easy to select the two designs. Figure 6.15 shows the filtered views.

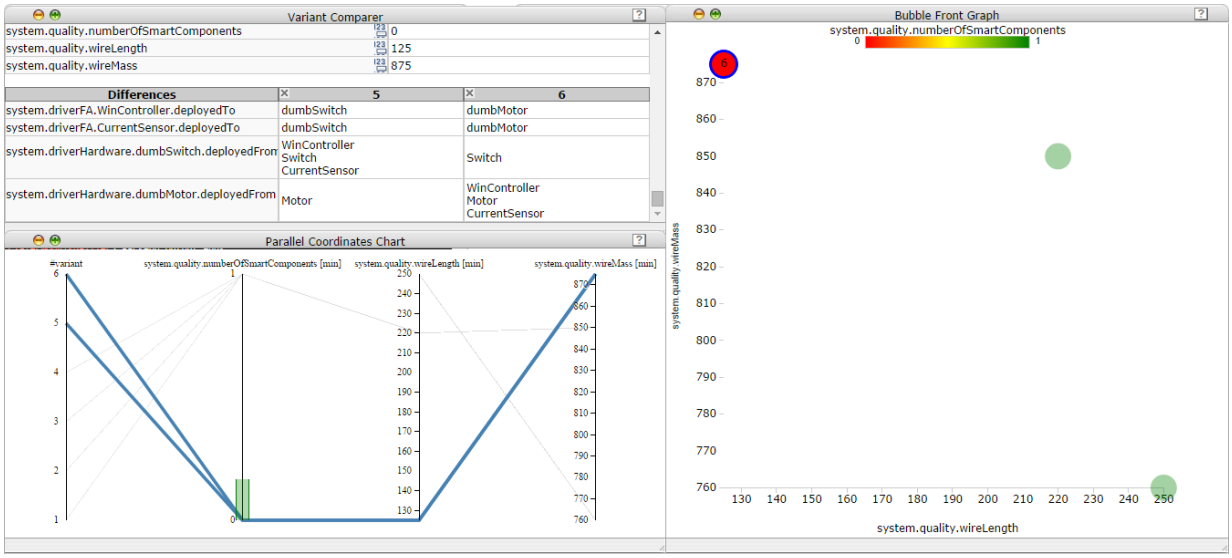



Figure 6.15: ClaferMooVisualizer: PW driver subsystem optimization. Designs 5 and 6 are selected

First, we choose instances without any smart component. By examining *Feature and Quality Matrix*, we identify that both designs have dumb switches and dumb motors, and there is a power wire `wireCommand` that connects a dumb switch to a dumb motor. There is also a load power supply coming into the dumb switch from a power fuse to make the switch powered and to make it able to supply power to the motor.

The difference between the two, as we can see front the *Variant Comparer* view, is deployment only. In the instance 5, `CurrentSensor` and `WinController` are deployed on the `dumbSwitch`. In the instance 6, `CurrentSensor` and `WinController` are deployed on the `dumdMotor`. Thesea are exactly the same as the basic electric designs depicted on Fig. 6.4 and Fig. 6.5, but also with power supply and wire length. Clafer tools generated these designs by performing optimization. Figure 6.14a illustrates the instance cluster.

The second cluster of designs (variants 3 and 4) is illustrated on Figure 6.16. Both has one smart component — `smartMotor`. The switch is connected to the motor via a discrete

wire. The smart motor drives itself and gets a load power from the fuse. The motor also includes all the control functionality: `WinController` is deployed on the motor.

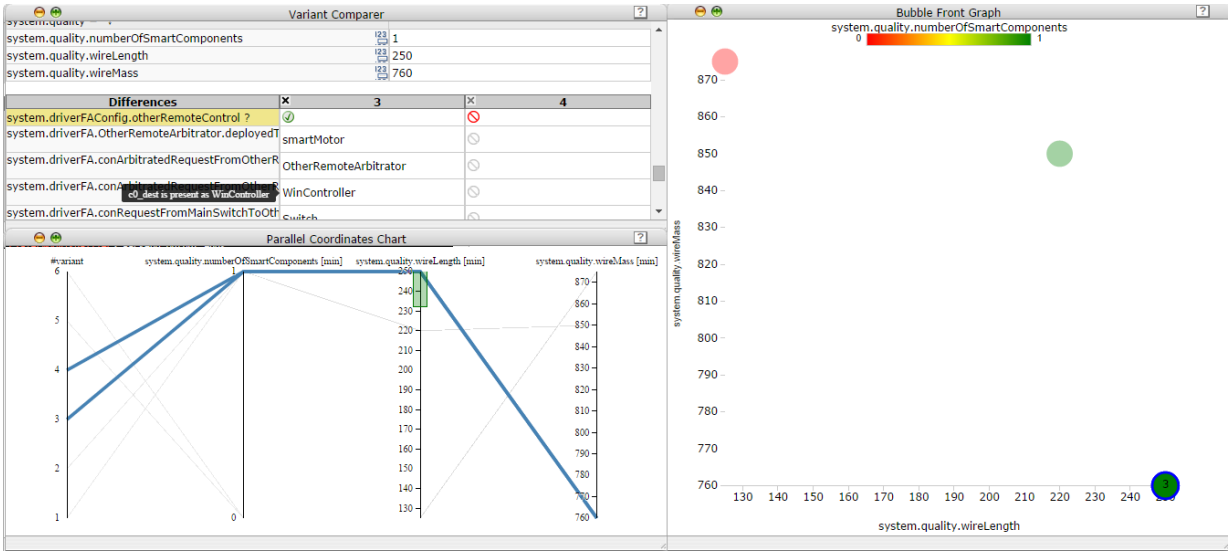



Figure 6.16: ClaferMooVisualizer: PW driver subsystem optimization. Designs 3 and 4 are selected

All the differences between the two designs come from presence of `otherRemoteControl`. If it is present, we have `OtherRemoteArbitrator` deployed on the motor, and our wire is called `wireRequestIndirect`. If remote control is not supported, we just have a wire called `wireRequestDirect`, which is equivalent to `wireRequestIndirect` in terms of length and mass. Figure 6.14b illustrates the instance cluster. Unfortunately, the tool shows implied differences as well; thus, the `src` and `dest` values of connections are also shown and distract attention.

Figure 6.17 shows the third cluster of designs (instances 1 and 2). Both has one smart component — `doorModule`. The switch is connected to the door module via a discrete wire. The dumb motor is driven by the doorModule that gets a load power from the fuse. The door module also includes all the control functionality: `WinController` is deployed on it.

As in the previous case, the difference between the designs is *other remote control* functionality. If it is present, it is deployed on the door module. Figure 6.14c illustrates the instance cluster.

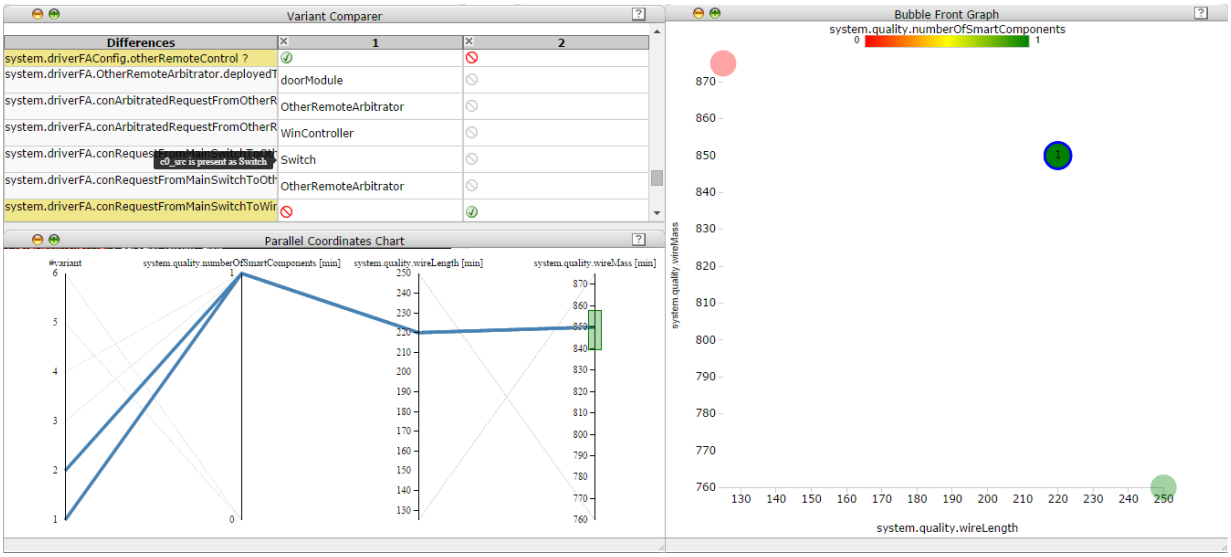



Figure 6.17: ClaferMooVisualizer: PW driver subsystem optimization. Designs 1 and 2 are selected

## 6.8 Transition to Multiple Subsystems

To show extensibility of our modeling approach, we show a transition from one window subsystem to a system of two window subsystems.

### 6.8.1 Vehicle Level: Features

Passenger subsystem features are not different from the driver subsystem features. However, some of them become restricted depending on driver subsystem configuration. For example, the passenger window does support express-up, then the driver window should also be express-up as driver subsystem is always at least as rich in terms of features as other subsystems. We can specify these constraints using normal implications:

```
[passengerFAConfig.express => driverFAConfig.express]
[passengerFAConfig.express.expressUp => driverFAConfig.express.expressUp]
```

### 6.8.2 Analysis Level: Functional Architecture

Front passenger window subsystem is a little bit more complex that driver window subsystem. It should have a function that combines requests from the local (passenger) switch and the driver. Driver can close passenger windows too, therefore, we need a connection from the master switch to that function. We name this function `SwitchArbitrator`. Figure 6.18 shows these differences.

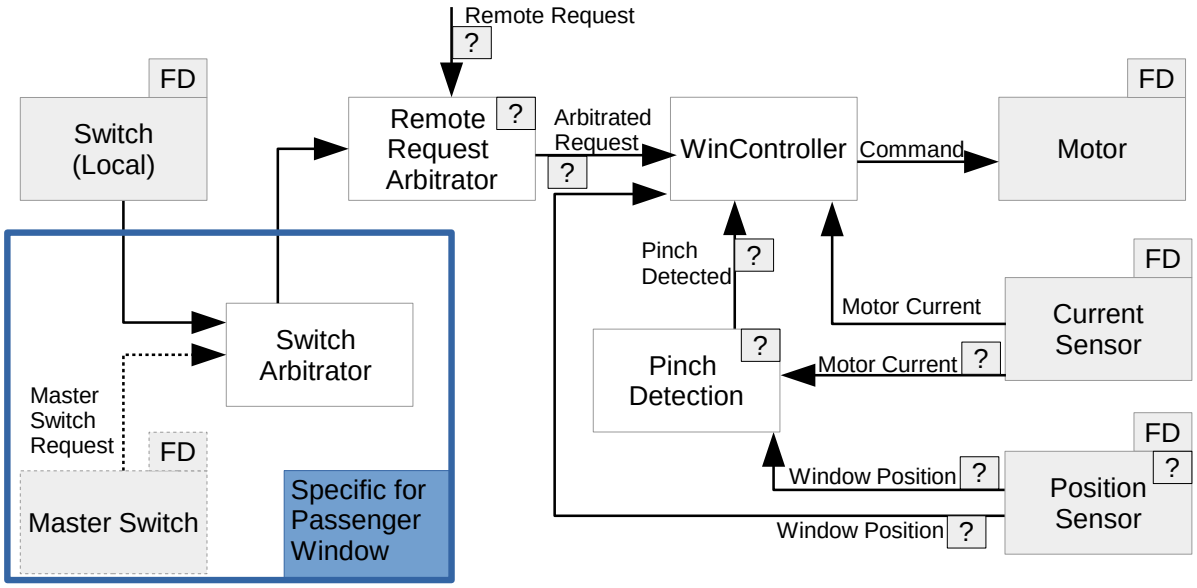



Figure 6.18: Front passenger subsystem architecture. Many components are common. The specific part is framed in blue. Master switch does not reside in the subsystem: it is a reference to the master switch

We implement this extension by creating two subtypes for each window and subtyping it from the common functional architecture part.

```
abstract DPWSubsystemFunctionalArchitecture : PWSubsystemFunctionalArchitecture
  ... (driver specific connectors)
abstract FPPWSubsystemFunctionalArchitecture : PWSubsystemFunctionalArchitecture
  MasterSwitch -> FunctionalDevice
  SwitchArbitrator : AnalysisFunction
  ... (passenger specific connectors)
```

### 6.8.3 Design Level: Hardware

Hardware can also be modeled via subtyping with extention. We need `masterSwitch` reference to the master switch device, as well as an optional junction component. We discussed this in the Principles chapter's Sec. 3.1, and Figure 3.5b reflects this extension.

### 6.8.4 Design Level: Deployment and Wiring

This `SwitchArbitrator` function can be deployed in various places:

1. On a passenger switch (dumb or smart). The request from the master switch has to come to the passenger switch in some way (a wire or a bus)
2. On a junction. Power wires from the master switch and the passenger switch can be joined into one.
3. On any smart device other than switch. The preference can be a smart motor or a door module.

As in previous models, we create subtypes of `PWDeploymentAndWiring` and model each of the cases above carefully with all the proper connections. We do not show these deployment and wiring rules in our thesis. However, we do show a modeling problem that arises.

When we define a subtype `FPPWDeploymentAndWiring : PWDeploymentAndWiring`, the properties `fa` (Functional Architecture) and `ht` (hardware topology) get inherited. However, we cannot access `fa.SwitchArbitrator` and `ht.switchRequestJunction` since the generic types `PWFunctionalArchitecture` and `PWHardwareTopology` do not have these properties. Therefore, we apply the typecasting pattern described in Sec. 4.4 twice:

```
fpfa -> FPPWSubsystemFunctionalArchitecture
[fpfa.ref = fa.ref] // typecast fa to fpfa

fpht -> FPPWSubsystemHardware
[fpht.ref = ht.ref] // typecast ht to fpht
```

Now we can access the properties that were inaccessible by stating `fpfa.SwitchArbitrator` and `fpht.switchRequestJunction` .

### 6.8.5 Integration

We nest our newly created clafers under the `PowerWindowSystem` as in Section 6.4. We also change the types to the newly created. The complete assembly looks as follows:

```
abstract PowerWindowSystem
  BCM : ECU ?

  driverFAConfig : PWSubsystemConfig
  driverFA : DPWSubsystemFunctionalArchitecture
    [config = driverFAConfig]

  frontPassengerFAConfig : PWSubsystemConfig
  frontPassengerFA : FPPWSubsystemFunctionalArchitecture
    [config = frontPassengerFAConfig]
    [MasterSwitch.ref = driverFA.Switch]

  driverHardware: PWSubsystemHardware
    [bcm = BCM]
    [dist.inlineToBCM = Dist.inlineDtoBCM]
    [dist.inlineToFuse = Dist.inlineDtoPowerFuse]
  frontPassengerHardware: FPPWSubsystemHardware
    [bcm = BCM]
    [masterSwitch.ref = driverHardware.switch.ref]
    [dist.inlineToBCM = Dist.inlineFPtoBCM]
    [dist.inlineToFuse = Dist.inlineFPtoPowerFuse]

  driverDeploymentAndWiring : DPWSubsystemDeploymentAndWiring
    [ht = driverHardware]
    [fa = driverFA]

  frontPassengerDeploymentAndWiring : FPPWSubsystemDeploymentAndWiring
    [ht = frontPassengerHardware]
    [fa = frontPassengerFA]

  allSmartComponents -> Device 0..8
  [allSmartComponents.ref = driverHardware.localSmartComponents.ref
     ++ frontPassengerHardware.localSmartComponents.ref]
```

We also adjust our quality metrics to make them take the front passenger subsystem into account:

```
quality
  numberOfSmartComponents ->> integer = #allSmartComponents
  wireLength ->> integer = sum WireConnector.length +
    driverHardware.busLength + frontPassengerHardware.busLength
  wireMass ->> integer = sum WireConnector.mass +
    driverHardware.busMass + frontPassengerHardware.busMass
```

### 6.8.6 Optimization-Based Design Exploration

We go to the optimization-based exploration right a way. We try unconstrained optimization first, and then switch to the constrained one.

#### Without Constraints

For the unconstrained optimization, we do not constrain the design: the tool should generate the entire Pareto front. We do that as follows:

```
system : PowerWindowSystem
<<min system.quality.numberOfSmartComponents >>
<<min system.quality.wireLength >>
<<min system.quality.wireMass >>
```

Unfortunately, ClaferMooVisualizer cannot produce any instance within a reasonable time. The reason is that the design space is big, and the solver has to consider many possible combinations which is hard to do. Thus, we ask the solver to follow the right direction by adding constraints.

#### With Constraints

Now we add constraints and partially specify the design we are interested in. We want a configuration with driver express-up, but front passenger just express-down. Both should not have a remote control. Both designs should have smart motors, BCM and a bus that connects all the smart components together. BCM should be used for communication only,

and we deny any deployment on BCM to avoid many isomorphic instances in terms of structure. We also disable switch request junction to reduce the design space. The complete configuration looks as follows below:

```
system : PowerWindowSystem
  [driverFAConfig.express.expressUp]
  [frontPassengerFAConfig.express]
  [no frontPassengerFAConfig.express.expressUp]
  [no driverFAConfig.otherRemoteControl]
  [no frontPassengerFAConfig.otherRemoteControl]

  [driverHardware.smartMotor]
  [frontPassengerHardware.smartMotor]
  [driverHardware.busType.branchToAllSmartComponents]
  [frontPassengerHardware.busType.branchToAllSmartComponents]
  [no frontPassengerHardware.switchRequestJunction]

  [BCM]
  [no BCM.deployedFrom]
```

This time, ClaferMooVisualizer generates **5** optimal designs relatively fast. The instances are separated into two clusers. One consists of a single instance only. Another one has four isomorphic instances in terms of structure and differ by deployment only. Figure 6.19 is a tool screenshot for this problem.

The first cluster — a red circle — is represented by a solely instance 3 smart components looks as on Figure 6.20a. The three smart devices are the driver smart motor, the front passenger smart motor and the BCM. Both switches are dumb. There are no door modules. The bus connects all the three smart devices together. For the master switch, there are two discrete wire coming out: one goes to the smart motor to control it, another one goes to the BCM for controlling the passenger motor. The BCM then sends the signals to the front passenger motor via the bus. The passenger switch has also a discrete wire to control its motor. Both smart motors get the load power from the power fuse and drive on their own.

The second cluster — a green circle — represented by 4 isomorphic instances and looks as on Figure 6.20b. It is almost the same as the previous one, but the master switch is smart. Now we do not need discrete wires from it, but we connect it to the bus. The wiring cost decreases, because we have only one wire to the bus instead of having two discrete

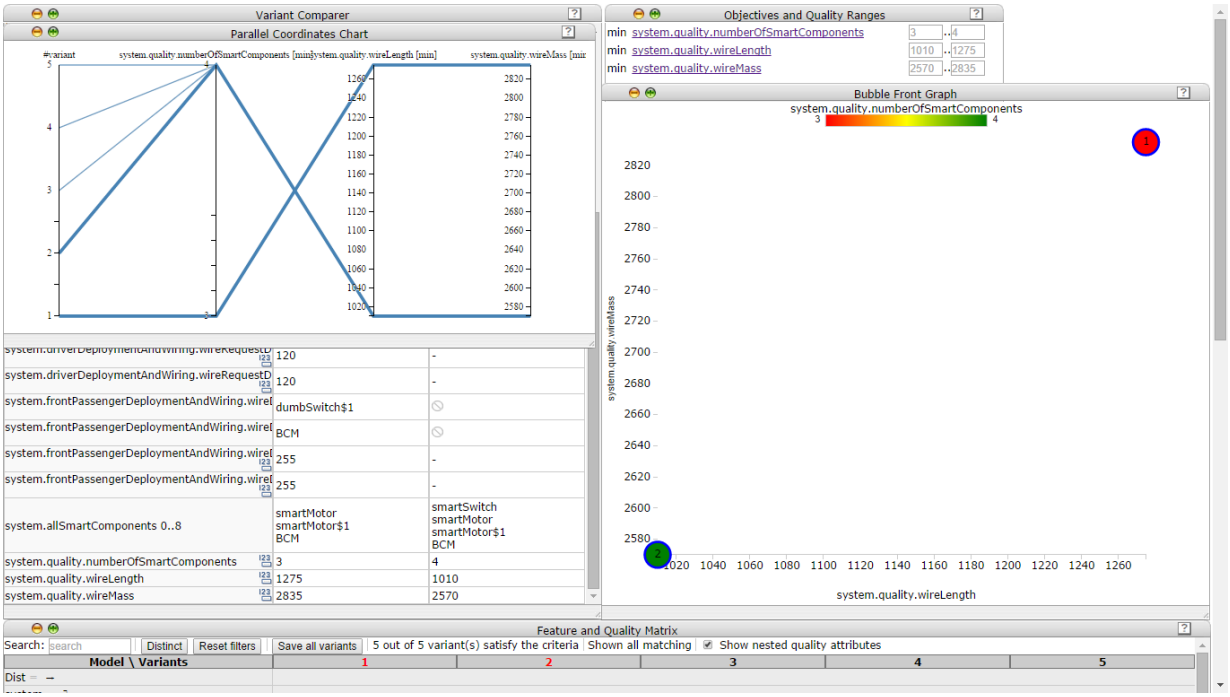



Figure 6.19: Complete PW system design visualization and exploration — a screenshot from ClaferMooVisualizer tool

wires to the motor and the bcm. Thus, by adding a smart device — turning the master switch into the smart master switch — we can save wiring cost.

The optimization backend did not generate any instance with the number of smart devices greater than 4: they are not necessary and cannot reduce the wiring cost anymore. Adding a smart component requires a device power from the fuse which negatively affects the wiring length. The tool still decided to make the master switch smart because it is located closer to the power supply and has two discrete wires, therefore, making it smart actually reduces the overall wiring length.

## 6.9 Conclusions

Performing the Power Window (PW) case study was a very comprehensive evaluation of our approach. In this section, we list our experience summary, lessons learned and limitations for each of the following criteria.

### 6.9.1 Domain Modeling

We successfully created meta-models for hardware and functional architectures. We demonstrated that relatively small metamodels combined with expressive power of Clafer is sufficient to create a model of the two-window PW system.

We successfully modeled function connectors with quality attributes — length, thickness and mass. Clafer does not support real numbers, thus we had to use scaling techniques. However, the techniques worked in our example with thickness and reflected the real semantics of thickness.

We successfully modeled variability at all EAST-ADL abstraction levels — vehicle, analysis and design levels. Also, variability were supported at all of the device, subsystem and system levels. Clafer was very intuitive when representing variability.

During modeling, bottom-up development approach (Sec. 5.1) reduced debugging time and improved model comprehension. First, by successfully testing individual parts and then integrating, we did not have any debugging problems described in Sec. 5.1. Second, we identified the two-window optimization scalability problem and resolved it relatively fast by adding constraints: we could easily turn some model parts off, which we would not able to do if started with a huge model. And third, we used inheritance with configuration thus thinking in object-oriented way.

### 6.9.2 Modeling Wiring and Deployment Problems

Clafer was expressive enough to represent all deployment and wiring constraints. We successfully applied the collaboration pattern (Sec. 5.2) for these two problems. In terms of modeling deployment and wiring constraints, Clafer lacks a comprehensive branching structure: we had to model conditions via a set of implications. The built-in if-then-else was not easy to use because of its unintuitive syntax and type-checking.

### 6.9.3 Reasoning Performance

With regards to instance generation performance: the process was instantaneous for both driver subsystem and the two-window system. Both constrained and unconstrained configurations were instantaneous.

With regards to optimization performance, we observed a perfect (instantaneous) reasoning performance for the driver subsystem. For the two-window system, unconstrained

optimization was slow and did not finish within the reasonable time. However, a reasonably constrained optimization was instantaneous for our experiments. Experiments can be made to analyze performance in more details.

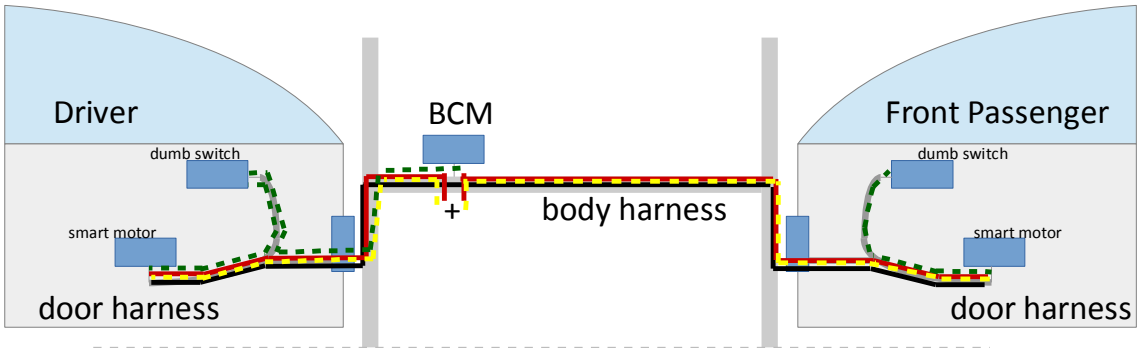
### 6.9.4 Design Generation

Clafer backends successfully generated correct designs from our specifications. Designs from both constraint-based generation and optimization-based instance generation in our experiments were satisfactory. However, we did not perform detailed testing, therefore, we cannot guarantee full correctness and validity of all our constraints. Moreover, our domain knowledge is limited and we are not experts in the electrical engineering area.

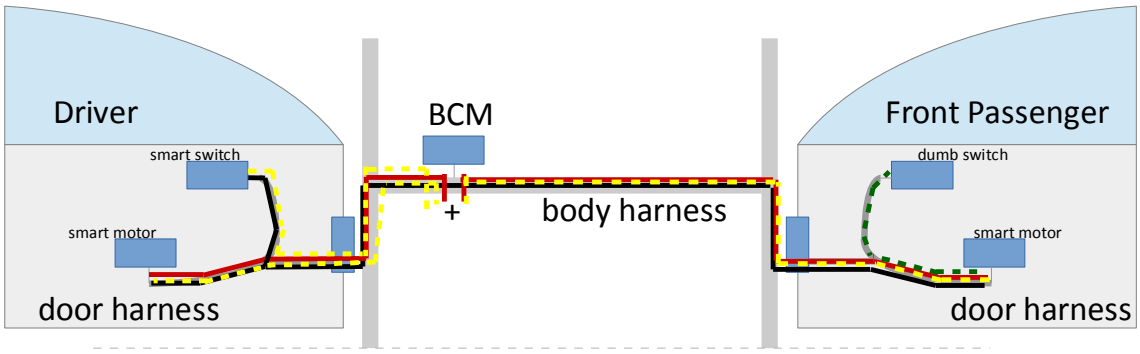
### 6.9.5 Visualization and Exploration

We were able to visualize small design spaces (up to 15 designs) and compare selected designs to each other using only the standard built-in visualizations of ClaferMooVisualizer and ClaferConfigurator. Three-dimensional visualization of Bubble Front Graph was intuitive and comprehensive. Multi-dimensional Parallel Coordinates Chart was very useful for observing quality ranges and filtering. Feature and Quality Matrix, in contrast, was less intuitive: it shows a lot of irrelevant clafers and does not allow easy comparison. Variant Comparer was useful for highlighting design differences, however, commonalities were harder to observe because of their big number.

The visualizations, however, were very generic. For thesis illustrations, we made various design diagrams manually. The tools could generate such diagrams automatically by supporting custom, problem-specific visualizations.




(a) Design 1




(b) Design 2

Figure 6.20: Representatives of two clusters of instances for the complete PW system optimization. Power wire connector is a solid red line, device power connector is a dashed yellow line, discrete wire connector is a dashed green line. Power wire connectors go to the power supply fuse located in the body harness close to BCM. Bus is a black solid line

# Chapter 7

# Conclusions, Limitations, Threats to Validity and Future Work

## 7.1 General Conclusions

We formulated architecture modeling principles and patterns in Clafer. We defined and demonstrated approaches of meta-modeling, subsystem and system domain modeling. We defined two Macro-Level patterns and four Micro-Level patterns that we successfully applied in the architecture modeling domain. We demonstrated extensibility of our modeling approach by showing a transition from modeling one-window system (driver only) to two-window system (driver and front passenger). We believe the modeling principles and patterns can be used outside the architecture domain.

We successfully applied Clafer and Clafer tools for two problems in architecture modeling. First, we modeled an automated hardware topology generation. Clafer tools were able to generate valid hardware topologies with wire connectors. Second, we formulated a function-to-hardware deployment problem, and Clafer tools were able to generate correct designs with function deployment. In our approach we combined the two problems together to generate complete designs with deployment.

We formulated and performed multi-objective optimization with respect to three minimization objectives — the number of smart components, wire length and wire mass. We demonstrated the ways of design visualization and exploration using ClaferMooVisualizer and ClaferConfigurator.

As a secondary contribution, we evaluated Clafer backends and visualization tools and

proposed extensions to the tools and the language itself.

## 7.2 Limitations

Scalability in terms of reasoning performance is the most significant limitation of our approach. We start having performance problems when modeling two-window Power Window systems. Also, our performance tests made for query modeling methods demonstrated performance decrease with increasing number of subsystems. Currently, in Clafer all reasoning and optimizations approaches are exact, therefore, they have scalability problems because of the exponential complexity.

We do not foresee other significant limitations of our approach.

## 7.3 Threats to Validity

For our case study, we have two threats to validity. First, we lack full-coverage tests for our case study models. We did not perform detailed testing, so there is no guarantee of full correctness and validity of our case study model. However, our approach is feasible with regards to the problems we considered (deployment and topology generation), and resulting designs we received from the tools are plausible. Second, we have a limited domain knowledge on the automotive systems and may not foresee some deployment or wiring rules. However, this is not a big threat since our case study is extensible in terms of constraints, and can easily be modified to accommodate new requirements.

The main threat to validity of our entire research is a bias towards the current implementation of Clafer tools, the release **0.3.6.1**. Clafer language and tools are gradually extended to support new features. First, the features include the ones that are present in Clafer semantics [13] but not implemented in the tools yet — such as, redefinition. Second, there are features that are theoretically supported by the solvers and useful in general, but not implemented yet — such as, real numbers. Our work is fully valid for the current Clafer release, however, for the next versions of Clafer, some of our conclusions on performance reasoning and modeling approaches may become invalid.

## 7.4 Future Work

There is a room for a future work can be done towards both Clafer modeling principles and the Power Window case study. With regards to modeling approaches and principles, new domains and new applications of Clafer can be explored. Some of our formulated modeling principles and patterns may be evaluated in other technical domains (e.g., aerospace domain).

With regards to the case study, it can be extended to support four-window Power Window system configurations and evaluated on these configurations. Moreover, the case study can be the basis for the future works in behavioral modeling, such as, latency analysis.

# Appendix A

# Full Source Codes of Models

Source codes in Clafer for the Power Window Case Study can be found online:

https://github.com/gsdlab/ClaferCaseStudies/tree/master/PowerWindow.Thesis

## A.1 Power Window Case Study: Driver Only Model

```
//=======================
/* Distance Data */
//=======================

Dist
  switchToMotor : integer = 40
  switchToDoorModule : integer = 20
  motorToDoorModule : integer = 30
  inlineToSwitch : integer = 45
  inlineToMotor : integer = 45
  inlineToDoorModule : integer = 35

  doorSpliceToSwitch : integer = 25
  doorSpliceToMotor : integer = 25
  doorSpliceToDoorModule : integer = 15

  inlineDtoInlineFP : integer = 170
```

```
  inlineDtoBCM : integer = 40
  inlineFPtoBCM : integer = 130

  /* for power supply */
  inlineDtoPowerFuse : integer = 40
  inlineFPtoPowerFuse : integer = 130

abstract PWSubsystemConfig
  basicUpDown  // mandatory feature
  express ?    // at least express down
    expressUp ?  // both express up and express down
  otherRemoteControl ?

abstract Device
  electronic ?
  smart ?
  [smart => electronic]
  deployedFrom -> AnalysisFunction *
    [this.deployedTo = parent]

abstract AnalysisFunction
  deployedTo -> Device
    [parent in this.deployedFrom]

abstract FunctionalDevice : AnalysisFunction

abstract AFConnector
  src -> AnalysisFunction
  dest -> AnalysisFunction

abstract ECU : Device
  [smart]

abstract PWSubsystemFunctionalArchitecture
  WinController : AnalysisFunction
  Motor : FunctionalDevice
  Switch : FunctionalDevice
  CurrentSensor : FunctionalDevice
```

```
PositionSensor : FunctionalDevice ?
PinchDetection : AnalysisFunction?

OtherRemoteArbitrator : AnalysisFunction ?

conCommand : AFConnector
  [src = WinController]
  [dest = Motor]

conCurrent : AFConnector
  [src = CurrentSensor]
  [dest = WinController]

conArbitratedRequestFromOtherRemoteArbitratorToWinController : AFConnector ?
  [src = OtherRemoteArbitrator]
  [dest = WinController]
conRequestFromSwitchToOtherRemoteArbitrator : AFConnector ?
  [src = Switch]
  [dest = OtherRemoteArbitrator]
conRequestFromSwitchToWinController : AFConnector ?
  [src = Switch]
  [dest = WinController]

pinchDetectionConnections ?
  conPositionToWinController : AFConnector
    [src = PositionSensor]
    [dest = WinController]
  conPositionToPinchDetection : AFConnector
    [src = PositionSensor]
    [dest = PinchDetection]
  conCurrentToPinchDetection : AFConnector
    [src = CurrentSensor]
    [dest = PinchDetection]
  conPinchDetectionToWinController : AFConnector
    [src = PinchDetection]
    [dest = WinController]
```

```
  config -> PWSubsystemConfig

  [config.otherRemoteControl <=> OtherRemoteArbitrator]
  [config.otherRemoteControl <=>
      conRequestFromSwitchToOtherRemoteArbitrator]
  [config.otherRemoteControl <=>
      conArbitratedRequestFromOtherRemoteArbitratorToWinController]

  [no config.otherRemoteControl <=> conRequestFromSwitchToWinController]

  [config.express.expressUp <=> PinchDetection]
  [config.express.expressUp <=> PositionSensor]
  [config.express.expressUp <=> pinchDetectionConnections]

abstract PWSubsystemHardware

  doorModule : ECU ?
  switch -> Device
  motor -> Device
  bcm -> ECU ?

  localComponents -> Device 0..4
  [localComponents = switch.ref, motor, bcm.ref, doorModule]
  localSmartComponents -> Device 0..4
  [localSmartComponents =  smartSwitch, smartMotor, bcm.ref, doorModule]

  smartSwitch : Device ?
    [smart]
    [switch = this]
  dumbSwitch : Device ?
    [no smart]
    [switch = this]
  [smartSwitch xor dumbSwitch]

  smartMotor : Device ?
    [smart]
    [motor = this]
  dumbMotor : Device ?
```

```
  [no smart]
  [motor = this]
[smartMotor xor dumbMotor]

powerFuse : Device ?
  [no electronic]
  [no smart]

/* Bus */
[busType <=> (#localSmartComponents > 1)]
[no busType => (busLength = 0)]
busLength ->> integer
busMass ->> integer
[busMass = busLength]
xor busType ?
  branchToDoorModule
    [bcm && doorModule]

    [(smartSwitch && smartMotor) =>
      (busLength = dist.inlineToBCM + Dist.inlineToDoorModule
          + Dist.motorToDoorModule + Dist.switchToDoorModule)]

    [(smartSwitch && !smartMotor) =>
      (busLength = dist.inlineToBCM + Dist.inlineToDoorModule
          + Dist.switchToDoorModule) ]

    [(!smartSwitch && smartMotor) =>
      (busLength = dist.inlineToBCM + Dist.inlineToDoorModule
          + Dist.motorToDoorModule) ]

    [(!smartSwitch && !smartMotor) =>
      (busLength = dist.inlineToBCM + Dist.inlineToDoorModule) ]

  branchToSwitch
    [bcm && smartSwitch && !doorModule]

    [(smartMotor) =>
      (busLength = dist.inlineToBCM + Dist.inlineToSwitch
```

```
        + Dist.switchToMotor)]

    [(!smartMotor) =>
      (busLength = dist.inlineToBCM + Dist.inlineToSwitch) ]

  branchToAllSmartComponents
    [bcm]
    [(no smartSwitch && no smartMotor && doorModule) =>
      busLength = dist.inlineToBCM + Dist.inlineToDoorModule]

    [(no smartSwitch && smartMotor && no doorModule) =>
      busLength = dist.inlineToBCM + Dist.inlineToMotor]
    [(no smartSwitch && smartMotor && doorModule) =>
      busLength = dist.inlineToBCM + Dist.inlineToMotor
      + Dist.doorSpliceToDoorModule]

    [(smartSwitch && no smartMotor && no doorModule) =>
      busLength = dist.inlineToBCM + Dist.inlineToSwitch]
    [(smartSwitch && no smartMotor && doorModule) =>
      busLength = dist.inlineToBCM + Dist.inlineToSwitch]

    [(smartSwitch && smartMotor && no doorModule) =>
      busLength = dist.inlineToBCM + Dist.inlineToSwitch
       + Dist.doorSpliceToMotor]
    [(smartSwitch && smartMotor && doorModule) =>
      busLength = dist.inlineToBCM + Dist.inlineToSwitch
       + Dist.doorSpliceToMotor]

  localBusOnly
    [no bcm]
    [(smartSwitch && smartMotor) =>
      busLength = Dist.switchToMotor]
    [(doorModule && smartSwitch && !smartMotor) =>
      busLength = Dist.switchToDoorModule]
    [(doorModule && smartMotor && !smartSwitch) =>
      busLength = Dist.motorToDoorModule]

dist
```

```
    inlineToBCM ->> integer = Dist.inlineDtoBCM
    inlineToFuse ->> integer = Dist.inlineDtoPowerFuse

abstract PWSubsystemDeploymentAndWiring
  ht -> PWSubsystemHardware
  fa -> PWSubsystemFunctionalArchitecture

  [fa.Switch.deployedTo.ref = ht.switch.ref] // to the local switch only
  [fa.Motor.deployedTo.ref  = ht.motor.ref ] // to the motor device only

  // Motor Driver
  motorDriver -> Device
  [motorDriver.ref in ht.localComponents.ref]
  // MotorDriver can be any of {BCM, switch, motor, doorModule}

  [(ht.doorModule && ht.bcm) => (motorDriver.ref != ht.bcm.ref)]
  // if we have a door module and BCM, then the motor driver is not BCM
  // ExpressUp: Position Sensor, Pinch Detection, WinCotnroller

  [fa.config.express.expressUp =>
    (
      (fa.PositionSensor.deployedTo.ref in ht.localComponents.ref) &&
      (fa.PinchDetection.deployedTo.ref in ht.localSmartComponents.ref) &&
      (fa.WinController.deployedTo.ref in ht.localSmartComponents.ref) &&
      ((fa.PositionSensor.deployedTo.ref != motorDriver.ref)
          => (fa.PositionSensor.deployedTo.ref = ht.motor.ref))
    )
  ]

  [fa.config.express => (fa.WinController.deployedTo.electronic)]

  [motorDriver.smart => (
      (fa.WinController.deployedTo.ref in ht.localSmartComponents.ref) &&
      (fa.CurrentSensor.deployedTo.ref = motorDriver.ref)
    )
  ]
  [!motorDriver.electronic => (
  // if the motor is driven by a dumb component (not even electronic)
```

```
  (
    (fa.WinController.deployedTo.ref = ht.motor.ref) &&
    (fa.CurrentSensor.deployedTo.ref = ht.motor.ref)
    // both WinController and CurrentSensor are on the motor (thermistor)
  )
  || // or
  (
    (fa.WinController.deployedTo.ref = ht.switch.ref) &&
    (fa.CurrentSensor.deployedTo.ref = ht.switch.ref)
    // both window controller and the current sensor are on the dumb switch
  )
)]

[(motorDriver.electronic && !motorDriver.smart) =>
  (fa.CurrentSensor.deployedTo.ref = motorDriver.ref) &&
  (fa.WinController.deployedTo.ref in ht.localComponents.ref)
]

[fa.OtherRemoteArbitrator =>
  (fa.WinController.deployedTo.ref != ht.motor.ref =>
      (fa.OtherRemoteArbitrator.deployedTo.ref
             = fa.WinController.deployedTo.ref)
  )
  &&
  (fa.OtherRemoteArbitrator.deployedTo.ref in ht.localSmartComponents.ref)
]

distFromMotorToMotorDriver ->> integer
[(motorDriver.ref = ht.switch.ref) =>
  (distFromMotorToMotorDriver = Dist.switchToMotor)]
[(motorDriver.ref = ht.motor.ref) =>
  (distFromMotorToMotorDriver = 0)]
[(motorDriver.ref = ht.doorModule) =>
  (distFromMotorToMotorDriver = Dist.motorToDoorModule)]
[(motorDriver.ref = ht.bcm.ref) =>
  (distFromMotorToMotorDriver = Dist.inlineToMotor + ht.dist.inlineToBCM)]

/* Command Wire */
```

```
wireCommand : PowerWireConnector ?
  [src = motorDriver.ref]
  [dest = ht.motor.ref]
  [length = distFromMotorToMotorDriver]

[(motorDriver.ref = ht.motor.ref) <=> ht.motor.smart]
// if the driver is not the motor, then we need a power wire for the command
// we are not making discrete connection to the motor
[ht.motor.smart <=> no wireCommand]

/* Position Wire */

wirePosition : AnalogWireConnector ?
  [src = ht.motor.ref]
  [dest = motorDriver.ref]
  [length = 2 * distFromMotorToMotorDriver]

[wirePosition <=> (fa.config.express.expressUp
  && (motorDriver.ref != ht.motor.ref) )]
// the wire is present if and only if we have express-up,
// and the motor driver is not on the motor
[(fa.config.express.expressUp && motorDriver.ref = ht.switch.ref) =>
  ht.switch.smart]

/* No Other Remote Control */

[!fa.config.otherRemoteControl =>
  ((fa.WinController.deployedTo.ref = ht.switch.ref)
  => no wireRequestDirect ) &&
  ((fa.WinController.deployedTo.ref != ht.switch.ref ) =>
    (
      ((ht.switch.ref != motorDriver.ref && !ht.switch.smart)
           => wireRequestDirect ) &&
      ((ht.switch.ref = motorDriver.ref) => no wireRequestDirect )
    ))
]

numberOfDiscreteWiresFromSwitch ->> integer
```

```
[numberOfDiscreteWiresFromSwitch = if fa.config.express then 3 else 2]

wireRequestDirect : DiscreteWireConnector ?
  [src = ht.switch.ref]
  [dest.ref in ht.localSmartComponents.ref]
  [!fa.config.otherRemoteControl]

  [(dest.ref = ht.motor.ref) =>
    (length = numberOfDiscreteWiresFromSwitch * Dist.switchToMotor)]
  [(dest.ref = ht.doorModule) =>
    (length = numberOfDiscreteWiresFromSwitch * Dist.switchToDoorModule)]
  [(dest.ref = ht.bcm.ref) =>
    (length = numberOfDiscreteWiresFromSwitch * (Dist.inlineToSwitch
                                          + ht.dist.inlineToBCM))]
/* Other Remote Control */

[fa.config.otherRemoteControl => (
    (!ht.switch.smart) <=> wireRequestIndirect)
]

wireRequestIndirect : DiscreteWireConnector ?
  [src.ref = ht.switch.ref]
  [dest.ref in ht.localSmartComponents.ref]
  // from switch to any other local smart component
  //{BCM, smartMotor, doorModule}
  [fa.config.otherRemoteControl]

  [(dest.ref = ht.motor.ref) =>
    (length = numberOfDiscreteWiresFromSwitch * Dist.switchToMotor)]
  [(dest.ref = ht.doorModule) =>
    (length = numberOfDiscreteWiresFromSwitch * Dist.switchToDoorModule)]
  [(dest.ref = ht.bcm.ref) =>
    (length = numberOfDiscreteWiresFromSwitch * (Dist.inlineToSwitch
                                          + ht.dist.inlineToBCM))]

////////////////////////////
// switch or motor being electronic
////////////////////////////
```

```
  [(!ht.switch.smart) =>
    // if the switch is not smart
    (ht.switch.electronic <=> (fa.config.express
        && fa.WinController.deployedTo.ref = ht.switch.ref))
    // it's electronic iff WnController is located to it, and we have express
  ]

  [(!ht.motor.smart) =>
    // if the motor is not smart
    (ht.motor.electronic <=> (
        // it's electronic iff WnController is located to it, and we have express
        // or we use hall sensor as a position sensor (it's placed on the motor)
        (fa.config.express && (fa.WinController.deployedTo.ref = ht.motor.ref))
        || (fa.config.express.expressUp &&
             (fa.PositionSensor.deployedTo.ref = ht.motor.ref))
      )
    )
  ]

////////////// Load Power Supply ////////////////////////
  // from the load power fuse, to the motor driver

  wireFromLoadPowerFuseToMotorDriver : PowerWireConnector
    [src.ref = ht.powerFuse]
    [dest.ref = motorDriver.ref]
    [(motorDriver.ref = ht.switch.ref) =>
       (length = ht.dist.inlineToFuse + Dist.inlineToSwitch)]
    // the door module is the switch
    [(motorDriver.ref = ht.motor.ref) =>
      (length = ht.dist.inlineToFuse + Dist.inlineToMotor)]
    // the motor driver is the motor itself
    [(motorDriver.ref = ht.doorModule) =>
      (length = ht.dist.inlineToFuse + Dist.inlineToDoorModule)]
    // the door module is the motor driver
    [(motorDriver.ref = ht.bcm) => (length = 0)]
    // BCM has it's own power supply very close to that
```

```
////////////// Device Power Supply ////////////////////////////////
  // from the device power fuse, to any electronic component

  [ht.switch.electronic <=> wireFromDevicePowerFuseToSwitch]
  wireFromDevicePowerFuseToSwitch: DevicePowerWireConnector ?
    [src.ref = ht.powerFuse]
    [dest.ref = ht.switch.ref]
    [length = ht.dist.inlineToFuse + Dist.inlineToSwitch]

  [ht.motor.electronic <=> wireFromDevicePowerFuseToMotor]
  wireFromDevicePowerFuseToMotor: DevicePowerWireConnector ?
    [src.ref = ht.powerFuse]
    [dest.ref = ht.motor.ref]
    [length = ht.dist.inlineToFuse + Dist.inlineToMotor]

  [ht.doorModule <=> wireFromDevicePowerFuseToDoorModule]
  wireFromDevicePowerFuseToDoorModule: DevicePowerWireConnector ?
    [src.ref = ht.powerFuse]
    [dest.ref = ht.doorModule]
    [length = ht.dist.inlineToFuse + Dist.inlineToDoorModule]

abstract WireConnector
  src -> Device
  dest -> Device
  [src.ref != dest.ref]
  length ->> integer                        // wire length
  thickness ->> integer                     // wire thickness
  mass ->> integer = length * thickness     // wire mass (thickness * length)

abstract DiscreteWireConnector : WireConnector// inherits WireConnector
  [thickness = 1]       // take as a base thickness

abstract AnalogWireConnector : WireConnector// inherits WireConnector
  [thickness = 1]       // take as a base thickness

abstract PowerWireConnector : WireConnector  // inherits WireConnector
  [thickness = 7]       // power wire is ~7 times thicker than the discrete one
```

```
abstract DevicePowerWireConnector : WireConnector  // inherits WireConnector
  [thickness = 1]       // devie power wire has the same thickness

abstract PowerWindowSystem
  BCM : ECU ?
  driverFAConfig : PWSubsystemConfig
  driverFA : PWSubsystemFunctionalArchitecture
    [config = driverFAConfig]
  driverHardware: PWSubsystemHardware
    [bcm = BCM]
  driverDeploymentAndWiring : PWSubsystemDeploymentAndWiring
    [ht = driverHardware]
    [fa = driverFA]

  quality
    numberOfSmartComponents ->> integer =
       #driverHardware.localSmartComponents
    wireLength ->> integer = sum WireConnector.length
       + driverHardware.busLength
    wireMass ->> integer = sum WireConnector.mass
       + driverHardware.busMass

system : PowerWindowSystem
<<min system.quality.numberOfSmartComponents >>
<<min system.quality.wireLength >>
<<min system.quality.wireMass >>

/*
PureElectric : PowerWindowSystem
  [driverHardware.dumbSwitch]
  [driverHardware.dumbMotor]
  [no driverHardware.doorModule]
  [no BCM]
  [no driverFAConfig.express]
  [no driverFAConfig.otherRemoteControl]

SmartSwitchAndMotorAndBCM : PowerWindowSystem
  [driverFAConfig.express.expressUp]
```

```
  [driverFAConfig.otherRemoteControl]
  [driverHardware.smartMotor]
  [driverHardware.smartSwitch]
  [no driverHardware.doorModule]

  [BCM]
*/
```

## A.2 Query Performance Test

### A.2.1 Method 1: Quantifiers

```
abstract Device
  smart ?

abstract ECU : Device
  [smart]

abstract Subsystem
  switch : Device
  motor : Device

  bcm -> ECU ?
  doorModule : ECU

  localComponents -> Device 2..4
  [localComponents = switch, motor, doorModule, bcm.ref]
  localSmartComponents -> Device 0..4
  [all d : localComponents | (d in localSmartComponents) <=> d.smart]  // C2
  [no d : localSmartComponents | !(d in localComponents)]              // C3

System
  s1 : Subsystem
    [bcm = BCM]
  s2 : Subsystem
    [bcm = BCM]
```

```
  s3 : Subsystem
    [bcm = BCM]
  s4 : Subsystem
    [bcm = BCM]
  ... and so on, s5, s6, ...
  BCM -> ECU ?
```

### A.2.2 Method 2: Instances

```
abstract Device
  smart ?

abstract ECU : Device
  [smart]

abstract Subsystem
  xor switch -> Device
    smartSwitch : Device
      [parent = this]
      [smart]
    dumbSwitch : Device
      [parent = this]
      [no smart]

  xor motor -> Device
    smartMotor : Device
      [parent = this]
      [smart]
    dumbMotor : Device
      [parent = this]
      [no smart]

  bcm -> ECU ?
  doorModule : ECU

  localComponents -> Device 2..4
  [localComponents = switch.ref, motor.ref, doorModule, bcm.ref]
```

```
  localSmartComponents -> Device 0..4
  [localSmartComponents = smartSwitch, smartMotor, doorModule, bcm.ref]

System
  s1 : Subsystem
    [bcm = BCM]
  s2 : Subsystem
    [bcm = BCM]
  s3 : Subsystem
    [bcm = BCM]
  s4 : Subsystem
    [bcm = BCM]
  ... and so on, s5, s6, ...

  BCM -> ECU ?
```

# References


[1] Clafer Compiler. https://github.com/gsdlab/clafer.

[2] ClaferChocoIG Project. https://github.com/gsdlab/claferchocoig.

[3] ClaferIG Project. https://github.com/gsdlab/claferIg.

[4] ClaferSMT Project. https://github.com/gsdlab/clafersmt.

[5] DOJO Visualization Toolkit. http://demos.dojotoolkit.org/demos/.

[6] GA Tech ASDL. http://www.asdl.gatech.edu/.

[7] Google Charts. https://developers.google.com/chart/.

[8] META Tool Suite. https://www.youtube.com/watch?v=yog270kMUhQ.

[9] RAVE Tool. http://www.rave.gatech.edu/gallery.shtml.

[10] Michał Antkiewicz, Kacper Bąk, Alexandr Murashkin, Rafael Olaechea, Jia Hui (Jimmy) Liang, and Krzysztof Czarnecki. Clafer tools for product line engineering. In *Proceedings of the 17th International Software Product Line Conference Co-located Workshops*, SPLC '13 Workshops, pages 130–135, New York, NY, USA, 2013. ACM.

[11] Michał Antkiewicz, Kacper Bąk, Dina Zayan, Krzysztof Czarnecki, Andrzej Wąsowski, and Zinovy Diskin. Example-driven modeling using clafer. In *First International Workshop on Model-driven Engineering By Example*, 2013.

[12] EAST-ADL Association. EAST-ADL domain model specification, version V2.1.12.

[13] Kacper Bąk. Modeling and analysis of software product line variability in clafer, 2013.

[14] Kacper Bąk, Krzysztof Czarnecki, and Andrzej Wasowski. Feature and meta-models in clafer: Mixed, specialized, and coupled. In *Proceedings of the Third International Conference on Software Language Engineering*, SLE'10, pages 102–122, Berlin, Heidelberg, 2011. Springer-Verlag.

[15] Kacper Bąk, Zinovy Diskin, Michał Antkiewicz, Krzysztof Czarnecki, and Andrzej Wąsowski. Partial instances via subclassing. In Martin Erwig, RichardF. Paige, and Eric Van Wyk, editors, *Software Language Engineering*, volume 8225 of *Lecture Notes in Computer Science*, pages 344–364. Springer International Publishing, 2013.

[16] X. Blasco, J. M. Herrero, J. Sanchis, and M. Martínez. A new graphical visualization of n-dimensional Pareto front for decision-making in multiobjective optimization. *Information Sciences*, 178(20), 2008.

[17] Hans Blom, Henrik Lnn, Frank Hagl, Yiannis Papadopoulos, Mark-Oliver Reiser, Carl-Johan Sjstedt, De-Jiu Chen, and Ramin Tavakoli Kolagari. EAST-ADL an architecture description language for automotive software-intensive systems. white paper, version M2.1.10.

[18] Manfred Broy. Challenges in automotive software engineering. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 33–42, New York, NY, USA, 2006. ACM.

[19] A.E. Bucovsky and D. Jesse & Mavris. Design Space Exploration for Boom Mitigation on a Quiet Supersonic Business Jet. http://arc.aiaa.org/doi/abs/10.2514/6.2003-6802.

[20] Bastian Florentz and Michaela Huhn. Embedded systems architecture: Evaluation and analysis. In *Proceedings of the Second International Conference on Quality of Software Architectures*, QoSA'06, pages 145–162, Berlin, Heidelberg, 2006. Springer-Verlag.

[21] Software Testing Fundamentals. Integration Testing. http://softwaretestingfundamentals.com/integration-testing/.

[22] Thomas Heurung and Stefan Walz. Designing and implementing architectures for distributed automotive e/e systems.

[23] Mavris D. Virasak J. & Schrage D.P. Kim, H.-S. Selection and design optimization of a high speed, highly maneuverable rotorcraft configuration. http://vtol.org/store/product/

selection-and-design-optimization-of-a-high-speed-highly-maneuverable-rotorcraft-c
cfm.

[24] Stefan Kugele and Gheorghe Pucea. Model-based optimization of automotive e/e-architectures. In *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis*, CSTVA 2014, pages 18–29, New York, NY, USA, 2014. ACM.

[25] J. Metzger L. S. Brandt, N. Krämer and U. Lindemann. Optimization approach for function partitioning in an automotive electric electronic system architecture. International Design Conference - Design 2012, 2013.

[26] Ralph Moritz, Tamara Ulrich, and Lothar Thiele. Evolutionary exploration of e/e-architectures in automotive design. In Diethard Klatte, Hans-Jakob Lüthi, and Karl Schmedders, editors, *Operations Research Proceedings 2011*, Operations Research Proceedings, pages 361–366. Springer Berlin Heidelberg, 2012.

[27] Alexandr Murashkin, Michał Antkiewicz, Derek Rayside, and Krzysztof Czarnecki. Visualization and exploration of optimal variants in product line engineering. In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, pages 111–115, New York, NY, USA, 2013. ACM.

[28] Alexander Pretschner, Manfred Broy, Ingolf H. Kruger, and Thomas Stauner. Software engineering for automotive systems: A roadmap. In *2007 Future of Software Engineering*, FOSE '07, pages 55–71, Washington, DC, USA, 2007. IEEE Computer Society.

[29] Andy Pryke, Sanaz Mostaghim, and Alireza Nazemi. Heatmap visualization of population based multi objective algorithms. In *Evolutionary Multi-Criterion Optimization*, vol. 4403, LNCS. 2007.

[30] Vladimir Rupanov, Christian Buckl, Ludger Fiege, Michael Armbruster, Alois Knoll, and Gernot Spiegelberg. Early safety evaluation of design decisions in e/e architecture according to iso 26262. In *Proceedings of the 3rd International ACM SIGSOFT Symposium on Architecting Critical Systems*, ISARCS '12, pages 1–10, New York, NY, USA, 2012. ACM.

[31] Maurice H. ter Beek, Alessandro Fantechi, Stefania Gnesi, and Franco Mazzanti. A collection of models of a bike-sharing case study, 2014. http://blog.inf.ed.ac.uk/quanticol/files/2014/05/TR-QC-07-2014.pdf.

[32] Tea Tušar and Bogdan Filipič. Visualizing 4D approximation sets of multiobjective optimizers with prosections. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, GECCO, 2011.

[33] Josiah T. VanderMey and Hassan J. Bukhari. Optimization and Design Space Exploration of a Supersonic Business Jet Planform. http://ocw.mit.edu/courses/engineering-systems-division/esd-77-multidisciplinary-system-design-optimization-spring-2010/projects/MITESD_77S10_paper03.pdf.

[34] Zaur; Barthels Andreas; Michel Hans-Ulrich; Stechele Walter; Herkersdorf Andreas Walla, Gregor; Molotnikov. A design space exploration framework for automotive embedded systems and their power management. 27th European Conference on Modelling and Simulation (ECMS 2013), 2013.