Synthesis and Exploration of Multi-Level, Multi-Perspective Architectures of Automotive Embedded Systems

by

Jordan Ross

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2016

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

In industry, evaluating candidate architectures of automotive embedded systems is routinely done during the design process. Today's engineers, however, are limited in the number of candidates that they are able to evaluate in order to find the optimal architectures. This limitation results from the difficulty in defining the candidates as it is a mostly manual process. In this work, we propose a way to synthesize multi-level, multi-perspective candidate architectures and to explore them across the different layers and perspectives. Using a reference model similar to the EAST-ADL domain model but with a focus on early design, we explore the candidate architectures for two case studies: an automotive power window system and the central door locking system. Further, we provide a comprehensive set of questions, based on the different layers and perspectives, that engineers can ask to synthesize only the candidates relevant to their task at hand. Finally, using the modeling language Clafer, which is supported by automated backend reasoners, we show that it is possible to synthesize and explore optimal candidate architectures for two highly configurable automotive subsystems.

Acknowledgements

I want to first and foremost thank my supervisor, **Krzysztof Czarnecki**, for giving me this wonderful opportunity to learn from him and deepen my understanding. I have learned more than I ever have these last couple of years because of him.

I also want to thank **Michał Antkiewicz** for all of the discussions and guidance he provided me these last couple of years, it was a pleasure.

Lastly, I want to thank my family for their patience as I embarked on this adventure.

Dedication

I dedicate this to my lovely wife Emily.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Automotive E/E (electrical and electronic) architectures are continuing to grow in size and complexity due to the increasing amount of software functions being deployed to cars. These architectures consist of sensors, actuators, electronic control units (ECUs), and the physical connection media such as discrete signal wires, analog signal wires, and digital network buses. Premium cars can have 3 km of wiring connecting more than 70 ECUs with at least 2,000 software functions running on them [17, 31]. Additionally, the architectures contain large amounts of variability in features, functions, devices, and the deployment, which further increases the complexity.

EAST-ADL is a reference architecture model and a set of domain-specific languages for modeling automotive E/E architectures. It was designed to capture E/E systems with sufficient detail to allow modeling for documentation, design, analysis, and synthesis [25]. Figure 1.1 shows the domain model for EAST-ADL and how the system model is composed of multiple abstraction levels which can be cross-cut by one or more perspectives (or extensions). Using EAST-ADL to model a system (or a family of systems) aids in managing the associated complexity [20]. In Chapter 3, we define a reference model for modeling architecture of E/E systems that is simplified (compared to EAST-ADL) for early design, where decomposition of software components may be too detailed to capture. However, the presented model could be extended to cover full EAST-ADL and the presented synthesis and optimization would be equally applicable. Additionally, while being simplified, the presented reference model is composed of multiple abstraction levels and perspectives. We apply the reference model to building architectural models which represent many possible candidate architectures and which affords the engineers to optimize quality attributes of the architecture such as total cost, mass, and timing and explore the tradeoffs among them.

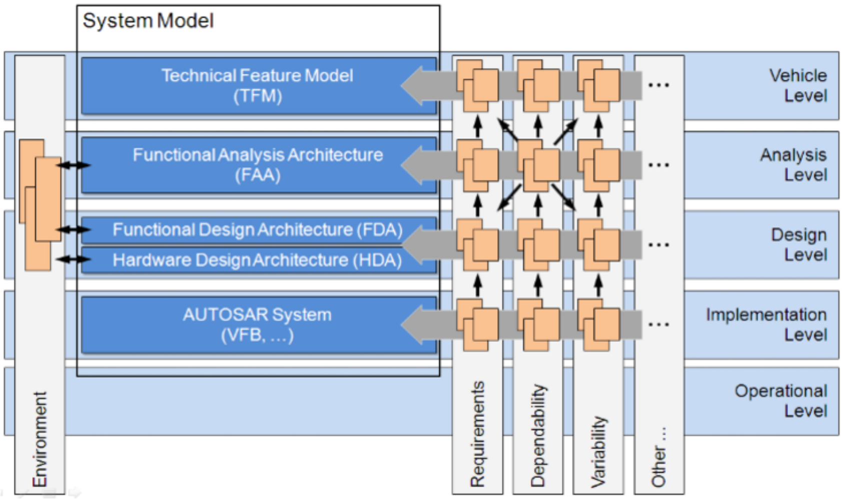


Figure 1.1: EAST-ADL Domain Model [25]

In Chapter 4, we present a mechanism for constructing a set of design specifications to use in exploring the possible candidate architectures captured in models constructed using the reference model from Chapter 3.

To appreciate the motivation, consider the following example scenario used throughout the remainder of the paper. Emily is an engineer at an automotive company. She is working on designing the E/E system architecture for the door power window. Emily wants to consider a set of candidate architectures and to compare them based on their mass and cost. The candidates themselves should be correct, meaning that they must meet any requirements and constraints specified by Emily (e.g., end-to-end latency and deployment restrictions).

While the EAST-ADL can help Emily in modeling multiple candidate E/E architectures, there is currently no approach allowing Emily to synthesize the candidate architectures, while considering design decisions at all layers, and explore trade-offs among them. In this work, we use a modeling language Clafer (see Chapter 2) to model multi-layer, multi-perspective architectures (Chapter 5). We show that by using Clafer we can model alternative design decisions for any component in addition to design constraints and objectives for a number of qualities such as latency, mass, and parts cost. Furthermore, we can join these decisions, constraints, and objectives together using first-order logic to write fine-grained design specifications for exploring the possible design space (Chapter 4).

In Chapter 6 we give the full details, including their Clafer encoding, for a power window system and door locks system. We evaluate our approach in Chapter 7 to show how it improves the state of the art. Additionally, we evaluate the performance of synthesizing the candidate architectures. We then provide the threats to validity, related work, and conclude in Chapters 8, 9, and 10, respectively.

This work provides the following contributions:

- Design exploration approach for E/E architectures based on a multi-layer and multi-perspective reference model; in contrast to previous work (see related work section), the approach explores many types of design decisions (at least 15) from multiple layers, constraints (at least 13), and objectives (at least 5) simultaneously
- The approach is supported by a modeling language, reasoning tool, and results visualization tool to enable a wide range of design exploration scenarios, which are characterized using specification templates.
- Two large case studies of realistic automotive E/E systems; the corresponding models and exploration scenarios can serve as a future benchmark for comparing design exploration tools in the E/E systems domain.
- We use one of the case studies to show the relevance of capturing multiple layers for obtaining more realistic exploration results.
- We show that the majority of the exploration scenarios of the case studies are feasible in terms of performance; we also show that exact optimization is feasible for most of the optimization scenarios for most of the studied models; however, future work will need to address scalability to larger models.

Chapter 2

Clafer Background

Clafer [4] is a lightweight, general-purpose, structural modeling language that was originally developed for feature modeling [15]. In this section we provide readers with an informal background and basic understanding of the Clafer language and its constructs. The formal semantics of the language can be found in [15].

Clafer currently supports two backend solvers that generate instances from Clafer models. The first solver (not used in this work) is Alloy [1] and the second is Choco3 CSP based solver called *chocosolver*[1]. Clafer compiler translates an input Clafer model into the input language of each backend solver. We use *chocosolver* to generate both non-optimal and optimal instances for the modeled E/E architectures. Figure 2.1 shows the workflow of how a Clafer file is compiled into one of the backend model formats then processed to generate instances. *Chocosolver* uses exact algorithms in order to find optimal instances based on single or multiple objectives.

In the following sections, the constructs of a Clafer model are explained along with examples. Additionally, we use the solver to show what instances are generated from the corresponding example models.

[1] https://github.com/gsdlab/chocosolver

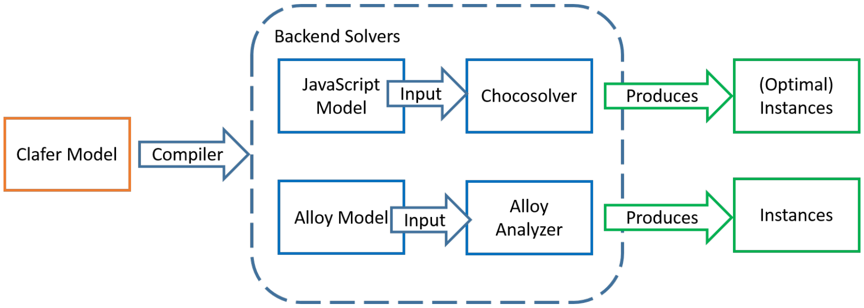


Figure 2.1: Overall workflow of Clafer and supported backend solvers.

2.1 Types of clafers and inheritance

In Clafer, a model consists of *clafers*[2]. The name "clafer" comes from the words class, feature, and reference because a clafer provides modeling capabilities of all these language constructs.

Clafers are organized in a containment hierarchy: root clafers can contain nested clafers, similarly to how classes can contain other classes and attributes. Each clafer defines a set of instances, similarly to how a class defines a set of its instances (objects). A clafer can be one of two types: abstract or concrete, similarly to classes. A concrete clafer results in an instance being generated, while an abstract one does not. Clafers are also organized in an inheritance hierarchy, like classes, with a restriction that only abstract clafers can be superclafers. Listing 2.1 shows an example of two abstract clafers `Car` and `ElectricCar` and two concrete clafers `JanesCar` and `JohnsCar`. Comments begin with `//`.

Listing 2.1: Example for concrete and abstract clafers

```
abstract Car                    // top-level abstract clafer
  engine                         // nested concrete clafer

// an abstract clafer inheriting from an abstract clafer
abstract ElectricCar : Car
  battery

// a concrete clafer inheriting from an abstract clafer
JanesCar : ElectricCar

// error: cannot inherit from a concrete clafer
```

[2] Throughout this paper if the word Clafer begins with an uppercase letter it describes the language, whereas a lowercase one denotes the language construct.

```
JohnsCar : JanesCar
```

The containment is specified using indentation, whereas inheritance if specified using colon (`:`). With respect to the containment hierarchy, we say that a clafer is a child of another clafer (e.g., `engine` is a child of `Car`) and a clafer is a parent of another clafer (e.g., `Car` is the parent of `engine`). With respect to the inheritance hierarchy, we say that a clafer is a superclafer of another one (e.g., `Car` is the superclafer of `ElectricCar`, which itself is a superclafer of `JanesCar`). Conversely, a clafer inherits from/is a subclafer of/extends another clafer (e.g., `ElectricCar` inherits from/is a subclafer of/extends `Car`).

2.1.1 Instance generation

Given a Clafer model, a backend reasoner can generate all instances of the model in a given finite scope. For example, for the model in Listing 2.1 (without the incorrect `JohnsCar`) we get the following instance:

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

JanesCar
  engine
  battery

--- Instance 1 End ---
```

As we can see, only instances of the concrete clafer `JanesCar` and the inherited concrete clafers are generated. For simplicity, the instances have the same names as the clafers. To be fully explicit one would have to write the following, which we usually omit (we overload : to mean *instanceOf* relationship).

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

JanesCar : JanesCar
  engine : engine
  battery : battery

--- Instance 1 End ---
```

Note, that because inheritance is transitive, the instance `JanesCar` is also an instance of `ElectricCar` and `Car`.

2.2 Clafer multiplicity and group cardinality

Up to this point none of the concepts introduced have allowed for expressing variability in the model. Clafer provides two constructs expressing variability: clafer multiplicity and group cardinality. Clafer multiplicity is a range `n..m` indicating the allowed number of instances of the given clafer with respect to its parent. The most common multiplicities are `1..1` (mandatory) which requires one instance of the given clafer per instance of its parent; and `0..1` (optional) which allows for at most one instance of the given clafer per instance of its parent.

Group cardinalities are used to express variability over a group of children of the given clafer, hence the name. All clafers, by default have the group cardinality of `0..*`, that is, they do not impose any constraints on their children.

In Listing 2.2, we explicitly show the multiplicities of all clafers. By default, all abstract clafers have the multiplicity of `0..*`, which we usually omit. Also, by default, clafers have the multiplicity of `1..1` unless they are children of a group with cardinality other than `0..*`, in which case the default multiplicity is `0..1`. For example, the clafer `engine` has multiplicity of `1` (short for `1..1`), `wheel` has multiplicity of `4`, and `seat` has the multiplicity of `2..4`. These multiplicities specify that every instance of `Car` contains exactly one instance of `engine`, four instances of `wheel`, and between two and four instances of `seat`, respectively.

In Listing 2.2, the clafer `transmission` has group cardinality `xor` (keyword for `1..1`) meaning that every instance of `transmission` must contain exactly one instance of its children (`automatic` or `manual`).

Listing 2.2: Example with explicit clafer multiplicities and a group cardinality

```
abstract Car 0..*
  engine 1
  xor transmission 1
    automatic 0..1
    manual ?
  wheel 4
  seat 2..4

abstract Chevy : Car 0..*

JohnsCar : Chevy 1
```

Using the instance generator, two of the 6 instances are:

```
=== Instance 1 Begin ===
JohnsCar
  engine
  transmission
    automatic
  wheel
  wheel$1
  wheel$2
  wheel$3
  seat
  seat$1

--- Instance 1 End ---

=== Instance 4 Begin ===

JohnsCar
  engine
  transmission
    manual
  wheel
  wheel$1
  wheel$2
  wheel$3
  seat
  seat$1
  seat$2

--- Instance 4 End ---
```

Note that in order to distinguish the multiple instances of the same clafer from each other, a suffix `$n` where n is the instance number is added.

Both model instances satisfy all multiplicity and group cardinality constraints; for example, the instance generator will never generate an instance which has both kinds of transmission or neither of them.

2.3 References

The last Clafer construct which allows for variability in models are reference clafers, that is, clafers whose instances can point to instances of other clafers or primitive values.

Listing 2.3 shows an example of a clafer `Car` that contains two references `driver` and `passenger`, which denote the driver and passengers of the car, respectively. The type of the reference clafer `driver` is `Person`, which means that every instance of the clafer `driver` points to one instance of the clafer `Person`. The multiplicity of `driver` is `1..1`, by default, so every instance of a `Car` will be always connected with one instance of `Person` via an instance of the reference clafer `driver`. The reference clafer `passenger` can have between zero and four instances, each pointing to an instance of `Person`. There are two kinds of reference clafers: set (specified using `->`) and bag (multiset, specified using `->>`). In our example, we do not want to allow the same person to be a passenger more than once, that is, the collection of passengers should be a set and we used `->` to express that constraint.

Listing 2.3: Example using references

```
abstract Person

abstract Car
  driver -> Person
  passenger -> Person 0..4

MyCar : Car
John : Person
Jane : Person
```

A correct instance of this model then would have `John` or `Jane` as the driver because they are the only clafers of type `Person` in the model. The reference clafer `passenger` points to a set of size 0 to 4 meaning that up to four passengers can be in the car. A correct instance of this model can have no passengers, both John and Jane as passengers, or only John or only Jane as a passenger.

Two of eight possible instances for Listing 2.3 are:

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

MyCar
  driver -> John
John
Jane

--- Instance 1 End ---

=== Instance 8 Begin ===

MyCar
  driver -> Jane
  passenger -> John
  passenger$1 -> Jane
John
Jane

--- Instance 8 End ---
```

We can see two instances of the reference clafer `passenger` (`passenger` and `passenger$1`) pointing to the instances of `John` and `Jane`, respectively. The following instance is incorrect.

```
MyCar
  passenger -> John
  passenger$1 -> John

John
Jane
```

First, a required instance of `driver` is missing, which violates the multiplicity `driver 1..1`; second, the same instance of `John` is a passenger twice, which violates the set constraint.

2.4 Writing Basic Constraints

So far all we have shown with Clafer is the ability to create models with large amounts of variability using references and cardinalities. When modeling real systems and problems a modeler also wants to use constraints in order to restrict the targets of references and the allowed configurations of the model. Additionally, constraints are used when wanting to query a model for one or more specific instances which satisfy the given constraints.

In this section, we give readers some of the basics for writing constraints but for conciseness it is not exhaustive. Readers should refer to other documentation found at http://www.clafer.org (we recommend "Clafer Cheat Sheet"[3] for an informal language reference). One general rule when writing constraints in Clafer, is that there are no scalars in the language, only sets, and therefore writing a number `1` really means a singleton set containing the number one. Similarly, we cannot directly access clafer instances (since they only exist at instance generation time) and instead we often use singleton sets. For example, the concrete clafer `John : Person 1` is a singleton set which will contain exactly one instance, thus by writing constraints about the clafer `John`, we write constraints about its only instance.

[3] http://t3-necsis.cs.uwaterloo.ca:8091/ClaferCheatSheet

Listing 2.4 builds on the previous two examples and adds some constraints. The constraints that we want to model are:

- C1: *John is the driver of MyCar*
- C2: *Only Dan and Jane can be passengers in MyCar*
- C3: *The driver should not be in the set of passengers in a car*
- C4: *The total number of passengers with the driver can't exceed the number of seats in a car*

A constraint is a Boolean expression surrounded by square brackets "[ ]". Just like clafers, constraints can be either top-level or nested. Nested constraints must hold for every instance of their context clafer (the clafer they are nested under); consequently an instance of the context clafer cannot exist unless all of its nested constraints hold.

In Listing 2.4, the constraint C1 is captured in line 10, whereby `.dref` gets the target value of the reference clafer (much like dereferencing a pointer in C/C++). C2 is then captured in line 11 in the model. Note the subtle differences between C1 and C2 in the constraints and the wording. For C1, set equality (the operator `=`) is used to restrict that the target of the reference *has to be* `John`. In C2, the wording is "can be" so subsetting is used (the keyword `in`). The difference between `in` and `=` is that the former can let the set be any of values to the right of the operation where the latter requires that the set is equal to the right of the operation (note the correct instances that can be generated). In C2, the expression `Jane, Dan` computes a union of instances of Jane and Dan (alternatively, it can be written as `Jane ++ Dan`).

Listing 2.4: Example using constraints

```
1  abstract Person
2  abstract Car
3    seat 2..4
4    driver -> Person
5    passenger -> Person 0..4
6    [driver.dref not in passenger.dref] // C3
7    [#(passenger, driver) <= #seat] // C4
8
9  MyCar : Car
10   [driver.dref = John] // C1
11   [passenger.dref in (Jane, Dan)] // C2
12 John : Person
13 Jane : Person
14 Dan : Person
```

Constraints C3 and C4 pertain to every instance of `Car` so the constraints are nested under `Car`. For C3 the `not in` enforces that the target of `driver` is not present in the set `passenger`.

C4 uses # to get the size of a set such that one can say the size of the union of sets `passenger` and `driver` is less than or equal to the size of the set `seat`. When using the instance generator 12 instances are found that satisfy the constraints and two of them are as follows:

```
=== Instance 3 Begin ===

MyCar
  seats
  seats$1
  driver -> John
  passenger -> Dan
John
Jane
Dan

--- Instance 3 End ---

=== Instance 11 Begin ===

MyCar
  seats
  seats$1
  seats$2
  seats$3
  driver -> John
  passenger -> Jane
  passenger$1 -> Dan
John
Jane
Dan

--- Instance 11 End ---
```

Constraints are one of the most important aspects of Clafer for modeling systems since restrictions on system configurations, deployments, etc. can be constrained such that only correct systems are synthesized.

2.5 Working with Integers

Clafer and its backends also support working with integers allowing for adding quantitative information to models. Currently, only integer arithmetic is supported, so working with numbers is cumbersome when modeling real systems.

Constraints can be used to model numerical relationships between components and get the total for a set of component. For example, Listing 2.5 takes the car example and models three features which have an associated cost.

Listing 2.5: Example using constraints

```
abstract Feature
  cost -> integer
abstract Car
  bluetooth : Feature
  heatedSeats : Feature
  passiveKeyEntry : Feature

MyCar : Car
  [bluetooth.cost = 5]
  [heatedSeats.cost = 10]
  [passiveKeyEntry.cost = 25]
```

The constraints nested under `MyCar` configure an instance of `Car` by giving values to the different feature costs. The dot "." operator navigates from a parent to a child clafer or to the target of a reference. Using the instance generator gives just one correct instance as there is no variability in the model (all references have been tightly constrained and all multiplicities are fixed).

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

MyCar
  passiveKeyEntry
    cost -> 25
  heatedSeats
    cost$1 -> 10
  bluetooth
    cost$2 -> 5

--- Instance 1 End ---
```

2.6 Optimization Objectives

In order to do optimization over qualities of a model, objectives must be defined. In Clafer, optimization objectives are captured through goals as in lines 26 and 27 of Listing 2.6. In this example the car with features is used but a second quality is added to represent the *comfort level* for a user. Also note that all of the features are optional (denoted by the `?`, meaning cardinality 0..1) such that there is variability among the available features. In order to populate the used feature set of the `Car` the constraint C1 on lines 9, 13, and 17 constrains that `this` (the Feature) is in the set of `parent`'s (the Car's) child `feature`. The constraint C2 on lines 21 and 22 sum the cost and comfort (respectively) of all the present features in the car to be used for the optimization goals.

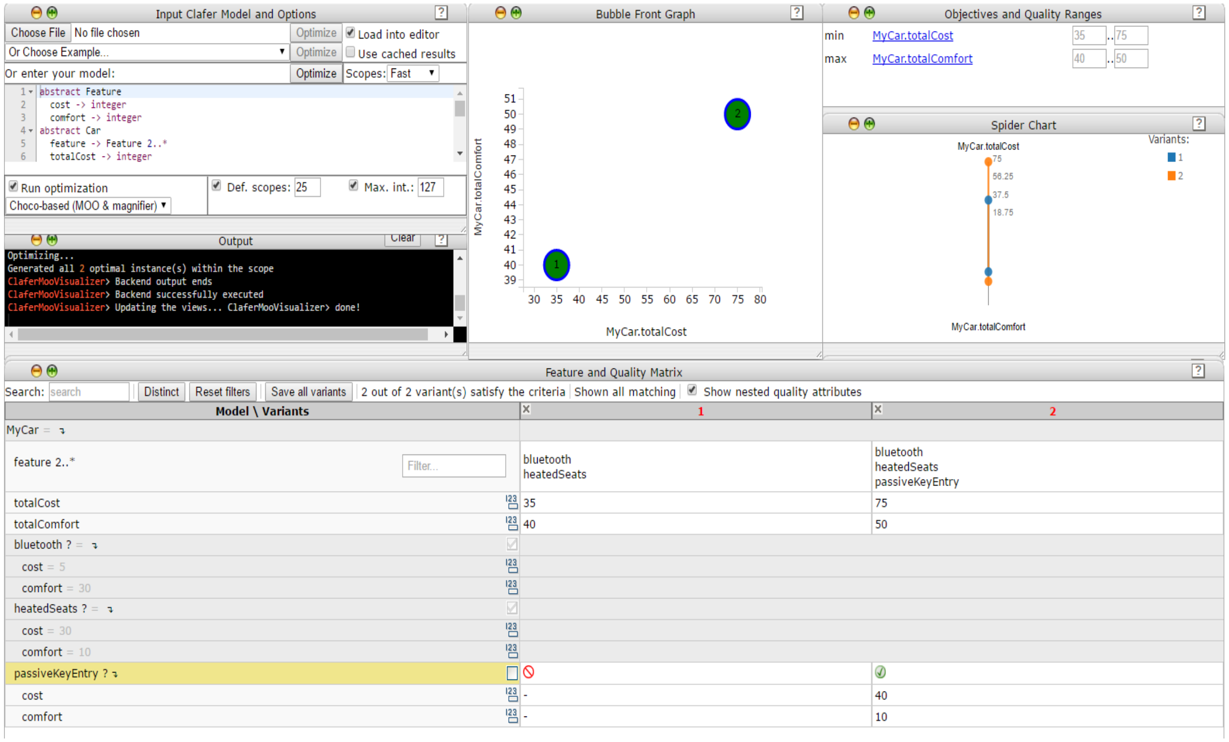
Listing 2.6: Example using optimization

```
1  abstract Feature
2    cost -> integer
3    comfort -> integer
4  abstract Car
5    feature -> Feature 2..*
6    totalCost -> integer
7    totalComfort -> integer
8    bluetooth : Feature ?
9      [this in parent.feature] // C1
10     [cost = 5]
11     [comfort = 30]
12   heatedSeats : Feature ?
13     [this in parent.feature] // C1
14     [cost = 30]
15     [comfort = 10]
16   passiveKeyEntry : Feature ?
17     [this in parent.feature] // C1
18     [cost = 40]
19     [comfort = 10]
20
21   [totalCost = sum feature.cost] // C2
22   [totalComfort = sum feature.comfort]
23
24 MyCar : Car
25
26 << minimize MyCar.totalCost >>
27 << maximize MyCar.totalComfort >>
```

The instance generator configured for optimization finds 2 Pareto-optimal instances which are shown below:

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

MyCar
  feature -> bluetooth
  feature$1 -> heatedSeats
  totalCost -> 35
  totalComfort -> 40
  bluetooth
    cost -> 5
    comfort -> 30
  heatedSeats
    cost$1 -> 30
    comfort$1 -> 10

--- Instance 1 End ---

=== Instance 2 Begin ===

MyCar
  feature -> bluetooth
  feature$1 -> heatedSeats
  feature$2 -> passiveKeyEntry
  totalCost -> 75
  totalComfort -> 50
  bluetooth
    cost -> 5
    comfort -> 30
  heatedSeats
    cost$1 -> 30
    comfort$1 -> 10
  passiveKeyEntry
    cost$2 -> 40
    comfort$2 -> 10

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

What are the tradeoffs between these two instances? Instance 1 has the lowest cost, while sacrificing the comfort. Instance 2 has the highest comfort at the higher cost. Clafer tools include Clafer Multi-Objective Optimization (MOO) Visualizer, a tool for visualizing and exploring the set of optimal instances of a model, which allows the users to perform tradeoff analysis and find the instances most suitable for their needs [6]. A screenshot from the tool is shown in Figure 2.2 when visualizing the instances generated from Listing 2.6.



Input Clafer Model and Options

Choose File No file chosen | Optimize | ☑ Load into editor

Or Choose Example... ▾ | Optimize | □ Use cached results

Or enter your model: | Optimize | Scopes: Fast ▾

```
abstract Feature
  cost -> integer
  comfort -> integer
abstract Car
  feature -> Feature 2..*
  totalCost -> integer
```

☑ Run optimization

Choco-based (MOO & magnifier) ▾

☑ Def. scopes: 25 ☑ Max. int.: 127

Output

```
Optimizing...
Generated all 2 optimal instance(s) within the scope
ClaferMooVisualizer> Backend output ends
ClaferMooVisualizer> Backend successfully executed
ClaferMooVisualizer> Updating the views... ClaferMooVisualizer> done!
```

Bubble Front Graph

Objectives and Quality Ranges

min	MyCar.totalCost	35	75
max	MyCar.totalComfort	40	50

Spider Chart

Feature and Quality Matrix

Search: search | Distinct | Reset filters | Save all variants | 2 out of 2 variant(s) satisfy the criteria | Shown all matching | ☑ Show nested quality attributes

Model \ Variants	1	2
MyCar = ↴		
feature 2..*	bluetooth heatedSeats	bluetooth heatedSeats passiveKeyEntry
totalCost	35	75
totalComfort	40	50
bluetooth ? = ↴		
cost = 5		
comfort = 30		
heatedSeats ? = ↴		
cost = 30		
comfort = 10		
passiveKeyEntry ? ↴	⊘	✓
cost	-	40
comfort	-	10

Figure 2.2: Screenshot of MOO Visualizer for Listing 2.6.

Chapter 3

Reference Model

Using the motivational scenario, in this section we define a reference model (for the remainder of this paper we refer to it as *the reference model*) that supports early design of E/E system architectures. Similar to the EAST-ADL, the reference model is made up of multiple abstraction layers. Each layer describes the system, but at a different level of abstraction. We use the term *multi-layer* to describe models consisting of multiple layers. The reference model also contains multiple perspectives, which augment the system with analysis-task or stakeholder-specific information such as points of variability, latency, and mass; we use the term *multi-perspective* to describe models that contain more than one perspective. Figure 3.1 shows the *multi-layered*, *multi-perspective* reference model with the supported perspectives covered in this paper. We use an ellipses in the perspectives to show that the reference model is not limited to the ones shown, but could be extended with additional ones.

When a perspective is combined with a multi-layered model, it may cross-cut some or all the layers. For example, the mass perspective applies primarily to the hardware architecture, where devices and wires are assigned mass; however, mass information can also be aggregated at the feature level.

The reference model we define is similar to the EAST-ADL domain model; however, the one defined here focuses on the early design stages and does not consider the lower-level software decomposition for the system. More precisely, the simplified model does not consider the functional design architecture of the EAST-ADL; instead, it maps the functional analysis architecture on the hardware design architecture. Additionally, the reference model contains cross-cutting concerns that are not present in the EAST-ADL such as cost and

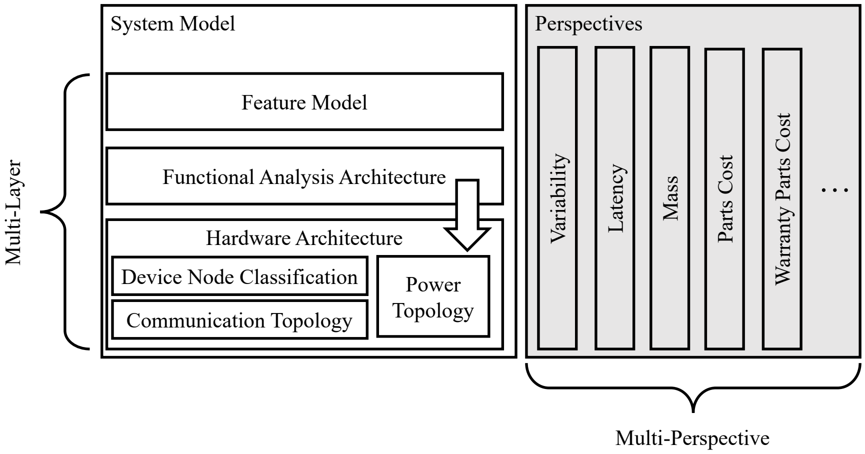


Figure 3.1: The reference model used for early design of automotive E/E systems architecture.

mass, which were introduced by Murashkin [49]. The following two sections detail the different layers and perspectives of the model, respectively, using Emily's power window as the running example.

3.1 Reference Model Layers

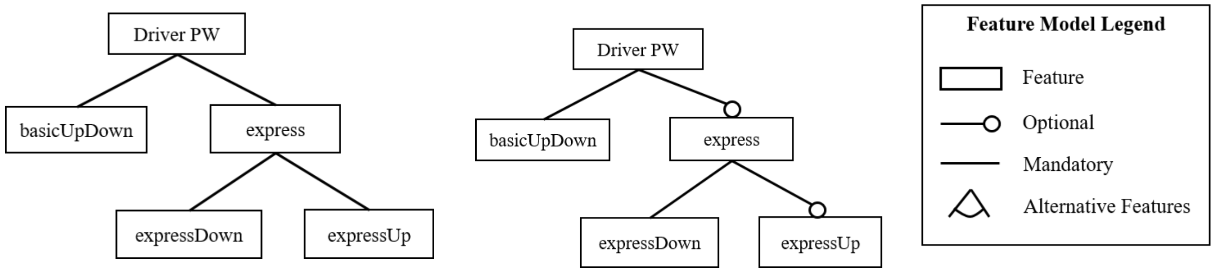
Starting with the top-most layer, the features of the system are captured in the *feature model layer*[1]. A *feature* is a high-level system characteristic relevant to some stakeholder, such as the customer or the user. Features may represent functionality or performance. When defining the features of the power window system, Emily chooses to have three features:

- `basicUpDown:`[2] the basic functionality of the power window to close or open it by pulling up or pressing the switch until the window glass reaches an end position;
- `expressDown:` the window glass retracts automatically when the user presses the switch once;
- `expressUp:` the window glass closes automatically, unless an object is detected, when the user pulls up and releases the switch once.

[1] We use *italics* to introduce a reference model *element*.

[2] We use `typeface` to denote Emily's model elements; a *concrete element*.

Figure 3.2a is a feature model, which organizes features into a hierarchy and indicates their variability, as Kang et. al. proposed in [35]. In Figure 3.2a, the features are all mandatory (i.e., no variability). In Figure 3.2b, express and expressUp are optional; the hierarchy also expresses an implication from expressUp to express, and from express to Driver PW. In this section, we will assume the feature mode representing a single system (Figure 3a) for our example.



(a) Single system.

(b) Three variants.

Figure 3.2: Power window feature models.

The features are then implemented by one or more functions in the *functional analysis architecture (FAA) layer*. There are two types of functions defined in the FAA: *analysis functions*, which model control functions with their inputs and outputs; and *functional devices*, which capture functions that represent sensors and actuators. Many of the analysis functions will be realized as software components; however, some functions or parts of them may be realized by specialized hardware (such as digital signal processor). Functional devices will typically be realized by hardware sensors and actuators plus additional software, such as device drivers or signal conditioning software. The FAA not only captures the functions, but also the communication among them in order to define a function graph (using *function connectors*). Figure 3.3 shows a FAA for the power window in which part of the feature `expressUp` is implemented by the functions `PositionSensor`, `PinchDetection`, and `WinControl`, including the communication between them.

The *hardware architecture* models the physical hardware devices and connectivity media in the system, which we decompose into the *device node classification*, *communication topology*, and *power topology*. The *device node classification* captures the *device nodes* in the system and their properties; one of which is the device node type. We define a device node as a piece of hardware such as a sensor, actuator, ECU, switch, electric center, or battery. We consider three types of device nodes:

- *Smart*: A device node that can be programmed with one or more analysis functions

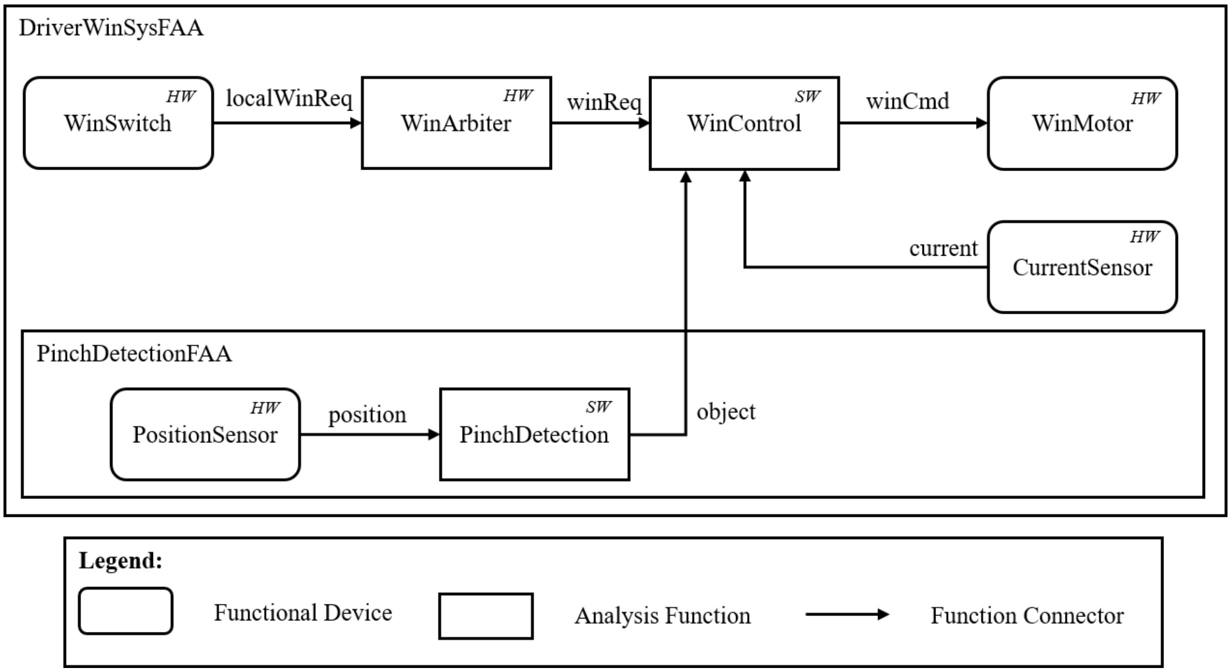


Figure 3.3: Power window functional analysis architecture. The "HW" or "SW" in upper right corner of a function indicates the implementation choice in hardware or software, respectively.

or functional devices which are implemented in software. Examples would be an ECU or a hardware device with embedded microcontroller.

- *Electric/Electronic (E/E)*: A device node that can not be programmed with executable software but rather implements some hardware functionality described by a functional device or analysis function that is implemented in hardware. Examples would be an analog sensor or actuator.
- *Power*: A device node that generates, stores, or relays power to other device nodes. In this paper, we do not allow a power device node to have any logic associated with it (i.e., no functional devices or analysis functions deployed to it). An example would be a battery or fuse box.

Figure 3.4 shows a class diagram of the classification for device nodes, analysis functions, and functional devices. It also shows the allowed deployments of the different functions to the various device node types. Notice from the figure that the classification is a lattice, that is, a smart device also has E/E and power capabilities, whereas an E/E device also has power capability but not smart capability. Thus, a functional device or analysis function implemented in hardware can be deployed to either an E/E or a smart device node.

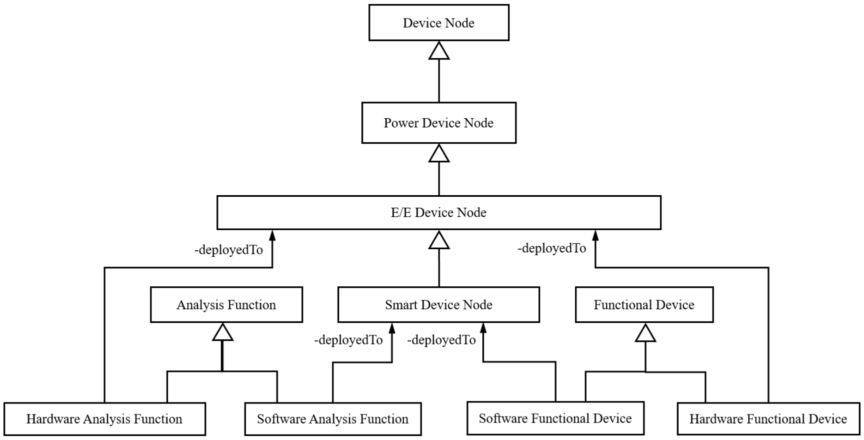


Figure 3.4: Class diagram showing classification of device nodes and functions.

In addition to classifying the device node type, we also differentiate between *local* and *remote* device nodes. A *local* device node is one that is owned by the modeled system whereas a *remote* node is owned by another system. In her power window, Emily chooses to have three local device nodes:

- `Motor` - an actuator that is *smart*.
- `Switch` - a sensor that is *E/E*.
- `DoorModule` - an ECU that is *smart*.
- `DoorInline` - an interconnect that connects the wiring from the main body harness to the door harness and that is a *power* device node.

In addition to the local nodes, Emily has two remote device nodes: the `BCM` (body control module) and the `EC` (electric center). The `BCM` is an ECU that is shared among all body-domain systems, including the power window, and the `EC` is a fuse box that acts like a main power source for the car.

The next part of the *hardware architecture* is the *communication topology*, which defines the physical media that function connectors are deployed to. The following hardware connectors make up the topology:

- *Discrete data connectors*: A connector used to indicate the status of a binary input, such as a switch. Figure 3.5 shows the different levels of abstraction for modeling

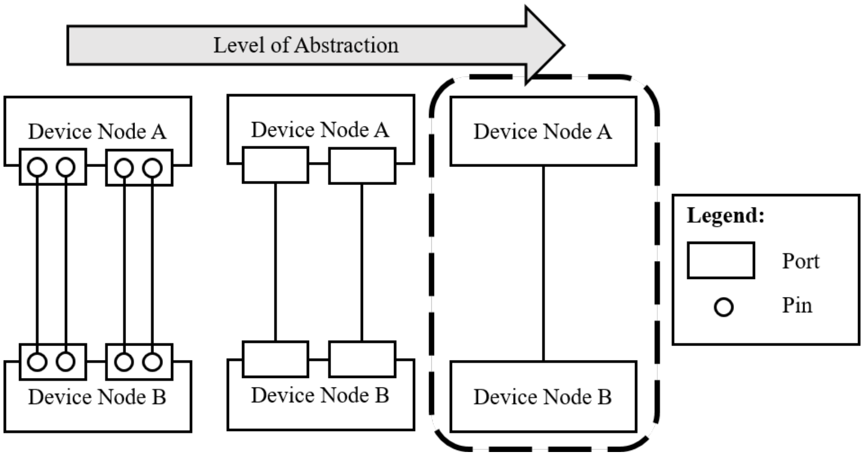


Figure 3.5: Modeling of data connectors at different levels of abstraction. The lowest level is shown on the left, where connectors are modeled between pins; the middle shows connectors between ports; the right is the highest level, where two device nodes are connected via a connector. The dashed selection shows the level of abstraction used for data and power connectors in this paper.

hardware connectors. We chose to model them at the highest level (shown in the dashed box) to reduce model complexity; thus we don't model ports or pins on a device node.

- *Analog data connectors*: A connector which sends an analog signal (as opposed to digital); typically encoded as voltage amplitude. This is also an abstract connector similar to the discrete data connector.
- *Bus connectors*: An abstract connector between two or more device nodes in which all connected nodes may pass messages; these are typically serial buses such as CAN [23] and LIN [7].

Figure 3.6 shows the communication topology Emily chooses to connect the device nodes. The smart nodes are connected together using the bus and only the switch and BCM are connected via a discrete data connector.

The last part of the hardware architecture we consider is the *power topology*, which models how device nodes are connected with power and in which we consider two types of power connectors:

- *Load Power Connector*: An abstraction for lower gauge wires that distributes higher

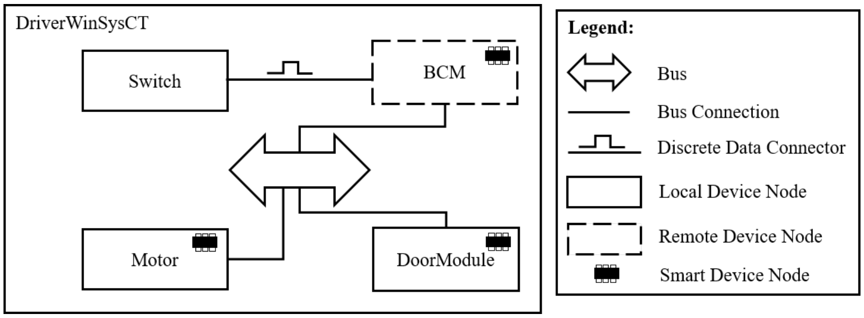


Figure 3.6: Power window communication topology.

power to device nodes. These connectors are often used to power motors, heaters, and lights.

- *Device Power Connector*: An abstraction for higher gauge wires that distributes lower power to device nodes. These connectors are often used to power smart devices.

As with the data connectors, we choose to model the power connectors with a high level of abstraction and disregard ports, pins, and routing through the wiring harness.

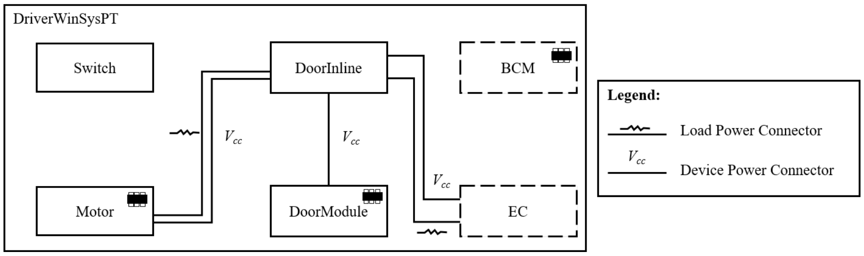


Figure 3.7: Power window power topology.

Figure 3.7 shows the power topology Emily models for her power window. Two load power connectors are used to provide the necessary voltage from the EC to the motor. Additionally, three device power connectors are used to connect the `EC` with the smart devices in the architecture. Even though the `BCM` is *smart*, device power is not modeled because the `EC` and `BCM` belong to other systems (shown as remote device nodes).

The last part of the system model is the *deployment* of the *FAA* to the hardware architecture, which is represented by the arrow in Figure 3.1. The analysis functions and functional devices are deployed onto device nodes, whereas function connectors are mapped onto the different communication topology connectors.

The deployment step may result in different allowed topologies and node classifications. For example, in Emily's power window, the communication topology only defines a discrete connector between `Switch` and `BCM` as well as the `Switch` being *E/E*. Therefore the function `WinArbiter` has to be deployed to the `BCM` if `WinSwitch` is deployed to the `Switch`. This is the only correct deployment because the reference model does not support deploying a single function connector to more than one hardware communication medium.

3.2 Reference Model Perspectives

Until this point, Emily has only defined a single candidate architecture for her power window. In order for her to model multiple candidates, variability (also known as *degrees of freedom* [10]) has to be expressed in the model. First, Emily adds variability to the feature model in which the express features are made optional and the feature `expressUp` is only present if the feature `express` is (c.f. Figure 3.2b). Just by adding variability at the feature level, Emily now has three possible candidates. Adding variability to the feature `expressUp` results in having optional functions and connectors which are used to implement the feature. Figure 3.8 shows the resulting variable functional analysis architecture in which the `PinchDetectionFAA` is optional; which implies all contained functions and connectors are optional as well (`PositionSensor`, `position`, `PinchDetection`, and `object`). Emily can also express variability in how a function is implemented; for example, in Figure 3.8 Emily choses to have `WinArbiter` be implemented in either hardware or software.

Additionally, variability can be expressed in the mapping of the FAA to the hardware architecture by allowing more than one valid deployment for a function to a device node or a function connector to a communication topology connector. We also consider variability for the type of device node in the device node classification. For example, Emily might have two types of `Switch` device nodes she can use in the system, either a smart or an electric/electronic one. Furthermore, variability can be expressed over the presence of a device node in the classification; such as the `DoorModule` being optional in the resulting candidate architectures. Lastly, the communication and power topologies can contain variability in the presence of connectors or in the case of a bus connector, the type of bus (i.e. LIN, CAN, or FlexRay). By adding variability to each of the layers in the power window model, Emily can end up modeling an exponential (in the number of components) number of candidate architectures. With adding variability to each individual layer, Emily also does not need to enumerate manually the complete variable configurations of the system; instead she can simply add variations to individual components.

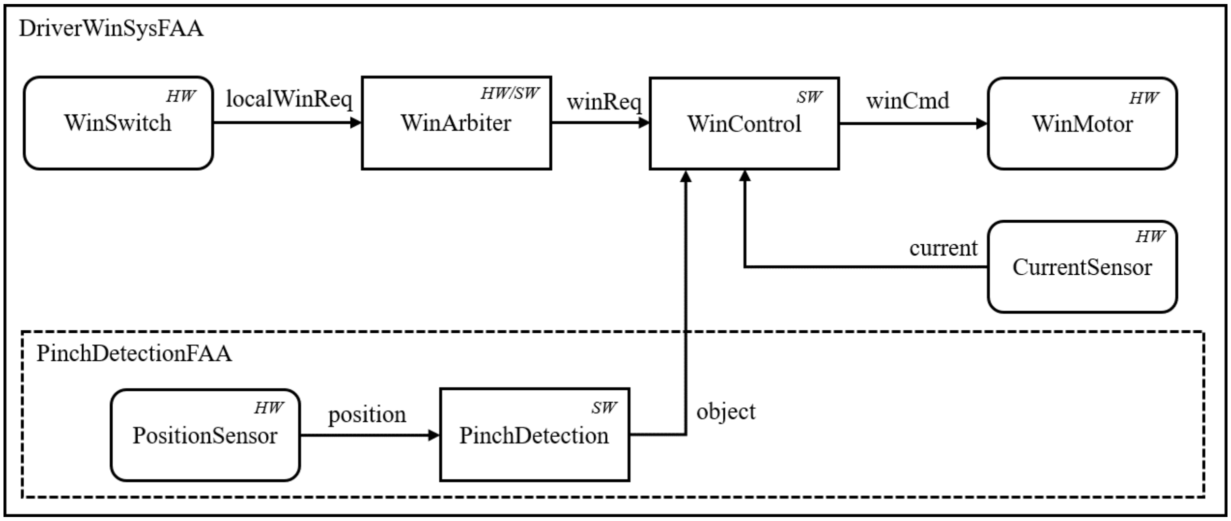


Figure 3.8: Power window FAA with variability. Optional components are shown by a dotted border/line. "HW/SW" indicates the function can be implemented in either software or hardware.

In order to evaluate the candidates based on metrics such as mass and cost, Emily must define qualities for the components of the different layers. When considering the mass, part unit cost, and part warranty cost of an architecture, we only assign values at the hardware architecture layer as it represents the physical elements in the architecture; we do not consider other costs in our model. Therefore, the components that add to the total mass and cost of a system are device nodes, power connectors, and communication connectors. We consider part warranty cost to be the cost incurred by the OEM when a defective part fails and calculate by taking the failure rate multiplied by the replacement part cost.

The last perspective that we define in this paper is for the latency of a system. We do this through *timing chains* — a sequence of executing functions with their communication as defined in the FAA. With the latency of a system being modeled, timing requirements such as *end-to-end latency* — the time taken to reach a target function from some source function — or *input synchronization latency* — the maximum difference in time taken to reach a target function from more than one source function. For the power window, Emily could model the timing requirement that the time taken from the function `WinSwitch` to function `WinMotor` is less than 500 ms.

In this work, we consider a simple latency model where schedulability and complex equations (such as the recurrence relation for worst case transmission delay [22]) are not modeled. Instead, we assume the deployment of functions to nodes to be schedulable and they are scheduled in a way to avoid any bus arbitration (no blocking times). Additionally, we do not consider the overhead associated with different communication protocols which is needed for a more precise timing model. Such a model is sufficient for very early exploration. We consider the following components to contribute to the latency of a system:

- *Functional device & analysis function*: a base latency is specified by the user. If the function is implemented in software, the base latency represents the latency a function is expected to take at some base ECU speed factor. If the function is implemented in hardware, the base latency is simply the hardware specification latency.
- *Function connector*: a size is specified by the user to represent the size of the message being sent. Then based on whether the function connector is deployed and to what type of hardware connector, the latency is calculated using the transfer rate of the hardware connector.
- *Device node*: a speed factor can be given to the device node which affects the resulting latency of deployed analysis functions.
- *Discrete/analog data connector*: the latency is assumed to be zero in our model.
- *Bus connector*: the transfer rate is dependent on the type of bus, which then affects the deployed to function connectors.

In the next chapter, we characterize the different design specifications, for exploration, that can be constructed for a model built using our reference model, such as Emily's power window.

Chapter 4

Characterizing the Possible Design Scenarios

For Emily, building an architectural model of her system has provided no exploration benefits, yet. However, the model represents all possible candidate designs; being able to ask or query for a subset of them based on some *decisions* would provide such a benefit. Making design decisions is routinely done throughout the early stages of design. For example, Emily might need to decide if having a dedicated ECU on each door is a cost-effective solution while meeting some high-level requirements such as end-to-end latency or mass. While making a decision is quite simple (e.g., Emily could just flip a coin), the challenge is finding the possible remaining candidates which satisfy the high-level requirements and the model constraints when the decision is made. Additionally, if more than one decision is made (e.g., Emily chooses to have the dedicated ECU and have the feature `expressUp`), which is often done, it may be challenging to determine if a candidate design is possible.

Another benefit Emily could reap from building the architectural model is comparing two or more sets of candidates. While an argument could be made that this comparison could be done manually (e.g., it is intuitive that adding a dedicated ECU will be of higher cost than not), it is limited to comparing a single candidate from each set. Consider the example shown in Figure 4.1 in which Emily is trying to compare designs with a dedicated door ECU and designs without such an ECU. Doing this manually would require generating, by hand, a candidate for each design choice (shown as black circles in the figure) and then comparing them one at a time (shown as the red rounded rectangles). Using the architectural model and a supporting reasoner, all candidates for both design decisions could be synthesized and the sets of architectures could be compared at one time (shown as the blue rounded

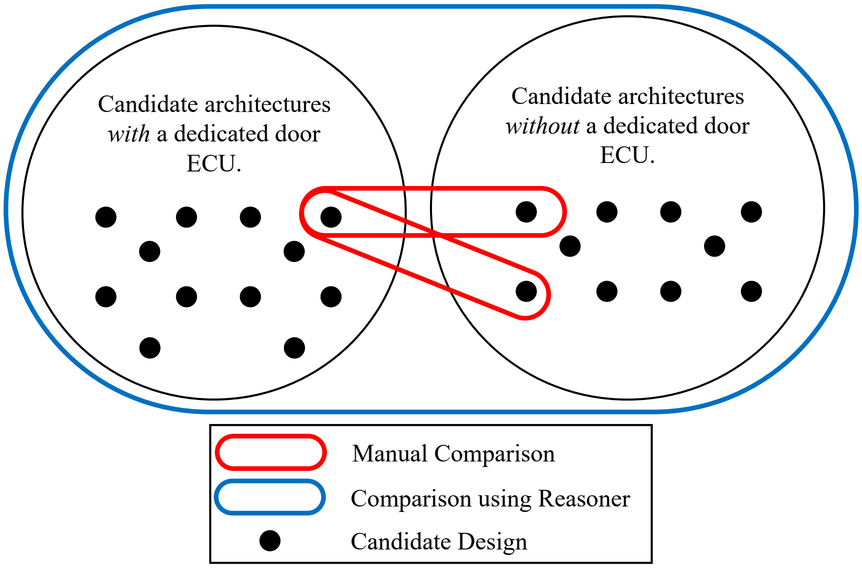

Candidate architectures *with* a dedicated door ECU.
Candidate architectures *without* a dedicated door ECU.
Manual Comparison
Comparison using Reasoner
Candidate Design

Figure 4.1: Illustration of comparing candidate designs for different sets of architectures using a manual versus an automated model-based approach.

rectangle). Then, since we are comparing sets, we can group candidates by part cost values to show if there exists any candidate with the dedicated ECU which costs less than any without.

Furthermore, Emily could ask for a comparison between only optimal candidates. This would allow her to compare the candidates with minimal part costs when either design decision holds. Doing this optimization manually would not be practical, unless the number of candidates is small; by considering all layers of the reference model a reasoner would also produce candidates that are globally optimal rather than locally within only one or two of the layers.

In the following subsections, we first give example exploration scenarios using design decisions that Emily, or another engineer, might make for the power window model built using the reference model. Secondly, we generalize the decisions, constraints, and objectives that can be reasoned about for any model constructed using the reference model from Chapter 3. This generalized set of decisions covers a subset of the possible degrees of freedom and qualities that are captured by the reference model.

4.1 Possible Design Exploration Scenarios

For the power window that Emily is designing, we give several design scenarios in which Emily would like to explore the possible designs based on her model using a set of decisions, constraints, and objectives.

1. Emily would like to investigate the possibility of adding a dedicated ECU to each door (we call the door module). Precisely, she would like to find out if it is a cost-effective solution while meeting the requirements for mass and latency.

2. Suppose that Emily has made a set of decisions which captures the architecture of her companies previous generation power window system. Now, Emily would like to know if it is possible to implement the express up feature on the previous generation architecture while satisfying latency constraints. If so, she would like to find out what designs they would be.

3. Emily would like to compare having a *dumb* architecture which uses electric/electronic switches and motors versus a *smart* architecture where they are smart. She would like to explore optimal designs of both to see how they compare in respect to the qualities: mass, parts costs, and part warranty cost.

4. Emily is tasked with designing the power window for a higher end car in which cost is irrelevant but mass should be minimized, she would like to explore the possible designs. Additionally, since it is a high end car, all features should be included. Lastly, the end-to-end latency for pinch detection to react and reverse the motor should be less than 200 ms.

5. Emily would like to minimize the cost, regardless of the features, to support an "economy class" vehicle her company is rolling out. Is there an optimal car design that does include all features?

6. Emily is investigating whether it would be better in terms of all qualities (latency, mass, parts cost, and part warranty cost) to have a distributed architecture, when considering multiple power window systems (e.g., the driver and passenger doors), or a centralized one.

By using the reference model we describe in the previous section, Emily can construct a model that can be used in each of these scenarios. This is possible because of the layers we consider along with the cross-cutting perspectives. By modeling from the features down to the power and communication topology, Emily can consider the comprehensive impact of making decisions. Also, she is able to make decisions since each layer is augmented with a variability perspective, thus she can consider adding or removing elements, such as a feature or device node. In the next subsection, we formally state the set of design decisions, constraints, and objectives that can be expressed over any model using the reference model in order to explore the possible candidate architectures.

4.2 Generalizing the Possible Design Exploration Scenarios

As we saw in the previous section, design space exploration entails exploring resulting candidates when certain decisions are made, constraints are held, and objectives are optimized. Before we state what possible decisions, constraints, and objectives one can make when using the reference model, we first define what design exploration are and its defining elements as follows:

- *Design Space Exploration*: The activity of discovering and evaluating design alternatives, captured by design specifications, during system development (adapted from [34]).
- *Design Specification*: A set of one or more design decisions, constraints, and objectives joined together by Boolean logic formed over a architectural model to specify the desired candidate architectures.
- *Design Decision*: A restriction on the variability allowed by one or more components in the modeled architecture.
- *Design Constraint*: A restriction on the value for a system or component quality in the modeled architecture such as mass or latency.
- *Design Objective*: An optimization goal for a system quality in the modeled architecture in which to maximize or minimize.

We consider two classes of design space exploration scenarios. The first class uses a *single design specification*. Scenarios 2, 4, and 5 from the previous section fall into this class. A common example of this class would be causal analysis where a certain specification is made in order to observe the resulting possible candidates. The second class uses *multiple design specifications*. Comparative analysis would be a common example of this class where two or more sets of designs (captured by design specifications) are compared. Scenarios 1, 3, and 6 from the previous section fall into this class.

In order to analyze the set of exploration scenarios afforded by the reference model, we first characterize the range of possible specifications. This, in turn, requires characterizing the possible decisions, constraints, and objectives. We achieve this goal using specification templates.

Table 4.1 contains the templates that express a subset of all possible design decisions that can be made when using the reference model and first-order logic. The parameters X and Y in each of the templates are names of elements such as features, functions, devices, etc., present in the model. In the templates we capture all possible presence variability for the the elements (DD1, DD3, DD4, DD8, DD9, DD11). Additionally, device nodes and bus connectors have associated types (DD7, DD14), functions have implementation decisions (DD5), and connectors have endpoints that can be variable (DD6, DD10, DD15). Lastly, the deployment can also be variable with respect to function and their connectors (DD2, DD12, DD13). This is a subset because we do not capture arbitrary Boolean formulas between the deployment of a set to another in this characterization; however, note that Clafer can express arbitrary Boolean expressions should a particular scenario need it. In Chapter 5 we show how it is possible to capture each of these variable design decisions in our implementation.

Table 4.2 contains the templates for a subset of all design constraints that can be made when using the reference model. The constraints that we consider are simple inequalities that involve individual values or sums of mass, part cost, part warranty cost (DC5 – DC13), and latency for end-to-end latency (DC1, DC3). Input synchronization (DC2, DC4) additionally use maximum of a set of values. The parameter X in the templates is a quality associated with an element such as a device node mass or the latency of a timing chain, and parameters Y and Z are numbers.

Table 4.3 then shows a subset of the possible design objectives that can be captured when using the reference model. Parameter X is a quality associated with an element, as in the previous table, and parameter Y is a number. Although Clafer allows using arbitrary numeric expressions as objectives, we only focus on the subset in the table. The objectives that we consider are additive with respect to the mass, part cost, and part warranty cost as well as timing margins. We define *total mass* as the summation of mass for all elements (that define mass). *Total part cost* and *total part warranty cost* are defined in a similar manner.

In order to combine the design decisions, constraints, and objectives together to form design specifications, we use a workflow diagram shown in Figure 4.2. The design decisions and constraints are joined together using Boolean operations. In the next subsection we show concretely, how these templates can be used to build a set of design specifications to

Table 4.1: A set of decision templates based on the reference architecture model

Reference Model Concept	**Design Decision Template**	**ID**
Features	Feature X <is|is not> present in the system	DD1
Functions & Connectors	Function X is deployed to device node(s) Y(1) or Y(2) or . . . Y(m)	DD2
	Function X <is|is not> present in the system	DD3
	Function connector X <is|is not> present in the system	DD4
	Function X is implemented in <hardware|software>	DD5
	Function connector X is provided <to|from> function(s) Y(1) or Y(2) or . . . Y(m)	DD6
Device Nodes	Device node X is <smart|electric/electronic|power>	DD7
	Device node X <is|is not> present in the system	DD8
Power Connectors	<Load|Device> power connector X <is|is not> present in the system	DD9
	The <load|device> power connector X is provided <to|from> device node(s) Y(1) or Y(2) or . . . Y(m)	DD10
Communication Connectors	<Bus|Discrete|Analog> connector X <is|is not> present in the system	DD11
	Function connector X does not use a hardware connector to communicate	DD12
	Function connector X uses connector(s) Y(1) or Y(2) or . . . Y(m) to communicate	DD13
	Bus Connector X is of type <LIN|Low Speed CAN|High Speed CAN|FlexRay>	DD14
	The <bus|discrete|analog> connector X should have device node(s) Y(1) or Y(2) or . . . Y(m) as endpoints	DD15

Table 4.2: A set of constraint templates based on the reference architecture model

Quality Attribute	**Design Constraint Template**	**ID**
Timing (End-to-End Latency)	The timing chain latency X must be less than or equal to Y	DC1
	The maximum difference for timing chain latencies X(1), X(2), . . . , X(n) must be less than or equal Y	DC2
Timing (Margin)	The margin between timing chain latency X and the required latency Y must be greater than or equal to Z	DC3
	For all margins between the timing chain latencies X(1), X(2), . . . X(n) and the requirement latencies Y(1), Y(2), . . . Y(n) the minimum must be greater than or equal to Z	DC4
Mass	The total mass of the system must be less than or equal to Y	DC5
	The mass of X must be less than or equal Y	DC6
	The sum of mass for components X(1), X(2), . . . X(n) must be less than or equal to Y	DC7
Part Cost	The total cost of the system must be less than or equal to Y	DC8
	The cost of X must be less than or equal to Y	DC9
	The sum of cost for components X(1), X(2), . . . X(n) must be less than or equal to Y	DC10
Warranty Part Cost	The total parts warranty cost of the system must be less than or equal to Y	DC11
	The parts warranty cost of X must be less than or equal to Y	DC12
	The sum of parts warranty cost for components X(1), X(2), . . . X(n) must be less than or equal to Y	DC13

Table 4.3: A set of the design objective templates based on the reference architecture model

Quality Attribute	**Template Design Objectives**	**ID**
Timing (margins)	<Maximize|Minimize> the margin between the timing chain X end-to-end latency and the requirement latency of Y	DO1
	<Maximize|Minimize> the <smallest|largest> margin between the timing chain(s) X(1), X(2), . . . X(n) end-to-end latency(s) and the requirement latency(s) Y(1), Y(2), . . . Y(n)	DO2
Mass	<Maximize|Minimize> the total mass of the system	DO3
Parts Cost	<Maximize|Minimize> the total cost of the system	DO4
Warranty Parts Cost	<Maximize|Minimize> the total parts warranty cost of the system	DO5

explore a concrete model constructed using the reference model.

4.3 Example Design Exploration using Templates

Returning to Emily's hypothetical design exploration scenarios in Section 4.1, we can now use the templates to form a set of specifications to be used in the exploration. As an example, we use scenario 4:

Emily is tasked with designing the power window for a higher end car in which cost is irrelevant but mass should be minimized; she would like to explore the possible designs. Additionally, since its a high end car, all features should be included. Lastly, the end-to-end latency for pinch detection to react and reverse the motor should be less than 200 ms.

This exploration requires just one design specification which includes one decision for wanting all features (i.e., having the feature `expressUp`), one constraint for the latency requirement, and one objective for minimizing the mass. Using template DD1, we can write the design decision as "Feature `expressUp` is present in the system", which we abbreviate with a shorthand notation as DD1(`expressUp`, is). This notation replaces the arguments inside the parentheses with the configurable parameters in the template, in the same order as they appear. The design constraint "The timing chain latency `PinchDetection_TC` must be less than or equal to 200 ms" can then be written as DC1(`PinchDetection_TC`, 200 ms) in

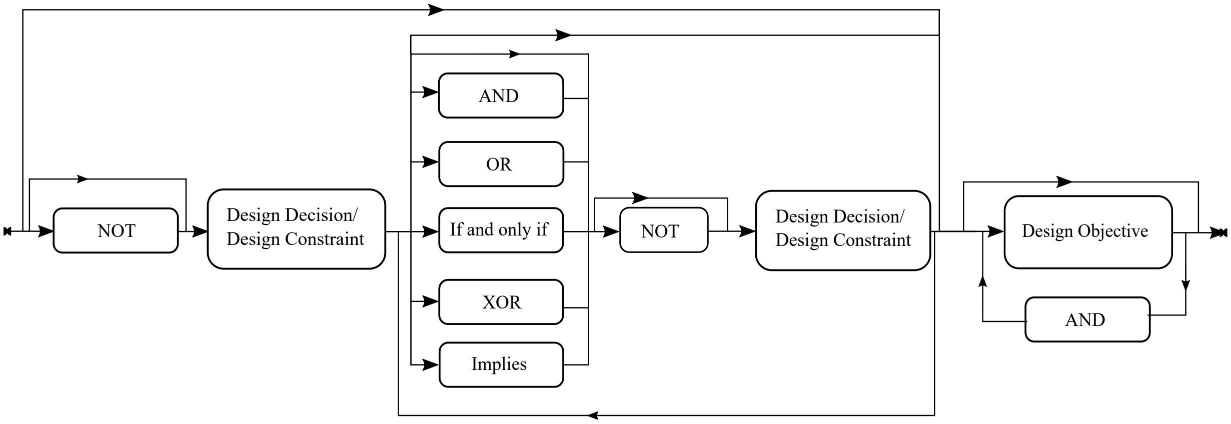


Figure 4.2: A workflow diagram for combining design decisions, constraints, and objectives together to create a design specification.

shorthand notation, where `PinchDetection_TC` is the latency for a timing chain constructed from the `PositionSensor` functional device to the `WinMotor` (Figure 3.8). Lastly, the design objective can be written as "Minimize the total mass of the system"; DO3(Minimize) in shorthand notation.

We construct the design specification by combining the instantiated templates use the flow diagram in Figure 11: "Feature `expressUp` is present in the system AND the timing chain latency `PinchDetection_TC` must be less than or equal to 200 ms AND Minimize the total mass of the system" or "DD1(`expressUp`, is) AND DC1(`PinchDetection_TC`, 200 ms) AND DO3(Minimize)" in shorthand notation.

In a different scenario, Emily may want the latency of pinch detection to be met only if the feature `expressUp` is present. This conditional design specification can be constructed using an implication between the two decisions as follows: "Feature `expressUp` is present in the system IMPLIES the timing chain latency `PinchDetection_TC` must be less than or equal to 200 ms" or "DD1(`expressUp`, is) IMPLIES DC1(`PinchDetection_TC`, 200 ms)" in shorthand notation.

Chapter 5

Encoding the Reference Model in Clafer

In the previous sections, we defined a reference model for early design and showed a subset of the possible design specifications that can be formed over a design-space model built using the reference model from Chapter 3. In this section, we present how to encode the reference model and *design-space models* (a model built using the reference model) using Clafer [4] in order to support the design specifications from Chapter 4.

Recall from Chapter 2, Clafer's language support for variability modeling using cardinalities and group cardinalities. Thus, when the reference model is encoded, there is no longer a separate variability model as there is with current EAST-ADL modeling tools. Thus, Figure 5.1 shows a unified view of the reference model diagram shown earlier. The unification symbolizes that the variability, and other perspectives are present in the Clafer encoding of each of the layers.

Next, we describe the implementation of each of the reference model layers in Clafer as well example use cases in the context of Emily's power window.

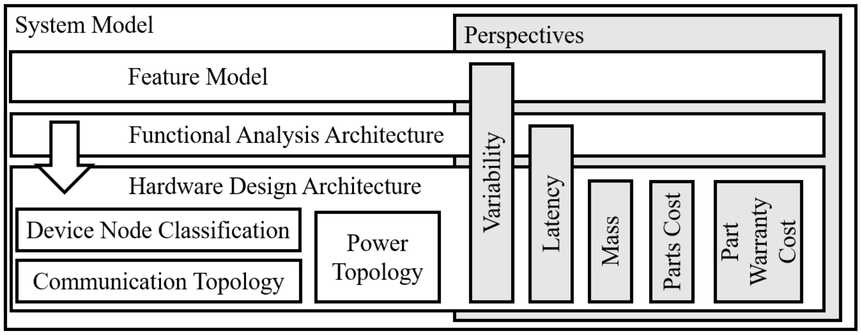


Figure 5.1: A unified view of the reference model used for early design of automotive E/E systems architecture

5.1 Encoding of Reference Model Layers in Clafer

The encoding for the feature model layer is shown in Listing 5.1. It defines two abstract clafers, **FeatureModel**[1] and **Feature**, for *feature model* and *feature* respectively. Neither clafer has quality attributes since they are not expressed at the feature model layer (as shown in Figure 5.1).

Listing 5.1: Clafer encoding of reference model concepts for the feature model layer

```
abstract FeatureModel
abstract Feature
```

Using these components, Emily can model her feature model (with variability) in Clafer as shown in Listing 5.2. With nesting, Emily is able to model the implication between the features `express` and `expressUp` and `expressDown`. Note, that anytime a cardinality is not shown it is assumed to be 1..1 by the compiler.

Listing 5.2: Power window feature model example

```
PowerWindowFM : FeatureModel
  basic : Feature
  express : Feature ?
    expressDown : Feature
    expressUp : Feature ?
```

By annotating features with some variable cardinality (such as, 0..1) we support design decision DD1 (Feature X <is|is not> present in the system).

[1] We use **bold typeface** to refer to a clafer in a listing

Encoding the FAA layer requires defining abstract clafers for analysis function, functional device, and function connector, which we define in Chapter 3. Listing 5.3 shows how we use inheritance, denoted by a colon (lines 12,13) to model common "properties" or attributes of analysis functions and functional devices in a common super clafer **`FunctionalAnalysisComponent`**. Design decisions DD3 and DD4 (Function/Function connector X <is|is not> present in the system) are possible, similar as with features, by using cardinality 0..1 (i.e., optional) when declaring the function or connector; this can be seen on line 8 of Listing 5.4, which is a fragment of Emily's power window FAA. For design decision DD5 (Function X is implemented in <hardware|software>), we capture the two alternatives using mutually exclusive group cardinality (`xor`) on line 4 of Listing 5.3. Note that in Clafer constraints are enclosed in brackets.

To capture the deployment of functions and function connectors to the hardware architecture, we need a relation between the two. This relation is captured using the reference **`deployedTo`** on lines 3 and 18 of Listing 5.3 (denoted by `->`). In Clafer, a reference can also be given a cardinality, which allows for a reference to point to a variable size set of targets. Furthermore, consider the constraint on line 4 of Listing 5.4; one can loosely restrict that the target of a reference must be *in* a set of possibilities. Thus, an instance of **`WinControl`** would be deployed to either **`DoorModule`** or **`Switch`**. This mechanism supports the design decision "Function X is deployed to device node(s) Y(1) or Y(2) or . . . Y(m)" (DD2). Further, since the endpoints of a function connector are references, it supports the decision "Function connector X is provided <to|from> function(s) Y(1) or Y(2) or . . . Y(m)" (DD6).

Additionally, constraints nested in the reference model concepts ensure that each use of the concept must satisfy the constraints (for a satisfiable instance). For example, the constraint on line 9 of Listing 5.3 enforces that functions implemented in software are deployed to a device that must have **`type`** (a child clafer of **`DeviceNode`** in Listing 5.5) must be *smart*. Note, the `.` operator navigates to the target of a reference or to a child of a clafer (see [15] for the full semantics).

In order to augment the FAA layer with the latency perspective, we explicitly define a latency property (captured by a child integer reference clafer) for functions and function connectors (lines 11 and 22 of Listing 5.3). Currently the Clafer backend solver is limited to working with integer values so all quality attributes have to be modeled using only integers. The constraints on lines 7, 10, and 24 along with the clafer **`baseLatency`** enforce the relationships between latency and hardware factors, as we described in Chapter 3. Recall that a designer only typically specifies the base latency for a function; however, since Clafer is a constraint language, the designer can also leave the base latency of one or

Listing 5.3: Clafer encoding of reference model concepts for the functional analysis architecture layer. Comments are denoted by "//" and the unit of the quality attribute is enclosed in the square brackets in the comments.

```
abstract FunctionalAnalysisArchitecture
abstract FunctionalAnalysisComponent
  deployedTo -> DeviceNode
  xor implementation
    hardware
      [deployedTo.type in (EEDeviceType, SmartDeviceType)]
      [latency = baseLatency]
    software
      [deployedTo.type in SmartDeviceType]
      [latency = baseLatency*deployedTo.speedFactor]
  latency -> integer // [ms]
  baseLatency -> integer // [ms]
abstract AnalysisFunction : FunctionalAnalysisComponent
abstract FunctionalDevice : FunctionalAnalysisComponent
abstract FunctionConnector
  sender -> FunctionalAnalysisComponent
  receiver -> FunctionalAnalysisComponent
  deployedTo -> HardwareDataConnector ?
    [parent in this.deployedFrom]
    [(sender.deployedTo.dref, receiver.deployedTo.dref) in (deployedTo.
        endpoint.dref)]
  [(sender.deployedTo.dref = receiver.deployedTo.dref) <=> no this.
      deployedTo]
  latency -> integer // [us]
  messageSize -> integer // [byte]
  [if (deployedTo) then (latency = messageSize*deployedTo.
      transferTimePerSize) else (latency = 0)]
```

Listing 5.4: Power window FAA example

```
PowerWindowFAA : FunctionalAnalysisArchitecture
  WinControl : AnalysisFunction
    [baseLatency = 2]
    [deployedTo in (PowerWindowDN.DoorModule, PowerWindowDN.Switch)]
  WinMotor : FunctionalDevice
    [latency = 10]
    [deployedTo = (PowerWindowDN.Motor)]
  PinchDetection : AnalysisFunction ?
    [baseLatency = 2]
    [deployedTo = (PowerWindowDN.DoorModule)]
  winCmd : FunctionConnector
    [sender = WinControl && receiver = WinMotor]
    [messageSize = 2]
    [deployedTo in (PowerWindowCT.logicalSwitchMotorDisc, PowerWindowCT.
       logicalMotorDoorModuleDisc)]
```

more functions opened and to be determined by the underlying solver.

The device node classification layer encoding is shown in Listing 5.5, where a device node has multiple properties to express mass, part cost, part warranty cost, and type. The integer properties explicitly augment the device node classification with the latency, mass, part cost, and part warranty cost. Design decision DD7 (Device node X <is|is not> present in the system) is afforded by using cardinality 0..1 (i.e., optional) when declaring the node. The target of reference **`type`** on line 6 of Listing 5.5 can be one (by the cardinality 1..1) of the types from the set **`DeviceNodeType`**. The set contains three possibilities: smart, electric/electronic, and power (see line 2 of Listing 5.5). This forms a mutually exclusive selection, identical to using an `xor` group cardinality as done for the implementation choice for a function; thus, supporting the design decision "Device node X is of type <smart|electric/electronic|power>" (DD7)

Mass, cost, ppm (part per million), replacement cost, and speed factor are all specified by the system designer. The part warranty cost is then constrained to be the number of failures per million (ppm) multiplied by the replacement cost, shown on line 10 of Listing 5.5.

Listing 5.6 shows part of the device node classification for Emily's power window, containing the nodes `Switch` and `DoorModule`. In order to show the variability that Emily has both a *smart* and *electric/electronic* `Switch`, we can use the constraint on line 3, which has identical form as a variable deployment constraint. Additionally, to express that the device node `DoorModule` is optional, a cardinality of 0..1 is expressed on line 9. Also take

Listing 5.5: Clafer encoding of reference model concepts for the device node classification.

```
1  abstract DeviceNodeClassification
2  enum DeviceNodeType = SmartDeviceType | EEDeviceType | PowerDeviceType
3  abstract DeviceNode
4    type -> DeviceNodeType
5    speedFactor -> integer // unitless
6    mass -> integer // [g]
7    cost -> integer // [dollar]
8    ppm -> integer // unitless
9    replaceCost -> integer // [dollar]
10   warrantyCost -> integer = ppm*replaceCost // [dollar per million]
11   [(type in (PowerDeviceType, EEDeviceType)) => (speedFactor = 0)]
```

note that quality attribute values do not have to be assigned to one possible value by the modeler: on line 7 it shows how the device node type can be used to select between two values for **ppm**.

For the power and communication topologies, we need an encoding for the different types of connectors. Similar to the FAA layer encoding, we use inheritance in Listings 5.7 and 5.9 to model the various connectors and capture similar properties.

Using constant integer values such as **Data.MassPer Length.LoadPowerConnector**, the total mass and cost of the different connectors can be computed given their length. Both the load power connector and device power connector define a source and sink reference, which allow the endpoints of the connectors to vary. Similar to before, these references allow design decision DD10 ("The <load|device> power connector X is provided <to|from> device node(s) Y(1) or Y(2) or . . . Y(m)"), and the presence variability can be captured via cardinalities, allowing for DD8 ("<Load|Device> power connector X <is|is not> present in the system").

Using the clafers from Listing 5.7, Emily can express a fragment of her power topology for the motor in Listing 5.8. The fragment shows the device power connector for the motor, that is optional, and is present if and only if the device node `Motor` is smart. It also shows the load power connector for the motor (the connector source is not shown).

Listing 5.9 shows the communication topology modeled in Clafer, which consists of multiple data connectors that we introduced in Chapter 3. Recall that the discrete and analog data connectors were abstractions; in order to improve the accuracy of cost and mass, the constraint on lines 7 and 9 shows how the number of function connectors deployed to a hardware connector is used to represent the number of connectors in the bundle. Similar to

Listing 5.6: Power window device node classification example.

```
PowerWindowDN : DeviceNodeClassification
  Switch : DeviceNode
    [type in (SmartDeviceType, EEDeviceType)]
    [mass = 173]
    [cost = 110]
    [replaceCost = 110]
    [if (type in SmartDeviceType) then (ppm = 50) else (ppm = 10)]
    [(type in SmartDeviceType) => (speedFactor = 10)]
  DoorModule : DeviceNode ?
    [type = SmartDeviceType]
    [mass = 368]
    [cost = 300]
    [ppm = 50]
    [replaceCost = 300]
    [speedFactor = 10]
```

Listing 5.7: Clafer encoding of reference model concepts for the device node classification.

```
abstract HardwareConnector
  length -> integer // [cm]
  mass -> integer // [mg]
  cost -> integer // [dollar per thousand]
abstract PowerConnector : HardwareConnector
  source -> DeviceNode
  sink -> DeviceNode
abstract LoadPowerConnector : PowerConnector
  [mass = Data.MassPerLength.LoadPowerConnector*length]
  [cost = Data.CostPerLength.LoadPowerConnector*length]
abstract DevicePowerConnector : PowerConnector
  [mass = Data.MassPerLength.DevicePowerConnector*length]
  [cost = Data.CostPerLength.DevicePowerConnector*length]
```

Listing 5.8: Example power topology for power window.

```
PowerWindowPT : PowerTopology
  MotorLoadPowerConnector : LoadPowerConnector
    [sink = PowerWindowDN.Motor]
  MotorDevicePowerConnector : DevicePowerConnector ?
    [source = PowerWindowDN.DoorInline]
    [sink = PowerWindowDN.Motor]
    [length = 45]
  [MotorDevicePowerConnector <=> (PowerWindowDN.Motor.type =
     SmartDeviceType)]

PowerWindowCT : CommunicationTopology
  logicalLowSpeedBus : BusConnector ?
    [type.LIN || type.LowSpeedCAN]
    [length = 70]
    [endpoint in (PowerWindowDN.Motor, PowerWindowDN.DoorModule)]

  logicalMotorDoorModuleDisc : DiscreteWireConnector ?
    [endpoint = (PowerWindowDN.Motor, PowerWindowDN.DoorModule)]
    [length = 30]
```

power and function connectors, the endpoints of a data connector are references; however, the encoding allows for two or more endpoints to support the design decision DD15. Further, we support four types of common buses, where each has a different mass per length, cost per length, and transmission rate. This encoding uses a group cardinality, similar to the function implementation encoding, allowing the design decision "Bus Connector X is of type <LIN|Low Speed CAN|High Speed CAN|FlexRay>" (DD14). Lastly, the function connector deployment is modeled similar to functions (DD13), with the exception that it can be optional, supporting design decision DD12.

Listing 5.10 shows a fragment of the communication topology for the power window. The fragment shows two alternatives, in which the device nodes `Motor` and `DoorModule` can be connected at the hardware level. The selection of the alternatives is based on the constraints on line 11 of Listing 5.9, which enforces that all device nodes used as endpoints for the bus connector have to be smart. Therefore, if the device node `Motor` is smart then the bus will be used, otherwise the discrete wire will be synthesized.

In the following subsection, we show an example for how element-level qualities can be aggregated to support the design constraints and objectives from Chapter 4. Additionally, we include an example for joining multiple decisions, constraints, and objectives together, Listing 5.9: Clafer encoding of reference model concepts for the communication topology

```
abstract HardwareDataConnector : HardwareConnector
  endpoint -> DeviceNode 2..*
  deployedFrom -> FunctionConnector 1..*
    [this.deployedTo = parent]
  transferTimePerSize -> integer // [us/byte]
abstract DiscreteDataConnector : HardwareDataConnector
  [mass = length*(#deployedFrom)*Data.MassPerLength.DiscreteDataConnector]
  [transferTimePerSize = 0]
  [cost = Data.CostPerLength.DiscreteDataConnector*length*(#deployedFrom)]
abstract BusConnector : HardwareDataConnector
  [all e : endpoint | e.type = SmartDeviceType]
  xor type
    LowSpeedCAN
      [transferTimePerSize = Data.TimePerSize.LowSpeedCANBus]
      [mass = Data.MassPerLength.LowSpeedCANBus*length]
      [cost = Data.CostPerLength.LowSpeedCANBus*length]
    HighSpeedCAN
      ...
    LIN
      ...
    FlexRay
      ...
abstract LogicalBusBridge
  transferTimePerSize -> integer // [us/byte]
  bus -> BusConnector 2
  gatewayTransferTimePerSize -> integer // [us/byte]
  [transferTimePerSize = gatewayTransferTimePerSize + bus.transferTimePerSize]
```

Listing 5.10: Example communication topology for power window.

```
PowerWindowCT : CommunicationTopology
  logicalLowSpeedBus : BusConnector ?
    [type.LIN || type.LowSpeedCAN]
    [length = 70]
    [endpoint in (PowerWindowDN.Motor, PowerWindowDN.DoorModule)]

  logicalMotorDoorModuleDisc : DiscreteWireConnector ?
    [endpoint = (PowerWindowDN.Motor, PowerWindowDN.DoorModule)]
    [length = 30]
```

in accordance to the earlier workflow diagram.

5.1.1 Using Clafer to Model a Simplified Power Window

Recall, from Chapter 4 that many of the design constraints and objectives were a summation of component values such as device node masses or function latencies. To showcase how these constraints and objectives can be written in Clafer, we simplify Emily's power window. We do so by showing the `WinControl` and `WinMotor` functions, with their connection, and two device nodes, `DoorModule` and `Motor`, as well as the possible media that connect them. Listing 5.11 shows the Clafer model for this simplified architecture.

Using the simplified model, we model a timing chain latency by adding the latency values of `WinControl`, `WinMotor`, and `winCmd` as shown on line 1 of Listing 5.12. Then using the timing chain, we encode the design constraint "The end-to-end latency for timing chain X must be less than or equal to Y" (DC1) on line 2 of Listing 5.12. Similarly, if we wanted to minimize the total mass of the architecture, as in DO3, we would need to sum up the device node and hardware connector masses. Line 3 shows how we can define a total mass, and line 4 then shows the Clafer encoding to minimize the value. Lastly, our example specification could state that we do not want the `WinControl` function to be deployed to the door module. This can be done using the quantifier `no`, as shown on line 5. The result of this specification would be an instance including only the motor, which is smart to contain the `WinControl` function, and no discrete data connector.

Listing 5.12: Example design specification encoding

```
1 exampleTC -> integer = WinControl.latency+winCmd.latency+WinMotor.latency
2 [exampleTC <= 34]
```

Listing 5.11: Simplified power window E/E architecture

```
WinControl : AnalysisFunction
  [implemented.software]
  [baseLatency = 2]
  [deployedTo in (DoorModule, Motor)]
WinMotor : FunctionalDevice
  [implemented.hardware]
  [baseLatency = 10]
  [deployedTo = Motor]
winCmd : FunctionConnector
  [sender = WinControl & receiver = WinMotor]
  [deployedTo in (doorModuleMotorDisc)]

DoorModule : DeviceNode ?
  [type = SmartDeviceType]
  [mass = 100]
  [cost = 100]
  [replaceCost = 100]
  [ppm = 50]
  [speedFactor = 10]
Motor : DeviceNode
  [type in (SmartDeviceType, EEDeviceType)]
  [mass = 100]
  [cost = 100]
  [replaceCost = 100]
  [ppm = 50]
  [(type in SmartDeviceType) => (speedFactor = 10)]

doorModuleMotorDisc : DiscreteDataConnector ?
  [endpoint = (Motor, DoorModule)]
  [length = 50]
```

```
totalMass -> integer = DoorModule.mass + Motor.mass + doorModuleMotorDisc
   .mass
<< minimize totalMass >>
[WinControl.deployedTo != DoorModule]
```

5.2 The Supporting Reasoner and Tools

Recall from Chapter 2 that *chocosolver* is a part of the Clafer toolchain. It compiles the model written in the Clafer modeling language down to a set of variables and constraints from the Choco constraint programming library [55]. Constraint programming is a powerful paradigm from artificial intelligence for performing search efficiently over a large search space. The Choco library is a notable open source constraint programming implementation and has won awards in the past three MiniZinc challenges (2013, 2014, 2015). Chocosolver extends the Choco library to handle Clafer-specific features, such as reasoning over relational logic and multi-objective optimization, in which the guided improvement algorithm (GIA) [33] is implemented. Additionally, any solution in the Choco domain gets mapped back to a solution in the Clafer domain.

Chocosolver explores the search space for solutions by incrementally assigning variables in hope of finding a complete solution. After each assignment, the solver will perform propagation by using the currently assigned variables to deduce the values of other variables, thus pruning the search space. If the solver detects that the current assignment of variables precludes all complete solutions, it will backtrack to an earlier state by unassigning some of the variables. This systematic backtracking search can be done efficiently by a combination of optimized data structures, efficient propagation of constraints, and effective search heuristics.

The solutions, or candidate architectures in our context that chocosolver finds can then be visualized using Clafer Web Tools [13, 50], another part of the Clafer toolchain. Clafer Multiple-Objective Optimization (MOO) Visualizer is one of the web tools that allows for visualizing optimal candidates using *bubble front charts* – a multi-dimensional bubble chart for visualizing up to four quality objectives simultaneously – and *parallel coordinate charts* – a visualization in which each instance is represented by a polyline connecting parallel quality objective axes. Additionally, the candidates can be filtered using the feature quality matrix in order to visualize only ones that contain certain features, functions, deployments, etc.

Chapter 6

Modeling Two Case Studies in Clafer

6.1 Power Window

The first case study we present in this thesis is the E/E architecture for a power window system in a car. First a single driver side door system is presented and then a second system for the front passenger side door power window is added with communication between the two. This case study has been developed and presented in two previous works, one by Murashkin [49] and one by Akhtar [8]. The case study presented in this work is an extension of the former while the latter uses the case study for a comprehensive design analysis, not just exploration. Further details for how we extend Murashkin's earlier work is discussed in the related work (Chapter 9).

The reason for choosing the power window as a case study is that it was self-contained and not overly complex. The material for the architecture designs and sources of variability was obtained through publicly available service manuals from companies such as Nissan, Infinity, BMW, and GM. The quality attribute information for cost and mass was obtained from OEM part supply websites and Amazon.com. Latency quality attributes were mainly created artificially using typical values as a base line (based on domain expertise). Lastly, the reliability attributes (ppm) were obtained from standard values and handbook calculations. In the last part of this section, we give details on how the values were formatted as inputs to the model.

6.1.1 Single Door: Driver

Feature Model

The features that are considered for the single door power window are as follows:

- `Basic Up/Down` The basic operation of the power window. When the switch is held in the up position the window retracts until closed or release of the switch. There is an identical reverse operation for holding the switch down.
- `Express` The express *down* feature in which when a user pushes the button down into an express position, the window opens until the window is completely open without the user having to continue pressing down the switch.
- `Express Up` The express *up* feature is similar to that of the express down except for the reverse operation. Additionally, if an object is detected in the window travel path then the window should stop and retract to be fully open again.

Figure 6.1 is the feature model for the system, note that for `expressUp` to exist the feature `express` must be present. This relationship can be captured using nesting in Clafer and is highlighted in Listing 6.1. Additionally, using clafer multiplicities the features `express` and `expressUp` are made optional.

Listing 6.1: Clafer feature model for single door power window

```
DWinSysFM : FeatureModel
  basicUpDown : Feature
  express : Feature ?
    expressUp : Feature ?
```

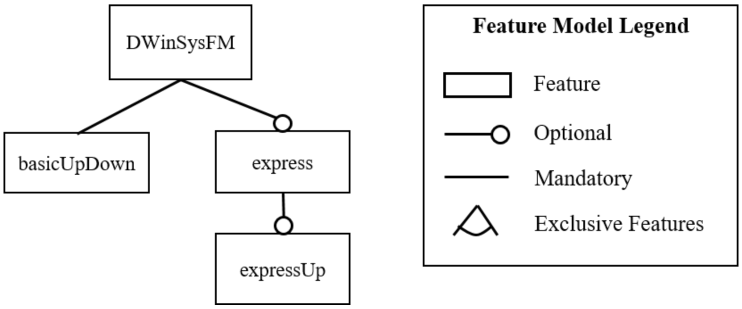


Figure 6.1: The feature model for a single door power window

Functional Analysis Architecture

The functional analysis architecture for the power window needs to support two sets of functions, one for the basic operation and one for detecting obstacles for the express up feature. Figure 6.2 shows the FAA for the system using a graphical domain-specific language (the legend is shown in Figure 6.3) that can be translated to Clafer.

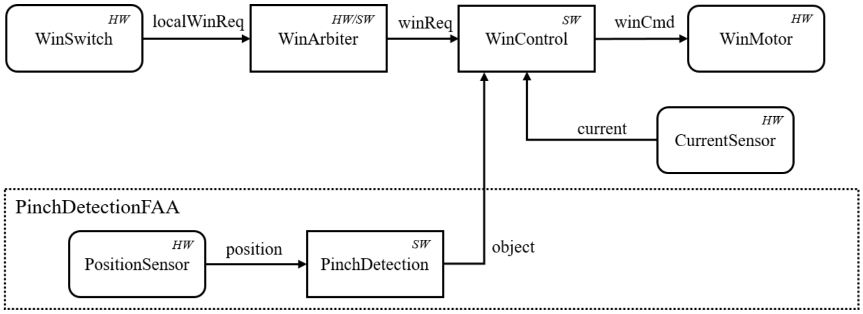
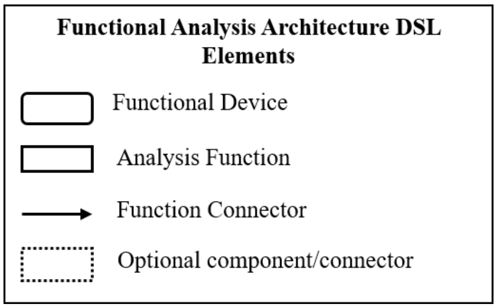


Figure 6.2: The functional analysis architecture for a single door power window. The "HW" or "SW" in upper right corner of a function indicates the implementation choice in hardware or software respectively. "HW/SW" indicates the function can be implemented in either software or hardware.



Functional Analysis Architecture DSL Elements
Functional Device
Analysis Function
Function Connector
Optional component/connector

Figure 6.3: Legend for the graphical domain-specific language (DSL) for describing the FAA for E/E architectures

The functions present in the FAA and their allowed implementations are as follows:

- `WinSwitch`: A sensor functional device that reads the switch position requested by the user. *Allowed implementation(s): hardware*
- `WinArbiter`: An analysis function that arbitrates which incoming signal should be sent to the controller. For the driver window this component is not needed since

only one input is present, however we still model it. *Allowed implementation(s): hardware, software*

- `WinControl`: The main control logic of the power window. It takes the switch position request, the current value from the motor and if an object is present (if pinch detection is required) and translates it to a command to send to the motor. *Allowed implementation(s): software*
- `WinMotor`: The actuator that takes the desired command from the controller and translates it to moving the motor to close and open the window. *Allowed implementation(s): hardware*
- `CurrentSensor`: A sensor placed on the motor to measure the current being pulled by the motor. This is used to ensure that the motor does not "stall" by continuing to move in a direction in which it can no longer move (i.e. the window is fully open or closed). *Allowed implementation(s): hardware*
- `PositionSensor`: A sensor to detect the current position of the window in the path of travel. *Allowed implementation(s): hardware*
- `PinchDetection`: Another piece of control logic that takes the position sensor reading and determines if an object is present or not. *Allowed implementation(s): software*

The `PinchDetectionFAA` is a functional analysis architecture nested inside the driver power window FAA and it is optional (since the express up feature is). By putting the `PositionSensor` and `PinchDetection` and the two function connectors in the FAA it allows for those components to be removed when `Pinch DetectionFAA` is not present.

Listing 6.2 shows a fragment of the FAA encoded using Clafer for the single door power window. Note the use of nesting for the components inside of `PinchDetectionFAA`.

Listing 6.2: Snippet of Clafer model for single door functional analysis architecture

```
DWinSysFAA : FunctionalAnalysisArchitecture
  WinSwitch : FunctionalDevice
    [implementation.hardware]
    [baseLatency = 20]
  WinController : AnalysisFunction
    [implementation.software]
    [baseLatency = 2]
  WinMotor : FunctionalDevice
    [implementation.hardware]
    [baseLatency = 10]
  ...
```

```
12   winReq : FunctionConnector
13     [sender = WinArbiter && receiver = WinController]
14     [messageSize = 1]
15   winCmd : FunctionConnector
16     [sender = WinController && receiver = WinMotor]
17     [messageSize = 2]
18   PinchDetectionFAA : FunctionalAnalysisArchitecture ?
19     PinchDetection : AnalysisFunction
20       [implementation.software]
21       [baseLatency = 2]
22     PositionSensor : FunctionalDevice
23       [implementation.hardware]
24       [baseLatency = 10]
25     object : FunctionConnector
26       [sender = PinchDetection && receiver = WinController]
27       [messageSize = 2]
28     ...
29   [DWinSysFAA.PinchDetectionFAA <=> DWinSysFM.express.expressUp]
```

The constraint on line 29 of Listing 6.2 expresses equivalence between the `PinchDetectionFAA` and the feature `expressUp` in the feature model, which ensures that pinch detection functionality is present if and only if the feature express up is present.

Device Node Classification

The device node classification consists of local nodes, which belong to the system, and remote nodes which are shared among many systems. The local nodes that belong to the driver door power window system are:

- `Switch`: The physical switch sensor that is present on the door for the driver to control their window. *Allowed type(s): smart, electric/electronic*
- `Motor`: The motor that moves the window armature to open and close the window. *Allowed type(s): smart, electric/electronic*
- `Door Module`: An optional ECU that is housed inside the door. *Allowed type(s): smart*
- `Door Inline`: An interconnect that connects the wiring from the main body harness to the door harness. *Allowed type(s): power*

The remote nodes in the system are:

- `BCM` (Body Control Module): The main ECU that houses much of the body control software. *Allowed type(s): smart*
- `EC` (Electric Center): The main fuse box that is the primary source of power *Allowed type(s): power*

A fragment of the device node classification encoding in Clafer is shown in Listing 6.3. In order to model the shared components, a clafer `Car` is defined which houses any remote components to the model. Then, the concrete `BCM` and `EC` are declared in `Car`. Inside the definition of the device node classification for the driver system, two references are defined to point to the central components. Using a reference instead of local declaration represents the fact that the driver device node classification has to be able to access the BCM and EC; however, since the BCM reference is optional, the driver system might not need it.

Listing 6.3: Snippet of Clafer model for single door device node classification

```
abstract SwitchNode : DeviceNode
  numSwitches -> integer

DWinSysDN : DeviceNodeClassification
  BCM -> DeviceNode ?
  EC -> DeviceNode
  Switch : DeviceNode
    [type in (SmartDeviceType, EEDeviceType)]
    [mass = 173]
    [cost = 110]
    [replaceCost = 110]
    [if (type in SmartDeviceType) then (ppm = 50) else (ppm = 10)]
    [(type in SmartDeviceType) => (speedFactor = 10)]
    [numSwitches = 2]
  Motor : DeviceNode
    [type in (SmartDeviceType, EEDeviceType)]
  ...
  DoorInline : DeviceNode
    [type = PowerDeviceType]
  ...
  DoorModule : DeviceNode ?
    [type = SmartDeviceType]
  ...
  [BCM = Car.BCM]
  [EC = Car.EC]

Car
  BCM : DeviceNode ?
```

```
    [type = SmartDeviceType]
    ...
  EC : DeviceNode
    [type = PowerDeviceType]
    ...
```

Listing 6.3 also contains a sub-type of `DeviceNode`, namely `SwitchNode`. This subtype allows the modeling of a general type of switch panel on a car and denotes the number of switches on the panel. The number of switches could have been captured by multiplicities but the solver exhibits a performance slow down.

Power Topology

The power topology for the power window models two types of power, load and device. The device power topology is quite straight forward; if a device node is smart then it must have a connection from the EC to the device node using a device power connector. The topology is modeled by defining optional power connectors from the EC to the door inline and then from the door inline to the respective nodes. Figure 6.4 shows the power topology for the single door window and the two sets of power connectors. Figure 6.5 is a legend for understanding the graphical symbols used to model the hardware architecture elements.

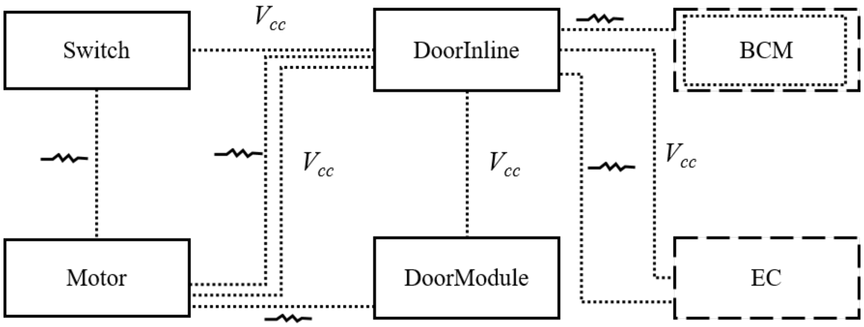


Figure 6.4: The power topology for a single door power window. The inside dotted box for the BCM denote that it is an optional remote device node.

In order to have the correct device power connectors present in the architecture the following rules are applied:

- **Rule 1:** A device power connector between a local device node and the door inline must exist if the device node is smart.

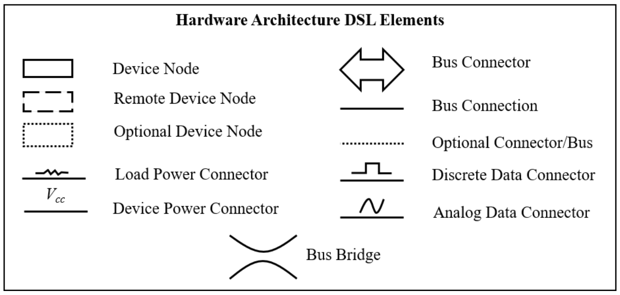


Figure 6.5: Legend for the graphical domain specific language (DSL) for describing the hardware architecture for E/E architectures.

- **Rule 2:** A device power connector between the EC and the door inline must exist if there is at least one device power connector leaving the door inline.

The second power connector that is of interest is the load power connector. Load power is required by the motor in order to move the window glass. The configuration of the load power is driven by the deployment of the `WinControl` analysis function. This is because the `WinControl` function acts as a driver for the motor. Therefore, a load power connector must be defined from each of the device nodes that can be smart to the motor (as seen in Figure 6.4). Secondly, the load power must get from the EC to the device node that has the `WinControl` function deployed. This results in four concrete topologies for the load power which can be described by the following rules:

- **Rule 3:** If the control is deployed to the motor then a load power connector must be present from the door inline to the motor and the EC to the door inline.
- **Rule 4:** If the control is deployed to the switch or door module, then a load power connector must exist from the door inline to the local device node, from the local device node to the motor, and the EC to the door inline.
- **Rule 5:** If control is deployed to the BCM then a load power connector must exist from the door inline to the motor, BCM to the door inline, and EC to the BCM.

Listing 6.4 shows a fragment of the power topology encoded in Clafer with the quality attribute values as well. Comments are used to show which constraint capture what rules for defining the allowed configurations. A group cardinality `xor` is used in order to model the exclusive configurations outlined in rules 3, 4, and 5.

Listing 6.4: Snippet of Clafer model for single door power topology

```
DWinSysPT : PowerTopology
```

```
dn -> DWinSysDN
MotorLoadPowerWire : LoadPowerConnector
  [sink = dn.Motor]
SwitchLoadPowerWire : LoadPowerConnector ?
  [source = dn.DoorInline && sink = dn.Switch]
  [length = 45]
DoorModuleLoadPowerWire : LoadPowerConnector ?
  [source = dn.DoorInline && sink = dn.DoorModule]
  [length = 35]
DoorInlineLoadPowerWire : LoadPowerConnector
  [sink = dn.DoorInline]

xor MotorLoadPowerConfig
  SwitchIsMotorDriver //R4
    [MotorLoadPowerWire.source = dn.Switch]
    [MotorLoadPowerWire.length = 40]
    [DoorInlineLoadPowerWire.source = dn.EC.dref]
    [DoorInlineLoadPowerWire.length = 40]
    [SwitchLoadPowerWire && DoorInlineLoadPowerWire && no
       DoorModuleLoadPowerWire]
  DoorModuleIsMotorDriver //R4
    ...
  BCMIsMotorDriver //R5
    ...
  MotorIsMotorDriver //R3
    ...

switchInlineDP : DevicePowerConnector ?
  [source = dn.DoorInline && sink = dn.Switch]
  [length = 45]
motorInlineDP : DevicePowerConnector ?
  ...
[switchInlineDP <=> (dn.Switch.type in SmartDeviceType)] //R1
[motorInlineDP <=> (dn.Motor.type in SmartDeviceType)] //R1
[ha.pt.inlineECDP <=> some(motorInlineDP, switchInlineDP,
   doorModuleInlineDP)] //R2
```

Communication Topology

The communication topology contains two mediums for communication, a bus and discrete connectors. For the driver door power window there is a single bus that allows for communication between the BCM and the local device nodes. The discrete connectors, as stated in the reference model, represent bundles of wires between two nodes. Unlike the power topology, a single connector is modeled between two communicating nodes so the door inline does not play any part in routing the connectors and it is abstracted away. Making this assumption allows for simpler models which accommodates early design.

Figure 6.6 shows the communication topology for the power window. Note that there is no possible discrete connector between the door module and BCM due to the assumption made that if a device is smart (when both are) they must use the bus.

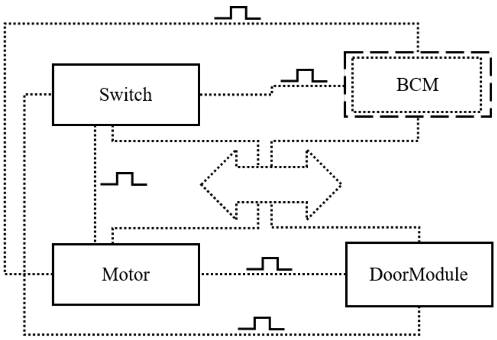


Figure 6.6: The communication topology for a single door power window

Listing 6.5 shows a fragment of the corresponding encoding in Clafer along with the quality attributes. The endpoints constraint on line 7 restricts the possible device nodes that *can be* connected to the bus. The endpoint constraints for the discrete wires explicitly gives the two endpoints for the connector such that the solver along with the constraint on line 16 in Listing 5.3 can synthesize the correct connectors needed. The deployment rules for the function connectors to the communication topology is then shown in the following section.

Listing 6.5: Snippet of Clafer model for single door communication topology

```
1  DWinSysCT : CommTopology
2    dn -> DWinSysDN
3
4    logicalLowSpeedBus : BusConnector ?
5      [type.LIN || type.LowSpeedCAN]
6      [length = 70]
7      [endpoint in (dn.Motor, dn.Switch, dn.DoorModule, dn.BCM.dref)]
8
9    logicalSwitchMotorDisc : DiscreteDataConnector ?
10     [length = 40]
11     [endpoint = (dn.Switch, dn.Motor)]
12
13   logicalSwitchBCMDisc: DiscreteDataConnector ?
14     ...
```

Deployment

To complete the single door power window case study the last part needing to be modeled is the deployment of the functional analysis architecture onto the hardware architecture. The deployment consists of only rules (or constraints) that details what the allowed mappings of functions to nodes and function connectors to mediums. The rules that need to be modeled are as follows:

- **Rule 1:** The analysis functions can be deployed to any of the local or central nodes.
- **Rule 2:** The `WinMotor`, `CurrentSensor`, and `Position Sensor` must be deployed to the `Motor`.
- **Rule 3:** The function connectors can be deployed to any of the possible discrete connectors or the bus.

The rules can then be encoded using constraints in Clafer; a fragment of the model is shown in Listing 6.6. Additionally on lines 12-15 the power topology configuration is constrained based on the deployment of the controller as detailed earlier.

Note the use of implication "=>" in the constraints pertaining to the pinch detection elements. This is such that the deployment constraints do not restrict that the pinch detection FAA must be present (these constraints are conditional on the presence of the pinch detection FAA).

6.1.2 Two Door: Driver & Front Passenger

In this section, the previous system is scaled up to a two door system. When building the second system, we observed that the passenger door was very close in structure to the driver. So in the first subsection this commonality is addressed and abstract clafers are used to generalize the core elements. The following section then gives the specifics for how the case study extends the common core to model the passenger door.

Generalizing the Core Elements

The passenger door is almost identical to the driver door with the exception of an added function and some additional communication. Thus, the generalized elements turn out to be the same as the driver system. Starting with the feature model, the Clafer model in

Listing 6.6: Snippet of Clafer model for single door deployment

```
DWinSysDpl : Deployment
  fa -> DWinSysFAA
  ha -> DWinSysHA

  [fa.WinArbiter.deployedTo.dref in (ha.dn.BCM.dref, ha.dn.Switch, ha.dn.Motor, ha.dn.DoorModule)]
  [fa.PinchDetectionFAA => (fa.PinchDetectionFAA.PinchDetection.deployedTo.dref in (ha.dn.BCM.dref, ha.dn.Switch, ha.dn.Motor, ha.dn.DoorModule))]
  [fa.WinSwitch.deployedTo.dref = ha.dn.Switch]
  [fa.PinchDetectionFAA => (fa.PinchDetectionFAA.PositionSensor.deployedTo.dref = ha.dn.Motor)]

  [(fa.WinController.deployedTo.dref = ha.dn.Switch) <=> ha.pt.MotorLoadPowerConfig.SwitchIsMotorDriver]
  [(fa.WinController.deployedTo.dref = ha.dn.Motor) <=> ha.pt.MotorLoadPowerConfig.MotorIsMotorDriver]
  [(fa.WinController.deployedTo.dref = ha.dn.BCM.dref) <=> ha.pt.MotorLoadPowerConfig.BCMIsMotorDriver]
  [(fa.WinController.deployedTo.dref = ha.dn.DoorModule) <=> ha.pt.MotorLoadPowerConfig.DoorModuleIsMotorDriver]

  [(fa.localWinReq.deployedTo.dref in (ha.ct.logicalLowSpeedBus, ha.ct.logicalSwitchMotorDisc, ha.ct.logicalSwitchBCMDisc, ha.ct.logicalMotorBCMDisc, ha.ct.logicalSwitchDoorModuleDisc, ha.ct.logicalMotorDoorModuleDisc))]
  ...
```

Listing 6.1 can be modified to be abstract and have two separate concretizations of it, as shown in Listing 6.7.

Listing 6.7: Generalized feature model and two concretizations

```
abstract WinSysFM : FeatureModel
  basicUpDown : Feature
  express : Feature ?
    expressUp : Feature ?

DWinSysFM : WinSysFM
PWinSysFM : WinSysFM
```

The same can be done for the remaining layers presented in the single door power window.

Listing A.2 in Appendix A.2 shows the full generalized architecture.

Extending for the Passenger System

The passenger door system requires some extensions to the base model which are not present in the driver door system. Starting with the feature model, it only makes sense for the passenger to have less features than the driver. The following rules can be derived then:

- **Rule 1:** The passenger should not have the feature `express` if the driver does not have it.
- **Rule 2:** The passenger should not have the feature `expressUp` if the driver does not have it.

These rules turn into constraints captured in the specialization of the passenger feature model in Listing 6.8.

Listing 6.8: Clafer feature model for two door system

```
DWinSysFM : WinSysFM
PWinSysFM : WinSysFM
  [express => DWinSysFM.express]
  [express.expressUp => DWinSysFM.express.expressUp]
```

The functional analysis architecture for the passenger system needs to include the additional switch and connector which models the driver side switch controlling the passenger window. Since the generalized FAA contains all other functionality, the only thing the passenger concretization needs is a functional device to represent the driver side switch and a function connector from the switch to the arbiter. Figure 6.7 shows such architecture and the corresponding Clafer model is shown in Listing 6.9. Observe that the arbiter can now always prefer the input from the driver-side switch, which will override the passenger's input.

Listing 6.9: Clafer functional analysis architecture for a two door system

```
DWinSysFAA : WinSysFAA
  [DriverWinSys.DWinSysFM.express.expressUp <=> DWinSysFAA.
     PinchDetectionFAA]

PWinSysFAA : WinSysFAa
  [PassengerWinSys.PWinSysFM.express.expressUp <=> PWinSysFAA.
     PinchDetectionFAA]
```

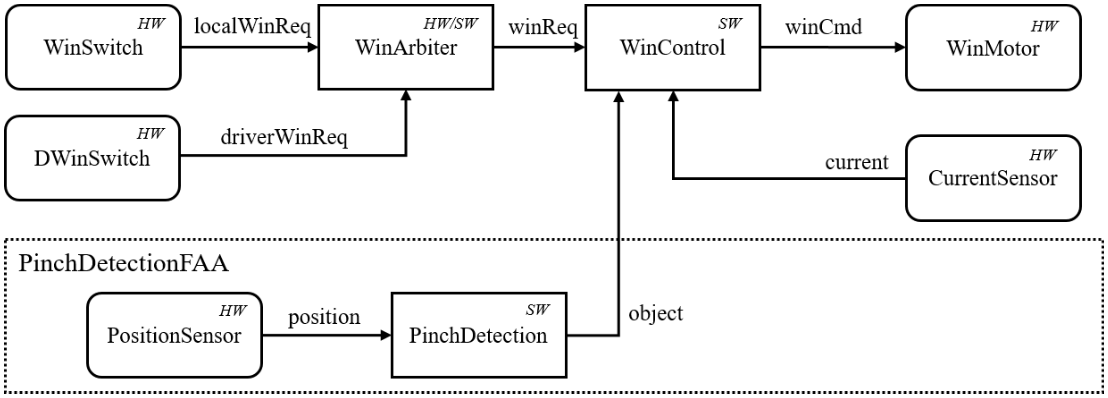


Figure 6.7: The functional analysis architecture for a passenger door power window system

```
DWinSwitch : FunctionalDevice
  [deployedTo.hardware]
  [baseLatency = 10]
driverWinReq : FunctionConnector
  [sender = DWinSwitch && receiver = WinArbiter]
  [messageSize = 1]
```

The only difference for the passenger device node classification is an addition of a reference to the driver side device node `Switch`. This is shown on lines 113 and 114 of Listing A.3 in Appendix A.3.

For the power topology, there are no extensions for either system from the generalized topology. However, the communication topology requires some additional connectors which is shown in Figure 6.8.

Listing 6.10 shows the concretization of the two common communication topologies and the extension for the passenger. Note the use of a logical bus bridge in order to model the communication between the passenger local bus and the driver local bus.

The added communication topology and additional functional device requires the deployment for the passenger system to be extended as well. Listing 6.11 gives a fragment of the Clafer model for the two deployments.

Listing 6.10: Clafer communication topology for two door system

```
DWinSysCT : WinSysCT
  ...
PWinSysCT : WinSysCT
  logicalDoorBusBridge : LogicalBusBridge ?
    [bus = (PWinSysCT.logicalLowSpeedBus, DWinSysCT.logicalLowSpeedBus)]
    [gatewayTransferTimePerSize = 10]
    [endpoint in (PWinSysDN.Motor, PWinSysDN.Switch, PWinSysDN.DoorModule
       , PWinSysDN.BCM.dref, DWinSysDN.Motor, DWinSysDN.Switch,
       DWinSysDN.DoorModule)]
  logicalDriveSwitchPassSwitch : DiscreteDataConnector ?
    [length = 260]
    [endpoint = (PWinSysHA.PWinSysDN.DSwitch.dref, PWinSysHA.PWinSysDN.
       Switch)]
  logicalDriveSwitchPassMotor : DiscreteDataConnector ?
    [length = 260]
    [endpoint = (PWinSysHA.PWinSysDN.DSwitch.dref, PWinSysHA.PWinSysDN.
       Motor)]
  logicalDriveSwitchPassDoorModule : DiscreteDataConnector ?
    [length = 250]
    [endpoint = (PWinSysHA.PWinSysDN.DSwitch.dref, PWinSysHA.PWinSysDN.
       DoorModule)]
  logicalDriveSwitchBCM : DiscreteDataConnector ?
    [length = 85]
    [endpoint = (PWinSysHA.PWinSysDN.DSwitch.dref, PWinSysHA.PWinSysDN.
       BCM.dref)]
  ...
```

Listing 6.11: Clafer functional analysis architecture for two door system

```
DWinSysDpl : WinSysDpl
  ...
PWinSysDpl : WinSysDpl
  [PWinSysFA.DWinSwitch.deployedTo.dref = PWinSysHA.PWinSysDN.DSwitch.
     dref]
  [PWinSysFA.dWinReq.deployedTo.dref in (
    PWinSysHA.PWinSysCT.logicalDoorBusBridge,
    PWinSysHA.PWinSysCT.logicalDriveSwitchPassSwitch,
    PWinSysHA.PWinSysCT.logicalDriveSwitchPassMotor,
    PWinSysHA.PWinSysCT.logicalDriveSwitchPassDoorModule,
    PWinSysHA.PWinSysCT.logicalDriveSwitchBCM)]
  ...
```

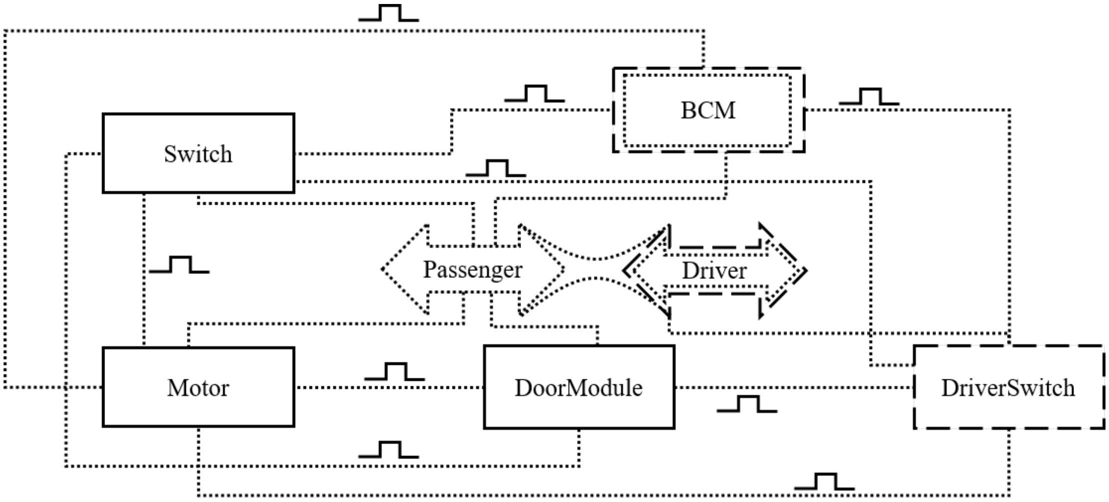


Figure 6.8: The communication topology for the passenger door power window system. The “Driver” bus is shown as a reference for the bridge between the two buses. It is assumed that any node connected to the “Passenger” bus can use the bridge to send a message over the bridge to the “Driver" bus.

6.1.3 Quality Attributes & Timing Analysis

Normalizing the Quality Attributes

As stated earlier, Clafer and *chocosolver* can currently reason over integers only. Therefore the gathered values for mass, cost, etc. must be scaled and rounded to provide meaningful numerical results while working with integers.

The first step is to choose some unit base for the different qualities such as grams for mass. Next, all the values are observed in the converted unit base and a check is made for any values that are less than 1. If any such values do exist then they must be scaled upwards in order to increase precision. For example, if a transfer rate was $0.0001 ms/byte$ then it could be converted to $1 \mu s/byte$. Then, any other qualities that are summed with the new unit base would need to be converted as well (i.e. all of the data connection latencies due to buses). If a unit is not scalable such as dollars, one can change the unit to be *dollars per thousand*.

An issue that arises with working with *chocosolver* is that the total of any summation or multiplication can not overflow. The overflow can happen in one of two ways:

- The result of a computation can not exceed the bounds set to be used by chocosolver. This is a result of using a CSP-based solver.
- The bounds set for the use of the chocosolver can't exceed the 32-bit integer maximum for any arithmetic expression.

Thus, care needs to be expressed when changing the base of numbers such that the numbers do not become too large to work with.

End-to-End Latency Constraints

For both power window systems there exist two end-to-end latency constraints that are of interest to model and constrain; they are as follows:

- **TR1:** The time it takes for the switch to read the pressed value to the time the motor begins actuation must be less than 750ms.
- **TR2:** The time it takes for the position sensor to read a value to the time the motor begins actuation must be less than 500ms.
- **TR3:** The time it takes the switch and position sensor to read the data and be sent to the control must be synchronized to less than 50ms.

The requirements are captured by creating two *timing chains*, one from `WinSwitch` to `WinMotor` and another from `PositionSensor` to `WinMotor`. These chains are captured in lines 12-26 for the driver power window in Listing A.3 in Appendix A.3. The requirements are then modeled in lines 29-32 of the same Listing.

For TR3 a input synchronization constraint must be modeled using Clafer which the fragment of interest is shown in Listing 6.12. The use of the set `min` and `max` are use to get the smallest and largest latencies from the three chains.

Listing 6.12: Snippet of Clafer to show the encoding of TR4.

```
ControlInputDifference -> integer
[ControlInputDifference = (max(SwitchToControlLatency.dref,
   PositionSensorToControlLatency.dref)
  - min(SwitchToControlLatency.dref, PositionSensorToControlLatency.dref)
     )]
```

Also of interest to engineers is the margin that exists between the end-to-end latency requirement and the actual end-to-end latency. This is a good optimization parameter when optimizing latency as it increases the robustness of the system when the margin is maximized by allowing more room for error. Lines 35-41 capture the margins for the driver power window in Listing A.3.

6.2 Door Locks

The second case study we present in this report is a E/E architecture for a central door locks system. Only the locking control for the driver and three passenger doors was considered; not the trunk or fuel lid. In this case study, we considered features such as remote key access (where a remote control is used to unlock and lock the car) and passive key entry (where a key fob is used to lock and unlock without touching the remote).

We chose this system in order to build on the power window by modeling a second system in the body domain. With two such systems, future work can be done in incorporating the two systems together and exploring trade-offs when sharing components.

Similar to the power window case study, the material was gathered from OEM service manuals such as GM, BMW, and Nissan. The domain knowledge for the passive key entry was obtained from various articles and suppliers.

6.2.1 Feature Model

The door locks system contains many more features than the power window which is expected since it is more complex. Figure 6.9 shows the feature model for the door locks system and the variability that exists. The feature model layer alone encodes 16 variants of the system.

The description for the features is as follows:

- `Basic:` The basic operation of the door locks system using the inside lock switch and cylinder key switch. It also includes unlocking all doors when the car is in park.
- `Speed Smart Lock:` The feature that will lock the car when a certain threshold speed has been reached.
- `Lock Switch Position:` The feature that dictates where the inside lock switch will be located. The possibilities are a lock switch on the driver and front passenger door or a shared switch in the center (i.e., the console).

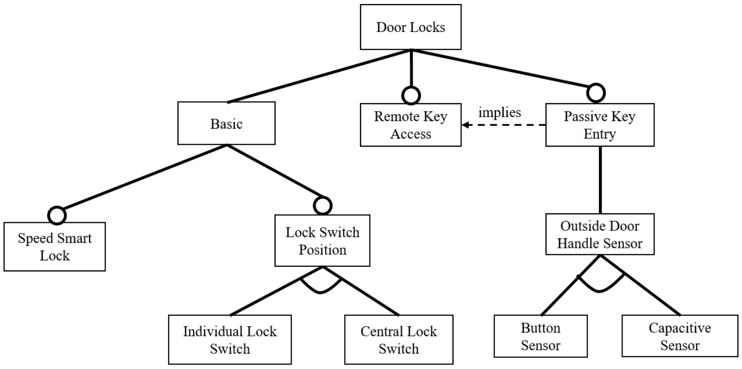


Figure 6.9: The feature model for a central door locks system

- `Remote Key Access (RKA)`: The feature that allows a car to unlock and lock when a remote control button is pressed.
- `Passive Key Entry (PKE)`: The feature that allows a car to unlock/lock when the door handle is touched/button pressed without touching the key fob.
- `Outside Door Handle Sensor`: The feature for what type of sensor is used to detect that the user wishes to lock/unlock the car using PKE. The capacitive sensor is a touch device that detects when a user has grabbed/touched the door handle. The button sensor is a physical push button placed on the outside door handle of the car.

Listing 6.13 shows the feature model encoded in Clafer. Note that instead of using an `xor` grouping for the `LockPositionSwitch`, a cardinality 0..1 was chosen to reduce the feature model size.

Listing 6.13: Clafer feature model for door locks

```
DLockFM : FeatureModel
  Basic : Feature
    IndividualLockSwitch : Feature ?
    SpeedSmartLock : Feature ?
  RKA : Feature ? // Remote Key Access
  PKE : Feature ? // Passive Key Entry
    xor OutsideDoorHandleSensor
      ButtonSensor : Feature
      CapacitiveSensor : Feature
  [PKE => RKA]
```

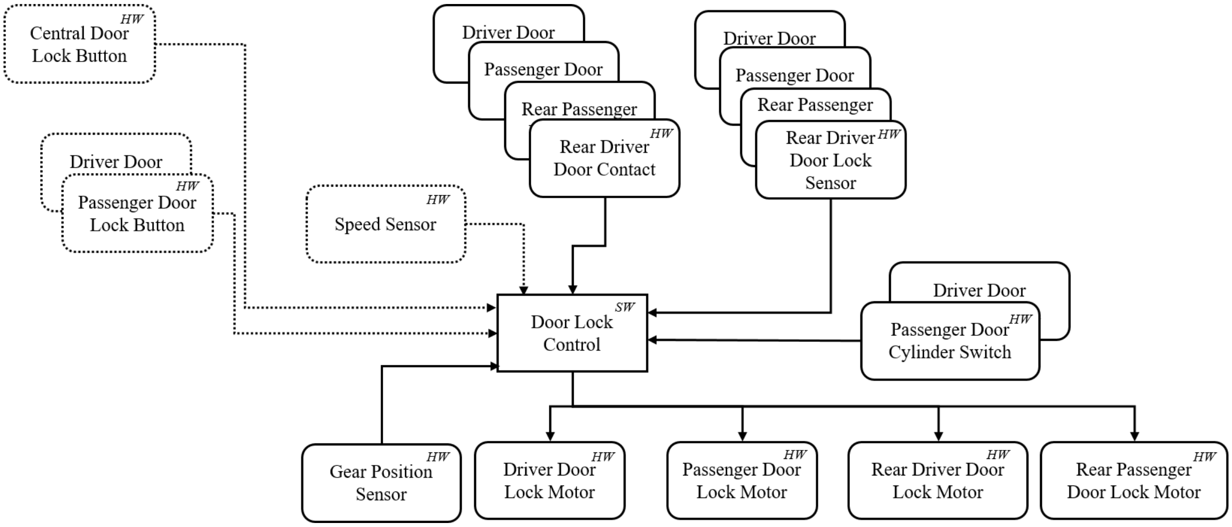


Figure 6.10: The functional analysis architecture for the basic features in the door locks system

6.2.2 Functional Analysis Architecture

The functional analysis architecture for the door locks is split into three fragments based on the features (`Basic`, `RKA`, and `PKE`). Figure 6.10 shows the basic functionality. For readability purposes, the common functional components that are identical across the four (or two) doors are grouped together with a single connector. Additionally, the connectors are not named for readability.

The following is a detailed description of the basic functional components (only one from a grouping is explained):

- `[Central]DoorLockButton`[1] A door lock button placed in the respective location to lock and unlock all doors. *Allowed implementation(s): hardware*
- `SpeedSensor` A functional device that reads the current speed of the car which is needed for the feature `SpeedSmartLock`. *Allowed implementation(s): hardware*
- `GearPositionSensor` A functional device that detects the current gear position of the car (Park, Reverse, Drive, etc.). *Allowed implementation(s): hardware*
- `[Driver]DoorLockMotor` A functional device that locks and unlocks the door. *Allowed*

[1] We use the square brackets to denote the variable portion of the name

implementation(s): hardware

- `[Driver]DoorCylinderSwitch` A functional device the detects the position of the cylinder switch when a key is inserted and turned. *Allowed implementation(s): hardware*

- `[Driver]DoorContact` A functional device that detects if the door is ajar or closed. *Allowed implementation(s): hardware*

- `[Driver]DoorLockSensor` A functional device that detects the current position of the lock for the door (i.e. the door is locked or unlocked). *Allowed implementation(s): hardware*

- `DoorLockControl` The control function that reads in the sensor inputs and gives the appropriate signal to the motors for locking/unlocking the doors. *Allowed implementation(s): software*

The second functional analysis fragment is for the RKA feature which is shown in Figure 6.11. The `DoorLockControl` analysis function is replicated as it is the only shared component with the basic FAA fragment.

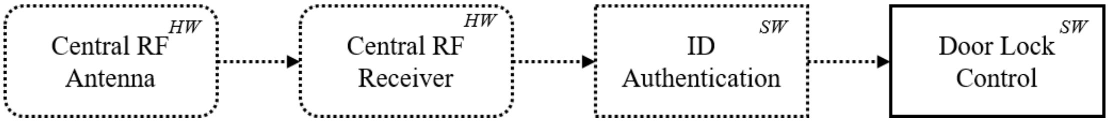


Figure 6.11: The functional analysis architecture for the RKA fragment in the door locks system

The functions for the RKA FAA are as follows:

- `CentralRFAntenna` A radio frequency (RF) antenna that receives signals from the key remote and are sent to a receiver/transciever for decoding. *Allowed implementation(s): hardware*

- `CentralRFReceiver` A receiver that decodes the antenna signal. *Allowed implementation(s): hardware*

- `IDAuthentication` The analysis function that takes the decoded signal from the receiver and determines if the key that sent the signal has permission to unlock/lock the car. *Allowed implementation(s): software*

The last functional analysis fragment, shown in Figure 6.12, contains the functionality for the PKE feature. The functions are described as follows:

- `[Driver]SideOutsideLFAntenna` A low frequency (LF) antenna that broadcasts a signal generated by the transmitter to the outside perimeter of the [driver] side door. *Allowed implementation(s): hardware*

- `[Driver]SideLFTransmitter` Transmitter that encodes a signal from the control to send to the antenna. *Allowed implementation(s): hardware*

- `Inside[Front]LFAntenna` A low frequency antenna that broadcasts a signal generated by the transmitter inside [front] region of the car. *Allowed implementation(s): hardware*

- `InsideLFTransmitter` Transmitter that encodes a signal from the control to send to the inside antennas. *Allowed implementation(s): hardware*

- `[Driver]SideDoorHandleButtonSensor` A functional device that detects when the button on the outside door handle has been pressed. *Allowed implementation(s): hardware*

- `[Driver]SideDoorHandleCapacitiveSensor` A functional device that detects when the handle is grabbed/touched. *Allowed implementation(s): hardware*

- `PKEControl` The control function that takes the inputs and determines what messages to broadcast to the antennas and dictates to the door lock control what the lock/unlock request is. *Allowed implementation(s): software*

The three fragments then can be encoded using Clafer with their links to the feature model. A fragment of the functional analysis architecture for the door locks is shown in Listing 6.14. The complete FAA can be found in Listing A.4 in Appendix A.4.

Listing 6.14: Clafer functional analysis architecture fragment for door locks

```
abstract DoorLockFAA : FunctionalAnalysisArchitecture
  // --  Core Components --//
  // Cylinder Switches
  DriverDoorCylinderSwitch : FunctionalDevice
    [implementation.hardware]
    [baseLatency = 10]
  ...
  // Door Lock Control
  DoorLockControl : AnalysisFunction
    [implementation.software]
    [baseLatency = 4]
  ...
  // -- Optional Fragments/Components --//
```

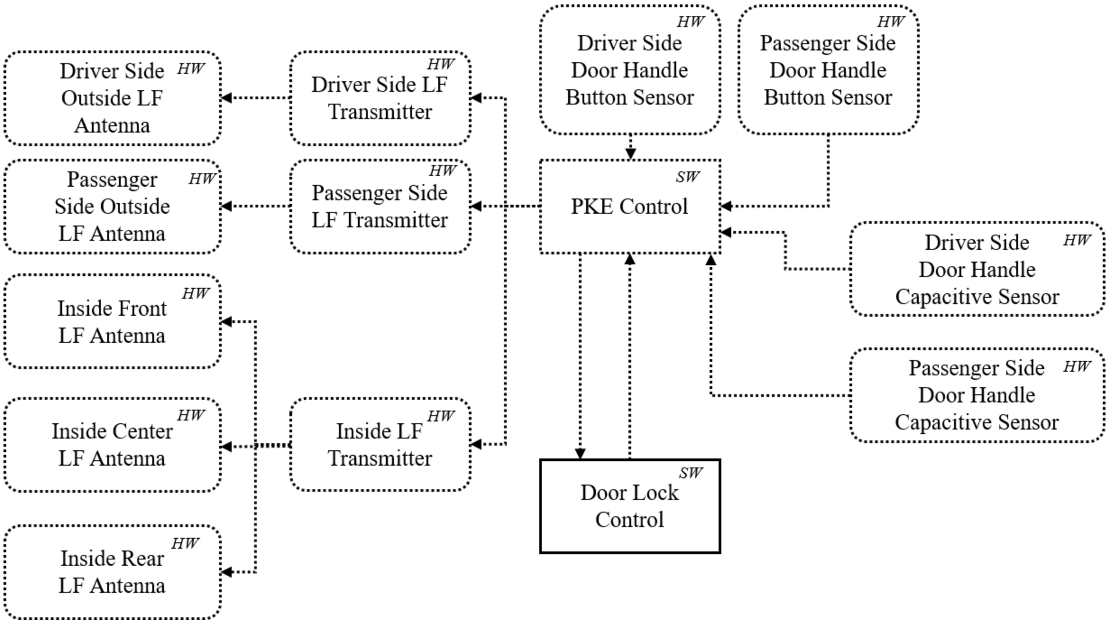

Driver Side Outside LF Antenna
HW
Driver Side LF Transmitter
HW
Driver Side Door Handle Button Sensor
HW
Passenger Side Door Handle Button Sensor
HW
Passenger Side Outside LF Antenna
HW
Passenger Side LF Transmitter
HW
PKE Control
SW
Driver Side Door Handle Capacitive Sensor
HW
Inside Front LF Antenna
HW
Passenger Side Door Handle Capacitive Sensor
HW
Inside Center LF Antenna
HW
Inside LF Transmitter
HW
Door Lock Control
SW
Inside Rear LF Antenna
HW

Figure 6.12: The functional analysis architecture for the PKE fragment in the door locks system

```
// Speed Smart Lock FA Components
SpeedSmartLockFA : FunctionalAnalysisArchitecture ?
  SpeedSensor : FunctionalDevice
    [implementation.hardware]
    [baseLatency = 10]
  speed : FunctionConnector
    [messageSize = 1]
    [sender = SpeedSensor && receiver = DoorLockControl]
// Central or Distributed Lock Switch
xor DoorLockButtonFA : FunctionalAnalysisArchitecture
  IndividualLockSwitchFA : FunctionalAnalysis
    DriverDoorLockButton : FunctionalDevice
      [implementation.hardware]
      [baseLatency = 10]
    PassDoorLockButton : FunctionalDevice
      ...
  CentralLockSwitchFA : FunctionalAnalysis
    CentralLockButton : FunctionalDevice
      ...
RemoteKeyAccessFA : FunctionalAnalysisArchitecture ?
  CentralRFAntenna : FunctionalDevice
```

```
      ...
  PassiveKeyEntryFA : FunctionalAnalysisArchitecture ?
    DriverOutsideLFAntenna : FunctionalDevice
      ...
    xor OutsideDoorHandleSensor
      ButtonSensor
        DriverDoorButtonSensor : FunctionalDevice
         ...
      CapacitiveSensor
         DriverDoorCapacitiveSensor : FunctionalDevice
          ...

DLockFAA : DoorLockFAA
  [DoorLockButtonFA.IndividualLockSwitchFA <=> DLockFM.Basic.
     IndividualLockSwitch]
  [SpeedSmartLockFA <=> DLockFM.Basic.SpeedSmartLock]
  [RemoteKeyAccessFA <=> DLockFM.RKA]
  [PassiveKeyEntryFA <=> DLockFM.PKE]
  [PassiveKeyEntryFA.OutsideDoorHandleSensor.ButtonSensor <=> DLockFM.PKE
     .OutsideDoorHandleSensor.ButtonSensor]
  [PassiveKeyEntryFA.OutsideDoorHandleSensor.CapacitiveSensor <=> DLockFM
     .PKE.OutsideDoorHandleSensor.CapacitiveSensor]
```

Similarly to the power window, a functional analysis architecture fragment is defined for an optional feature. This allows for a group of functions and connectors to be made optional by just having the fragment being optional. In addition to just defining fragments, the door locks FAA is also using `xor` groupings to model exclusive functions that stem from the feature model which was not done for the power window.

6.2.3 Device Node Classification

Similar to the functional analysis architecture, the device node classification splits the local device nodes into three fragments. In addition to the local nodes, there exists four remote nodes which are shared with other systems. Two of the four are the electric center and BCM which were described in the power window case study. A detailed description of each of the local device nodes and their allowed types is as follows:

- `[Driver]SideDoorLockMotorAssembly`: A hardware device that contains the motor and various sensors for the locking mechanism. *Allowed type(s): electric/electronic.*

- `[Driver]LockPowerSwitch`: A physical switch that can lock or unlock the car. *Allowed type(s): electric/electronic.*

- `CentralRFAntennaModule`: A hardware module that contains an RF antenna, a receiver, and some processing power. *Allowed type(s): smart.*

- `Transmitter`: A transmitter hardware module. *Allowed type(s): electric/electronic.*

- `PassiveKeyModule`: An ECU dedicated for PKE functions. *Allowed type(s): smart.*

- `[Driver]DoorButtonHandleModule`: A hardware module embedded in the door handle that contains an LF antenna and a button sensor. *Allowed type(s): electric/electronic.*

- `[Driver]DoorCapacitiveHandleModule`: A hardware module embedded in the door hanlde that contains a LF antenna and a capacitive sensor. *Allowed type(s): electric/electronic.*

- `Inside[Front]LFAntenna`: An LF antenna hardware device. *Allowed type(s): electric/electronic.*

In contrast to the power window case study, the device nodes are of fixed types. The only variability that exists at the device node level is the presence. The remote device nodes that are unique to the door locks case study are as follows:

- `Transmission Control Module (TCM)` A device node responsible for handling any control associated with the transmission. For this case study it is used to house the gear position sensor functional device. *Allowed type(s): smart.*

- `Combination Meter` A hardware device responsible for measuring the current speed of the car. *Allowed type(s): smart.*

The Clafer encoding of the device node classification is different than in the power window since for the door locks it is split into multiple fragments to group the nodes associated with the features. A fragment of the Clafer is shown in Listing 6.15. Like the FAA, `xor` groupings are used to model the exclusive sets of device nodes that follow from the feature model.

Listing 6.15: Clafer device node classification fragment for door locks

```
abstract DoorLockDN : DeviceNodeClassification
  //-- Core Device Nodes --//
  DriverDoorLockMotorAssembly : DeviceNode
    ...
  TCM -> DeviceNode
  BCM -> DeviceNode
  EC -> DeviceNode
  // -- Optional Device Nodes --//
  // Speed Smart Lock Nodes
  CombinationMeter -> DeviceNode ?
  // Central or Individual Lock Nodes
  xor DoorLockButtonDN
    IndividualLockSwitchDN : DeviceNodeClassification
      DriverLockPowerSwitch : DeviceNode
      ...
    CentralLockSwitchDN
      CenterLockPowerSwitch : DeviceNode
        ...
  RemoteKeyAccessDN : DeviceNodeClassification ?
    CentralRFAntennaModule : DeviceNode
      ...
  PassiveKeyEntryDN : DeviceNodeClassification ?
    Transmitter : DeviceNode ?
      ...
    xor OutsideDoorHandleSensor
      ButtonSensor
        DriverDoorButtonHandleModule : DeviceNode
        ...
      CapacitiveSensor
        DriverDoorCapacitiveHandleModule : DeviceNode
        ...
    InsideFrontLFAntenna : DeviceNode
    ...
DLockDN : DoorLockDN
  [BCM = Car.BCM]
  [TCM = Car.TCM]
  [EC = Car.EC]
  [CombinationMeter => CombinationMeter = Car.CombinationMeter]
  [DoorLockButtonDN.IndividualLockSwitchDN <=> DLockFM.Basic.
     IndividualLockSwitch]
  [CombinationMeter <=> DLockFM.Basic.SpeedSmartLock]
  [RemoteKeyAccessDN <=> DLockFM.RKA]
  [PassiveKeyEntryDN <=> DLockFM.PKE]
  [PassiveKeyEntryDN.OutsideDoorHandleSensor.ButtonSensor <=> DLockFM.PKE
     .OutsideDoorHandleSensor.ButtonSensor]
  [PassiveKeyEntryDN.OutsideDoorHandleSensor.CapacitiveSensor <=> DLockFM
     .PKE.OutsideDoorHandleSensor.CapacitiveSensor]
```

6.2.4 Power Topology

In the device node classification, the inline that was modeled in the power window was not for the door locks. An inline was not modeled in this case study as there was no interesting power configurations that were affected by the door inline as there were in the power window. The power topology for the door locks did not contain load configurations, as with the power window, due to the `DoorLockControl` analysis function always being deployed to the `BCM`.

The power topology, shown in Figure 6.13 contains no variability itself as the only optional components are driven by the selection of the device nodes used (i.e. there is no variability in the configuration of the topology given a set of device nodes). The only unique piece to the door locks is the need for a device power connector when using a capacitive sensor.

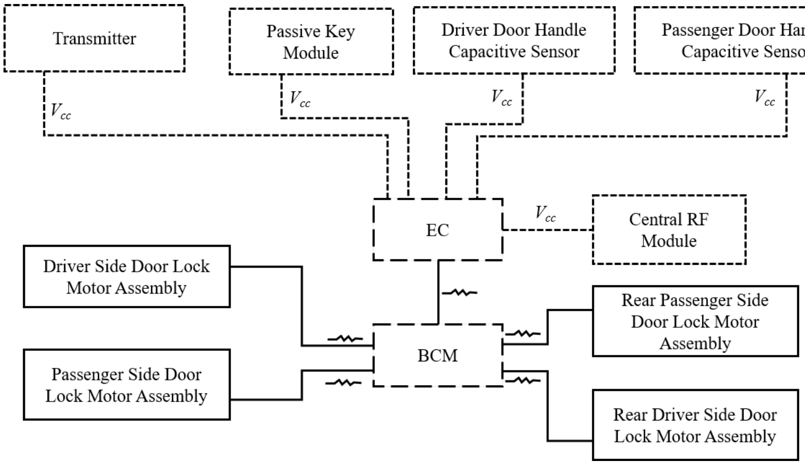

Transmitter
Passive Key Module
Driver Door Handle Capacitive Sensor
Passenger Door Handle Capacitive Sensor
V_{cc}
V_{cc}
V_{cc}
V_{cc}
EC
V_{cc}
Central RF Module
Driver Side Door Lock Motor Assembly
Rear Passenger Side Door Lock Motor Assembly
BCM
Passenger Side Door Lock Motor Assembly
Rear Driver Side Door Lock Motor Assembly

Figure 6.13: The power topology for the door locks system

The full Clafer encoding of the power topology is shown in Listing A.4 in Appendix A.4.

6.2.5 Communication Topology

As opposed to the power topology, the communication topology is interesting and also larger than the power window. In order to manage the size and complexity, it is split into three figures. First, Figure 6.14 shows the communication topology fragment for the basic and RKA features. Two buses are needed: the first is a high speed bus (which would be either high speed CAN or FlexRay) which handles safety critical nodes; the second is a low speed bus (either low speed CAN or LIN) which handles the non-critical body domain nodes. This low speed bus is assumed to be routed through the main body harness and not to the doors as was so in the power window.

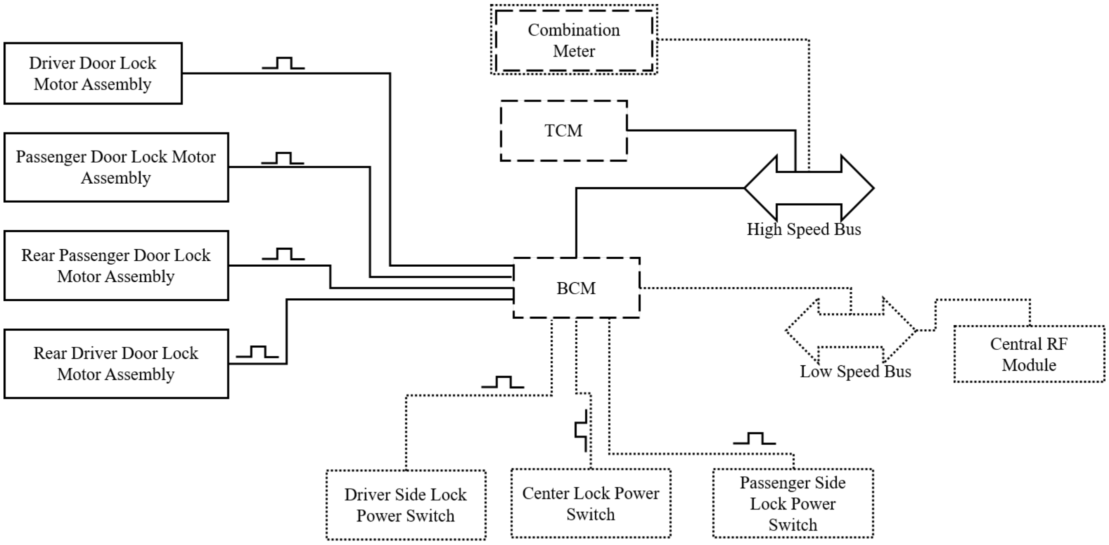

Combination Meter
Driver Door Lock Motor Assembly
TCM
Passenger Door Lock Motor Assembly
High Speed Bus
Rear Passenger Door Lock Motor Assembly
BCM
Rear Driver Door Lock Motor Assembly
Central RF Module
Low Speed Bus
Driver Side Lock Power Switch
Center Lock Power Switch
Passenger Side Lock Power Switch

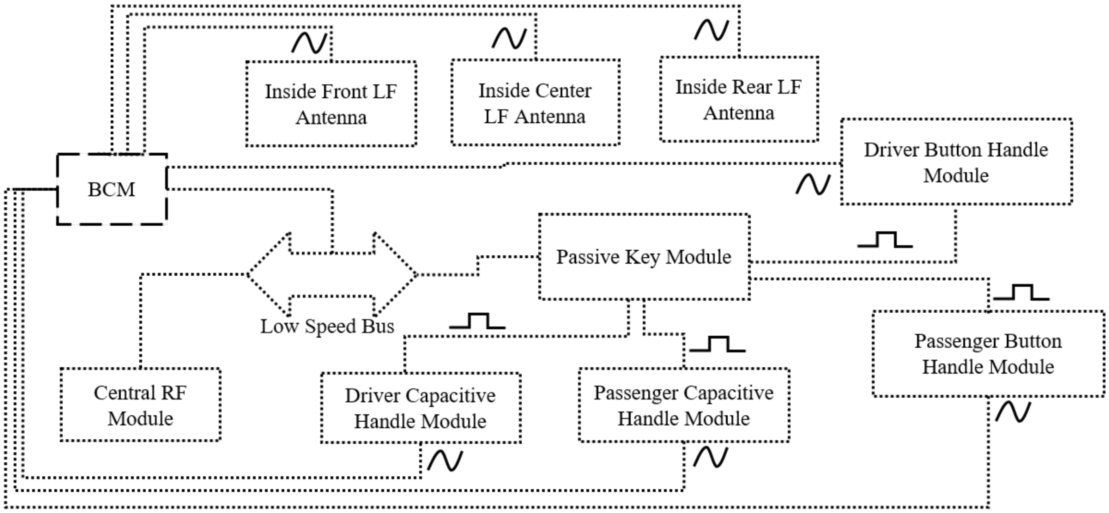
Figure 6.14: The communication topology for the door locks system basic and RKA fragments

The other two figures (Figures 6.15 and 6.16) show two views which can be overlaid to describe the complete communication architecture of the PKE feature. The first view shows the possible communication connectors needed when the BCM acts as a LF transmitter (i.e. the `DriverSideLFTransmitter`, `PassengerSideLFTransmitter`, and `InsideLFTransmitter` are all deployed to the `BCM`). The second view, shows when the transmitter device node is used instead.

The full Clafer encoding of the communication topology is shown in Listing A.4 in Appendix A.4.

6.2.6 Deployment

To complete the door locks case study, the last part to be modeled is the deployment. As in the power window, the deployment for the door locks is just a set of constraints that restricts the set of possible targets for the functional analysis components. The unique part of the door locks encoding is using fragments to split up the deployment by features.



Inside Front LF Antenna
Inside Center LF Antenna
Inside Rear LF Antenna
Driver Button Handle Module
BCM
Passive Key Module
Low Speed Bus
Passenger Button Handle Module
Central RF Module
Driver Capacitive Handle Module
Passenger Capacitive Handle Module

Figure 6.15: The communication topology for the door locks system PKE fragment which uses the BCM as a transmitter

It allows us to drop the repeated use of implication (denoted by `=>`) in the constraints. For example, the Clafer fragments shown below are identical for modeling the deployment constraints. Listing 6.16 shows how the constraint is written using an implies (as in the power window model), whereas Listing 6.17 shows how the constraints can be nested inside a fragment.

Listing 6.16: Using implies in constraint expression to handle conditional deployment.

```
[DLockFM.PKE => (fa.PassiveKeyEntryFA.DriverLFTransmitter.deployedTo in (
   ha.dn.PassiveKeyEntryDN.Transmitter, ha.dn.BCM.dref))]
```

Listing 6.17: Nesting the constraint under a fragment to handle conditional deployment.

```
PassiveKeyEntryDpl ?
  [fa.PassiveKeyEntryFA.DriverLFTransmitter.deployedTo in (ha.dn.
     PassiveKeyEntryDN.Transmitter, ha.dn.BCM.dref)]
  [PassiveKeyEntryDpl <=> DLockFM.PKE]
```

The full Clafer encoding for the deployment is shown in Listing A.4 in Appendix A.4.

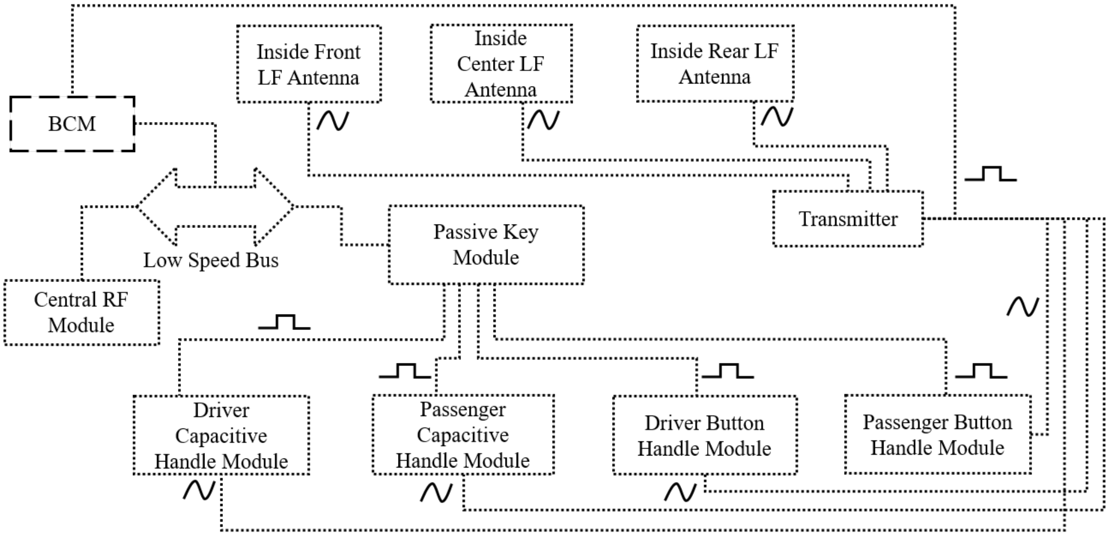


Figure 6.16: The communication topology for the door locks system PKE fragment which uses the transmitter device node

6.2.7 Quality Attributes & Timing Analysis

Similar to that of the power window case study, the values for the different quality attributes were obtained from various sources then normalized so that they could be used in integer operations. Also, for the door locks case study end-to-end timing constraints were constructed for three requirements as well as an input synchronization constraint as follows:

- **TR1:** The time it takes for the individual switch to read the pressed value to the time the motor begins to move to lock/unlock the door must be less than 500ms.
- **TR2:** The time it takes for the central switch to read the pressed value to the time the motor begins to move to lock/unlock the door must be less than 500ms.
- **TR3:** The time it takes for the handles sensor the read the user request to the time the motor begins to lock/unlock the door must be less than 750ms.
- **TR4:** The time it takes the door contacts, door sensors, and the lock switch to read the data and send to the control must by synchronized to less than 50ms.

TR1 through TR3 are end-to-end timing constraints that are quite similar to that of the power window. Lines 700-731 in Listing A.4 in Appendix A.4 shows the required Clafer to encode the requirements.

Chapter 7

Evaluation

We had three main objectives for evaluating the use of the reference model and Clafer and its supported tool-chain to synthesize and explore automotive E/E architectures:

1. Comparison to state-of-the-art tools
2. Role of multiple layers
3. Performance evaluation

Then, to meet these objectives we formulated a set of research questions, which are as follows:

- **RQ1:** What aspects of our reference model are unique and not found in current meta-models for E/E architecture or are found but not supported by reasoning?
- **RQ2:** What exploration scenarios that are supported by the presented reference model and Clafer encoding are not possible with other existing reference models and tools?
- **RQ3:** Does considering a subset of the layers from our reference model, when exploring the candidate architectures, produce different candidates in comparison to when considering all layers? If so, how different are these sets of candidates?
- **RQ4:** Is it *feasible* to explore the individual design decisions, constraints, and objectives put forth in Chapter 4 for realistic models? We define *feasible* as the ability for the solver to find the first 10 non-optimal solutions in less than 10 minutes and the first 10 optimal solutions within 45 min. We want to obtain a handful of instances

within minutes to enable quick iterations for engineers, and therefore we pick 10 instances in 10 minutes. We give a higher time budget to optimization since it is a harder problem; we pick 45 minutes since waiting for hours would seriously slow down the exploration process.

- **RQ5:** Is it feasible to explore the example design scenarios we presented in Chapter 4 using realistic power window models?

The first two research questions target our first objective: comparing Clafer to state-of-the-art tools. The third research question focuses on the second objective, whereas the last two are centered around evaluating the performance of using Clafer and its supported tools. In the following section, we outline our methodology for each objective. We follow it up with our findings for each research question.

7.1 Research Methodology

As a part of our research methodology, we modeled two automotive E/E architectures in the body domain: a power window system and a central door locks system, as presented in Chapter 6. We chose to model these two systems as they are self-contained and provide a rich design space to explore potentially interesting tradeoffs.

7.1.1 Comparison to State-of-the-Art Tools

To compare using Clafer and its supported tool-chain to state-of-the-art tooling, we first conducted a literature review. We searched journal and conference proceedings in the following digital libraries: IEEE Xplore, Google Scholar, Elsevier Science Direct, Springer Digital Library, and ACM Digital Library. The literature was collected using the following set of keywords: E/E architecture, architecture modeling, design space exploration, optimization, and tool support. From the gathered literature, we selected only tool-supported approaches targeting architectural modeling and design exploration for automotive E/E systems and similar domains, such as avionics, railway systems, building and factory automation, and oil and gas exploration.

Using the selected tools, we then attempted to model the single door variant of the power window system. Next, we attempted to explore the models using the Clafer supported decision, constraint, and objective templates from Chapter 4. For each template, we determined if it was *natively* possible, meaning without changing the tools' original capabilities, and commented why.

7.1.2 Role of Multiple Layers

To investigate the role of multiple layers in design space exploration, we systematically applied the following procedure:

1. Remove one of the following from the single door power window design-space model: feature model, power topology, or communication topology. Additionally, remove all dependencies to the removed layer from the remaining layers.
2. Identify the quality attributes associated with the removed layer.
3. Write a design specification that minimizes mass, part cost, and part warranty cost qualities. Additionally, constrain the end-to-end latency for the timing chain constructed from the switch to the motor to be less than 500 ms. Gather the resulting instances for the single door power window design-space model that has a layer removed (step 1) and the given specification.
4. Gather the resulting instances for the complete single door power window design-space model and the given specification.
5. Compare the identified quality attributes, from step 2, for both sets of instances (steps 3 and 4).

We did not consider removing the functional analysis architecture or device node classification since the remaining information would not be sufficient for meaningful exploration.

7.1.3 Performance Evaluation

To evaluate the performance of exploring realistic models using Clafer, and its supported tool-chain, we use the power window and central door locks system design-space models as benchmarks. In order for readers to appreciate the size and complexity of the models, Table 7.1 shows the number of concrete components for a given reference model element, number of deployment configurations (for functions to device nodes), and number of possible candidates. For the concrete components, we give how many contain *presence variability*; that is, the component may or may not exist in the synthesized architecture.

Table 7.1: Number of concrete components for each reference model element, number of deployment configurations, and number of possible candidates for each design-space model. (n) denotes that n of the concrete components have presence variability.

	Single Door Power Window	**Two Door Power Window**	**Central Door Locks**
Features	3 (2)	6 (4)	7 (6)
Analysis Functions	3 (1)	6 (2)	3 (2)
Functional Devices	4 (1)	9 (2)	33 (15)
Function Connectors	6 (2)	7 (4)	33 (18)
Device Nodes	6 (2)	10 (3)	21 (14)
Discrete/Analog Connectors	13 (13)	18 (18)	34 (30)
Bus Connectors	1 (1)	2 (1)	2 (1)
Power Connectors	8 (6)	16 (12)	9 (5)
Deployment Configurations	64	4096	96
Possible Architectures	32,000	959,714,800	2,028

Table 7.2: Size of Clafer encoding for each model. The design-space model numbers are reported minus the reference model elements.

	Reference Model	**Single Door Power Window**	**Two Door Power Window**	**Central Door Locks**
Abstract clafers	25	8	8	6
Concrete clafers	66	85	114	195
Constraints	64	161	211	444
References	50	33	47	28

For the number of possible candidate architectures, expressed by the design-space model, we were able to run the solver and find all solutions for both the single door variant of the power window and for the central door locks. The solver was run with no latency constraint specified. For the two door variant, we computed the number of possible solutions using the single door variant and taking into account the allowed feature variants between the two doors (e.g., the passenger door can not have the feature express if the driver door does not).

Table 7.2 gives statistics for encoding the models in Clafer. The table indicates a large number of constraints for the central door locks model in comparison to both variants of the power window model. As a result of the constraints, there are significantly fewer possible candidate architectures as seen in Table 7.1.

Using these models as benchmarks, we constructed two experiments for research questions RQ4 and RQ5. In the first experiment, we measured the time taken to find the first 10 instances for each of the design decision, constraint and objective templates. However, we did not test each one with each of its possible parameters due to the exponential number of combinations. Rather, we tested each decision using a single selection of parameters (i.e., either the device was chosen to be smart or EE, not both); for constraints and objectives, we only tested one of the mass, part cost, and part warranty cost constraints due to their similar nature. We did, however, test each of the latency constraints with a single selection of the parameters. For both the single and two door variant of the power window model, the same parameters were selected for the template specification.

The time to find the first 10 instances was measured ten times and the average, standard deviation, and number of instances found was recorded. A *timeout* was reported if the first 10 instances could not be found in 10 minutes or under, for the non-optimal specifications, and 45 minutes or under for the optimal specifications. If a timeout occurred, the number of instances found before time expired was reported. As mentioned earlier, we wanted to obtain a handful of instances within minutes to enable quick iterations for engineers, and therefore we pick 10 instances in 10 minutes. We gave a higher time budget to optimization since it is a harder problem; we pick 45 minutes since waiting for hours would seriously slow down the exploration process.

The second experiment measured the time taken to find the first 10 instances for each of the example power window design specifications found in Chapter 4. For each of the multiple design specification scenarios (1, 3, and 6), we split the scenario into multiple specifications and measured the solving time individually. Additionally, for scenario 6, we do not consider the single door power window model as a distributed vs. centralized tradeoff is not applicable to a single window. Lastly, both experiments were carried out on a laptop with the following specs:

- Intel® Core™ i7-3520M CPU @ 2.90 GHz
- 4.00 GB RAM
- Windows 10 - 64 bit

7.2 Comparison to State-of-the-Art Tools

We selected four tools, as a result of our literature review, to compare with Clafer. The description, version used in the evaluation, and other relevant details is provided in Table 7.3.

Tool Name	**Version Used For Evaluation**	**Description**	**Targeted Domain**	**Tool Released to Public?**	**Sources**
Arche/-Opterix	Not Provided	A tool that supports modeling software components, communication between software components, ECUs, buses, and services. Additionally, the tool can optimize the deployment of software components to ECUs using objectives such as redundancy allocation, energy consumption, and cost.	Automotive E/E Architecture	Yes	[2,9,42–46]

Tool Name	**Version Used For Evaluation**	**Description**	**Targeted Domain**	**Tool Released to Public?**	**Sources**
Auto/-FOCUS – AF3	2.9	A model-based development tool that supports architecture modeling from the requirements to code generation. Additionally, the tool can synthesize hardware platform architectures, given a functional architecture based on objectives such as end-to-end latency and number of nodes. The tool is also able to synthesize and explore optimal deployments and schedules.	Automotive E/E Architecture	Yes	[3, 64, 65]
OSATE	2.2.1	A model-based development tool that implements the AADL (architecture analysis & design language), and supports modeling both aerospace and automotive systems. It does not have any optimization support; however, it supports model checking, schedulability analysis, and flow latency analysis.	Automotive/ Aerospace E/E Architecture	Yes	[26,59]

Tool Name	Version Used For Evaluation	Description	Targeted Domain	Tool Released to Public?	Sources
AAOL	N/A	A domain-specific constraint language used for modeling and optimization of automotive E/E architectures. The supporting tool can optimize based on user-defined objectives, and support design constraints, such as memory capacity, ASIL requirements, and predefined deployments.	Automotive E/E Architecture	No	[36]

Table 7.3: Details for selected state-of-the-art tools to compare with Clafer

For ArcheOpterix, AutoFOCUS – AF3, and OSATE, we modeled the power window system (the single door variant) up to what was supported by the tool. For AAOL, the tool was not publicly available so it was not possible to recreate the power window system. Instead, the evaluation was done using the published details about the language and case study, a four door power window [36].

When modeling the single door variant of the power window system, we developed a single domain-space model (as we did with Clafer, c.f. [56]) for the system. For example, in ArcheOpterix a single domain-space model consisted of one XML file containing the model elements and their qualities. If variability for a certain element could not be captured, we did not consider the possibility of creating a second XML file with the new variant since the underlying solver accepted a single file. In the context of AF3 a single domain-space model consisted of one component architecture and one platform architecture. For OSATE, we considered a single system implementation to be one domain-space model as the analyzers worked only with one implementation at a time. In AAOL, we consider a single domain-space model to consist of a single set of constraints, objectives, and orders; however, since the constraints can be written over a set of software architectures and hardware architectures we allow for multiple of both.

Table 7.4 shows the limitations we encountered when modeling using each of the tools. The resulting models are available for readers online.[1]

	ArcheOpterix	**AutoFOCUS – AF3**	**OSATE**	**AAOL**
Feature Model	The tool did not allow for modeling the features of the system.	The tool did not allow for modeling the features of the system.	The tool did not allow for modeling the features of the system.	The tool did not allow for modeling the features of the system.
FAA	The design-space model had to contain a fixed FAA. There was no distinction between functional devices or analysis functions, only software components were considered. It was also not possible to model functions implemented in hardware.	The design-space model had to contain a fixed FAA. There was no distinction between functional devices or analysis functions, only software components were considered. It was also not possible to model functions implemented in hardware.	The design-space model had to contain a fixed FAA. There was no distinction between functional devices or analysis functions, only processes and threads were considered.	The tool made no distinction between functional devices or analysis functions.

[1] https://github.com/ross2jd/Thesis_PWModels/

	ArcheOpterix	**AutoFOCUS – AF3**	**OSATE**	**AAOL**
Device Node Classification	The design-space model had to contain a fixed hardware architecture including its type. Also, E/E and power devices could not be modeled, all nodes were considered smart.	The design-space model had to contain a fixed hardware architecture, including its type. When using the generated hardware architecture, it was not possible to model node qualities. Lastly, power devices could not be modeled.	The design-space model had to contain a fixed hardware architecture. The device node classification was not entirely separate from the FAA since the switch and motor were *devices* — an AADL concept representing an actuator or sensor — and could be used in both layers. Lastly, power devices could not be modeled.	The tool did not consider power devices.
Power Topology	The tool did not allow for modeling distribution of power from sources to devices.	The tool did not allow for modeling distribution of power from sources to devices.	The tool did not allow for modeling distribution of power from sources to devices.	The language did not allow for modeling distribution of power from sources to devices.
Comm. Topology	The tool did not allow for variable communication media.	The tool did not allow for variable communication media. Additionally, the tool did not consider discrete/analog connectors.	The tool did not allow for variable communication media. Additionally, the tool did not consider discrete/analog connectors.	There were no limitations in modeling this layer.

	ArcheOpterix	**AutoFOCUS – AF3**	**OSATE**	**AAOL**
Deployment	The tool did not allow for modeling restrictions on deployments for function connectors to hardware data connectors.	The tool did not allow alternatives to be specified for deployments of a software components connector. Additionally, deployment of software components to E/E devices was disallowed.	There were no limitations in modeling this layer.	The tool did not allow for modeling a restriction for what set, size greater than 1, of buses a software component could produce traffic on.
Objectives & Constraints	The tool did not consider mass and part warranty cost as objectives. Also, no end-to-end latency or input synchronization constraint could be specified.	The tool did not consider mass, part cost, or warranty part cost as objectives. No end-to-end latency or input synchronization constraint could be specified.	The tool did not support optimization over any objective. Constraints for part cost, mass and warranty part cost were also unsupported.	The input synchronization constraint could not be expressed in the tool.

Table 7.4: Limitations we encountered when modeling the single door variant of the power window system introduced in Chapter 3

7.2.1 Research Question 1

Using Table 7.4, the unique elements along with a short explanation for each, are as follows:

- **Feature model and features:** Feature models and features were not explicitly a part of any reference model; however, in AADL there is a concept, *feature*, but it does not match our definition from Chapter 3. Rather, an AADL feature is used to model the incoming and outgoing data for a process, thread, or device.

- **Power Topology:** The distribution of power from sources to devices was not considered by any of the tools.
- **Power Devices:** None of the tools allowed modeling a device for relaying or distributing power, such as the door inline in the power window.
- **Deployment of Function Communication:** None of the tools allowed restricting the possible targets for which a function connector should be deployed, while taking account both buses and discrete/analog connectors.

While our reference model does contain unique elements not found in the surveyed tools, we also want to state that the surveyed tools contain elements not found in our reference model. For example, we do not consider energy consumption, bus reliability, memory capacity, or schedulability; these could be incorporated in future work.

7.2.2 Research Question 2

Table 7.5 shows our findings for what decision templates are supported, indicated by a "Yes" in the table, by each of the surveyed tools. If the decision was not supported, we listed one of two reasons for why:

1. The template pertained to a model element that was not supported by the tool.
2. The tool could not capture the variability associated with the model element in the template.

Table 7.6 shows the supported constraint and objective templates, indicated with a "Yes" in the table, for each of the surveyed tools. As with the decisions, we listed one of two reasons for why the template was unsupported:

1. The template considered a quality attribute that was not supported by the tool.
2. The template contained a quantifier or optimization objective that was not supported by the tool.

Both tables show that none of surveyed tools are able to capture all decisions, constraints, and objectives that are possible when using Clafer.

In summary, the difference between Clafer and the compared tools is: Clafer synthesizes the valid candidate architectures based on a set of constraints and objectives, and allows modeling variability at all layers. The compared tools, ArcheOpterix, OSATE, and AF3, only allow for variability in the deployment of functions to device nodes. AAOL is limited

Table 7.5: Supported (Yes) and unsupported (No) decision templates for each of the surveyed tools. If the decision is unsupported, one of the following reasons is listed: 1) Model element not supported 2) Variability not captured.

ID	**Arche-Opterix**	**AF3**	**OSATE**	**AAOL**
DD1	No (1)	No (1)	No (1)	No (1)
DD2	Yes	Yes	Yes	Yes
DD3	No (2)	No (2)	No (2)	Yes
DD4	No (2)	No (2)	No (2)	No (1)
DD5	No (1)	No (1)	No (2)	No (1)
DD6	No (2)	No (2)	No (2)	No (1)
DD7	No (1)	No (1)	No (1)	No (1)
DD8	No (2)	No (2)	No (2)	Yes
DD9	No (1)	No (1)	No (1)	No (1)
DD10	No (1)	No (1)	No (1)	No (1)
DD11	No (2)	No (2)	No (2)	Yes
DD12	No (2)	No (2)	Yes	No (1)
DD13	No (2)	No (2)	Yes	No (1)
DD14	No (1)	No (1)	No (1)	No (2)
DD15	No (2)	No (2)	No (2)	Yes

Table 7.6: Supported (Yes) and unsupported (No) constraint and objective templates for each of the surveyed tools. If unsupported, one of the following reasons is listed: 1) Quality not supported 2) Expression not supported.

ID	**Arche-Opterix**	**AF3**	**OSATE**	**AAOL**
DC1	No (1)	No (2)	Yes	Yes
DC2	No (1)	No (2)	No (2)	No (2)
DC3	No (1)	No (1)	Yes	Yes
DC4	No (1)	No (1)	No (2)	No (2)
DC5	No (1)	No (1)	No (1)	Yes
DC6	No (1)	No (1)	No (1)	Yes
DC7	No (1)	No (1)	No (1)	Yes
DC8	No (2)	No (1)	No (1)	Yes
DC9	No (2)	No (1)	No (1)	Yes
DC10	No (2)	No (1)	No (1)	Yes
DC11	No (1)	No (1)	No (1)	Yes
DC12	No (1)	No (1)	No (1)	Yes
DC13	No (1)	No (1)	No (1)	Yes
DO1	No (1)	No (1)	No (2)	Yes
DO2	No (1)	No (1)	No (2)	No (2)
DO3	No (1)	No (1)	No (1)	Yes
DO4	No (2)	No (1)	No (1)	Yes
DO5	No (1)	No (1)	No (1)	Yes

by the elements considered in the language, rather than being able express variability as evident from Table 7.5. However, expressing the presence variability for components such as functions and device nodes required defining multiple software and hardware architectures to be used by the solver.

7.3 Role of Multiple Layers (Research Question 3)

In Table 7.7, we show the results of our experiment in removing the feature model, power topology, and communication topology from the single door variant of the power window system model. Each row of the table shows the resulting quality attribute values when minimizing mass, part cost, and warranty part cost, while ensuring a end-to-end latency of 500 ms from the switch to the motor.

From the table, we found that the feature model layer does not play a role in the qualities, which we expected since no quality attributes were captured in this layer; however, the feature model layer contributed additional candidate architectures that were possible. The power topology, we found, did affect the mass, part cost, and warranty part cost. Furthermore, by removing the power topology, a tradeoff was introduced, meaning that two sets of candidate architectures existed on the Pareto front with differing quality attributes. The two sets of values are captured by curly braces in Table 7.7. This shows the importance of including the power topology in the overall model, as it eliminates designs that are not optimal when considering the full model. Lastly, by removing the communication layer, we found the values for mass and part cost were reduced. This was expected since less of the model is present; however, it was interesting to see no difference in the margin latency. The reason for this is due to the loss of precision when working with integers to represent smaller quantities.

7.4 Performance Evaluation

7.4.1 Research Question 4

Table 7.8 contains the results from running our outlined experiment on the design decision and constraint templates. The results show that for the single door variant of the power window and the door locks models, all sample templates tested for each of the decisions and constraints were considered feasible. The two door variant, however, did experience

Table 7.7: Resulting quality attribute values when removing portions of the power window model (single door variant). The margin latency is for the timing chain from the switch to the motor. The margin latency value is given as a range denoting the smallest and largest values for the reported instances. We use the curly braces to denote an ordered set of values when more than one set of values exists on the Pareto front.

Removed Portion	**Associated Quality Attributes**	**Number of Instances**	**Quality Attribute Values**			
			Mass (g)	Parts Cost ($)	Warranty Parts Cost ($ per million)	Margin Latency (ms)
None	Mass, Parts Cost, Warranty Parts Cost, Margin Latency	9	837	223	6452	440-445
Feature Model	None	6	837	223	6452	44-445
Power Topology	Mass, Parts Cost	41	{813,811}	{221,223}	{6452, 10852}	{440-445,439-445}
Communicat/ion Topology	Mass, Parts Cost, Margin Latency	9	833	222	6452	440-445

limitations in which one of the templates was not feasible. DD15 experiences a timeout because by requiring certain devices to communicate via the bus it increases the possible deployment targets for function connectors on both doors. We see this design decision that experiences scalability issues since this decision is more challenging for the single door variant, as well, evidenced by taking 3.5 times longer, on average, than the other decisions.

	Single Door Power Window			**Two Door Power Window**			**Door Locks**		
Design Spec.	*Solving Time (s)*	*Std. Dev. (s)*	*Number of Instances*	*Solving Time (s)*	*Std. Dev. (s)*	*Number of Instances*	*Solving Time (s)*	*Std. Dev. (s)*	*Number of Instances*
DD1	6	0.6	10	20	1.4	10	314	7	10
DD2	6	1.2	10	20	1.9	10	91	2.2	10
DD3	6	0.3	10	21	3.3	10	315	7.8	10
DD4	6	0.3	10	20	2.8	10	476	7.9	10
DD5	6	0.2	10	20	2.1	10	93	1.4	10
DD7	5	0.1	10	19	1.3	10	313	9.2	10
DD8	6	0.8	10	503	16.4	10	317	8.1	10
DD9	5	0.4	10	18	1.2	10	27	1.5	10
DD10	6	0.3	10	20	1.3	10	94	2.3	10
DD11	6	0.4	10	17	2.2	10	32	2.2	10
DD12	6	0.3	10	19	1.3	10	101	1.6	10
DD13	6	0.7	10	19	2.9	10	96	4.2	10
DD14	6	0.9	10	18	1.0	10	33	3.4	10
DD15	21	1.1	10	Timeout	–	0	92	5	10
DC1	5	0.5	10	17	2.5	10	93	1.5	10
DC2	6	1	10	18	2.5	10	94	2.4	10
DC3	6	0.9	10	20	2.3	10	94	2.7	10
DC4	6	0.6	10	19	2.4	10	94	4.6	10

	Single Door Power Window			**Two Door Power Window**			**Door Locks**		
Design Spec.	*Solving Time (s)*	*Std. Dev. (s)*	*Number of Instances*	*Solving Time (s)*	*Std. Dev. (s)*	*Number of Instances*	*Solving Time (s)*	*Std. Dev. (s)*	*Number of Instances*
DC5	6	0.8	10	19	1.9	10	116	3.7	10
DC6	6	0.9	10	19	1.2	10	92	3.1	10
DC8	7	1	10	19	2.0	10	339	4.8	2

Table 7.8: Design decision and constraint solving time for three case study models. A timeout is reported if the solver takes more than 10 minutes to find the first 10 non-optimal solutions.

Table 7.9 shows the results from executing the same experiment as before, but for the optimization templates. From the table, we see that each model does experience some limitations in performance. Observing the different specifications, we see that DO1 and DO2 (optimization over the latency qualities) have by far the worst performance. Additionally, we see that as the models grow in the amount of variability (the two door variant encodes overs 959 million variants opposed to roughly 32 thousand for the single door), it is no longer feasible to optimize with respect to one or more objectives. As the models grow in size (the door locks contains over 133 concrete elements), we find the design decision and constraint specifications, on average, take longer than in the smaller models (36 concrete elements for the single power window and 70 for the two door power window); however they are all still feasible. Future work is needed to investigate techniques to improve the scalability in order for all optimization and decision templates to be considered feasible.

7.4.2 Research Question 5

From Table 7.10, we see that again it is feasible to ask all design exploration scenarios for the single door variant of the power window. For the two door variant, we see that only one out of the six scenarios are feasible. The reason for this is related to our earlier observation that due to the encoding of the large number of variants, the chocosolver is unable to feasibly find the first 10 optimal instances for single or multiple objectives. Therefore, we find that it is feasible to ask Emily's design scenarios only if we are considering a smaller

Table 7.9: Design objective solving time for three case study models. A timeout is reported if the solver takes more than 45 minutes to find the first 10 optimal solutions.

	Single Door Power Window			**Two Door Power Window**			**Door Locks**		
Design Spec.	*Solving Time (s)*	*Std. Dev. (s)*	*Number of Instances*	*Solving Time (s)*	*Std. Dev. (s)*	*Number of Instances*	*Solving Time (s)*	*Std. Dev. (s)*	*Number of Instances*
DO1	2690	0	10	Timeout	–	0	Timeout	–	0
DO2	Timeout	–	0	Timeout	–	0	Timeout	–	0
DO3	59	1.5	9	Timeout	–	0	314	9.7	1
DO3, DO4, DO5	56	1.2	9	Timeout	–	0	Timeout	–	1

model such as the single door power window which still encodes roughly 32 thousand candidate architectures.

Table 7.10: Design exploration scenario specification solving time for the power window case study models. A timeout is reported if the solver takes more than 10 minutes to find the first 10 non-optimal solutions and more than 45 minutes to find the first 10 optimal solutions.

Scenario	**Single Door Power Window**			**Two Door Power Window**		
	Solving Time (s)	*Std. Dev. (s)*	*Number of Instances*	*Solving Time (s)*	*Std. Dev. (s)*	*Number of Instances*
1 - With ECU	176	2.7	10	Timeout	-	0
2 - No ECU	24	0.4	10	Timeout	-	0
2	6	0.3	10	20	1.8	10
3 - Smart	86	3.6	10	Timeout	-	0
3 - Dumb	26	0.8	6	Timeout	-	0
4	26	0.6	3	Timeout	-	0
5	25	1.1	10	Timeout	-	0
6 - Centralized	-	-	0	Timeout	-	0
6 - Distributed	-	-	0	Timeout	-	0

Chapter 8

Threats to Validity

8.1 Research Questions 1 & 2

In our comparison to state-of-the-art we have two main threats to validity; the first being an incomplete literature review. We mitigated this threat by consulting multiple search engines, in addition to using a previously done systematic literature review by Aleti et. al. [10].

The second threat, is a potential bias in our expertise with Clafer opposed to the compared state-of-the-art tools. It is to the best of our knowledge that our attempted models of the power window are accurate, and demonstrate the full capabilities of the tool, in the scope of design space exploration, as we have consulted with tutorials, user guides, and published works for each tool. However, in future work the threat could be further mitigated by having an expert, for each tool carry, out the exercise of modeling and exploring the power window system.

8.2 Research Question 3

The main threat to validity for the third research question is the sensitivity of quality attribute values used in the comparison between the original and partial design-space models. We were able to partially mitigate this by using real-world quality attribute values found from OEM part supply stores, reliability handbooks, and other online shopping sites such as Amazon. Since Clafer, currently, can work only with integer values, a loss of precision was introduced when encoding these values. In the accompanying technical report [56], we detail how the values are scaled.

8.3 Research Questions 4 & 5

We have three main threats in our evaluation of performance for Clafer. The first is a lack of full coverage for the decision, constraint, and objective templates tested in the first experiment. We were able to mitigate this threat by ensuring we covered each of the decision templates, and each objective at least once. In the experiment, not all constraints were tested since mass, parts cost, and part warranty cost all had the same expression, but with different values. Additionally, not all parameters were tested for research question four, and a bias could have been introduced for the choice of parameters selected. We attempted to mitigate this threat by choosing similar parameters between each of the tested models; however, future work could be done to measure the sensitivity of the selected parameters. For the attribute values, we also ensured that at least 1 instance would be found, such that we did not report a trivially unsatisfiable result.

The second threat is a bias towards the current implementation of Clafer and its supported tool-chain, the release *0.4.3*. Since Clafer and its tools are continually evolving over time, our results may become invalid; however, the results are valid for the chosen release.

The last threat is the use of a CSP based solver, chocosolver, opposed to a possibly more suitable one such as an SMT or ILP solver. We chose to use a CSP over SMT solver since our earlier work of encoding Clafer models to the SMT solver Z3 resulted in a worse performance compared to chocosolver. In particular, the Clafer backend using chocosolver has been optimized using custom constraint propagators, which are not available in SMT. However, a more efficient encoding of Clafer to SMT might have achieved better performance and should be investigated, along with an efficient encoding to ILP, in future works. Furthermore, a direct encoding of the case studies to SMT or ILP was not considered since one of the objectives of this work was to create high-level models.

Chapter 9

Related Work

9.1 Survey of Architecture Optimization

In 2013, Aleti et. al. [10] conducted a survey of the current literature in the area of architecture optimization. In the review, they considered not only embedded systems but also information systems. Many of the works surveyed are related to ours; however, in this section we highlight works with the most decisions, constraints, and objectives.

From the survey, we found the most design decisions (also known as degrees of freedom or points of variability) simultaneously considered by any work, in the embedded systems domain, was four [18,19,53]. These works considered degrees of freedom such as: allocation, component selection, hardware selection, software selection, software replication, hardware replication, and scheduling. In our work, we consider additional degrees of freedom (see Table 4.1); however, we do not take into account software or hardware replication.

From the survey, we found the most qualities considered in optimization, in the embedded systems domain, was four. This was work by Dave and Jha [21], in which they optimized the allocation of a hierarchal task graph to hardware, and its schedule with respect to performance, cost, reliability, and energy. In our work, we consider a simplified set of optimization objectives using mainly aggregation; however, we consider 15 degrees of freedom in our optimization while Dave and Jha only consider three.

Lastly, we found that many of the works considered constraints over qualities such as: performance, cost, realiabilty, energy, safety, and weight. However, from the surveyed works, in the embedded systems domain, they only considered a handful of constraints; on the other hand, in our work we consider 13 different quality constraints.

9.2 Recent Advances in E/E Architecture Evaluation & Optimization

Li et. al. [38] presented a framework which takes as input an AADL model, using OSATE, and optimizes with respect to cost, processor utilization, and data latency. This framework is limited by only considering the removal or addition of a processor or bus to the system, allocation of services, and replacement between hardware components; compared to our work, which we take into account feature variability and variable types of physical media. As stated earlier in Chapter 7, Aleti et. al. [9] also used OSATE to create AADL models and then translated them to their optimization framework, ArcheOpterix, to optimize based on data transmission reliability and communication overhead using evolutionary algorithms. Our work differed by covering variability and multiple layers and also using an exact optimization method.

Zeller et. al. [66] compared SAT-solving and different heuristic search algorithms, in which they do not optimize but rather synthesize a satisfying deployment given a set of constraints. The constraints they consider are timing, resource utilization, and schedulability. They do not optimize, or consider variable hardware topologies, analog/discrete wires for communication, or quality constraints such as mass, part cost, and warranty parts cost.

Glaß et. al. [28] optimized hardware topologies w.r.t. area cost, power consumption, and reliability using evolutionary algorithms. However, they do not consider variability in the functional architecture, which could potentially lead to different optimal topologies due to deployment based on what functions are present. Additionally, they only consider networked topologies, whereas in this work we consider both buses and discrete/analog connectors.

Lin et. al [41] considered the wire routing problem, in which they optimized w.r.t cost, which was dependent on wire selection, wire length, and number of splices used. This work goes into more detail of the wire selection and routing through the vehicle, which we disregarded given our focus on early design.

As stated in Chapter 7, Kugele and Pucea introduced AAOL in [36]. The case study used in their work was a power window system, similar to ours; however, ours considered more than two concrete designs. In our model, the single door variant alone encoded over 36,000 different candidate architectures as a result of being able to augment architectural elements with variability.

Walla et. al. [29] present a novel framework for early design exploration with respect to power consumption by ECUs. This work generates the inputs needed for their earlier work of simulating the power consumption and also provides some visualizations of the results. However, this exploration is less automated than ours since users have to clone a modeled E/E architecture and then make modifications to add differences. On the other hand, Clafer only models the structure of the architectures and does not support simulation or behavioral analysis.

In [27], Florentz and Huhn present meta-models to describe embedded systems architectures as well as an evaluation method. The case study used to demonstrate is similar to our power window model in which they consider two distinct variants and only two objectives: cost and performance. In comparison to our work, we provide an automated approach using a backend reasoner such that we can synthesize the candidates rather than having to explicitly model the variants.

Brandt et. al. in [16] present an optimization approach for cost of an automotive E/E architecture and consider variable deployments and number of ECUs. Unlike in our work, they consider a comprehensive cost metric including development and manufacturing costs. However, their work is not tool supported and does not consider optional functions as we do in our work.

In [58] Schäuffele presents PREEvision for E/E architecture modeling and optimization. In this work, Schäuffele shows a rich modeling environment that captures the architecture in a multi-layered approach as we do in Clafer as well as multiple optimization objective such as weight and cost. While this work considers more layers (wire routing and layout), it does not model variability for architectural elements and consequently does not support synthesis of functional designs and device topologies; rather, the architectural variants to be compared have to be modeled explicitly.

Overall, these works only considered a limited amount of variability, only very few perspectives, and many did not consider the entire multi-layer architecture when doing the optimization (from the features down to the hardware topology).

9.3 Extensions to Previous Work

This paper extends work previously done by Murashkin in [49] in the following ways:

- Extended the original power window case study by adding:
  - Latency and warranty cost quality attributes;
  - Implementation choice in the definition of a analysis function and functional device.
- Revised the original reference model by providing structure for the power and communication topology layers, adding the deployment of function connectors to hardware connectors, and adding quality attributes into the reference model;
- Additional case study model for a central door locks system;
- Systematic characterization of the possible exploration scenarios using specification templates;
- Evaluation addressing five research questions;
- Scalability of the optimization improved by optimizing the chocosolver in response to the experience with the case studies;
- New strategies of using visualizer tools (such as including the design decisions in the parallel coordinate diagrams).

Chapter 10

Conclusions & Future Work

We presented a workflow using the Clafer modeling language and its supported tool chain to synthesize multi-level, multi-perspective candidate E/E architectures for two automotive body domain sub-systems. Additionally, we defined a reference model that supports early design such that models can be created with limited amount of information. We then used the reference model and gave the full details for two automotive body domain sub-systems: a power window system and a door locks system This work also allows for synthesizing candidates with variability at all layers of the E/E architecture (i.e. features, functions, deployment, etc.) and not just deployment or a single layer as with many previous works. By considering all layers and augmenting elements with variability in each of them, we are able to synthesize globally optimal candidate architectures captured by the reference model. Furthermore, by using Clafer, we are able to synthesize candidates based on many types of design decision, constraints, and objectives to support much richer design exploration scenarios compared to other tools.

In the future, we would like to improve on the solving performance of chocosolver to make it feasible to construct and find solutions for models of larger size and variability. One possibility is to exploit the modularity of systems in the models, such that individual sub-systems are solved first and then the results are used to explore their composition. By improving the solver, larger and more intricate case studies can be explored such as combining the power window and door locks systems as well as modeling the remainder of the body domain. In addition, new perspectives such as safety, memory, and energy consumption should be added to the reference model to further enrich the design space exploration possibilities.

APPENDICES

Appendix A

Full Source Code for Clafer Models

A.1 Reference Model

Listing A.1: Complete reference model

```
//---------------------------- Meta-Model Elements --------------------------//
// Meta-Model Elements - This section contains all meta-model elements that
// are used to model a general automotive E/E architecture. Most of the
// elements are adapted from the EAST-ADL v2 specification.

// System is our version of the EAST-ADL "System Model". The two are similiar
// but have a couple differences:
//  - The implementation level is ignored.
//  - The analysis level and design level have been combined into the
//      architecture
abstract System
abstract FeatureModel
abstract Architecture
abstract FunctionalAnalysis
abstract HardwareArchitecture
abstract DeviceNodeClassification
abstract CommTopology
abstract PowerTopology
abstract Deployment

```

```
// Some generic "types" of clafers. Some types don't have properties but
// are used to improve readability
abstract Feature

abstract FunctionalAnalysisComponent
    deployedTo -> DeviceNode
    xor implementation
        hardware
            [latency = baseLatency]
            [deployedTo.type in (EEDeviceType, SmartDeviceType)]
        software
            [latency = baseLatency*deployedTo.speedFactor]
            [deployedTo.type in SmartDeviceType]
    baseLatency -> integer // [ms]
    latency -> integer // [ms]
abstract AnalysisFunction : FunctionalAnalysisComponent
abstract FunctionalDevice : FunctionalAnalysisComponent
abstract FunctionConnector
    sender -> FunctionalAnalysisComponent
    receiver -> FunctionalAnalysisComponent
    deployedTo -> HardwareDataConnector ?
        [parent in this.deployedFrom]
        [(sender.deployedTo.dref, receiver.deployedTo.dref) in (deployedTo.endpoint.dref)]
    [(sender.deployedTo.dref = receiver.deployedTo.dref) <=> no this.deployedTo]
    latency -> integer // [us]
    messageSize -> integer // [byte]
    [if (deployedTo) then (latency = messageSize*deployedTo.transferTimePerSize) else (latency = 0)]


enum DeviceNodeType = SmartDeviceType | EEDeviceType | PowerDeviceType

abstract DeviceNode
    type -> DeviceNodeType
    speedFactor -> integer // unitless
    mass -> integer // [g]
    cost -> integer // [dollar]
    ppm -> integer // unitless
    replaceCost -> integer // [dollar]
    warrantyCost -> integer = ppm*replaceCost // [dollar per million]
    [(type in (PowerDeviceType, EEDeviceType)) => (speedFactor = 0)]

// Hardware Connection Mediums
```

```
abstract HardwareConnector
    length -> integer // [cm]
    mass -> integer // [mg]
    cost -> integer // [dollar per thousand]
abstract PowerConnector : HardwareConnector
    source -> DeviceNode
    sink -> DeviceNode
abstract LoadPowerConnector : PowerConnector
    [mass = Data.MassPerLength.LoadPowerConnector*length]
    [cost = Data.CostPerLength.LoadPowerConnector*length]
abstract DevicePowerConnector : PowerConnector
    [mass = Data.MassPerLength.DevicePowerConnector*length]
    [cost = Data.CostPerLength.DevicePowerConnector*length]

abstract HardwareDataConnector : HardwareConnector
    endpoint -> DeviceNode 2..*
    deployedFrom -> FunctionConnector 1..*
        [this.deployedTo = parent]
    transferTimePerSize -> integer // [us/byte]

abstract DiscreteDataConnector : HardwareDataConnector
    [mass = length*(#deployedFrom)*Data.MassPerLength.DiscreteDataConnector]
    [transferTimePerSize = 0]
    [cost = Data.CostPerLength.DiscreteDataConnector*length*(#deployedFrom)]

abstract AnalogDataConnector : HardwareDataConnector
    [mass = length*(#deployedFrom)*Data.MassPerLength.AnalogDataConnector]
    [transferTimePerSize = 0]
    [cost = Data.CostPerLength.AnalogDataConnector*length*(#deployedFrom)]

abstract BusConnector : HardwareDataConnector
    [endpoint.type = SmartDeviceType]
    xor type
        LowSpeedCAN
            [transferTimePerSize = Data.TimePerSize.LowSpeedCANBus]
            [mass = Data.MassPerLength.LowSpeedCANBus*length]
            [cost = Data.CostPerLength.LowSpeedCANBus*length]
        HighSpeedCAN
            [transferTimePerSize = Data.TimePerSize.HighSpeedCANBus]
            [mass = Data.MassPerLength.HighSpeedCANBus*length]
            [cost = Data.CostPerLength.HighSpeedCANBus*length]
        LIN
            [transferTimePerSize = Data.TimePerSize.LINBus]
            [mass = Data.MassPerLength.LINBus*length]
            [cost = Data.CostPerLength.LINBus*length]
        FlexRay
            [transferTimePerSize = Data.TimePerSize.FlexRayBus]
            [mass = Data.MassPerLength.FlexRayBus*length]
            [cost = Data.CostPerLength.FlexRayBus*length]

abstract LogicalBusBridge : HardwareDataConnector
    [endpoint.type = SmartDeviceType]
    bus -> BusConnector 2
    gatewayTransferTimePerSize -> integer // [us/byte]
    [transferTimePerSize = gatewayTransferTimePerSize + sum(bus.transferTimePerSize)]
    [length = 0]
    [mass = 0]
    [cost = 0]


// -------------------- Quality Attribute Data ---------------------------//
Data
    MassPerLength // [mg/cm]
        LoadPowerConnector -> integer = 185
        DevicePowerConnector -> integer = 104
        DiscreteDataConnector -> integer = 110
        AnalogDataConnector -> integer = 110
        LowSpeedCANBus -> integer = 20
        HighSpeedCANBus -> integer = 20
        LINBus -> integer = 20
        FlexRayBus -> integer = 40
    CostPerLength // [dollar per thousand / cm]
        LoadPowerConnector -> integer = 9
        DevicePowerConnector -> integer = 9
        DiscreteDataConnector -> integer = 13
        AnalogDataConnector -> integer = 13
        LowSpeedCANBus -> integer = 52
        HighSpeedCANBus -> integer = 104
        LINBus -> integer = 26
        FlexRayBus -> integer = 208
    TimePerSize // [us/byte]
        LowSpeedCANBus -> integer = 64
        HighSpeedCANBus -> integer = 32
        LINBus -> integer = 400
        FlexRayBus -> integer = 1
    ReferenceSpeedFactor -> integer = 10
```

A.2 Generalized Power Window

Listing A.2: Complete generalized E/E architecture for power window

```
//------------------- Power Window Abstract Clafer --------------------------//
// Power Window Abstract Clafer - This section contains all abstract clafers
// that detail a generic system/component that can be used in the concerete
// system model.

abstract WinSysFM : FeatureModel
    basicUpDown : Feature
    express : Feature ?
        expressUp : Feature ?

abstract WinSysFA : FunctionalAnalysis
    WinSwitch : FunctionalDevice
        [implementation.hardware]
        [baseLatency = 20]
    WinArbiter : AnalysisFunction
        [baseLatency = (if implementation.software then 1 else 5)]
    WinController : AnalysisFunction
        [implementation.software]
        [baseLatency = 2]
    WinMotor : FunctionalDevice
        [implementation.hardware]
        [baseLatency = 10]
    CurrentSensor : FunctionalDevice
        [implementation.hardware]
        [baseLatency = 5]

    localWinReq : FunctionConnector
        [sender = WinSwitch && receiver = WinArbiter]
        [messageSize = 1]
    winReq : FunctionConnector
        [sender = WinArbiter && receiver = WinController]
        [messageSize = 1]
    winCmd : FunctionConnector
        [sender = WinController && receiver = WinMotor]
        [messageSize = 2]
    current : FunctionConnector
        [sender = CurrentSensor && receiver = WinController]
        [messageSize = 1]

    PinchDetectionFA : FunctionalAnalysis ?
        PinchDetection : AnalysisFunction
            [implementation.software]
            [baseLatency = 2]
        PositionSensor : FunctionalDevice
            [implementation.hardware]
            [baseLatency = 10]
        object : FunctionConnector
            [sender = PinchDetection && receiver = WinController]
            [messageSize = 2]
        position : FunctionConnector
            [sender = PositionSensor && receiver = PinchDetection]
            [messageSize = 1]

abstract WinSysDN : DeviceNodeClassification
    BCM -> DeviceNode ?
    EC -> DeviceNode
    Switch : SwitchNode
        [type in (SmartDeviceType, EEDeviceType)]
        [baseMass = 173]
        [cost = 110]
        [replaceCost = 110]
        [if (type in SmartDeviceType) then (ppm = 50) else (ppm = 10)]
        [(type in SmartDeviceType) => (speedFactor = 10)]
    Motor : DeviceNode
        [type in (SmartDeviceType, EEDeviceType)]
        [mass = 453]
        [if (type in SmartDeviceType) then (cost = 107) else (cost = 122)
           ]
        [if (type in SmartDeviceType) then (ppm = 50) else (ppm = 20)]
        [if (type in SmartDeviceType) then (replaceCost = 107) else (
           replaceCost = 122)]
        [(type in SmartDeviceType) => (speedFactor = 10)]
    DoorInline : DeviceNode
        [type = PowerDeviceType]
        [mass = 10] //TODO: Not a realistic number
        [cost = 4] //TODO: Not a realistic number
        [ppm = 1]
        [replaceCost = 2] //TODO: Not a realistic number
    DoorModule : DeviceNode ?
```

```
        [type = SmartDeviceType]
        [mass = 368]
        [cost = 300]
        [ppm = 50]
        [replaceCost = 300]
        [speedFactor = 10]

abstract WinSysPT : PowerTopology
    dn -> WinSysDN

    inlineECDist -> integer
    inlineBCMDist -> integer


    MotorLoadPowerWire : LoadPowerConnector
        [sink = dn.Motor]
    SwitchLoadPowerWire : LoadPowerConnector ?
        [source = dn.DoorInline && sink = dn.Switch]
        [length = 45]
    DoorModuleLoadPowerWire : LoadPowerConnector ?
        [source = dn.DoorInline && sink = dn.DoorModule]
        [length = 35]
    DoorInlineLoadPowerWire : LoadPowerConnector
        [sink = dn.DoorInline]

    xor MotorLoadPowerConfig
        SwitchIsMotorDriver
            [MotorLoadPowerWire.source = dn.Switch]
            [MotorLoadPowerWire.length = 40]
            [DoorInlineLoadPowerWire.source = dn.EC.dref]
            [DoorInlineLoadPowerWire.length = inlineECDist]
            [SwitchLoadPowerWire && DoorInlineLoadPowerWire && no
               DoorModuleLoadPowerWire]
        DoorModuleIsMotorDriver
            [MotorLoadPowerWire.source = dn.DoorModule]
            [MotorLoadPowerWire.length = 30]
            [DoorInlineLoadPowerWire.source = dn.EC.dref]
            [DoorInlineLoadPowerWire.length = inlineECDist]
            [no SwitchLoadPowerWire && DoorInlineLoadPowerWire &&
               DoorModuleLoadPowerWire]
        BCMIsMotorDriver
            [MotorLoadPowerWire.source = dn.DoorInline]
            [MotorLoadPowerWire.length = 45]
            [DoorInlineLoadPowerWire.source = dn.BCM.dref]
            [DoorInlineLoadPowerWire.length = inlineBCMDist]
            [no SwitchLoadPowerWire && DoorInlineLoadPowerWire && no DoorModuleLoadPowerWire]
        MotorIsMotorDriver
            [MotorLoadPowerWire.source = dn.DoorInline]
            [MotorLoadPowerWire.length = 45]
            [DoorInlineLoadPowerWire.source = dn.EC.dref]
            [DoorInlineLoadPowerWire.length = inlineECDist]
            [no SwitchLoadPowerWire && DoorInlineLoadPowerWire && no DoorModuleLoadPowerWire]

    switchInlineDP : DevicePowerConnector ?
        [source = dn.DoorInline && sink = dn.Switch]
        [length = 45]

    motorInlineDP : DevicePowerConnector ?
        [source = dn.DoorInline && sink = dn.Motor]
        [length = 45]

    doorModuleInlineDP : DevicePowerConnector ?
        [source = dn.DoorInline && sink = dn.DoorModule]
        [length = 35]

    [doorModuleInlineDP <=> dn.DoorModule]

    inlineECDP : DevicePowerConnector ?
        [source = dn.EC.dref && sink = dn.DoorInline]
        [length = WinSysPT.inlineECDist]

abstract WinSysCT : CommTopology
    dn -> WinSysDN
    inlineBCMDist -> integer


    logicalLowSpeedBus : BusConnector ?
        [type.LIN || type.LowSpeedCAN]
        [length = 70+inlineBCMDist]
        [endpoint in (dn.Motor, dn.Switch, dn.DoorModule, dn.BCM.dref)]

    logicalSwitchMotorDisc : DiscreteDataConnector ?
        [endpoint = (dn.Switch, dn.Motor)]
        [length = 40]
    logicalSwitchBCMDisc: DiscreteDataConnector ?
        [endpoint = (dn.Switch, dn.BCM.dref)]
        [length = 45+inlineBCMDist]
    logicalMotorBCMDisc : DiscreteDataConnector ?
```

```
        [endpoint = (dn.Motor, dn.BCM.dref)]
        [length = 45+inlineBCMDist]
    logicalSwitchDoorModuleDisc : DiscreteDataConnector ?
        [endpoint = (dn.Switch, dn.DoorModule)]
        [length = 25]
    logicalMotorDoorModuleDisc : DiscreteDataConnector ?
        [endpoint = (dn.Motor, dn.DoorModule)]
        [length = 30]

abstract WinSysHA : HardwareArchitecture
    dn -> WinSysDN
    pt -> WinSysPT
    ct -> WinSysCT


abstract WinSysDpl : Deployment
    fa -> WinSysFA
    ha -> WinSysHA

    // The most general deployment constraint that we have is that the
    // FunctionalAnalysisComponents must be deployed to its own HardwareTopology
    [fa.WinArbiter.deployedTo.dref in (ha.dn.BCM.dref, ha.dn.Switch, ha.dn.Motor, ha.dn.DoorModule)]
    [fa.WinController.deployedTo.dref in (ha.dn.BCM.dref, ha.dn.Switch, ha.dn.Motor, ha.dn.DoorModule)]
    [fa.PinchDetectionFA => (fa.PinchDetectionFA.PinchDetection.deployedTo.dref in (ha.dn.BCM.dref, ha.dn.Switch, ha.dn.Motor, ha.dn.DoorModule))]

    // More specific constraints on functional analysis component...
    [fa.WinSwitch.deployedTo.dref = ha.dn.Switch]
    [fa.WinMotor.deployedTo.dref = ha.dn.Motor]
    [fa.CurrentSensor.deployedTo.dref = ha.dn.Motor]
    [fa.PinchDetectionFA => (fa.PinchDetectionFA.PositionSensor.deployedTo.dref = ha.dn.Motor)]

    // Constraints pertaining to the power topology selection based on analysis function deployment
    [(fa.WinController.deployedTo.dref = ha.dn.Switch) <=> ha.pt.MotorLoadPowerConfig.SwitchIsMotorDriver]
    [(fa.WinController.deployedTo.dref = ha.dn.Motor) <=> ha.pt.MotorLoadPowerConfig.MotorIsMotorDriver]
    [(fa.WinController.deployedTo.dref = ha.dn.BCM.dref) <=> ha.pt.MotorLoadPowerConfig.BCMIsMotorDriver]
    [(fa.WinController.deployedTo.dref = ha.dn.DoorModule) <=> ha.pt.
        MotorLoadPowerConfig.DoorModuleIsMotorDriver]

    [ha.pt.switchInlineDP <=> (ha.dn.Switch.type in SmartDeviceType)]
    [ha.pt.motorInlineDP <=> (ha.dn.Motor.type in SmartDeviceType)]
    [ha.pt.inlineECDP <=> some(ha.pt.motorInlineDP, ha.pt.switchInlineDP,
         ha.pt.doorModuleInlineDP)]

    // Constraints pertaining to the communication topology selected
        based on analysis function deployement
    [(fa.localWinReq.deployedTo.dref in (ha.ct.logicalLowSpeedBus, ha.ct.
        logicalSwitchMotorDisc, ha.ct.logicalSwitchBCMDisc, ha.ct.
        logicalMotorBCMDisc, ha.ct.logicalSwitchDoorModuleDisc, ha.ct.
        logicalMotorDoorModuleDisc))]
    [(fa.winReq.deployedTo.dref in (ha.ct.logicalLowSpeedBus, ha.ct.
        logicalSwitchMotorDisc, ha.ct.logicalSwitchBCMDisc, ha.ct.
        logicalMotorBCMDisc, ha.ct.logicalSwitchDoorModuleDisc, ha.ct.
        logicalMotorDoorModuleDisc))]
    [(fa.winCmd.deployedTo.dref in (ha.ct.logicalLowSpeedBus, ha.ct.
        logicalSwitchMotorDisc, ha.ct.logicalSwitchBCMDisc, ha.ct.
        logicalMotorBCMDisc, ha.ct.logicalSwitchDoorModuleDisc, ha.ct.
        logicalMotorDoorModuleDisc))]
    [(fa.current.deployedTo.dref in (ha.ct.logicalLowSpeedBus, ha.ct.
        logicalSwitchMotorDisc, ha.ct.logicalSwitchBCMDisc, ha.ct.
        logicalMotorBCMDisc, ha.ct.logicalSwitchDoorModuleDisc, ha.ct.
        logicalMotorDoorModuleDisc))]
    [(fa.PinchDetectionFA.object.deployedTo.dref in (ha.ct.
        logicalLowSpeedBus, ha.ct.logicalSwitchMotorDisc, ha.ct.
        logicalSwitchBCMDisc, ha.ct.logicalMotorBCMDisc, ha.ct.
        logicalSwitchDoorModuleDisc, ha.ct.logicalMotorDoorModuleDisc))]
    [(fa.PinchDetectionFA.position.deployedTo.dref in (ha.ct.
        logicalLowSpeedBus, ha.ct.logicalSwitchMotorDisc, ha.ct.
        logicalSwitchBCMDisc, ha.ct.logicalMotorBCMDisc, ha.ct.
        logicalSwitchDoorModuleDisc, ha.ct.logicalMotorDoorModuleDisc))]

abstract SwitchNode : DeviceNode
    numSwitches -> integer
    baseMass -> integer
    [mass = baseMass*numSwitches]
```

A.3 Two Door Power Window

Listing A.3: Complete E/E architecture for two door power window case study

```
//------------------- Power Window System Model ----------------------------//
// Power Window System Model - This section is the concrete model of the power
// window system. This is the model that instances will be generated for. It
// will heavily use the previous two sections.

// Driver Window System
DriverWinSys : System
    DWinSysFM : WinSysFM
    DWinSysFA : WinSysFA
        [DriverWinSys.DWinSysFM.express.expressUp <=> DWinSysFA.PinchDetectionFA]
        // Timing Chains
        SwitchToControlDeviceLatency -> integer = WinSwitch.latency + WinArbiter.latency
        ControlToMotorDeviceLatency -> integer = WinController.latency + WinMotor.latency
        SwitchToControlCommLatency -> integer = localWinReq.latency + winReq.latency
        ControlToMotorCommLatency -> integer = winCmd.latency
        SwitchToMotorEndToEndLatency -> integer = SwitchToControlDeviceLatency + ControlToMotorDeviceLatency + (SwitchToControlCommLatency+ControlToMotorCommLatency)/1000

        PositionSensorToControlDeviceLatency -> integer = PositionSensor.latency + PinchDetection.latency
        PositionSensorToControlCommLatency -> integer = position.latency + object.latency
        PositionSensorToMotorEndToEndLatency -> integer = PositionSensorToControlDeviceLatency + ControlToMotorDeviceLatency + (PositionSensorToControlCommLatency+ControlToMotorCommLatency)/1000

        SwitchToControlLatency -> integer = SwitchToControlDeviceLatency + SwitchToControlCommLatency/1000
        PositionSensorToControlLatency -> integer = PositionSensorToControlDeviceLatency + PositionSensorToControlCommLatency/1000
        ControlInputDifference -> integer
        [ControlInputDifference = (max(SwitchToControlLatency.dref, PositionSensorToControlLatency.dref)
            - min(SwitchToControlLatency.dref,
               PositionSensorToControlLatency.dref))]

        // End-to-End Timing Constraint(s)
        [(TimingRequirements.BasicEndToEndLatency) => (
           SwitchToMotorEndToEndLatency <= TimingRequirements.
           BasicEndToEndLatency)]
        [(PinchDetectionFA && TimingRequirements.
           PinchDetectionEndToEndLatency) => (
           PositionSensorToMotorEndToEndLatency <= TimingRequirements.
           PinchDetectionEndToEndLatency)]
        // Input Synchronization Contraint(s)
        [(PinchDetectionFA && TimingRequirements.ControlInputSynchLatency
           ) => ControlInputDifference <= TimingRequirements.
           ControlInputSynchLatency]

        // Timing Margins
        BasicEndToEndLatencyMargin -> integer ?
        [if TimingRequirements.BasicEndToEndLatency then (
           BasicEndToEndLatencyMargin = (TimingRequirements.
           BasicEndToEndLatency - SwitchToMotorEndToEndLatency))
            else (no BasicEndToEndLatencyMargin)]
        PinchDetectionEndToEndLatencyMargin -> integer ?
        [if TimingRequirements.PinchDetectionEndToEndLatency then (
            PinchDetectionEndToEndLatencyMargin = (TimingRequirements.
               PinchDetectionEndToEndLatency -
               PositionSensorToMotorEndToEndLatency))
            else (no PinchDetectionEndToEndLatencyMargin)]
    DWinSysHA : WinSysHA
        DWinSysDN : WinSysDN
            [this.BCM = Car.BCM]
            [this.EC = Car.EC]
            [this.Switch.numSwitches = 2]
        DWinSysPT : WinSysPT
            [dn = DWinSysDN]
            [inlineECDist = 40]
            [inlineBCMDist = 40]
        DWinSysCT : WinSysCT
            [dn = DWinSysDN]
            [inlineBCMDist = 40]
        [dn = DWinSysDN]
        [pt = DWinSysPT]
        [ct = DWinSysCT]
    DWinSysDpl : WinSysDpl
        [fa = DWinSysFA]
        [ha = DWinSysHA]
```

```


// Passenger Window System
PassengerWinSys : System
    PWinSysFM : WinSysFM
        [express => DriverWinSys.DWinSysFM.express]
        [express.expressUp => DriverWinSys.DWinSysFM.express.expressUp]
    PWinSysFA : WinSysFA
        [PassengerWinSys.PWinSysFM.express.expressUp <=> PWinSysFA.
           PinchDetectionFA]
        DWinSwitch : FunctionalDevice
            [implementation.hardware]
            [baseLatency = 10]
        dWinReq : FunctionConnector
            [sender = DWinSwitch && receiver = WinArbiter]
            [messageSize = 1]

        // Timing Chains
        SwitchToControlDeviceLatency -> integer = WinSwitch.latency +
           WinArbiter.latency
        ControlToMotorDeviceLatency -> integer = WinController.latency +
           WinMotor.latency
        SwitchToControlCommLatency -> integer = localWinReq.latency +
           winReq.latency
        ControlToMotorCommLatency -> integer = winCmd.latency
        SwitchToMotorEndToEndLatency -> integer =
           SwitchToControlDeviceLatency + ControlToMotorDeviceLatency +
           (SwitchToControlCommLatency+ControlToMotorCommLatency)/1000

        PositionSensorToControlDeviceLatency -> integer = PositionSensor.
           latency + PinchDetection.latency
        PositionSensorToControlCommLatency -> integer = position.latency
           + object.latency
        PositionSensorToMotorEndToEndLatency -> integer =
           PositionSensorToControlDeviceLatency +
           ControlToMotorDeviceLatency + (
           PositionSensorToControlCommLatency+ControlToMotorCommLatency)
           /1000

        SwitchToControlLatency -> integer = SwitchToControlDeviceLatency
           + SwitchToControlCommLatency/1000
        PositionSensorToControlLatency -> integer =
           PositionSensorToControlDeviceLatency +
           PositionSensorToControlCommLatency/1000
        ControlInputDifference -> integer
        [ControlInputDifference = (max(SwitchToControlLatency.dref,
           PositionSensorToControlLatency.dref)
            - min(SwitchToControlLatency.dref,
               PositionSensorToControlLatency.dref))]

        // End-to-End Timing Constraint(s)
        [(TimingRequirements.BasicEndToEndLatency) => (
           SwitchToMotorEndToEndLatency <= TimingRequirements.
           BasicEndToEndLatency)]
        [(PinchDetectionFA && TimingRequirements.
           PinchDetectionEndToEndLatency) => (
           PositionSensorToMotorEndToEndLatency <= TimingRequirements.
           PinchDetectionEndToEndLatency)]
        // Input Synchronization Contraint(s)
        [(PinchDetectionFA && TimingRequirements.ControlInputSynchLatency
           ) => ControlInputDifference <= TimingRequirements.
           ControlInputSynchLatency]

        // Timing Margins
        BasicEndToEndLatencyMargin -> integer ?
        [if TimingRequirements.BasicEndToEndLatency then (
           BasicEndToEndLatencyMargin = (TimingRequirements.
           BasicEndToEndLatency - SwitchToMotorEndToEndLatency))
            else (no BasicEndToEndLatencyMargin)]
        PinchDetectionEndToEndLatencyMargin -> integer ?
        [if TimingRequirements.PinchDetectionEndToEndLatency then (
            PinchDetectionEndToEndLatencyMargin = (TimingRequirements.
               PinchDetectionEndToEndLatency -
               PositionSensorToMotorEndToEndLatency))
            else (no PinchDetectionEndToEndLatencyMargin)]

    PWinSysHA : WinSysHA
        PWinSysDN : WinSysDN
            [this.BCM = Car.BCM]
            [this.EC = Car.EC]
            [this.Switch.numSwitches = 1]
            DSwitch -> SwitchNode
            [DSwitch = DriverWinSys.DWinSysHA.DWinSysDN.Switch]
        PWinSysPT : WinSysPT
            [dn = PWinSysDN]
            [inlineECDist = 130]
            [inlineBCMDist = 130]
        PWinSysCT : WinSysCT
            [dn = PWinSysDN]
            [inlineBCMDist = 130]
```

```
            logicalDoorBusJoin : LogicalBusBridge ?
                [bus = (PWinSysCT.logicalLowSpeedBus, DWinSysCT.logicalLowSpeedBus)]
                [gatewayTransferTimePerSize = 10] // This is the time to transfer a unit size over the gateway
                [endpoint in (PWinSysDN.Motor, PWinSysDN.Switch, PWinSysDN.DoorModule, PWinSysDN.BCM.dref, DWinSysDN.Motor, DWinSysDN.Switch, DWinSysDN.DoorModule)]
            logicalDriveSwitchPassSwitch : DiscreteDataConnector ?
                [length = 260]
                [endpoint = (PWinSysHA.PWinSysDN.DSwitch.dref, PWinSysHA.PWinSysDN.Switch)]
            logicalDriveSwitchPassMotor : DiscreteDataConnector ?
                [length = 260]
                [endpoint = (PWinSysHA.PWinSysDN.DSwitch.dref, PWinSysHA.PWinSysDN.Motor)]
            logicalDriveSwitchPassDoorModule : DiscreteDataConnector ?
                [length = 250]
                [endpoint = (PWinSysHA.PWinSysDN.DSwitch.dref, PWinSysHA.PWinSysDN.DoorModule)]
            logicalDriveSwitchBCM : DiscreteDataConnector ?
                [length = 85]
                [endpoint = (PWinSysHA.PWinSysDN.DSwitch.dref, PWinSysHA.PWinSysDN.BCM.dref)]
        [dn = PWinSysDN]
        [pt = PWinSysPT]
        [ct = PWinSysCT]
    PWinSysDpl : WinSysDpl
        [fa = PWinSysFA]
        [ha = PWinSysHA]
        [PWinSysFA.DWinSwitch.deployedTo.dref = PWinSysHA.PWinSysDN.DSwitch.dref]
        [PWinSysFA.dWinReq.deployedTo.dref in (
            PWinSysHA.PWinSysCT.logicalDoorBusJoin,
            PWinSysHA.PWinSysCT.logicalDriveSwitchPassSwitch,
            PWinSysHA.PWinSysCT.logicalDriveSwitchPassMotor,
            PWinSysHA.PWinSysCT.logicalDriveSwitchPassDoorModule,
            PWinSysHA.PWinSysCT.logicalDriveSwitchBCM)]

//----------------------------- Car System Model ----------------------------//
Car
    BCM : DeviceNode ?
        [type = SmartDeviceType]
        [mass = 408]
        [cost = 460]
        [ppm = 50]
        [replaceCost = 460]
        [speedFactor = 10]
    EC : DeviceNode
        [type = PowerDeviceType]
        [mass = 0]
        [cost = 0]
        [ppm = 10]
        [replaceCost = 0]


totalCarMass -> integer = sum(DeviceNode.mass) + (sum(HardwareConnector.mass)/1000)
totalCarCost -> integer = sum(DeviceNode.cost) + (sum(HardwareConnector.cost)/1000)
totalCarWarrantyCost -> integer = sum(DeviceNode.warrantyCost)


// Timing Requirements
TimingRequirements
    BasicEndToEndLatency -> integer ?
    PinchDetectionEndToEndLatency -> integer ?
    ControlInputSynchLatency -> integer ?

// Optimization Goals:
// Comment out these goals if optimization should not be perfromed (no other modifications are necessary)
// << minimize totalCarMass >>
// << minimize totalCarCost >>
// << minimize totalCarWarrantyCost >>
```

A.4 Central Door Locks

Listing A.4: Complete E/E architecture for central door locks case study

```
//------------------- Door Lock Abstract Clafer --------------------------//
// Door Lock Abstract Clafer - This section contains all abstract clafers
// that detail a generic system/component that can be used in the concerete
// system model.
abstract DoorLockFA : FunctionalAnalysis
    // -------  Core Components ---------------//
    // Cylinder Switches
    DriverDoorCylinderSwitch : FunctionalDevice
        [implementation.hardware]
        [baseLatency = 10]
    PassDoorCylinderSwitch : FunctionalDevice
        [implementation.hardware]
        [baseLatency = 10]
    driverCylReq : FunctionConnector
        [messageSize = 1]
        [sender = DriverDoorCylinderSwitch && receiver = DoorLockControl]
    passCylReq : FunctionConnector
        [messageSize = 1]
        [sender = PassDoorCylinderSwitch && receiver = DoorLockControl]
    // Door Contacts
    DriverDoorContact : FunctionalDevice
        [implementation.hardware]
        [baseLatency = 10]
    PassDoorContact : FunctionalDevice
        [implementation.hardware]
        [baseLatency = 10]
    RearRightPassDoorContact : FunctionalDevice
        [implementation.hardware]
        [baseLatency = 10]
    RearLeftPassDoorContact : FunctionalDevice
        [implementation.hardware]
        [baseLatency = 10]
    driverContactSignal : FunctionConnector
        [messageSize = 1]
        [sender = DriverDoorContact && receiver = DoorLockControl]
    passContactSignal : FunctionConnector
        [messageSize = 1]
        [sender = PassDoorContact && receiver = DoorLockControl]
    rearRightPassContactSignal : FunctionConnector
        [messageSize = 1]
        [sender = RearRightPassDoorContact && receiver = DoorLockControl]
    rearLeftPassContactSignal : FunctionConnector
        [messageSize = 1]
        [sender = RearLeftPassDoorContact && receiver = DoorLockControl]
    // Door Lock Sensors
    DriverDoorLockSensor : FunctionalDevice
        [implementation.hardware]
        [baseLatency = 10]
    PassDoorLockSensor : FunctionalDevice
        [implementation.hardware]
        [baseLatency = 10]
    RearRightPassDoorLockSensor : FunctionalDevice
        [implementation.hardware]
        [baseLatency = 10]
    RearLeftPassDoorLockSensor : FunctionalDevice
        [implementation.hardware]
        [baseLatency = 10]
    driverLockPosition : FunctionConnector
        [messageSize = 1]
        [sender = DriverDoorLockSensor && receiver = DoorLockControl]
    passLockPosition : FunctionConnector
        [messageSize = 1]
        [sender = PassDoorLockSensor && receiver = DoorLockControl]
    rearRightPassLockPosition : FunctionConnector
        [messageSize = 1]
        [sender = RearRightPassDoorLockSensor && receiver =
           DoorLockControl]
    rearLeftPassLockPosition : FunctionConnector
        [messageSize = 1]
        [sender = RearLeftPassDoorLockSensor && receiver =
           DoorLockControl]
    // Door Lock Control
    DoorLockControl : AnalysisFunction
        [implementation.software]
        [baseLatency = 4]
    driverLockCmd : FunctionConnector
        [messageSize = 1]
        [sender = DoorLockControl && receiver = DriverDoorLockMotor]
    passLockCmd : FunctionConnector
        [messageSize = 1]
        [sender = DoorLockControl && receiver = PassDoorLockMotor]
    rearRightLockCmd : FunctionConnector
        [messageSize = 1]
        [sender = DoorLockControl && receiver =
           RearRightPassDoorLockMotor]
    rearLeftLockCmd : FunctionConnector
        [messageSize = 1]
        [sender = DoorLockControl && receiver = RearLeftPassDoorLockMotor
           ]
    // Door Lock Motor
    DriverDoorLockMotor : FunctionalDevice
        [implementation.hardware]
        [baseLatency = 10]
    PassDoorLockMotor : FunctionalDevice
        [implementation.hardware]
        [baseLatency = 10]
    RearRightPassDoorLockMotor : FunctionalDevice
        [implementation.hardware]
        [baseLatency = 10]
    RearLeftPassDoorLockMotor : FunctionalDevice
        [implementation.hardware]
        [baseLatency = 10]
    // Gear Position Sensor
    GearPositionSensor : FunctionalDevice
        [implementation.hardware]
        [baseLatency = 10]
    gearPostion : FunctionConnector
        [messageSize = 1]
        [sender = GearPositionSensor && receiver = DoorLockControl]

    // ----- Optional Fragments/Components --------//
    // Speed Smart Lock FA Components
    SpeedSmartLockFA : FunctionalAnalysis ?
        SpeedSensor : FunctionalDevice
            [implementation.hardware]
            [baseLatency = 10]
        speed : FunctionConnector
            [messageSize = 1]
            [sender = SpeedSensor && receiver = DoorLockControl]
    // Central or Distributed Lock Switch
    xor DoorLockButtonFA
        IndividualLockSwitchFA : FunctionalAnalysis
            DriverDoorLockButton : FunctionalDevice
                [implementation.hardware]
                [baseLatency = 10]
            PassDoorLockButton : FunctionalDevice
                [implementation.hardware]
                [baseLatency = 10]
            driverDoorLockReq : FunctionConnector
                [messageSize = 1]
                [sender = DriverDoorLockButton && receiver =
                   DoorLockControl]
            passDoorLockReq : FunctionConnector
                [messageSize = 1]
                [sender = PassDoorLockButton && receiver =
                   DoorLockControl]
        CentralLockSwitchFA : FunctionalAnalysis
            CentralLockButton : FunctionalDevice
                [implementation.hardware]
                [baseLatency = 10]
            centralDoorLockReq : FunctionConnector
```

```
                [messageSize = 1]
                [sender = CentralLockButton && receiver = DoorLockControl]

    RemoteKeyAccessFA : FunctionalAnalysis ?
        CentralRFAntenna : FunctionalDevice
            [implementation.hardware]
            [baseLatency = 10]
        CentralRFReceiver : FunctionalDevice
            [implementation.hardware]
            [baseLatency = 10]
        IDAuthentication : AnalysisFunction
            [implementation.software]
            [baseLatency = 4]

        centralAntennaSignal : FunctionConnector
            [messageSize = 1]
            [sender = CentralRFAntenna && receiver = CentralRFReceiver]
        centralReceiverMsg : FunctionConnector
            [messageSize = 1]
            [sender = CentralRFReceiver && receiver = IDAuthentication]
        authenticationMsg : FunctionConnector
            [messageSize = 1]
            [sender = IDAuthentication && receiver = DoorLockControl]

    PassiveKeyEntryFA : FunctionalAnalysis ?
        DriverOutsideLFAntenna : FunctionalDevice
            [implementation.hardware]
            [baseLatency = 10]
        DriverLFTransmitter : FunctionalDevice
            [implementation.hardware]
            [baseLatency = 10]
        PassOutsideLFAntenna : FunctionalDevice
            [implementation.hardware]
            [baseLatency = 10]
        PassLFTransmitter : FunctionalDevice
            [implementation.hardware]
            [baseLatency = 10]
        InsideFrontLFAntenna : FunctionalDevice
            [implementation.hardware]
            [baseLatency = 10]
        InsideCenterLFAntenna : FunctionalDevice
            [implementation.hardware]
            [baseLatency = 10]
        InsideRearLFAntenna : FunctionalDevice
```

```
            [implementation.hardware]
            [baseLatency = 10]
        InsideLFTransmitter : FunctionalDevice
            [implementation.hardware]
            [baseLatency = 10]

        driverTransMsg : FunctionConnector
            [messageSize = 1]
            [sender = DriverLFTransmitter && receiver = DriverOutsideLFAntenna]
        passTransMsg : FunctionConnector
            [messageSize = 1]
            [sender = PassLFTransmitter && receiver = PassOutsideLFAntenna]
        insideFrontTransMsg : FunctionConnector
            [messageSize = 1]
            [sender = InsideLFTransmitter && receiver = InsideFrontLFAntenna]
        insideCenterTransMsg : FunctionConnector
            [messageSize = 1]
            [sender = InsideLFTransmitter && receiver = InsideCenterLFAntenna]
        insideRearTransMsg : FunctionConnector
            [messageSize = 1]
            [sender = InsideLFTransmitter && receiver = InsideRearLFAntenna]


        xor OutsideDoorHandleSensor
            ButtonSensor
                DriverDoorButtonSensor : FunctionalDevice
                    [implementation.hardware]
                    [baseLatency = 10]
                PassDoorButtonSensor : FunctionalDevice
                    [implementation.hardware]
                    [baseLatency = 10]
            CapacitiveSensor
                DriverDoorCapacitiveSensor : FunctionalDevice
                    [implementation.hardware]
                    [baseLatency = 10]
                PassDoorCapacitiveSensor : FunctionalDevice
                    [implementation.hardware]
                    [baseLatency = 10]

        PKEControl : AnalysisFunction
```

```
            [implementation.software]
            [baseLatency = 4]

        driverDoorHandleReq : FunctionConnector
            [messageSize = 1]
            [sender in (OutsideDoorHandleSensor.ButtonSensor.
               DriverDoorButtonSensor,
                OutsideDoorHandleSensor.CapacitiveSensor.
                   DriverDoorCapacitiveSensor) && receiver = PKEControl]
        passDoorHandleReq : FunctionConnector
            [messageSize = 1]
            [sender in (OutsideDoorHandleSensor.ButtonSensor.
               PassDoorButtonSensor,
                OutsideDoorHandleSensor.CapacitiveSensor.
                   PassDoorCapacitiveSensor) && receiver = PKEControl]
        driverPKEReq : FunctionConnector
            [messageSize = 1]
            [sender = PKEControl && receiver = DriverLFTransmitter]
        passPKEReq : FunctionConnector
            [messageSize = 1]
            [sender = PKEControl && receiver = PassLFTransmitter]
        insidePKEReq : FunctionConnector
            [messageSize = 1]
            [sender = PKEControl && receiver = InsideLFTransmitter]
        doorLockControlReq : FunctionConnector
            [messageSize = 1]
            [sender = DoorLockControl && receiver = PKEControl]

abstract DoorLockDN : DeviceNodeClassification
    //---------- Core Device Nodes ----------------//
    DriverDoorLockMotorAssembly : DeviceNode
        [type = EEDeviceType]
        [cost = 144]
        [ppm = 20]
        [replaceCost = 144]
        [mass = 104]
    PassengerDoorLockMotorAssembly : DeviceNode
        [type = EEDeviceType]
        [cost = 144]
        [ppm = 20]
        [replaceCost = 144]
        [mass = 104]
    RearRightPassengerDoorLockMotorAssembly : DeviceNode
        [type = EEDeviceType]
        [cost = 144]
        [ppm = 20]
        [replaceCost = 144]
        [mass = 104]
    RearLeftPassengerDoorLockMotorAssembly : DeviceNode
        [type = EEDeviceType]
        [cost = 144]
        [ppm = 20]
        [replaceCost = 144]
        [mass = 104]
    TCM -> DeviceNode
    BCM -> DeviceNode
    EC -> DeviceNode

    // ------- Optional Device Nodes ---------------//
    // Speed Smart Lock Nodes
    CombinationMeter -> DeviceNode ?

    // Central or Individual Lock Nodes
    xor DoorLockButtonDN
        IndividualLockSwitchDN : DeviceNodeClassification
            DriverLockPowerSwitch : DeviceNode
                [type = EEDeviceType]
                [cost = 23]
                [replaceCost = 23]
                [ppm = 10]
                [mass = 28]
            PassLockPowerSwitch : DeviceNode
                [type = EEDeviceType]
                [cost = 23]
                [replaceCost = 23]
                [ppm = 10]
                [mass = 28]
        CentralLockSwitchDN
            CenterLockPowerSwitch : DeviceNode
                [type = EEDeviceType]
                [cost = 23]
                [replaceCost = 23]
                [ppm = 10]
                [mass = 28]

    RemoteKeyAccessDN : DeviceNodeClassification ?
        CentralRFAntennaModule : DeviceNode
            [type = SmartDeviceType]
            [mass = 91]
            [cost = 57]
            [ppm = 50]
            [replaceCost = 57]
            [speedFactor = 10]

    PassiveKeyEntryDN : DeviceNodeClassification ?
        Transmitter : DeviceNode ?
            [type = EEDeviceType]
            [mass = 397]
            [cost = 239]
            [ppm = 50]
            [replaceCost = 293]
        PassiveKeyModule : DeviceNode ?
            [type = SmartDeviceType]
            [mass = 408]
            [cost = 191]
            [ppm = 50]
            [replaceCost = 191]
            [speedFactor = 50]
        xor OutsideDoorHandleSensor
            ButtonSensor
                DriverDoorButtonHandleModule : DeviceNode
                    [type = EEDeviceType]
                    [mass = 408]
                    [cost = 41]
                    [ppm = 10]
                    [replaceCost = 41]
                PassDoorButtonHandleModule : DeviceNode
                    [type = EEDeviceType]
                    [mass = 408]
                    [cost = 41]
                    [ppm = 10]
                    [replaceCost = 41]
            CapacitiveSensor
                DriverDoorCapacitiveHandleModule : DeviceNode
                    [type = EEDeviceType]
                    [mass = 198]
                    [cost = 218]
                    [ppm = 10]
                    [replaceCost = 218]
                PassDoorCapacitiveHandleModule : DeviceNode
                    [type = EEDeviceType]
                    [mass = 198]
                    [cost = 218]
                    [ppm = 10]
                    [replaceCost = 218]
        InsideFrontLFAntenna : DeviceNode
            [type = EEDeviceType]
            [mass = 198]
            [cost = 57]
            [ppm = 10]
            [replaceCost = 57]
        InsideCenterLFAntenna : DeviceNode
            [type = EEDeviceType]
            [mass = 198]
            [cost = 57]
            [ppm = 10]
            [replaceCost = 57]
        InsideRearLFAntenna : DeviceNode
            [type = EEDeviceType]
            [mass = 198]
            [cost = 57]
            [ppm = 10]
            [replaceCost = 57]

abstract DoorLockPT : PowerTopology
    dn -> DoorLockDN

    // Motor Load Power
    driverMotorLP : LoadPowerConnector
        [length = 10]
        [source = dn.BCM.dref && sink = dn.DriverDoorLockMotorAssembly]
    passMotorLP : LoadPowerConnector
        [length = 15]
        [source = dn.BCM.dref && sink = dn.PassengerDoorLockMotorAssembly
           ]
    rearRightPassMotorLP : LoadPowerConnector
        [length = 25]
        [source = dn.BCM.dref && sink = dn.
           RearRightPassengerDoorLockMotorAssembly]
    rearLeftPassMotorLP : LoadPowerConnector
        [length = 30]
        [source = dn.BCM.dref && sink = dn.
           RearLeftPassengerDoorLockMotorAssembly]

    // Remote Key Access Device Power
    centralRFModuleDP : DevicePowerConnector ?
        [length = 10]
        [source = dn.EC.dref && sink = dn.RemoteKeyAccessDN.
           CentralRFAntennaModule]

    // Passive Key Entry Device Power
    pkeModuleDP : DevicePowerConnector ?
        [length = 4]
        [source = dn.EC.dref && sink = dn.PassiveKeyEntryDN.PassiveKeyModule]
    transmitterDP : DevicePowerConnector ?
        [length = 5]
        [source = dn.EC.dref && sink = dn.PassiveKeyEntryDN.Transmitter]
    driverCapacitiveSensorDP : DevicePowerConnector ?
        [length = 11]
        [source = dn.EC.dref && sink = dn.PassiveKeyEntryDN.OutsideDoorHandleSensor.CapacitiveSensor.DriverDoorCapacitiveHandleModule]
    passCapacitiveSensorDP : DevicePowerConnector ?
        [length = 16]
        [source = dn.EC.dref && sink = dn.PassiveKeyEntryDN.OutsideDoorHandleSensor.CapacitiveSensor.PassDoorCapacitiveHandleModule]

abstract DoorLockCT : CommTopology
    dn -> DoorLockDN

    // Busses
    logicalLowSpeedBus : BusConnector ? // This is the logical bus connecting lower priority ECU's such as in the body domain
        [type.LIN || type.LowSpeedCAN]
        [length = 45]
        [endpoint in (dn.BCM.dref, dn.RemoteKeyAccessDN.CentralRFAntennaModule, dn.PassiveKeyEntryDN.PassiveKeyModule)]
    logicalHighSpeedBus : BusConnector // This is the logical bus connecting high priority ECU's such as vehicle control
        [type.HighSpeedCAN || type.FlexRay]
        [length = 30]
        [endpoint in (dn.BCM.dref, dn.TCM.dref, dn.CombinationMeter.dref)]

    // Logical Discrete Wires
    logicalBCMDriverMotorAssemblyDW : DiscreteDataConnector
        [length = 12]
        [endpoint = (dn.BCM.dref, dn.DriverDoorLockMotorAssembly)]
    logicalBCMPassMotorAssemblyDW : DiscreteDataConnector
        [length = 17]
        [endpoint = (dn.BCM.dref, dn.PassengerDoorLockMotorAssembly)]
    logicalBCMRearRightPassMotorAssemblyDW : DiscreteDataConnector
```

```
        [length = 27]
        [endpoint = (dn.BCM.dref, dn.
           RearRightPassengerDoorLockMotorAssembly)]
    logicalBCMRearLeftPassMotorAssemblyDW : DiscreteDataConnector
        [length = 32]
        [endpoint = (dn.BCM.dref, dn.
           RearLeftPassengerDoorLockMotorAssembly)]
    logicalBCMDriverLockPowerSwitchDW : DiscreteDataConnector ?
        [length = 14]
        [endpoint = (dn.BCM.dref, dn.DoorLockButtonDN.
           IndividualLockSwitchDN.DriverLockPowerSwitch)]
    logicalBCMPassLockPowerSwitchDW : DiscreteDataConnector ?
        [length = 19]
        [endpoint = (dn.BCM.dref, dn.DoorLockButtonDN.
           IndividualLockSwitchDN.PassLockPowerSwitch)]
    logicalBCMCenterLockPowerSwitchDW : DiscreteDataConnector ?
        [length = 3]
        [endpoint = (dn.BCM.dref, dn.DoorLockButtonDN.CentralLockSwitchDN
           .CenterLockPowerSwitch)]

    logicalBCMDriverCapacitiveSensorModule : AnalogDataConnector ?
        [length = 15]
        [endpoint = (dn.BCM.dref, dn.PassiveKeyEntryDN.
           OutsideDoorHandleSensor.CapacitiveSensor.
           DriverDoorCapacitiveHandleModule)]
    logicalBCMPassCapacitiveSensorModule : AnalogDataConnector ?
        [length = 20]
        [endpoint = (dn.BCM.dref, dn.PassiveKeyEntryDN.
           OutsideDoorHandleSensor.CapacitiveSensor.
           PassDoorCapacitiveHandleModule)]
    logicalBCMDriverButtonSensorModule : AnalogDataConnector ?
        [length = 15]
        [endpoint = (dn.BCM.dref, dn.PassiveKeyEntryDN.
           OutsideDoorHandleSensor.ButtonSensor.
           DriverDoorButtonHandleModule)]
    logicalBCMPassButtonSensorModule : AnalogDataConnector ?
        [length = 20]
        [endpoint = (dn.BCM.dref, dn.PassiveKeyEntryDN.
           OutsideDoorHandleSensor.ButtonSensor.
           PassDoorButtonHandleModule)]

    logicalPKEModuleDriverCapacitiveSensorModule : DiscreteDataConnector
       ?
        [length = 15]
        [endpoint = (dn.PassiveKeyEntryDN.PassiveKeyModule, dn.
           PassiveKeyEntryDN.OutsideDoorHandleSensor.CapacitiveSensor.
           DriverDoorCapacitiveHandleModule)]
    logicalPKEModulePassCapacitiveSensorModule : DiscreteDataConnector ?
        [length = 20]
        [endpoint = (dn.PassiveKeyEntryDN.PassiveKeyModule, dn.
           PassiveKeyEntryDN.OutsideDoorHandleSensor.CapacitiveSensor.
           PassDoorCapacitiveHandleModule)]
    logicalPKEModuleDriverButtonSensorModule : DiscreteDataConnector ?
        [length = 15]
        [endpoint = (dn.PassiveKeyEntryDN.PassiveKeyModule, dn.
           PassiveKeyEntryDN.OutsideDoorHandleSensor.ButtonSensor.
           DriverDoorButtonHandleModule)]
    logicalPKEModulePassButtonSensorModule : DiscreteDataConnector ?
        [length = 20]
        [endpoint = (dn.PassiveKeyEntryDN.PassiveKeyModule, dn.
           PassiveKeyEntryDN.OutsideDoorHandleSensor.ButtonSensor.
           PassDoorButtonHandleModule)]

    logicalTransmitterDriverCapacitiveSensorModule : AnalogDataConnector
       ?
        [length = 15]
        [endpoint = (dn.PassiveKeyEntryDN.Transmitter, dn.
           PassiveKeyEntryDN.OutsideDoorHandleSensor.CapacitiveSensor.
           DriverDoorCapacitiveHandleModule)]
    logicalTransmitterPassCapacitiveSensorModule : AnalogDataConnector ?
        [length = 20]
        [endpoint = (dn.PassiveKeyEntryDN.Transmitter, dn.
           PassiveKeyEntryDN.OutsideDoorHandleSensor.CapacitiveSensor.
           PassDoorCapacitiveHandleModule)]
    logicalTransmitterDriverButtonSensorModule : AnalogDataConnector ?
        [length = 15]
        [endpoint = (dn.PassiveKeyEntryDN.Transmitter, dn.
           PassiveKeyEntryDN.OutsideDoorHandleSensor.ButtonSensor.
           DriverDoorButtonHandleModule)]
    logicalTransmitterPassButtonSensorModule : AnalogDataConnector ?
        [length = 20]
        [endpoint = (dn.PassiveKeyEntryDN.Transmitter, dn.
           PassiveKeyEntryDN.OutsideDoorHandleSensor.ButtonSensor.
           PassDoorButtonHandleModule)]

    logicalPKEModuleTransmitter : DiscreteDataConnector ?
        [length = 5]
        [endpoint = (dn.PassiveKeyEntryDN.PassiveKeyModule, dn.
           PassiveKeyEntryDN.Transmitter)]

    logicalBCMInsideFrontAntenna : AnalogDataConnector ?
        [length = 13]
        [endpoint = (dn.BCM.dref, dn.PassiveKeyEntryDN.InsideFrontLFAntenna)]
    logicalTransmitterInsideFrontAntenna : AnalogDataConnector ?
        [length = 1]
        [endpoint = (dn.PassiveKeyEntryDN.Transmitter, dn.PassiveKeyEntryDN.InsideFrontLFAntenna)]
    logicalBCMInsideCenterAntenna : AnalogDataConnector ?
        [length = 6]
        [endpoint = (dn.BCM.dref, dn.PassiveKeyEntryDN.InsideCenterLFAntenna)]
    logicalTransmitterInsideCenterAntenna : AnalogDataConnector ?
        [length = 4]
        [endpoint = (dn.PassiveKeyEntryDN.Transmitter, dn.PassiveKeyEntryDN.InsideCenterLFAntenna)]
    logicalBCMInsideRearAntenna : AnalogDataConnector ?
        [length = 14]
        [endpoint = (dn.BCM.dref, dn.PassiveKeyEntryDN.InsideRearLFAntenna)]
    logicalTransmitterInsideRearAntenna : AnalogDataConnector ?
        [length = 12]
        [endpoint = (dn.PassiveKeyEntryDN.Transmitter, dn.PassiveKeyEntryDN.InsideRearLFAntenna)]



abstract DoorLockHA : HardwareArchitecture
    dn -> DoorLockDN
    pt -> DoorLockPT
    ct -> DoorLockCT

abstract DoorLockDpl : Deployment
    fa -> DoorLockFA
    ha -> DoorLockHA

    // Cylinder Switch Deployment
    [fa.DriverDoorCylinderSwitch.deployedTo = ha.dn.DriverDoorLockMotorAssembly]
    [fa.PassDoorCylinderSwitch.deployedTo = ha.dn.PassengerDoorLockMotorAssembly]

    // Door Contacts Deployment
    [fa.DriverDoorContact.deployedTo = ha.dn.DriverDoorLockMotorAssembly]
    [fa.PassDoorContact.deployedTo = ha.dn.PassengerDoorLockMotorAssembly
```

```
       ]
    [fa.RearRightPassDoorContact.deployedTo = ha.dn.
       RearRightPassengerDoorLockMotorAssembly]
    [fa.RearLeftPassDoorContact.deployedTo = ha.dn.
       RearLeftPassengerDoorLockMotorAssembly]

    // Door Lock Sensors Deployment
    [fa.DriverDoorLockSensor.deployedTo = ha.dn.
       DriverDoorLockMotorAssembly]
    [fa.PassDoorLockSensor.deployedTo = ha.dn.
       PassengerDoorLockMotorAssembly]
    [fa.RearRightPassDoorLockSensor.deployedTo = ha.dn.
       RearRightPassengerDoorLockMotorAssembly]
    [fa.RearLeftPassDoorLockSensor.deployedTo = ha.dn.
       RearLeftPassengerDoorLockMotorAssembly]

    // Door Lock Control Deployment
    [fa.DoorLockControl.deployedTo = ha.dn.BCM.dref]


    // Door Lock Motor Deployment
    [fa.DriverDoorLockMotor.deployedTo = ha.dn.
       DriverDoorLockMotorAssembly]
    [fa.PassDoorLockMotor.deployedTo = ha.dn.
       PassengerDoorLockMotorAssembly]
    [fa.RearRightPassDoorLockMotor.deployedTo = ha.dn.
       RearRightPassengerDoorLockMotorAssembly]
    [fa.RearLeftPassDoorLockMotor.deployedTo = ha.dn.
       RearLeftPassengerDoorLockMotorAssembly]

    // Gear Position Sensor Deployment
    [fa.GearPositionSensor.deployedTo = ha.dn.TCM.dref]

    // Speed Sensor Deployment
    [fa.SpeedSmartLockFA => (fa.SpeedSmartLockFA.SpeedSensor.deployedTo =
        ha.dn.CombinationMeter.dref)]

    // Power Button Unlock Deployment
    [fa.DoorLockButtonFA.IndividualLockSwitchFA => (fa.DoorLockButtonFA.
       IndividualLockSwitchFA.DriverDoorLockButton.deployedTo = ha.dn.
       DoorLockButtonDN.IndividualLockSwitchDN.DriverLockPowerSwitch)]
    [fa.DoorLockButtonFA.IndividualLockSwitchFA => (fa.DoorLockButtonFA.
       IndividualLockSwitchFA.PassDoorLockButton.deployedTo = ha.dn.
       DoorLockButtonDN.IndividualLockSwitchDN.PassLockPowerSwitch)]
    [fa.DoorLockButtonFA.CentralLockSwitchFA => (fa.DoorLockButtonFA.
       CentralLockSwitchFA.CentralLockButton.deployedTo = ha.dn.
       DoorLockButtonDN.CentralLockSwitchDN.CenterLockPowerSwitch)]

    // Remote Key Access Deployment
    [fa.RemoteKeyAccessFA => (fa.RemoteKeyAccessFA.CentralRFAntenna.
       deployedTo = ha.dn.RemoteKeyAccessDN.CentralRFAntennaModule)]
    [fa.RemoteKeyAccessFA => (fa.RemoteKeyAccessFA.CentralRFReceiver.
       deployedTo = ha.dn.RemoteKeyAccessDN.CentralRFAntennaModule)]
    [fa.RemoteKeyAccessFA => (fa.RemoteKeyAccessFA.IDAuthentication.
       deployedTo in (ha.dn.BCM.dref, ha.dn.RemoteKeyAccessDN.
       CentralRFAntennaModule, ha.dn.PassiveKeyEntryDN.PassiveKeyModule)
       )]

    // Passive Key Entry Deployment
    PassiveKeyEntryDpl ?
        xor OutsideDoorHandleSensor
            ButtonSensor
                [ha.dn.PassiveKeyEntryDN.OutsideDoorHandleSensor.
                   ButtonSensor && fa.PassiveKeyEntryFA.
                   OutsideDoorHandleSensor.ButtonSensor]
                [fa.PassiveKeyEntryFA.OutsideDoorHandleSensor.
                   ButtonSensor.DriverDoorButtonSensor.deployedTo = ha.
                   dn.PassiveKeyEntryDN.OutsideDoorHandleSensor.
                   ButtonSensor.DriverDoorButtonHandleModule]
                [fa.PassiveKeyEntryFA.OutsideDoorHandleSensor.
                   ButtonSensor.PassDoorButtonSensor.deployedTo = ha.dn.
                   PassiveKeyEntryDN.OutsideDoorHandleSensor.
                   ButtonSensor.PassDoorButtonHandleModule]
                [fa.PassiveKeyEntryFA.DriverOutsideLFAntenna.deployedTo =
                    ha.dn.PassiveKeyEntryDN.OutsideDoorHandleSensor.
                   ButtonSensor.DriverDoorButtonHandleModule]
                [fa.PassiveKeyEntryFA.PassOutsideLFAntenna.deployedTo =
                   ha.dn.PassiveKeyEntryDN.OutsideDoorHandleSensor.
                   ButtonSensor.PassDoorButtonHandleModule]
            CapacitiveSensor
                [ha.dn.PassiveKeyEntryDN.OutsideDoorHandleSensor.
                   CapacitiveSensor && fa.PassiveKeyEntryFA.
                   OutsideDoorHandleSensor.CapacitiveSensor]
                [fa.PassiveKeyEntryFA.OutsideDoorHandleSensor.
                   CapacitiveSensor.DriverDoorCapacitiveSensor.
                   deployedTo = ha.dn.PassiveKeyEntryDN.
                   OutsideDoorHandleSensor.CapacitiveSensor.
                   DriverDoorCapacitiveHandleModule]
                [fa.PassiveKeyEntryFA.OutsideDoorHandleSensor.
                   CapacitiveSensor.PassDoorCapacitiveSensor.deployedTo
                   = ha.dn.PassiveKeyEntryDN.OutsideDoorHandleSensor.
                   CapacitiveSensor.PassDoorCapacitiveHandleModule]
                [fa.PassiveKeyEntryFA.DriverOutsideLFAntenna.deployedTo =
                    ha.dn.PassiveKeyEntryDN.OutsideDoorHandleSensor.
                   CapacitiveSensor.DriverDoorCapacitiveHandleModule]
                [fa.PassiveKeyEntryFA.PassOutsideLFAntenna.deployedTo =
                   ha.dn.PassiveKeyEntryDN.OutsideDoorHandleSensor.
                   CapacitiveSensor.PassDoorCapacitiveHandleModule]


        [fa.PassiveKeyEntryFA.DriverLFTransmitter.deployedTo in (ha.dn.
           PassiveKeyEntryDN.Transmitter, ha.dn.BCM.dref)]
        [fa.PassiveKeyEntryFA.PassLFTransmitter.deployedTo in (ha.dn.
           PassiveKeyEntryDN.Transmitter, ha.dn.BCM.dref)]

        [fa.PassiveKeyEntryFA.InsideFrontLFAntenna.deployedTo = ha.dn.
           PassiveKeyEntryDN.InsideFrontLFAntenna]
        [fa.PassiveKeyEntryFA.InsideCenterLFAntenna.deployedTo = ha.dn.
           PassiveKeyEntryDN.InsideCenterLFAntenna]
        [fa.PassiveKeyEntryFA.InsideRearLFAntenna.deployedTo = ha.dn.
           PassiveKeyEntryDN.InsideRearLFAntenna]
        [fa.PassiveKeyEntryFA.InsideLFTransmitter.deployedTo in (ha.dn.
           PassiveKeyEntryDN.Transmitter, ha.dn.BCM.dref)]

        [fa.PassiveKeyEntryFA.PKEControl.deployedTo in (ha.dn.BCM.dref,
           ha.dn.PassiveKeyEntryDN.PassiveKeyModule)]

    // Power Topology Deployment
    [ha.pt.pkeModuleDP <=> ha.dn.PassiveKeyEntryDN.PassiveKeyModule]
    [ha.pt.driverCapacitiveSensorDP <=> ha.dn.PassiveKeyEntryDN.
       OutsideDoorHandleSensor.CapacitiveSensor.
       DriverDoorCapacitiveHandleModule]
    [ha.pt.passCapacitiveSensorDP <=> ha.dn.PassiveKeyEntryDN.
       OutsideDoorHandleSensor.CapacitiveSensor.
       PassDoorCapacitiveHandleModule]
    [ha.pt.centralRFModuleDP <=> ha.dn.RemoteKeyAccessDN]
    [ha.pt.transmitterDP <=> ha.dn.PassiveKeyEntryDN.Transmitter]

    // Communication Deployment
    [fa.driverCylReq.deployedTo = ha.ct.logicalBCMDriverMotorAssemblyDW]
    [fa.passCylReq.deployedTo = ha.ct.logicalBCMPassMotorAssemblyDW]

    [fa.driverContactSignal.deployedTo = ha.ct.
       logicalBCMDriverMotorAssemblyDW]
    [fa.passContactSignal.deployedTo  = ha.ct.
```

```
       logicalBCMPassMotorAssemblyDW]
    [fa.rearRightPassContactSignal.deployedTo = ha.ct.
       logicalBCMRearRightPassMotorAssemblyDW]
    [fa.rearLeftPassContactSignal.deployedTo = ha.ct.
       logicalBCMRearLeftPassMotorAssemblyDW]

    [fa.driverLockPosition.deployedTo = ha.ct.
       logicalBCMDriverMotorAssemblyDW]
    [fa.passLockPosition.deployedTo = ha.ct.logicalBCMPassMotorAssemblyDW
       ]
    [fa.rearRightPassLockPosition.deployedTo = ha.ct.
       logicalBCMRearRightPassMotorAssemblyDW]
    [fa.rearLeftPassLockPosition.deployedTo = ha.ct.
       logicalBCMRearLeftPassMotorAssemblyDW]

    [fa.driverLockCmd.deployedTo = ha.ct.logicalBCMDriverMotorAssemblyDW]
    [fa.passLockCmd.deployedTo = ha.ct.logicalBCMPassMotorAssemblyDW]
    [fa.rearRightLockCmd.deployedTo = ha.ct.
       logicalBCMRearRightPassMotorAssemblyDW]
    [fa.rearLeftLockCmd.deployedTo = ha.ct.
       logicalBCMRearLeftPassMotorAssemblyDW]

    [fa.gearPostion.deployedTo = ha.ct.logicalHighSpeedBus]

    [fa.SpeedSmartLockFA => (fa.SpeedSmartLockFA.speed.deployedTo in (ha.
       ct.logicalHighSpeedBus))]

    [fa.DoorLockButtonFA.IndividualLockSwitchFA => (fa.DoorLockButtonFA.
       IndividualLockSwitchFA.driverDoorLockReq.deployedTo = ha.ct.
       logicalBCMDriverLockPowerSwitchDW)]
    [fa.DoorLockButtonFA.IndividualLockSwitchFA => (fa.DoorLockButtonFA.
       IndividualLockSwitchFA.passDoorLockReq.deployedTo = ha.ct.
       logicalBCMPassLockPowerSwitchDW)]
    [fa.DoorLockButtonFA.CentralLockSwitchFA => (fa.DoorLockButtonFA.
       CentralLockSwitchFA.centralDoorLockReq.deployedTo = ha.ct.
       logicalBCMCenterLockPowerSwitchDW)]

    [fa.RemoteKeyAccessFA => (no fa.RemoteKeyAccessFA.
       centralAntennaSignal.deployedTo)]
    [fa.RemoteKeyAccessFA => (fa.RemoteKeyAccessFA.centralReceiverMsg.
       deployedTo in (ha.ct.logicalLowSpeedBus))]
    [fa.RemoteKeyAccessFA => (fa.RemoteKeyAccessFA.authenticationMsg.
       deployedTo in (ha.ct.logicalLowSpeedBus))]

    [fa.PassiveKeyEntryFA => fa.PassiveKeyEntryFA.driverTransMsg.
```

```
           deployedTo in (ha.ct.
           logicalTransmitterDriverCapacitiveSensorModule, ha.ct.
           logicalTransmitterDriverButtonSensorModule, ha.ct.
           logicalBCMDriverCapacitiveSensorModule, ha.ct.
           logicalBCMDriverButtonSensorModule)]
       [fa.PassiveKeyEntryFA => fa.PassiveKeyEntryFA.driverPKEReq.deployedTo
            in (ha.ct.logicalPKEModuleTransmitter, ha.ct.logicalLowSpeedBus)
           ]
       [fa.PassiveKeyEntryFA => fa.PassiveKeyEntryFA.passTransMsg.deployedTo
            in (ha.ct.logicalTransmitterPassCapacitiveSensorModule, ha.ct.
           logicalTransmitterPassButtonSensorModule, ha.ct.
           logicalBCMPassCapacitiveSensorModule, ha.ct.
           logicalBCMPassButtonSensorModule)]
       [fa.PassiveKeyEntryFA => fa.PassiveKeyEntryFA.passPKEReq.deployedTo
           in (ha.ct.logicalPKEModuleTransmitter, ha.ct.logicalLowSpeedBus)]
       [fa.PassiveKeyEntryFA => fa.PassiveKeyEntryFA.insideFrontTransMsg.
           deployedTo in (ha.ct.logicalTransmitterInsideFrontAntenna, ha.ct.
           logicalBCMInsideFrontAntenna)]
       [fa.PassiveKeyEntryFA => fa.PassiveKeyEntryFA.insideCenterTransMsg.
           deployedTo in (ha.ct.logicalTransmitterInsideCenterAntenna, ha.ct
           .logicalBCMInsideCenterAntenna)]
       [fa.PassiveKeyEntryFA => fa.PassiveKeyEntryFA.insideRearTransMsg.
           deployedTo in (ha.ct.logicalTransmitterInsideRearAntenna, ha.ct.
           logicalBCMInsideRearAntenna)]
       [fa.PassiveKeyEntryFA => fa.PassiveKeyEntryFA.insidePKEReq.deployedTo
            in (ha.ct.logicalPKEModuleTransmitter, ha.ct.logicalLowSpeedBus)
           ]
       [fa.PassiveKeyEntryFA => fa.PassiveKeyEntryFA.driverDoorHandleReq.
           deployedTo in (ha.ct.logicalPKEModuleDriverButtonSensorModule, ha
           .ct.logicalPKEModuleDriverCapacitiveSensorModule, ha.ct.
           logicalBCMDriverButtonSensorModule, ha.ct.
           logicalBCMDriverCapacitiveSensorModule)]
       [fa.PassiveKeyEntryFA => fa.PassiveKeyEntryFA.passDoorHandleReq.
           deployedTo in (ha.ct.logicalPKEModulePassButtonSensorModule, ha.
           ct.logicalPKEModulePassCapacitiveSensorModule, ha.ct.
           logicalBCMPassButtonSensorModule, ha.ct.
           logicalBCMPassCapacitiveSensorModule)]
       [fa.PassiveKeyEntryFA => fa.PassiveKeyEntryFA.doorLockControlReq.
           deployedTo in (ha.ct.logicalLowSpeedBus)]


//-------------------- Door Lock System Model
    ----------------------------//
DoorLockSys : System
    DLockFM : FeatureModel
```

```
        Basic : Feature
            IndividualLockSwitch : Feature ? // This feature is to determine if the driver and passenger should have individual door lock switches or use a central lock switch.
            SpeedSmartLock : Feature ? // This feature is if the door should lock when the car is above a certain speed.
        RKA : Feature ? // Remote Key Access
        PKE : Feature ? // Passive Key Entry
            xor OutsideDoorHandleSensor
                ButtonSensor : Feature
                CapacitiveSensor : Feature
        [PKE => RKA]
    DLockFA : DoorLockFA
        [DoorLockButtonFA.IndividualLockSwitchFA <=> DLockFM.Basic.IndividualLockSwitch]
        [SpeedSmartLockFA <=> DLockFM.Basic.SpeedSmartLock]
        [RemoteKeyAccessFA <=> DLockFM.RKA]
        [PassiveKeyEntryFA <=> DLockFM.PKE]
        [PassiveKeyEntryFA.OutsideDoorHandleSensor.ButtonSensor <=> DLockFM.PKE.OutsideDoorHandleSensor.ButtonSensor]
        [PassiveKeyEntryFA.OutsideDoorHandleSensor.CapacitiveSensor <=> DLockFM.PKE.OutsideDoorHandleSensor.CapacitiveSensor]

        // Timing Chains
        DriverSwitchToControl -> integer
            [if (DLockFM.Basic.IndividualLockSwitch) then (
                this = DoorLockButtonFA.IndividualLockSwitchFA.DriverDoorLockButton.latency +
                DoorLockButtonFA.IndividualLockSwitchFA.driverDoorLockReq.latency/1000
            ) else (
                this = DoorLockButtonFA.CentralLockSwitchFA.CentralLockButton.latency +
                DoorLockButtonFA.CentralLockSwitchFA.centralDoorLockReq.latency/1000
            )]
        DriverContactToControl -> integer = DriverDoorContact.latency + driverContactSignal.latency/1000
        DriverLockSensorToControl -> integer = DriverDoorLockSensor.latency + driverLockPosition.latency/1000

        DriverSwitchToMotor -> integer
            [if (DLockFM.Basic.IndividualLockSwitch) then (
                this = DoorLockButtonFA.IndividualLockSwitchFA.
```

```
                   DriverDoorLockButton.latency +
                DoorLockControl.latency +
                DriverDoorLockMotor.latency +
                ((driverLockCmd.latency + DoorLockButtonFA.
                   IndividualLockSwitchFA.driverDoorLockReq.latency)
                   /1000)
            ) else (
                this = DoorLockButtonFA.CentralLockSwitchFA.
                   CentralLockButton.latency +
                DoorLockControl.latency +
                DriverDoorLockMotor.latency +
                ((DoorLockButtonFA.CentralLockSwitchFA.centralDoorLockReq
                   .latency + driverLockCmd.latency)/1000)
            )]

        ControlInputDifference -> integer
        [ControlInputDifference = (max(DriverSwitchToControl.dref,
           DriverContactToControl.dref, DriverLockSensorToControl.dref)
            - min(DriverSwitchToControl.dref, DriverContactToControl.dref
               , DriverLockSensorToControl.dref))]

        PassiveKeyCapacitiveSensorToMotor -> integer ?
        [if (DLockFM.PKE.OutsideDoorHandleSensor.CapacitiveSensor) then (
            PassiveKeyCapacitiveSensorToMotor = PassiveKeyEntryFA.
               OutsideDoorHandleSensor.CapacitiveSensor.
               DriverDoorCapacitiveSensor.latency +
            PassiveKeyEntryFA.PKEControl.latency +
            PassiveKeyEntryFA.DriverLFTransmitter.latency +
            PassiveKeyEntryFA.DriverOutsideLFAntenna.latency + 50 +
            RemoteKeyAccessFA.CentralRFAntenna.latency +
            RemoteKeyAccessFA.CentralRFReceiver.latency +
            RemoteKeyAccessFA.IDAuthentication.latency +
            DoorLockControl.latency +
            DriverDoorLockMotor.latency +
            ((PassiveKeyEntryFA.driverDoorHandleReq.latency +
               PassiveKeyEntryFA.driverPKEReq.latency +
               PassiveKeyEntryFA.driverTransMsg.latency +
                RemoteKeyAccessFA.centralAntennaSignal.latency +
                   RemoteKeyAccessFA.centralReceiverMsg.latency +
                RemoteKeyAccessFA.authenticationMsg.latency +
                   driverLockCmd.latency)/1000))
        else (no PassiveKeyCapacitiveSensorToMotor)]

        // Timing Constraints
        // Driver lock switch to driver motor timing constraint
```

```
    [(DLockFM.Basic.IndividualLockSwitch && DoorLockRequirements.
       TimingRequirements.BasicIndividualSwitchLatency) => (
        DriverSwitchToMotor <= DoorLockRequirements.
           TimingRequirements.BasicIndividualSwitchLatency
    )]
    // Central lock switch to driver motor timing constraint
    [(no DLockFM.Basic.IndividualLockSwitch && DoorLockRequirements.
       TimingRequirements.BasicCentralSwitchLatency)=> (
        DriverSwitchToMotor <= DoorLockRequirements.
           TimingRequirements.BasicCentralSwitchLatency
    )]
    // Switch Unlock Input Synchronization Timing Constraint
    [DoorLockRequirements.TimingRequirements.
       SwitchUnlockInputSynchLatency => (
         ControlInputDifference <= DoorLockRequirements.
            TimingRequirements.SwitchUnlockInputSynchLatency
    )]
    // Passive Key Capacitive Sensor to Motor Timing Constraint
    [(DoorLockRequirements.TimingRequirements.PKELatency && DLockFM.
       PKE.OutsideDoorHandleSensor.CapacitiveSensor) => (
        PassiveKeyCapacitiveSensorToMotor <= DoorLockRequirements.
           TimingRequirements.PKELatency
    )]

    // Timing Margins
    BasicIndividualSwitchLatencyMargin -> integer ?
    [if DoorLockRequirements.TimingRequirements.
       BasicIndividualSwitchLatency then (
       BasicIndividualSwitchLatencyMargin = (DoorLockRequirements.
       TimingRequirements.BasicIndividualSwitchLatency -
       DriverSwitchToMotor))
        else (no BasicIndividualSwitchLatencyMargin)]
    BasicCentralSwitchLatencyMargin -> integer ?
    [if DoorLockRequirements.TimingRequirements.
       SwitchUnlockInputSynchLatency then (
       BasicCentralSwitchLatencyMargin = (DoorLockRequirements.
       TimingRequirements.SwitchUnlockInputSynchLatency -
       DriverSwitchToMotor))
        else (no BasicCentralSwitchLatencyMargin)]
    SwitchUnlockInputSynchLatencyMargin -> integer ?
    [if DoorLockRequirements.TimingRequirements.
       SwitchUnlockInputSynchLatency then (
        SwitchUnlockInputSynchLatencyMargin = (DoorLockRequirements.
           TimingRequirements.SwitchUnlockInputSynchLatency -
           ControlInputDifference))
```

```
            else (no SwitchUnlockInputSynchLatencyMargin)]
        PKELatencyMargin -> integer ?
        [if DoorLockRequirements.TimingRequirements.PKELatency then (PKELatencyMargin = (DoorLockRequirements.TimingRequirements.PKELatency - PassiveKeyCapacitiveSensorToMotor))
            else (no PKELatencyMargin)]

    DLockHA : DoorLockHA
        DLockDN : DoorLockDN
            [BCM = Car.BCM]
            [TCM = Car.TCM]
            [EC = Car.EC]
            [CombinationMeter => CombinationMeter = Car.CombinationMeter]
            [DoorLockButtonDN.IndividualLockSwitchDN <=> DLockFM.Basic.IndividualLockSwitch]
            [CombinationMeter <=> DLockFM.Basic.SpeedSmartLock]
            [RemoteKeyAccessDN <=> DLockFM.RKA]
            [PassiveKeyEntryDN <=> DLockFM.PKE]
            [PassiveKeyEntryDN.OutsideDoorHandleSensor.ButtonSensor <=> DLockFM.PKE.OutsideDoorHandleSensor.ButtonSensor]
            [PassiveKeyEntryDN.OutsideDoorHandleSensor.CapacitiveSensor <=> DLockFM.PKE.OutsideDoorHandleSensor.CapacitiveSensor]
        DLockPT : DoorLockPT
            [dn = DLockDN]
        DLockCT : DoorLockCT
            [dn = DLockDN]
        [dn = DLockDN]
        [pt = DLockPT]
        [ct = DLockCT]
    DLockDpl : DoorLockDpl
        [fa = DLockFA]
        [ha = DLockHA]
        [DLockFM.PKE <=> PassiveKeyEntryDpl]

DoorLockRequirements
    TimingRequirements
        BasicIndividualSwitchLatency -> integer ?
        BasicCentralSwitchLatency -> integer ?
        SwitchUnlockInputSynchLatency -> integer ?
        PKELatency -> integer ?

Car
    BCM : DeviceNode
        [type = SmartDeviceType]
        [mass = 408]
        [cost = 261]
        [ppm = 50]
        [replaceCost = 261]
        [speedFactor = 10]
    TCM : DeviceNode
        [type = SmartDeviceType]
        [mass = 204]
        [cost = 117]
        [ppm = 50]
        [replaceCost = 117]
        [speedFactor = 10]
    CombinationMeter : DeviceNode ?
        [type = SmartDeviceType]
        [mass = 198]
        [cost = 649]
        [ppm = 50]
        [replaceCost = 649]
        [speedFactor = 10]
    EC : DeviceNode
        [type = PowerDeviceType]
        [mass = 0]
        [cost = 0]
        [ppm = 10]
        [replaceCost = 0]

totalCarMass -> integer = sum(DeviceNode.mass) + sum(HardwareConnector.mass)/1000
totalCarCost -> integer = sum(DeviceNode.cost) + sum(HardwareDataConnector.cost)/1000
totalCarWarrantyCost -> integer = sum(DeviceNode.warrantyCost)/1000

// Optimization Goals:
// Comment out these goals if optimization should not be perfromed (no other modifications are necessary)
// << minimize totalCarMass >>
// << minimize totalCarCost >>
// << minimize totalCarWarrantyCost >>
```

References

[1] alloy: a language and tools for relational models. http://alloy.mit.edu/alloy/.

[2] Archeopterix. http://users.monash.edu.au/~aldeidaa/ArcheOpterix.html.

[3] Autofocus 3. http://af3.fortiss.org.

[4] Clafer. http://clafer.org.

[5] Clafer configurator. http://t3-necsis.cs.uwaterloo.ca:8093/.

[6] Clafer moo visualizer. http://t3-necsis.cs.uwaterloo.ca:8092/.

[7] Road vehicles – local interconnect network (lin) – part 6: Protocol conformance test specification, 2015.

[8] Zubair Akhtar. Model based automotive system design: A power window controller case study. Master's thesis, University of Waterloo, 2015. https://uwspace.uwaterloo.ca/handle/10012/9215.

[9] A. Aleti, S. Bjornander, Lars Grunske, and I. Meedeniya. Archeopterix: An extendable tool for architecture optimization of aadl models. In *Model-Based Methodologies for Pervasive and Embedded Software, 2009. MOMPES '09. ICSE Workshop on*, pages 61–71, 2009.

[10] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, and I. Meedeniya. Software architecture optimization methods: A systematic literature review. *IEEE Transactions on Software Engineering*, 39(5):658–683, 2013.

[11] A. Aleti, Lars Grunske, I. Meedeniya, and I. Moser. Let the ants deploy your software - an aco based deployment optimisation strategy. In *Automated Software Engineering, 2009. ASE '09. 24th IEEE/ACM International Conference on*, pages 505–509, 2009.

[12] Aldeida Aleti and Indika Meedeniya. Component deployment optimisation with bayesian learning. In *Proceedings of the 14th International ACM Sigsoft Symposium on Component Based Software Engineering*, CBSE '11, pages 11–20, 2011.

[13] Michał Antkiewicz, Kacper Bąk, Alexandr Murashkin, Rafael Olaechea, Jia Liang, and Krzysztof Czarnecki. Clafer tools for product line engineering. In *Software Product Line Conference*, 2013.

[14] Alessandro Biondi, Marco Di Natale, and Youcheng Sun. Moving from single-core to multicore: Initial findings on a fuel injection case study. Technical report, SAE Technical Paper, 2016.

[15] Kacper Bąk, Zinovy Diskin, Michał Antkiewicz, Krzysztof Czarnecki, and Andrzej Wąsowski. Clafer: Unifying class and feature modeling. *Software and Systems Modeling*, 2014. The final publication is available at Springer via DOI.

[16] LS Brandt, N Krämer, J Metzger, U Lindemann, et al. Optimization approach for function-partitioning in an automotive electric electronic system architecture. In *DS 70: Proceedings of DESIGN 2012, the 12th International Design Conference, Dubrovnik, Croatia*, 2012.

[17] Manfred Broy. Challenges in automotive software engineering. In *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, pages 33–42, 2006.

[18] D. W. Coit and A. E. Smith. Redundancy allocation to maximize a lower percentile of the system time-to-failure distribution. *IEEE Transactions on Reliability*, 47(1):79–87, 1998.

[19] David W. Coit and Alice E. Smith. Solving the redundancy allocation problem using a combined neural network/genetic algorithm approach. *Comput. Oper. Res.*, 23(6):515–526, 1996.

[20] P. Cuenot, DeJiu Chen, S. Gerard, Henrik Lonn, M.-O. Reiser, David Servat, C.-J. Sjostedt, R.T. Kolagari, M. Torngren, and M. Weber. Managing complexity of automotive electronics using the east-adl. In *Engineering Complex Computer Systems, 2007. 12th IEEE International Conference on*, pages 353–358, 2007.

[21] B. P. Dave and N. K. Jha. Cohra: hardware-software cosynthesis of hierarchical heterogeneous distributed embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(10):900–919, 1998.

[22] Robert I. Davis, Alan Burns, Reinder J. Bril, and Johan J. Lukkien. Controller area network (can) schedulability analysis: Refuted, revisited and revised. *Real-Time Syst.*, 35(3):239–272, 2007. http://dx.doi.org/10.1007/s11241-007-9012-7.

[23] Marco Di Natale. Understanding and using the controller area network. *inst. eecs. berkeley. edu/˜ ee249/fa08/Lectures/handout_canbus2. pdf*, 2008.

[24] J. Drake, M. Harbour, J. Gutierrez, P. Martinez, J. Medina, and J. Palencia. Modeling and analysis suite for real time applications. http://mast.unican.es/mast_description.pdf, 2014.

[25] EAST-ADL Association. *EAST-ADL domain model specification, version V2.1.12*, 2013. http://east-adl.info/Specification/V2.1.12/EAST-ADL-Specification_V2.1.12.pdf.

[26] Peter Feiler and Jörgen Hansson. Flow latency analysis with the architecture analysis and design language (aadl). Technical Report CMU/SEI-2007-TN-010, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2007. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=8229.

[27] 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.

[28] M. Glaß, M. Lukasiewycz, R. Wanka, C. Haubelt, and J. Teich. Multi-objective routing and topology optimization in networked embedded systems. In *Embedded Computer Systems: Architectures, Modeling, and Simulation, 2008. SAMOS 2008. International Conference on*, pages 74–81, 2008.

[29] Sebastian Graf, Michael Glaß, Jürgen Teich, and Christoph Lauer. Multi-variant-based design space exploration for automotive embedded systems. In *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '14, pages 7:1–7:6, 3001 Leuven, Belgium, Belgium, 2014. European Design and Automation Association.

[30] A. Hamann. *Iterative Design Space Exploration and Robustness Optimization for Embedded Systems*. Cuvillier, 2008.

[31] Kabsu Han, Yongseop Kwon, Wooyeon Kim, and Jeonghun Cho. Distributed hierarchical service network for automotive embedded system. In *Information Networking (ICOIN), 2012 International Conference on*, pages 188–192, 2012.

[32] V. Izosimov, P. Pop, P. Eles, and Z. Peng. Design optimization of time- and cost-constrained fault-tolerant distributed embedded systems. In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 864–869 Vol. 2, 2005.

[33] Daniel Jackson, H Estler, Derek Rayside, et al. The guided improvement algorithm for exact, general-purpose, many-objective combinatorial optimization. 2009.

[34] Eunsuk Kang, Ethan Jackson, and Wolfram Schulte. An approach for effective design space exploration. In *Foundations of Computer Software. Modeling, Development, and Verification of Adaptive Systems*, pages 33–54. Springer, 2010.

[35] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990.

[36] 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.

[37] Sadan Kulturel-Konak, David W. Coit, and Fatema Baheranwala. Pruned pareto-optimal sets for the system redundancy allocation problem based on multiple prioritized objectives. *Journal of Heuristics*, 14(4):335–357, 2008.

[38] R. Li, R. Etemaadi, M. T. M. Emmerich, and M. R. V. Chaudron. An evolutionary multiobjective optimization approach to component-based software architecture design. In *Evolutionary Computation (CEC), 2011 IEEE Congress on*, pages 432–439, 2011.

[39] Jia Liang. Solving clafer models with choco. (GSDLab-TR 2012-12-30), 2012. http://gsd.uwaterloo.ca/node/509.

[40] Yun-Chia Liang and Min-Hua Lo. Multi-objective redundancy allocation optimization using a variable neighborhood search algorithm. *Journal of Heuristics*, 16(3):511–535, 2009.

[41] C. W. Lin, L. Rao, P. Giusto, J. D'Ambrosio, and A. L. Sangiovanni-Vincentelli. Efficient wire routing and wire sizing for weight minimization of automotive systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(11):1730–1741, 2015.

[42] Indika Meedeniya. *Architecture Optimisation of Embedded Systems under Uncertainty in Probabilistic Reliability Evaluation Model Parameters*. PhD thesis, Swinburne University of Technology, Melbourne, Australia, 2012.

[43] Indika Meedeniya, Barbora Buhnova, Aldeida Aleti, and Lars Grunske. Architecture-driven reliability and energy optimization for complex embedded systems. In *Proceedings of the 6th International Conference on Quality of Software Architectures: Research into Practice - Reality and Gaps*, QoSA'10, pages 52–67, 2010.

[44] Indika Meedeniya, Barbora Buhnova, Aldeida Aleti, and Lars Grunske. Reliability-driven deployment optimization for embedded systems. *J. Syst. Softw.*, 84(5):835–846, 2011.

[45] Indika Meedeniya, Irene Moser, Aldeida Aleti, and Lars Grunske. Architecture-based reliability evaluation under uncertainty. In *Proceedings of the Joint ACM SIGSOFT Conference – QoSA and ACM SIGSOFT Symposium – ISARCS on Quality of Software Architectures – QoSA and Architecting Critical Systems – ISARCS*, QoSA-ISARCS '11, pages 85–94, 2011.

[46] J. Montgomery and I. Moser. Parallel constraint handling in a multiobjective evolutionary algorithm for the automotive deployment problem. In *e-Science Workshops, 2010 Sixth IEEE International Conference on*, pages 104–109, 2010.

[47] Ralph Moritz, Tamara Ulrich, and Lothar Thiele. Evolutionary exploration of e/e-architectures in automotive design. In *Operations Research Proceedings 2011*, pages 361–366. Springer, 2012.

[48] I. Moser and S. Mostaghim. The automotive deployment problem: A practical application for constrained multiobjective evolutionary optimisation. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pages 1–8, 2010.

[49] Alexandr Murashkin. Automotive electronic/electric architecture modeling, design exploration and optimization using Clafer. Master's thesis, University of Waterloo, 2014. https://uwspace.uwaterloo.ca/handle/10012/8780.

[50] Alexandr Murashkin, Michał Antkiewicz, Derek Rayside, and Krzysztof Czarnecki. Visualization and exploration of optimal variants in product line engineering. In *Software Product Line Conference*, 2013.

[51] Alexandr Murashkin, Michał Antkiewicz, Derek Rayside, and Krzysztof Czarnecki. Visualization and exploration of optimal variants in product line engineering. In *Software Product Line Conference*, Tokyo, Japan, 2013.

[52] Alexandr Murashkin, Luis Silva Azevedo, Jianmei Guo, Edward Zulkoski, Jia Liang, Krzysztof Czarnecki, and David Parker. Automated decomposition and allocation of automotive safety integrity levels using exact solvers. In *SAE 2015 World Congress & Exhibition*, Detroit, Michigan, USA, 2015. SAE, SAE. http://papers.sae.org/2015-01-0156/.

[53] Mark Nicholson, Alan Burns, and Yo Dd. Emergence of an architectural topology for safety-critical real-time systems, 1997.

[54] Yiannis Papadopoulos and Christian Grante. Evolving car designs using model-based automated safety analysis and optimisation techniques. *J. Syst. Softw.*, 76(1):77–89, 2005.

[55] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco3 Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2014.

[56] Jordan Ross. Case studies on E/E architectures for power window and central door locks systems, 2016. http://gsd.uwaterloo.ca/node/667.

[57] 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*, pages 1–10. ACM, 2012.

[58] Jörg Schäuffele. E/e architectural design and optimization using preevision. Technical report, SAE Technical Paper, 2016.

[59] Software Engineering Institute. OSATE, version 2. http://osate.github.io/.

[60] Thilo Streichert, Michael Glaß, Christian Haubelt, and Jürgen Teich. Design space exploration of reliable networked embedded systems. *Journ. on Systems Architecture*, pages 751–763, 2007.

[61] H. A. Taboada, J. F. Espiritu, and D. W. Coit. Moms-ga: A multi-objective multi-state genetic algorithm for system reliability optimization design problems. *IEEE Transactions on Reliability*, 57(1):182–191, 2008.

[62] Heidi A. Taboada, Fatema Baheranwala, David W. Coit, and Naruemon Wattanapongsakorn. Practical solutions for multi-objective optimization: An application to system reliability design problems. *Reliability Engineering & System Safety*, 92(3):314 – 322, 2007. Selected Papers Presented at the Fourth International Conference on Quality and ReliabilityICQR2005Fourth International Conference on Quality and Reliability.

[63] Heidi A. Taboada and David W. Coit. Data clustering of solutions for multiple objective system reliability optimization problems. *Quality Technology & Quantitative Management Journal*, pages 35–54, 2007.

[64] S. Voss, J. Eder, and B. Schaetz, editors. *Scheduling Synthesis for Multi-Period SW Components*, 2016.

[65] S. Voss and B. Schatz. Deployment and scheduling synthesis for mixed-critical shared-memory applications. In *Engineering of Computer Based Systems (ECBS), 2013 20th IEEE International Conference and Workshops on the*, pages 100–109, 2013.

[66] Marc Zeller and Christian Prehofer. Modeling and efficient solving of extra-functional properties for adaptation in networked embedded real-time systems. *J. Syst. Archit.*, 59(10):1067–1082, 2013.