ClaferMPS: Modeling and Optimizing Automotive Electric/Electronic Architectures Using Domain-Specific Languages

by

Eldar Khalilov

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

Waterloo, Ontario, Canada, 2017

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

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

Abstract

Modern automotive electric/electronic (E/E) architectures are growing to the point where architects can no longer manually predict the effects of their design decisions. Thus, in addition to applying an architecture reference model to decompose their architectures, they also require tools for synthesizing and evaluating candidate architectures during the design process. Clafer is a modeling language, which has been used to model variable multi-layer, multi-perspective automotive system architectures according to an architecture reference model. Clafer tools allow architects to synthesize optimal candidates and evaluate effects of their design decisions. However, since Clafer is a general-purpose structural modeling language, it does not help the architects in building models conforming to the given architecture reference model.

In this work, we present ClaferMPS, a set of extensible languages and IDE for modeling E/E architectures using Clafer. First, we present an E/E architecture domain-specific language (DSL) built on top of Clafer, which embodies the reference model and which guides the architects in correctly applying the reference model. We then evaluate the DSL and its implementation by modeling two existing automotive systems, which were originally modeled in plain Clafer. The evaluation showed that by using the DSL, an evaluator obtained correct models by construction because the DSL helped prevent typical errors that are easy to make in plain Clafer. The evaluator was also able to synthesize and evaluate candidate architectures as with plain Clafer. Finally, we demonstrate extensibility capabilities of ClaferMPS. Our implementation is built on top of the JetBrains Meta Programming System, which supports language modularization and composition, multi-stage transformations and projectional editing. As a result, ClaferMPS allows third parties to seamlessly add extensions to both Clafer and the E/E architecture DSL without invasive changes. To illustrate this approach, we consider the Robot Operating System (ROS) communications infrastructure, a case study, which is outside the scope of the existing reference model. We show how the E/E architecture DSL can be adapted to the new domain using MPS language modularization and composition.

Acknowledgements

First of all, I would like to express my sincere gratitude to my supervisor, **Professor Krzysztof Czarnecki**, for giving me this wonderful opportunity, for his continuous support, motivation, and immense knowledge.

Next, I would like to thank **Professor Joanne Atlee** and **Professor Derek Rayside** for being reviewers of this thesis and providing very constructive feedback.

I would like to thank **Michał Antkiewicz** for his collaboration, guidance, and all of the discussions he provided me throughout my Master's degree completion.

I also want to thank my fellow labmates in Generative Software Development Lab for collaboration, support and motivation during my studies: **Jordan Ross**, **Ian Colwell**, and **Pavel Valov**.

A special thanks to **Markus Voelter** from Itemis AG for collaboration, support and domain knowledge transfer.

Lastly, I want to thank my family for all their love and encouragement.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

With the increasing number of intelligent automotive features and the push towards autonomous cars, modern automotive electric/electronic (E/E) architectures are becoming increasingly complex. The architects can no longer create and evaluate candidate architectures manually to understand the effects of their design decisions. Thus, architects require powerful modeling and reasoning tools to allow them to synthesize candidate architectures given some design decisions and discover the correct and optimal ones automatically.

One approach to conquering the complexity is using a *reference model* which prescribes a certain way of decomposing the overall architecture into layers and captures the crosscutting concerns, including variability and quality. We present such a reference model in Chapter 2. Furthermore, in order to be able to automatically reason about an E/E architecture (evaluate the effect of design decisions), the architecture must be represented using a formal modeling language which is supported by a scalable automated reasoner. One such language is Clafer [5]. However, Clafer is a general-purpose structural modeling language which does not provide the architectural concepts from the reference model as first-class language constructs. Thus, to make the modeling and reasoning power of Clafer available to practitioners who are not Clafer experts, we implemented an architecture domain-specific language (DSL) based on the reference model to guide users in correctly and consistently applying the reference model (Chapter 5). Our implementation, which we have named ClaferMPS, relies on the JetBrains MPS language workbench [43], whereby we implemented Clafer as an MPS language and the Architecture DSL as an extension of Clafer in MPS. We evaluate our work by using the DSL to model two existing architectures of two automotive subsystems which were previously modeled in plain Clafer [30]. The goal of the evaluation is to see whether the DSL improves the modeling experience compared to plain Clafer while still supporting the reasoning capabilities.

Another challenge discussed in this thesis is flexibility of the DSL-based approach. DSLs are designed for a specific domain, which causes some extensibility problems when adapting the system to other scope. One way to address this challenge is to decompose the DSL into a set of smaller extensible languages. This approach allows for adapting the DSL to a changing domain by extending existing languages or integrating new ones. We consider another case study, which is slightly different from the Architecture DSL scope, and demonstrate how the DSL can be adapted to a new domain via language extensions and compositions.

This thesis is organized as follows. First, we introduce background information about the reference model and the Clafer language in Chapters 2 and 3, respectively. Chapter 4 is dedicated to challenges of architecture modeling in Clafer. In Chapter 5 we provide a brief overview of the MPS/mbeddr platform, as well as the structure of ClaferMPS. In Chapter 6, we present the design of the Architecture DSL and how it addresses the challenges of applying plain Clafer to architectural modeling. Chapter 7 describes implementation of the ClaferMPS languages. In Chapter 9, we discuss ClaferMPS extensibility. We present the key observations and discussion in Chapter 8. Finally, we briefly summarize the related work in Chapter 10, and conclude the thesis in Chapter 11.

The contributions of this thesis are the following:

- Design and implementation of the DSL on top of Clafer, which aids domain specialists in creating E/E architectures based on their domain knowledge. Since our system complements to the previous work [30], it is supported by existing reasoning and visualization tools to provide a complete workflow for modeling and optimizing E/E architectures.
- We have evaluated expressiveness and flexibility of the Architecture DSL using three case studies
- Our work is a proof-of-concept implementation of Clafer extensions that demonstrates applicability of DSLs in supporting Clafer-based approaches for architecture exploration and optimization. We also show that additional extensions can be easily added to ClaferMPS; however, applicability of ClaferMPS to other domains needs to be evaluated in future work.

Chapter 2

A Reference Model for E/E Architecture Modeling

In this chapter, we provide a brief overview of the reference model for modeling architectures of E/E systems (for the remainder of this thesis we refer to it as "the reference model"). This model, which was introduced by Ross [31], is an adaptation of EAST-ADL simplified for early

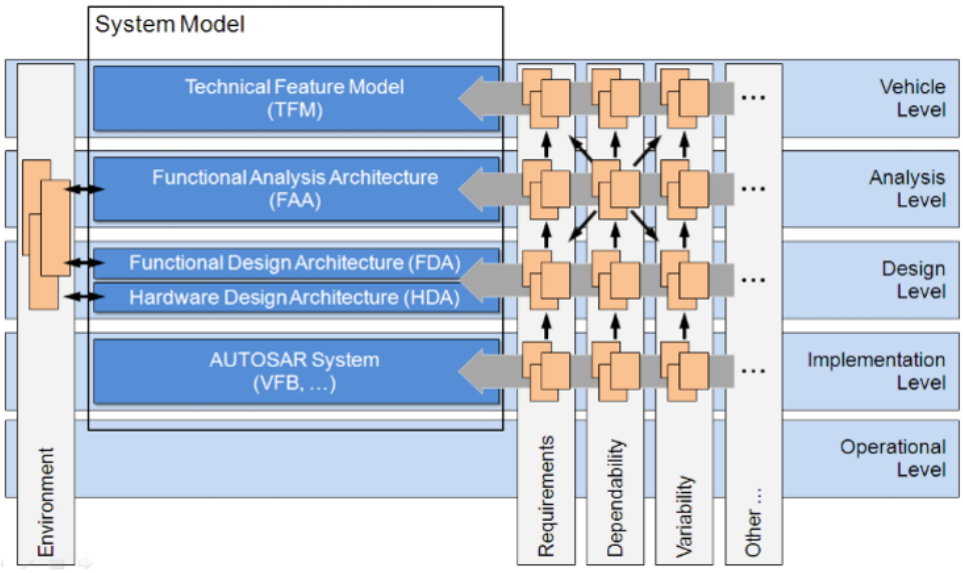


Figure 2.1: EAST-ADL Domain Model [11]

design of E/E system architectures.

EAST-ADL is an architecture definition language (ADL) for modeling automotive E/E architectures, which was designed to capture E/E systems with sufficient detail to allow modeling for documentation, design, analysis, and synthesis [11]. Figure 2.1 shows a diagram of the EAST-ADL domain model. The model is composed of multiple abstraction levels which can be cross-cut by one or more perspectives. Using EAST-ADL to model a system (or a product line of systems) assists in managing the associated complexity [10].

Similar to EAST-ADL, the reference model is *multi-layer*, and it prescribes dividing the architecture into multiple abstraction layers. Each layer describes the system, but at a different level of abstraction. Additionally, the reference model includes cross-cutting concerns that are not present in EAST-ADL, such as cost or mass, which were introduced by Murashkin [27].

The reference model is also *multi-perspective* in that it contains multiple perspectives, which augment the system with analysis-task or stakeholder-specific information such as points of variability, latency, and mass. Figure 2.2 shows the multi-layered, multi-perspective reference model used in this thesis with some of the supported perspectives.

2.1 Case Studies

The reference model used in this work is based on two case studies of automotive E/E subsystems. The first case study is an E/E architecture of a power window system in a car. The case study first considers a single driver side door system and then scales it up to a two door system by

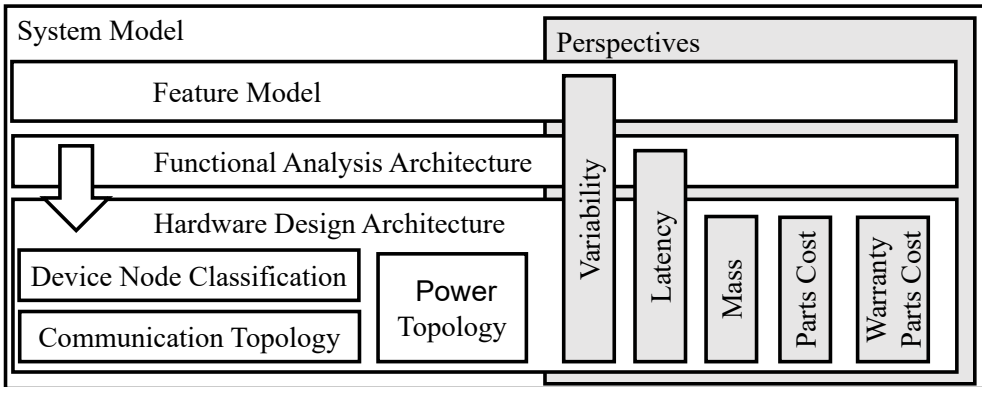


Figure 2.2: An automotive E/E system architecture reference model. The block arrow denotes the deployment of the functional analysis architecture to the hardware design architecture [31]

adding a front passenger window and communication between the doors. This case study has been developed and presented by Murashkin [27], and later extended by Ross [31]. Originally, the power window system has been chosen because it is self-contained and not overly complex.

The second case study is an E/E architecture for a central door locks system. It has been developed and presented by Ross [31]. This case study was chosen, because it is closely related to the power window study, and both these systems can be later consolidated in a larger system, and additional exploration of components sharing trade-offs can be performed.

In this thesis, we use these case studies to provide code examples of E/E architecture modeling in Clafer. Furthermore, Clafer models of the power window and door locks systems have been used for evaluating our work as discussed in Section 8

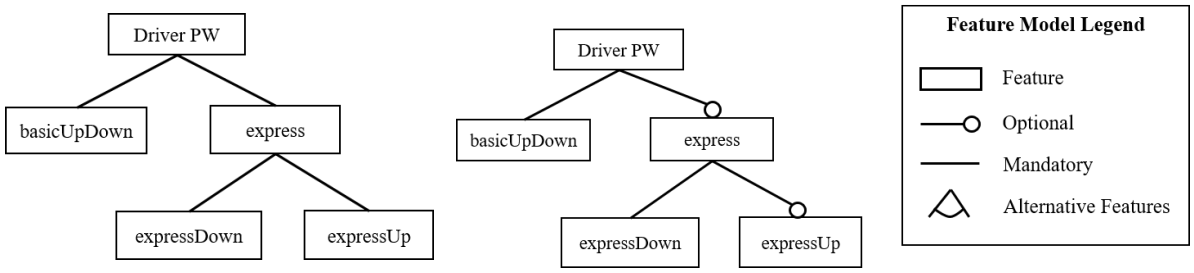
2.2 Reference Model Layers

In this section, we provide a brief introduction and basic understanding of the reference model layers. Additional details about the reference model constructs will be given later in the Section 6.

Starting with the top-most layer, the feature model contains *user-facing features*, such as *express up* and *pinch protection* in a power window system. Features are high-level functional or nonfunctional system characteristics requested by stakeholders, such as customers or users.

Figure 2.3 shows a feature model for the power window system, which is organized in a tree-like structure [19]. First, each power window supports basic up and down features represented by `basicUpDown` to close or open it by pressing up or down the switch. Next, some power windows support `express` features. It means that a user can completely close or open the window by pressing the switch only once. In Fig. 2.3a, all features are mandatory, that is, no variability is introduced. In Fig. 2.3b, `express` and `expressUp` are optional, which means three possible configurations: no any express functionality, only `expressDown`, and both `expressDown` and `expressUp` features.

These features are then implemented using one or more functions in the *functional analysis architecture* (FAA) layer. The FAA defines two types of functions: *analysis function* and *functional device*. The former models control functions with their inputs and outputs, while the latter captures functions that represent sensors and actuators. Apart from the functions, the FAA also captures the communication among them using *function connectors* in order to define a function graph. For example, Figure 2.4 shows a FAA for the power window, where the feature `expressUp` is implemented by functions `PositionSensor`, `PinchDetection`, and `WinControl`,



(a) Single system (b) Three variants

Figure 2.3: Power window feature models [31]

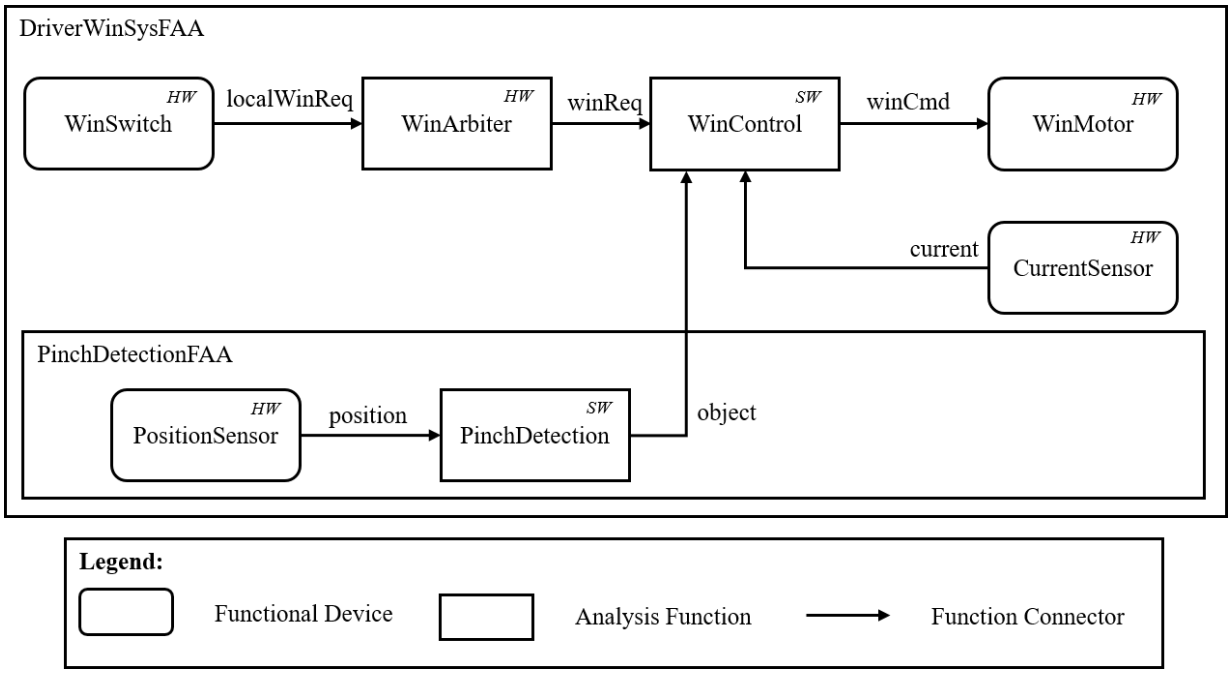


Figure 2.4: Power window functional analysis architecture [31]

as well as communications between them. The "HW" or "SW" in upper right corner of a functional component denotes the implementation method in hardware or software, respectively.

The functional analysis components are subsequently deployed onto hardware (the block arrow in the Fig. 2.2) defined in the *hardware architecture* (HA). The HA captures all physical devices and connection media, and can be decomposed into three sublayers:

- The *device node classification* contains *device nodes* of the system. The reference model

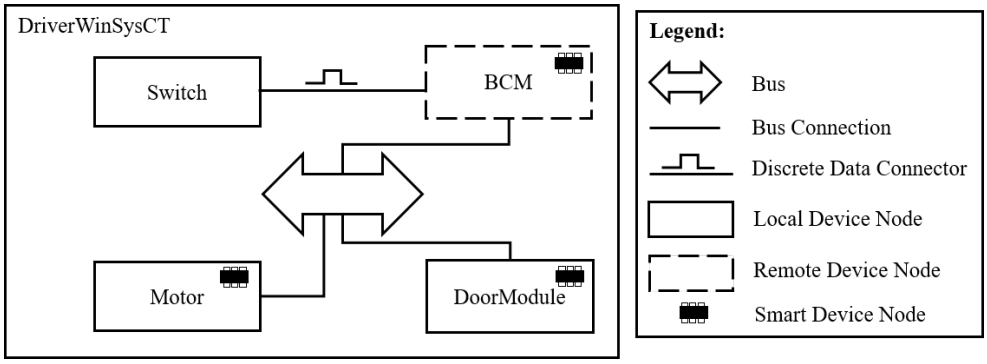


Figure 2.5: Power window communication topology [31]

considers three types of devices:

  - A programmable *Smart* device node that implements one or more software analysis functions or functional devices. Examples would be an ECU, or a hardware with embedded microcontroller.
  - An *Electric/Electronic* (E/E) device node that implements hardware functional devices or analysis functions. Examples would be an actuator or analog sensor.
  - A *Power* device node that generates, stores, or relays power to other device nodes. Examples would be a battery or fuse box.

- The *communication topology* captures the physical media that function connectors are deployed to. It includes discrete and analog data connectors between two devices, as well as serial buses (e.g. LIN, CAN, etc.) to connect two or more device nodes. Fig. 2.5 shows the possible communication topology of the power window system. The boxes with dashed border indicates remote nodes that are not physically located in the system.

- The *Power Topology* models power communications between device nodes. Figure 2.6 shows an example of the power topology. Note that in order to reduce model complexity, the power connectors, as well as the data connectors in the communication topology, are modeled at a high level of abstraction by omitting ports, pins, and routing through the wiring harness.

2.3 Reference Model Perspectives

In order to model multiple candidate architectures, variability has to be expressed in the model. Variability crosscuts all layers of the architecture. For example, in Figure 2.3b, `express` features

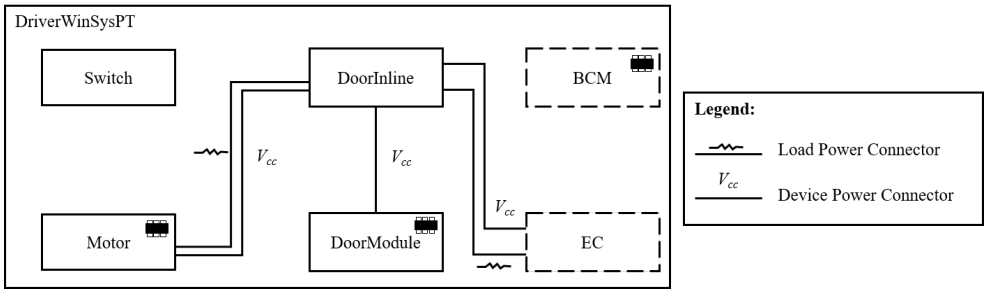


Figure 2.6: Power window power topology [31]

are optional, and the feature `expressUp` is only present if the feature `express` is. As a result, we now have three possible variants. The optional express features are then implemented by functions and hardware which also have to be optional as they are not needed when the features are not selected. Furthermore, there may exist alternative ways of realizing the features as different functions (e.g., different techniques of pinch detection). In Figure 2.7, the architecture fragment `PinchDetectionFAA` is optional as indicated by the dashed line (c.f. in Figure 2.4, the line is solid because the fragment is mandatory), which implies all nested functions and connectors are optional as well (`PositionSensor`, `position`, `PinchDetection`, and `object`).

In addition, there may exist alternative ways of deploying the functions onto hardware, as well as different hardware configurations. First, we consider variability for the types of device nodes used in the device node classifications. Second, the variability also exists in the presence of device nodes, as well as connectors in the power and communication topology. Furthermore, for some components we can vary their properties such as type of bus (i.e. LIN, CAN, or FlexRay) for a bus connector. As a result of adding variability to individual layer, the system architect is able to model an exponential (in the number of components) number of candidate architectures.

In order to evaluate the candidates based on metrics such as mass or cost, the architect must define quality attributes for the components. Similarly to variability, the quality perspectives may crosscut each layer of the architecture. For example, the latency of end-to-end flows (from sensors to actuators) depends on the functional connectors among the functions and functional devices, as well as on the particular deployment of these connectors onto the communication media (e.g., shared memory communication within an ECU vs. network communication between ECUs). Some qualities may also be confined to a particular layer, for example, hardware part cost, mass, and warranty cost only apply to the hardware design architecture.

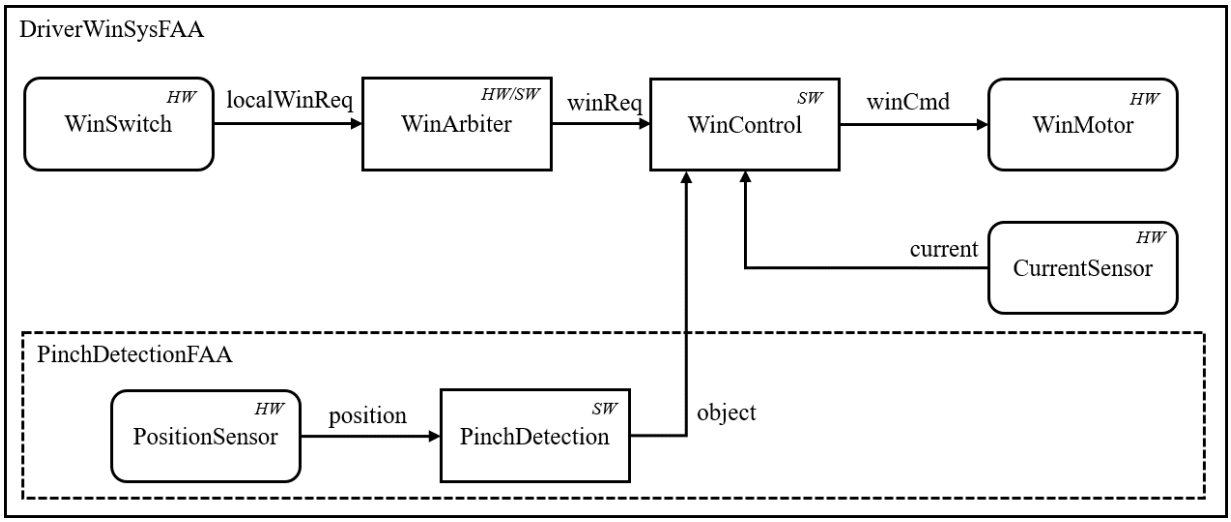


Figure 2.7: Power window functional analysis architecture with variability [31]

Chapter 3

Clafer Overview

Clafer[1] is a lightweight, general-purpose, textual, structural modeling language. 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. In this section, we briefly introduce the modeling and reasoning workflow when using Clafer. The formal semantics of the language can be found in [5].

3.1 Types of clafers, containment hierarchy, and inheritance

Clafer distinguishes two types of clafers: abstract and concrete, similarly to classes. A concrete clafer defines a new type and the number of possible instances of that clafer, while an abstract clafer represents only a type. Syntactically the only difference between these two types is the use of the `abstract` keyword.

Listing 3.1 shows an example of abstract and concrete clafers. Clafers are organized in a containment hierarchy: root clafers can contain nested clafers. The containment is specified via indentation. Clafers are also organized in an inheritance hierarchy specified using colon (:). For example, `ElectricCar` extends `Car` with an additional clafer `battery`.

[1] http://clafer.org/

[2] Throughout this paper if the word Clafer begins with an uppercase letter it refers to the language while Clafer in lowercase refers to the language construct.

Listing 3.1: Example Clafer model 1

```
abstract Car                    // defining a top level abstract clafer
   engine                       // nested concrete clafer

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

JanesCar : ElectricCar          // a concrete clafer inheriting from an abstract clafer
JohnsCar : JanesCar             // error: cannot inherit from a concrete clafer
```

3.2 Clafer multiplicity and group cardinality

An important feature of Clafer is variability support. There are two constructs expressing variability in the model: clafer multiplicity and group cardinality. Clafer multiplicity is a range `n..m` restricting how many instances of that clafer are allowed per instance of its parent. For example, a mandatory element has multiplicity 1..1 which requires exactly one instance of that clafer per instance of its parent; and optional clafers have multiplicity 0..1, thus they may or may not be present in model instances.

Group cardinalities are used to restrict how many instances of the clafer's children are allowed, for example, `mux` (short for 0..1) — at most one, `xor` (1..1) — exactly one, and `or` (1..*) — at least one, etc. By default, any clafer has the group cardinality 0..*, which means that it does not enforce any restrictions on its children.

Listing 3.2 shows an example of group cardinality and multiplicity. Here, `Car` has the multiplicity of `0..*`, which is a default value for all abstract clafers and usually omitted. Other multiplicities specify that every instance of `Car` includes one instance of `engine`, four instances of `wheel`, and from two to four instances of `seat`. The `xor` cardinality of `transmition` indicates that any of its instances contains either `automatic` or `manual` child.

Listing 3.2: Example Clafer model 2

```
abstract Car 0..*               // default clafer multiplicity (usually omitted)
   engine 1                     // short for 1..1
   xor transmition              // group cardinality xor - only one transition type can be present
      automatic 0..1            // optional clafer
      manual ?                  // optional clafer multiplicity expressed via "?" alias
   wheel 4
   seat 2..4                    // variable clafer multiplicity: from 2 to 4
```

The example model showed in Listing 3.2 can produce six different instances in total, and two of them are showed below:

```
=== 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 ---
```

To separate the multiple instances of the same clafer from each other, we use suffix `$n`, where n is the instance number.

3.3 References

A reference is a special type of clafer whose instances point to primitive values (e.g., integers) or instances of other clafers. In Listing 3.3, the clafer `Car` contains two references `driver` and `passenger`, which indicate the driver and passengers of the car, respectively. Both references have the type `Person`, which means that every instance of `driver` or `passenger` points to an instance of `Person`. In Clafer, a reference can be defined as set-valued (specified using ->) or bag-valued (multiset, specified using ->>). In our example, we used -> for the collection of passengers, because the same person can not be a passenger more than once.

Listing 3.3: Example using references

```
abstract Person

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

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

Apart from the abstract clafer `Car`, the model also includes two clafers of type `Person`, `John` and `Jane`. Thus, the example model generates eight different instances of `MyCar` in total, which can have no passengers, only John or only Jane as a passenger, or both John and Jane as passengers. Two of eight possible instances are shown below:

```
=== 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 ---
```

3.4 Constraints

Clafer also provides a powerful constraint language (first-order relational logic). A constraint is a boolean expression wrapped with square brackets "[ ]", which contains operations on sets and also on integers. Constraints are often used when a modeler wants to restrict targets of references and the allowed configurations of the model. Moreover, constraints are also useful for querying a model for specific instances that satisfy the given constraints.

Listing 3.4 shows examples of Clafer constraints. First, C1 assigns John as the driver of the `MyCar`. We use `.dref` to get the target of the reference clafer (similarly to dereferencing pointers in languages like C/C++). In C2, we say that only Jane and Dan can be passengers in `MyCar`. Here, the keyword `in` means a "subset of" the union set of instances of Jane and Dan. Next, C3 enforces that in every instance of `Car`, the target of `driver` is not presented in the `passenger` set. Finally, C4 states that the total number of persons can not exceed the number of seats in a car. Here, a keyword # is used to get the number of instances of a clafer.

Listing 3.4: Example Clafer model using constraints

```
abstract Person
abstract Car
   seat 2..4
   driver -> Person
   passenger -> Person 0..4
   [ driver . dref not in passenger . dref ] // C3
   [#( passenger , driver ) <= # seat ] // C4

MyCar : Car
   [ driver . dref = John ] // C1
   [ passenger . dref in (Jane , Dan)] // C2
John : Person
Jane : Person
Dan : Person
```

3.5 Working with integers and quality attributes

In order to add quantitative information to models (e.g. mass, cost, etc.), Clafer supports working with integers numbers. Integer quality attributes can be defined via the reference

notation (-> or -≫). Currently, the Clafer reasoner does not support real numbers and floating point; therefore, additional techniques like scaling or rounding to the nearest integer need to be used when modeling real systems.

Listing 3.1a shows an example of the car model with three features, which have an associated cost and warranty cost. We used constraints under `MyCar` to configure an instance of `Car` by giving values to the target of a reference. Another way to set value to the reference target is using clafer initializers (e.g. `warrantyCost` under the `Feature`). Since there is no variability in the model, just one correct instance is generated (Listing 3.1b).

```
abstract Feature
   cost -> integer
   warrantyCost -> integer = 10 // Clafer initializer

abstract Car
   bluetooth : Feature
   heatedSeats : Feature
   passiveKeyEntry : Feature

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

(a)

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

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

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

(b)

Figure 3.1: Example using integers

3.6 Tools

Clafer is supported by a set of tools [2], which include a scalable and exact instance generator and optimizer. Given a model expressed in Clafer, the instance generator can synthesize correct instances of the model. Furthermore, if the model contains optimization objectives, the instance generator can perform multi-objective optimization and generate a set of Pareto-optimal instances of the model. Finally, Clafer MOO (Multi-Objective Optimization) Visualizer [28] is a tool for visually exploring the set of optimal instances and performing trade-off analysis.

All these capabilities make Clafer suitable for expressing architectural models, which include representing variability and quality attributes, stating optimization objectives, synthesizing (optimal and non-optimal) candidate architectures, and evaluating the impact of design decisions, and performing design space exploration [27, 30].

Chapter 4

Domain-Specific Languages for the Definition of E/E Architectures

4.1 Challenges with E/E Architecture Modeling using Plain Clafer

Clafer is not domain-specific for the E/E modeling domain: it does not provide the architectural concepts from the reference model as first-class language constructs. Furthermore, Clafer can only express the structure (metamodel) of the architectural concepts from the reference model; Clafer and its general-purpose tools (compiler, instance generator, visualizer) cannot guide users in correctly applying the reference model rules. For example, Clafer will allow a user to leave references unconstrained and the instance generator will produce instances not intended by the modeler.

Model creators have to manually apply certain modeling idioms (conventions), such as, using inheritance to indicate the role of a given clafer (a clafer which inherits from `AnalysisFunction` represents an analysis function), appropriately setting the references, and using the right constraints. Murashkin first described such micro- and macro- level modeling patterns ([27], Chapters 4 and 5). Furthermore, they must have deep knowledge of Clafer and know how to achieve certain effects using just clafers and constraints. Finally, currently[1] Clafer does not provide a module system and the entire architecture must be contained in a single textual file. This makes testing fragments of the architecture difficult as modelers have to temporarily comment out parts they wish to ignore in a given test.

[1] As of version 0.4.4.

On the other hand, model readers have to recognize certain patterns and interpret the textual syntax to "see" the reference model concepts. The textual notation of Clafer emphasizes the containment hierarchy, which makes it hard to see inheritance and reference relationships among clafers. Clafer provides a generic graphical rendering which emphasizes inherintace and reference relationships; however, it is not suitable for visualizing architectural models. Finally, the lack of modularity makes it difficult to use parts of the architecture.

4.2 Domain Specific Languages

To address the challenges described above, a model-driven development (MDD) techniques [17] can be applied. In MDD, developers work in terms of high-level abstractions related to a particular domain, and then source code is generated using model-to-model and model-to-text transformations. The MDD approach is closely related to a concept of domain-specific languages (DSL).

A DSL is a language designed for a particular domain, which allows users to create models based on their domain knowledge and eliminate additional complexity resulting from implementation details. Since modeling idioms are encoded in the language itself, a model expressed with a DSL is more concise than the same program written in a general-purpose language. DSLs substantially improve productivity, comprehension, and maintenance of domain-specific models [7, 25].

4.3 Language Workbenches and Projectional editing

A language workbench is a set of tools for efficient definition, composition, evolution and reuse of domain-specific and general-purpose languages. Using language workbenches, new languages and their associated tools (such as IDEs, debuggers, visualizers, etc.) can be created with comparably little effort.

The term *language workbench* was first introduced by Martin Fowler [15]. According to Fowler, a DSL is defined in a language workbench as a combination of three aspects:

- A *Schema*, which is an abstract syntax of the language
- An *Editor*, which defines a graphical or textual representation of the abstract syntax and allows user manipulate the abstract syntax tree through *projections* (discussed below)
- A *Generator*, which translates the abstract syntax into a low-level executable representation

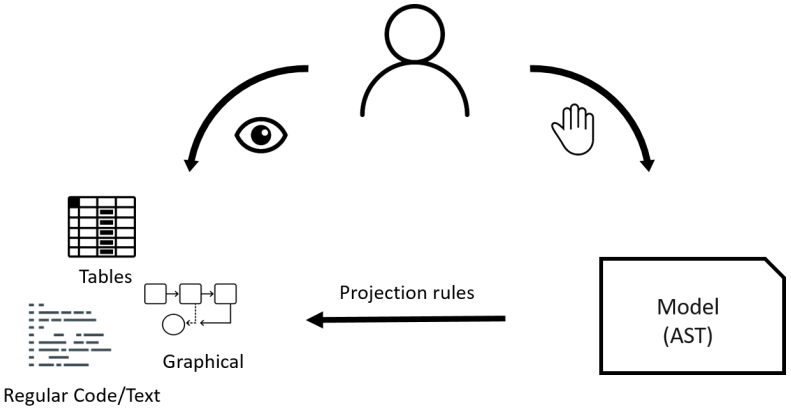


Figure 4.1: Projectional editing

To present the abstract syntax to users, language workbenches use a mechanism called *projectional editing*. In a traditional parser-based approach, a compiler parses character sequences typed by users to check the text for syntactic correctness, and then constructs an abstract syntax tree (AST), which contains all the semantic data expressed by the program. Any further analysis and processing, such as transformations or type checking, is based on the AST. In contrast, projectional editors do not involve parsers. Users modify the AST directly when editing a program. To interact with users, the projectional engine creates some concrete representation (textual or graphical) of the abstract syntax, which reflects the resulting changes (Figure 4.1).

Projectional editing has several advantages. First, since no parsers are used, the language syntax is free from the ambiguity problems associated with traditional parser-based approaches [15]. In projectional editors, any model element has a unique identifier (UID) and all references are based on the UID. Furthermore, programs are stored using XML or other tree persistence format. Second, the projectional editor can represent the AST in arbitrary syntactic formats, such as textual, graphical, tabular, or symbolic. This means that users can seamlessly mix different notations together, which significantly improves usability, and makes editing more flexible and intuitive. A well-known example of a projectional editor is MS Word, which saves documents in XML, and the users directly modify the AST. It also supports a variety of different notations including rich text, tables, figures, and mathematical formulas.

One challenge when dealing with projectional editors is making them convenient for end users. In the projectional editor, users cannot "just type" their code; they have to build the syntax tree of the model more or less manually. To address this challenge and simulate traditional textual editing, language workbenches provide various IDE features, such as aliases, code completion, side transformations, etc.

Chapter 5

Overview of ClaferMPS

Instead of writing user manuals and relying on modeling idioms, we decided to formally encode the reference model and its rules as an Architecture DSL [22], which guides the users in correctly and consistently applying the reference model rules. We implemented the DSL using JetBrains' Meta Programming System (MPS) [43] language workbench and mbeddr extensions. In this chapter, we present a brief overview of MPS and mbeddr, as well as components of ClaferMPS.

5.1 Meta Programming System (MPS)

As a language workbench, MPS is a tool which allows for efficiently developing domain-specific and general-purpose languages. MPS supports the definition of abstract syntax, textual, visual or tabular concrete syntax, type system, various rules and constraints, transformations, and code generators. All ingredients of powerful IDEs are also supported. MPS relies on *projectional editing*, where users directly modify the abstract syntax through a projected concrete syntax (discussed in Section 4.3); no parsing is involved. This allows MPS to support a wide range of notations [40] and various ways of language composition [36]. In particular, it supports language extension, where additional language concepts are added to a base language without invasively modifying this base language. MPS has been used to build ecosystems of integrated languages in various domains including embedded software, system specification, requirements engineering, safety and security analysis, insurance contract specification, medical software and public benefits calculations [41, 43].

5.2 MPS Language Aspects

In MPS, elements of language syntax are called *Concepts*. A concept has several aspects, such as structure, editor, type system, etc. Each AST node refers to its concept. In this section, we briefly describe basic language aspects of MPS language definition.

Structure The aspect *Structure* specifies an abstract syntax or metamodel of a language. In terms of structure, each concept has a name, properties (integer, string, or enumeration), children, and references to other concepts. Similar to classes, a concept can extend other concepts and implement multiple concept interfaces.

Editor The aspect *Editor* allows language designers to define the concrete syntax (aka *projection rules*). An editor for a node represents its view, as well as its controller. Editors consist of atomic elements (aka *editor cells*) arranged in various layouts (e.g. vertical, horizontal, indent, etc.). A cell may contain a property value (such as the concept's name), a child cell, a reference, a constant (e.g a keyword), or a collection of other cells. If the editor for the specific concept is not defined, its instance will be represented with an editor for the concept's nearest super concept that has an editor declaration.

Actions The aspect *Actions* allows for improving the projectional editing experience and making it similar to traditional text editors. Since the MPS editor manipulates the AST directly, users need some mechanisms to enter non-trivial tree changes linearly. For example, consider a simple numeric expression `1`. In order to encode `1 + 2`, the `1` needs to be replaced with the instance of binary `+` expression. Next, the user should append the `1` to the left slot of the `+` and the `2` to the right. Obviously, such type of editing is inconvenient. By using side transformations, users can just type the plus sign next to the `1` and the system will automatically restructure the AST by moving the `+` node to the root of the subtree with the `1` in the left slot, and then putting the cursor to the empty right slot. The Actions aspect holds such types of AST transformations.

Behavior The aspect *Behavior* contains user-defined virtual and non-virtual instance methods, static methods, and instance constructors.

Constraints This aspect defines constraints for properties (i.e. setter and getter functions, valid value ranges, etc.) and scopes for references (i.e. valid targets for a node). Additionally, the constraint aspect comprises concept context dependencies, which determine whether instances of the concept can be attached to other nodes as children/parents/ancestors in the current context.

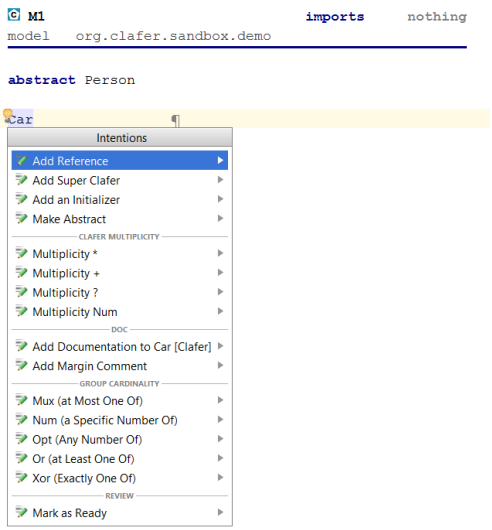


Figure 5.1: An example of the intention menu for the clafer node

Type System The aspect *Type System* specifies typing rules for concepts. It includes inference rules (i.e. computing a type for instances of a concept), subtyping rules, and checking (or non-typesystem) rules. The latter are used for detecting semantic errors and reporting them to users. For example, the Architecture DSL ensures that software function analysis components are deployed only to smart device nodes; otherwise, an error will be displayed in the editor. The type system also includes quick fixes that can be used to resolve type system errors.

Intentions *Intentions* are instant code manipulating actions available from the contextual menu that can be activated via `Alt + Enter` key shortcut. Intentions allows for fast access to frequently used operations with language constructs. For example, to make a clafer abstract, a user can open the intention menu under the clafer instance and choose the "Make Abstract" menu item (Figure 5.1). Another way to perform the same action is using side transformations by typing the `abstract` keyword before the clafer's name.

Generator The aspect *Generator* defines rules to map one AST to another AST. As has been discussed before, MPS follows model-to-model transformation, where user models are gradually transformed from one language to another lower-level language until the bottom-line level is reached. The last step is model-to-text transformation (discussed below), which is handled by the output language. The generation process of the Architecture DSL is discussed in Section 6.7

Textgen This aspect is used to define the mapping of a language concept to its textual representation and save the resulting text files on disk. It comes with constructs to transform nodes into text values, and print them out into some reasonable layout. The textgen aspect should be defined in the bottom-line language, such as Clafer, Java, C, etc. Any other high-level concepts (such as architecture constructs in the Architecture DSL) should use generators.

Scripts This aspect holds migration or enhancement scripts for language concepts. For example, they can be used to migrate outdated concepts to the most recent version of the language.

Additionally, MPS includes other aspects, such as Find Usage, Dataflow, Refactoring, etc. However, these aspects are not used in our work. For more details about MPS language aspects, we refer readers to elsewhere [37].

5.3 mbeddr

mbeddr[1] is a set of integrated and extensible languages built on top of the MPS language workbench for C-based embedded software development.

mbeddr comes with the C language, which is an MPS implementation of C99, as well as fully featured IDE for C extensions and any other languages developed in mbeddr. Having the C language available as an MPS language allows for extending programs with various domain-specific constructs, and at the same time being able to write low-level C code, if needed.

mbeddr also comes with a variety of extensions related to embedded-software development. Figure 5.2 shows an mbeddr components stack [37]. It is organized into concerns, which address C code implementation, formal analysis, and processing. The mbeddr stack is also multi-layered. Apart from the MPS platform, C-based core facilities, and integrated backend tools, mbeddr provides a number of default extensions, such as state machines, physical units, model-checking, visualizations, documentation (including html and LaTeX), etc. Furthermore, since mbeddr is built on top of a language workbench, it allows users to create their own extensions (the topmost layer in Fig. 5.2) of C or default extensions, without modifying any existing languages. Finally, mbeddr is organized as a set of modular languages, which means that developers can integrate any of the mbeddr components to their own projects, even if they are not based on C. In our work, we use a variety of mbeddr extensions including a modular system, diagrams, and primitive types.

The mbeddr project is open source and maintained by Itemis AG[2]. It has been successfully applied within several commercial projects as discussed in [38, 39].

[1] http://mbeddr.com/
[2] https://itemis.com

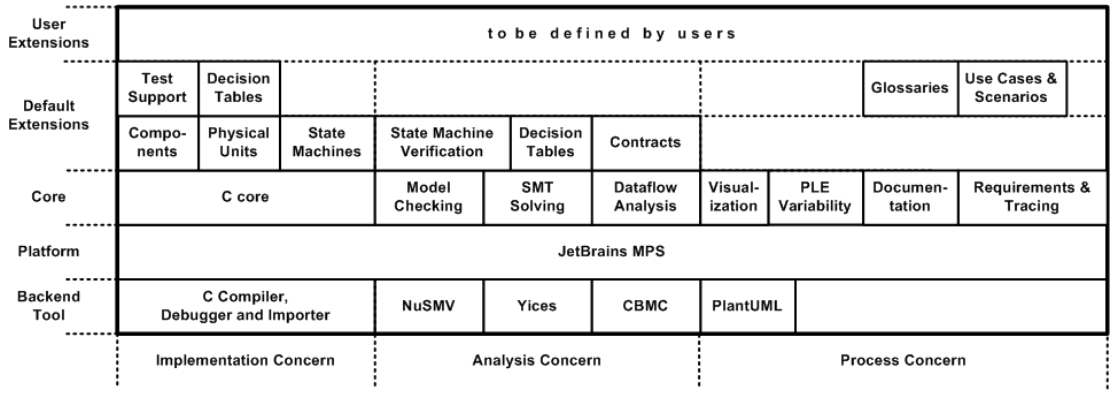


Figure 5.2: The mbeddr stack [37]

5.4 Components of ClaferMPS

Fig. 5.3 shows the components of our implementation. The boxes with gray background represent existing tools. On the plain Clafer side (right), we are using the Clafer compiler, which works with plain-text files, the instance generator, and the visualizer [2, 28].

On the MPS side (left), we build on top of MPS and use some utilities of mbeddr [41] such as the module system and graphical notation. The boxes with a pattern background represent the new components we developed: Clafer language, which implements full Clafer and provides a textual syntax; and Architecture DSL, which provides textual and graphical syntaxes. A model created in Architecture DSL is first transformed into a model expressed in the Clafer language in MPS, from which a plain-text Clafer model is generated. Generating a plain-text Clafer model allows us to leverage the existing Clafer toolchain (shown in Fig. 5.3 on the right). We refer to both the Clafer implementation in MPS and the Architecture DSL collectively as "ClaferMPS".

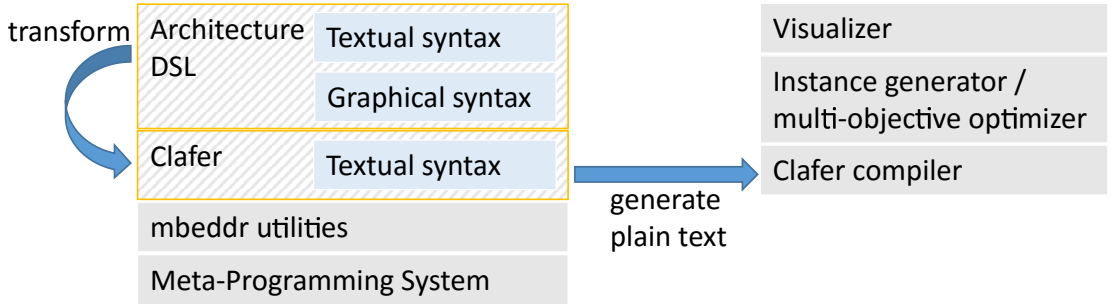


Figure 5.3: ClaferMPS (left) and plain Clafer (right) tool stacks

Chapter 6

E/E Architecture Modeling: Clafer vs. ClaferMPS

In this section, we demonstrate how the challenges of using plain Clafer for E/E architectural modeling are solved with ClaferMPS by comparing both approaches. For this purpose, we decomposed the modeling process into several steps. First, we define general concepts and domain-specific rules from the reference model. Next, to describe multiple candidate architectures, we add variability to each of the layers. In order to evaluate architectures based on various metrics, we define qualities for the layer components. Finally, we perform optimization and exploration over the possible candidates. In addition, we discuss other modeling aspects, such as project modularity, DSL extensibility, and presentation to user.

6.1 Applying E/E Reference Model Concepts

Since Clafer does not have first-class support for the E/E reference model concepts, we must define these abstractions first. In addition, both the reference model concepts and the concrete model must be contained in the same file, because Clafer currently lacks a module system. Listing 6.1 shows the feature modeling concepts *feature model* and *feature* encoded as abstract clafers.

Listing 6.1: Feature modeling concepts defined in plain Clafer

```
abstract FeatureModel
abstract Feature
```

Listing 6.2: Feature modeling (Clafer)

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

Concrete feature models can then be created by extending the abstract clafers `FeatureModel` and `Feature` as shown in Listing 6.2 (the symbol `:` indicates inheritance; `?` indicates optionality). In Clafer, we use indentation to nest clafers (i.e., establish containment) and to indicate dependency that the feature `expressUp` requires `express`.

Similarly, the remainder of the reference model can be encoded in Clafer using abstract clafers [30]. While this approach is valid for modeling E/E architectures, it is limited by its inability to guide users in applying these concepts correctly. For example, a plain Clafer alone cannot ensure that a *feature* can only be defined inside (i.e., nested under) a *feature model* or another *feature*, because this requirement is specific to the reference model.

In addition to being correctly nested, reference model concepts must also be constrained properly. For example, Listing 6.3 shows the definition of the functional analysis architecture concepts in plain Clafer. It consists of *analysis function*, *functional device*, and *function connector*. The latter has two nested reference clafers (indicated by `->`), which represent the connector's endpoints (lines 14-15). Moreover, each functional analysis component or connector can be deployed into the hardware architecture (lines 2 and 16, respectively). The concepts also include many constraints, such as, that analysis functions can only be deployed to smart device nodes or that function connectors should not be deployed to anything when their sender and receiver are deployed to the same device node (e.g., the same ECU).

Listing 6.3: Encoding of functional analysis architecture concepts in plain Clafer

```
1  abstract FunctionalAnalysisComponent
2    deployedTo -> DeviceNode
3    latency -> integer
4
5  abstract AnalysisFunction : FunctionalAnalysisComponent
6    [deployedTo.type in SmartDeviceNode]
7    baseLatency -> integer
8    [latency = baseLatency*deployedTo.speedFactor]
9
10 abstract FunctionalDevice : FunctionalAnalysisComponent
11   [deployedTo.type in (SmartDeviceNode, EEDeviceNode)]
12
13 abstract FunctionConnector
14   sender -> FunctionalAnalysisComponent
15   receiver -> FunctionalAnalysisComponent
16   deployedTo -> HardwareDataConnector ?
17     [parent in this.deployedFrom]
18     [(sender.deployedTo.dref,receiver.deployedTo.dref) in (deployedTo.endpoint.dref)]
19   [(sender.deployedTo.dref = receiver.deployedTo.dref) <=> no this.deployedTo]
20   latency -> integer
21   messageSize -> integer
22   [latency = (if deployedTo then messageSize*deployedTo.transferTimePerSize else 0)]
```

Violating these constraints (e.g., deploying a function to a power device) will prevent the Clafer instance generator from producing any instances; instead it will report the set of mutually contradicting constraints, which then require model debugging. Furthermore, even if no constraints are violated, the instance generator can still produce correct (i.e., satisfying all constraints) but invalid instances (i.e., not making sense in terms of the domain) because the model can be underconstrained. For example, if the reference clafers on lines 14 and 15 for a function connector

Listing 6.4: A valid and an invalid function connector model example

```
1  WinSwitch : FunctionalDevice
2  WinArbiter : AnalysisFunction
3    [latency = 10]
4  WinControl : AnalysisFunction
5
6  // valid connector
7  winReq : FunctionConnector
8    [sender = WinArbiter]
9    [receiver = WinControl]
10
11 // underconstrained (invalid) connector
12 localWinReq : FunctionConnector
```

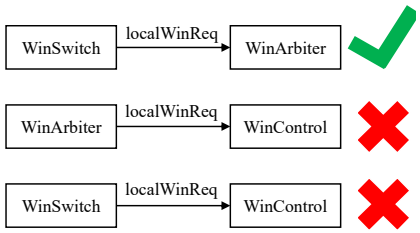


Figure 6.1: Instances generated from Listing 6.4

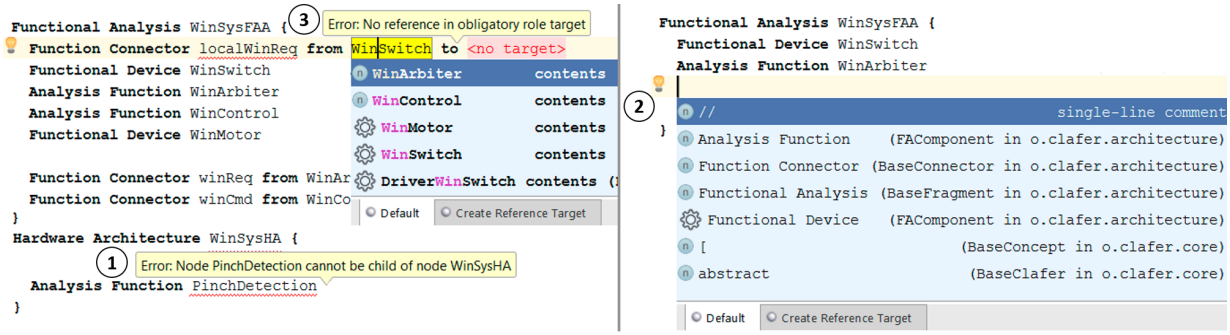


Figure 6.2: ClaferMPS functional analysis example.

are not constrained to point to valid targets, the instance generator will be free to choose any function as a target, which is likely to result in a nonsensical architecture. Listing 6.4 contains a concrete example showing a correct (with respect to the stated constraints) yet invalid Clafer declaration of `localWinReq` (the connector should only be allowed between the `WinSwitch` and `WinArbiter` functions) and Fig. 6.1 shows the resulting instances. This example is *invalid* since it did not reflect the domain adequately. However, `winReq` is an example of a correct and valid function connector since the sender and receiver are properly constrained.

ClaferMPS solution In order to minimize the need for writing constraints manually, we have designed and implemented a DSL on top of Clafer (using MPS' support for language extension), which provides E/E architecture concepts as first-class concepts to cover most of the reference model rules. Figure 6.2 shows a snippet of a functional analysis architecture modeled with the DSL. To ensure that users nest the reference model elements correctly, we restrict the usage context of the reference model concepts. This means that the DSL's auto completion menu shows only those concepts that are valid in the current context (i.e., *analysis function* is only shown in the context of *functional analysis architecture*) which can be seen at location ② in Fig. 6.2. If the user copy-/pastes an element into the wrong context, the error will be presented as shown in Fig. 6.2, ①.

Next, the DSL syntax was designed to include values for all required reference clafers (from

the plain Clafer approach) for the different concepts. Using the earlier example of function connectors, the DSL ensures that the user does not forget to set the targets of the sender and receiver, which are mandatory in the syntax. Additionally, the type system of the DSL ensures that the types of the chosen targets are correct, otherwise an error is reported (Fig. 6.2, ③). Thus, the DSL eliminates many common errors and minimizes the need for manually writing constraints and, consequently, model debugging.

Finally, the DSL supports semantic error detection. A simple example of a semantic rule can be formed from the constrains on lines 6 and 11 of Listing 6.3; it states that a device node of type *power* can't be given as a deployment target for an analysis function or functional device. Checking such rules informs users that there is a semantic error in the model (Fig. 6.4, ⑦).

6.2 Variability

In order to model more than one candidate architecture, the model must be augmented with variability. In plain Clafer, variability is expressed using multiplicities, group cardinalities, and reference clafers. For example, Listing 6.2 shows how variability can be expressed for the feature `express` by using a clafer multiplicity of 0..1 (denoted by `?`). In ClaferMPS, we chose to model variability the same way as in plain Clafer, but using different keywords such as `optional` to help architects.

6.3 Quality Attributes

To evaluate the quality of candidate architectures, we need to annotate the different reference model components with *quality attributes*. These attributes can then differ among domains and even systems within a domain. For example, a power window system might not consider *security* as a quality, whereas a doorlocks system might.

In Clafer, quality attributes can be added to the reference model by nesting a clafer under the component type as shown on line 3 of Listing 6.3. Then, in the definitions of concrete components, the values can be defined using constraints as shown on line 3 of Listing 6.4. The challenge with this approach is that users have to directly modify the reference model and nest clafers appropriately without any guidance. Additionally, these modifications can lead to inconsistencies in the reference model over time or introduce subtle errors.

ClaferMPS solution In ClaferMPS, users do not need to edit the reference model; instead, we provide a table shown in Fig. 6.3 whereby users define one or more *integer-valued* quality

attributes ② for the chosen architectural concepts ①. Furthermore, it is also possible to extend the reference model with additional plain clafer constructs ③, for example, to calculate values for quality attributes. Then, a user can immediately use the intention menu for a defined architectural concept (for example, the device node `Switch`) to add a value for that quality ④, ⑤. The intention menu is a contextual menu that allows users to perform various modifications of the model. Finally, quality attributes are properly inherited by subconcepts: the intention menu shows both concept-specific and inherited attributes.

Additionally, since the quality attributes are separate from the reference model, users can generate plain Clafer with or without the quality attributes. This allows the users to validate

Element	Content	Quality Attributes
Device Node ①	[this.warrantyCost = this.ppm * this.replaceCost] ③ [(this.type.power || this.type.electr) => this.speedFactor = 0]	mass ② cost warrantyCost replaceCost ppm speedFactor
Hardware Connector		mass cost length
Hardware Data Connector		transferTimePerSize
Function Connector	[if (some(this.deployedTo)) then (this.latency = this.messageSize * this.deployedTo.transferTimePerSize) else (this.latency = 0)]	messageSize latency
Discrete Data Connector	[this.mass = this.length * MassPerUnitLength.DiscreteDataConnector * #(this.deployedFrom)] [this.cost = this.length * CostPerUnitLength.DiscreteDataConnector * #(this.deployedFrom)] [this.transferTimePerSize = 0]	<< ... >>
Functional Analysis Component		latency baseLatency

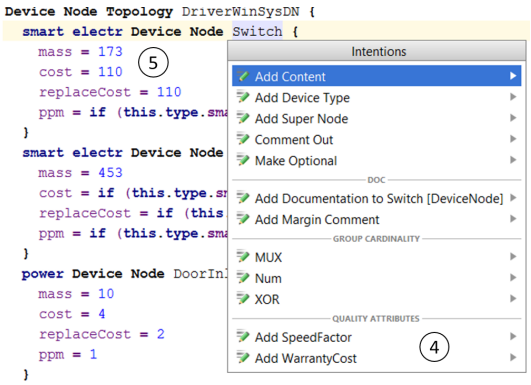


Figure 6.3: User-defined quality attribute declarations for the architectural concepts (left). An intention menu for assigning values of quality attributes to model elements (right).

their architectural model (i.e., ensure that their model captures all possible candidates they intended to model) without taking the qualities into account. In plain Clafer, such a task requires manually commenting out the quality attributes in the reference model and all constraints which set their values. In Section 6.7, we describe how the generation process supports this functionality.

6.4 Extensibility

In plain Clafer, since the reference model is a set of abstract clafers included in the same file as the concrete system, users can perform arbitrary changes to the reference model and use all capabilities of the language in unrestricted ways. It is both an advantage for users who are Clafer experts as well as a disadvantage for non-expert users because they lack guidance and they can suffer from common errors.

ClaferMPS solution As a result of building the Architecture DSL on top of the Clafer language in MPS, clafers and constraints can be mixed with the architectural elements. This is a common case when a modeler wants to use Clafer's constraint language to write additional constraints that are not expressible using the Architecture DSL. For example, Figure 6.4 shows a deployment specification of a functional architecture `WinSysFAA` to a hardware architecture `WinSysHA` ①. The concepts `Deployment` and `Deploy` ① belong to the Architecture DSL; however, the element `patterns` ② is simply a clafer which, in this case, is used to group rules for the `distributed` ③ and `centralized` ④ deployment patterns. Also, the figure shows a few constraints which go beyond what is expressible using the `Deploy` concept: some of them must always hold because

```
Deployment WinSysDpl of WinSysFAA to WinSysHA {
① Deploy CurrentSensor to deployment of <WinMotor>
   [(WinControl.deployedTo = Motor) <=> MotorIsDriver] ⑤
   xor patterns ②
     distributed
   ③ Deploy WinMotor to Motor
     Deploy WinControl to Switch, Motor, BCM, DoorModule
     centralized
   ④ Deploy PinchDetection to BCM , DoorInline
     Deploy position to motorBCMDisc
     [(WinControl.deployedTo = BCM) <=> BCMIsDriver] ⑥
-------------------------------------------------------------
     Error: WinSwitch cannot be deployed to a power device node DoorInline ⑦
   Deploy WinSwitch to DoorInline
}
```

Figure 6.4: Mixing clafers and constraints within a deployment (above the gray separator) and an example of semantic error for an invalid function deployment target (below the separator).

they are nested directly under deployment ⑤, some of them must only hold when an instance of the clafer `centralized` is present ⑥.

Additionally, ClaferMPS still provides guidance when adding clafers to an architectural model through auto-completion and type checking.

The ability to mix clafers and constraints with DSL elements allows for lightweight extensibility of the reference model. In Figure 6.4, the intention of the modeler is to specify a few alternative `DeploymentPattern`s, which is a concept currently not available in the reference model. Thanks to MPS, organizations can modularly extend the Architecture DSL by creating their own reference model which imports our reference model and adds new concepts, such as the `DeploymentPattern`. Next, they can create a new DSL which extends our Architecture DSL and adds the `DeploymentPattern` as a first-class concept together with an editor, typing, and other rules. The ability to mix clafers and constraints within the architectural models allows the DSL designers to work with the proposed extension before formally implementing it as a DSL in MPS.

6.5 Modularity

E/E architecture models can be quite large. Currently, Clafer does not have a module system and thus users have to define their model in a single, potentially large, text file. The model then becomes cumbersome to navigate, especially when modeling multiple subsystem architectures together.

ClaferMPS solution ClaferMPS provides a simple module system that allows users to create *modules*, which export all contained definitions and which can import definitions from other modules. The modules are combined together during the generation process which we detail next in Section 6.7.

6.6 Presentation

The Clafer compiler can generate a static graphical representation of a model which shows the inheritance hierarchy and references as shown in Fig. 6.5. This graphical representation is complementary to the textual syntax which emphasizes clafer nesting; however, it is not suitable for visualizing architectures.

ClaferMPS Solution In addition to textual syntax, the Architecture DSL provides a graphical representation of E/E architectures. This allows for architects to visualize the relationships and

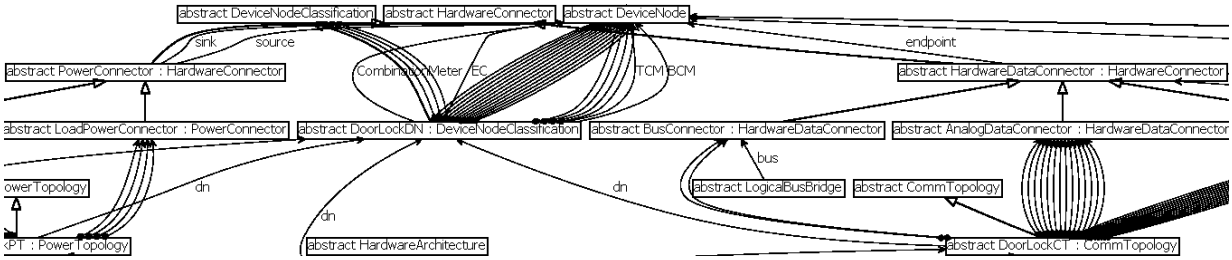


Figure 6.5: Snippet of the graph for door locks generated by Clafer compiler

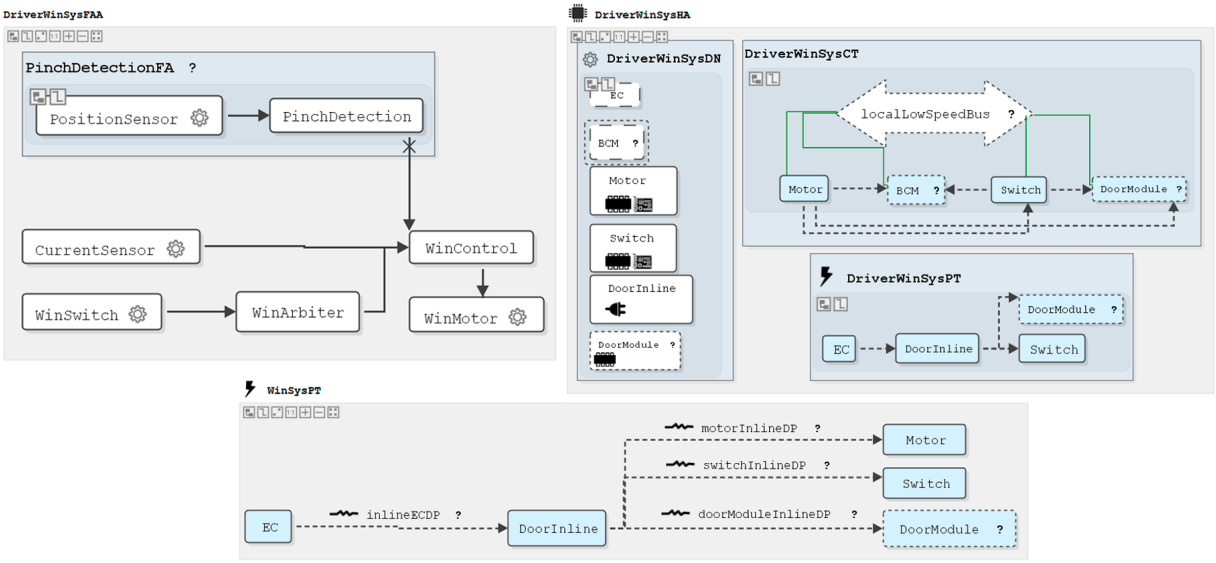


Figure 6.6: ClaferMPS Architecture DSL Diagrams

connections between different elements to ensure that their model matches what they intended. Figure 6.6 shows snippets of a few kinds of diagrams expressed with the graphical notations of the DSL; the diagrams are fully editable and, since they are projections of the same underlying model, they are always synchronized with the textual representation. Users can switch between textual and graphical projections and even view and edit both side-by-side.

The graphical editor is implemented using an MPS extension provided by the *mbeddr.platform*[1]. It does not only provide basic rendering functionality but also a set of helper tools such as automatic layout, alignment, and snapping, which reduce the effort for manually arranging the diagram elements. However, some manual layouting is still necessary.

[1] http://mbeddr.com/platform.html

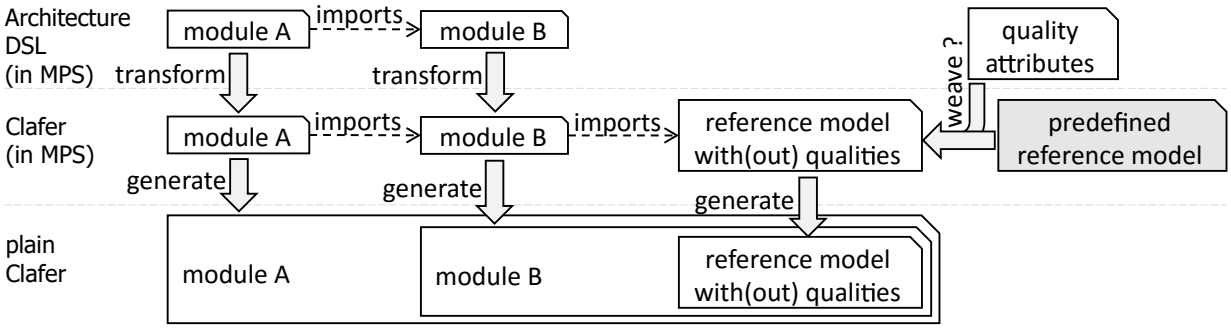


Figure 6.7: Plain Clafer generation process

In the Architecture DSL, diagrams focus on the structure and hide other information such as quality attributes or plain clafers. This allows users to view the model from a different perspective than offered by the textual representation. Additionally, thanks to support for modularity, users can see a graphical projection of their current module allowing them to work with a specific portion of the overall architecture.

6.7 Reasoning, Debugging, and Multi-Objective Optimization

The typical workflow when modeling in plain Clafer is to write a small model fragment or temporarily comment out parts of a larger model irrelevant for the task at hand, execute the compiler to check whether the model is correct (syntax, name, type checking), execute the instance generator to validate the model (check that only valid instances are produced), and repeat. Murashkin described such micro-level and macro-level modeling patterns [27]. Next, users perform multi-objective optimization and impact and trade-off analyses [2, 30].

Furthermore, the modelers often validate the model logic without the quality attributes, which requires commenting them out (cf. Section 6.3).

ClaferMPS Solution The current reasoning, debugging, and multi-objective optimization tools require plain Clafer as input. Moreover, since Clafer tools do not have a module system, all imported modules, quality attributes, and the reference model must be combined into a single file.

Figure 6.7 shows how ClaferMPS generates plain Clafer. First, the Architecture DSL takes the predefined reference model without quality attributes expressed in Clafer in MPS (it contains Clafer code as shown in Listings 6.1 and 6.3, but written in MPS). If the user chooses to include

quality attributes, ClaferMPS weaves them into the predefined reference model resulting in a reference model with quality attributes; otherwise, the predefined reference model is used directly. Next, the DSL transforms the modules expressed in the Architecture DSL into equivalent modules expressed in Clafer in MPS, while preserving the import structure. If the user has configured the DSL to exclude the quality attributes, ClaferMPS ignores all quality-related expressions during the transformation. Also, the resulting modules must now import the reference model. Finally, Clafer in MPS generates plain Clafer files for every module such that the resulting file contains all of its imported modules. For example, the generated plain Clafer for the module B contains the module's contents and the reference model with or without the quality attributes, whereas the plain Clafer for the module A has its contents as well as all contents of the module B.

This process allows the users to reason about, debug, and optimize each layer of the architecture separately with or without quality attributes. While the users still need to comment out unused parts of the reference model; their workload is greatly reduced.

Chapter 7

ClaferMPS Implementation

In this chapter, we discuss the implementation of ClaferMPS (version 0.4.4.3). It can be split into two main parts. The first part is called *Clafer core*. It is essentially an implementation of the Clafer language in MPS. Similar to the C language in mbeddr, having the MPS version of Clafer allows for extending models with various domain-specific constructs that can be later transformed into plain clafer code. The second part is actually the *Architecture DSL*, which implements E/E domain-specific constructs, as well as essential abstractions for Clafer-based extensions. Both the Clafer language and the Architecture DSL are expressed using standard MPS DSLs and mbeddr extensions. In this section, we provide a brief overview of the ClaferMPS languages, and also illustrate how most important concepts are implemented in MPS. The complete source code of ClaferMPS can be found in [22].

7.1 Clafer Core

Figure 7.1 shows an abstract syntax of the Clafer core. It is organized in two languages: **org.clafer.core** and **org.clafer.expressions**. The **org.clafer.expressions** language defines primitive types (integer, string, and boolean), literals, and operators. This language is based on the mbeddr expression language. To adjust it for our purposes, we removed all C-based expressions, and extended the language with additional Clafer constructs such as binary set expressions (e.g. `in`, `nin`, `++`, etc.), quantifiers (e.g. `some`, `one`, etc.), minimum and maximum functions (`min`, `max`), and other expressions. The **org.clafer.core** language contains all basic constructs of Clafer such as clafer, modules, clafer reference expression, cardinalities, etc. In addition, the core language defines a textgen aspect for plain text generation, which is the final step of multi-stage transformation for all Clafer-based DSLs.

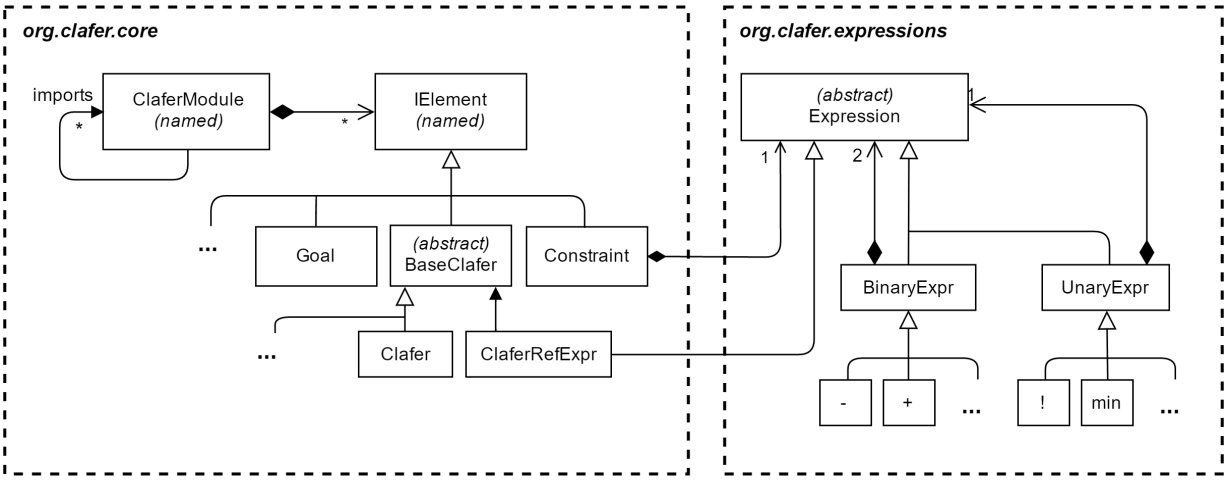


Figure 7.1: The abstract syntax of the ClaferMPS core languages.

7.1.1 Clafer Module

`ClaferModule` is a root concept, which represents modules in ClaferMPS. The modular system has been discussed previously in Section 6.5. Each module contains a collection of `IElements`. The interface concept `IElement` is implemented by all constructs that can be inserted into modules. The `IElement` extends an MPS' `IIdentifierNamedConcept` interface, which provides a `name` property. `ClaferModule` also implements `IContainerOfUniqueNames` interface, so that conflicts among global names are detected automatically.

The `IElements` is implemented by three basic concepts: `Goal`, `Constraint`, and `BaseClafer`. The latter is an abstract concept, which acts as a super type of all clafer-like constructs, as well as of a clafer itself. Any subconcept of the `BaseClafer` will be later transformed to a plain clafer.

7.1.2 Clafer

Figure 7.2 shows an abstract syntax of `Clafer` in MPS, and Figure 7.3 demonstrates the actual structure of the `Clafer` concept.

The clafer syntax can be decomposed into a set of syntactic components:

```
<abstract?> <group cardinality?> <name> <super?> <reference?> <multiplicity?> <initializer?>
        <elements*>
```

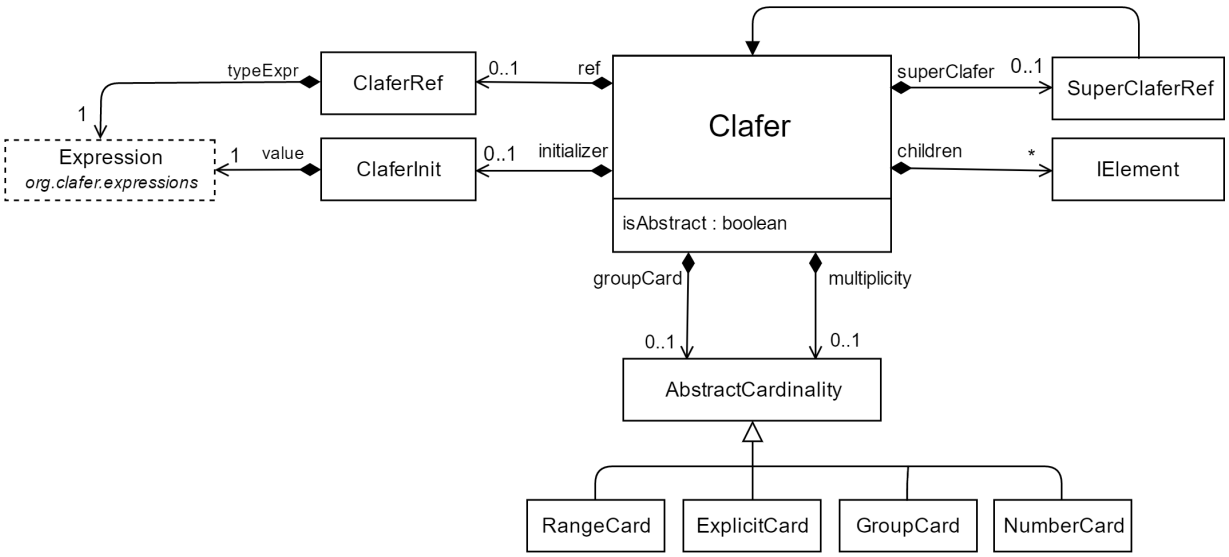


Figure 7.2: The abstract syntax of the `Clafer` element.

Table 7.1 shows the syntax of each of the clafer components, as well as their corresponding MPS representations. The only mandatory part of the clafer declaration is <name>, which is represented as a string `name` property. The <abstract> component is also a boolean `isAbstract` property of the `Clafer` concept. A concept `SuperClaferRef` has exactly one reference to any other abstract clafer that is visible in a given scope. Concepts `ClaferRef` and `ClaferInit` represent a reference to another clafer and a value of the given clafer, respectively, and contain exactly one `Expression`. Finally, both <multiplicity> and <group cardinality> are represented by subconcepts of `AbstractCardinality` such as `RangeCard` (<int literal>..<int literal>), `ExplicitCard` (?, *, +), `GroupCard` (e.g xor, or, etc.), and `NumberCard` (<int literal>). To ensure that Clafer cardinality rules are satisfied, the Constraint aspect is used, as shown in Figure 7.4.

```
concept Clafer extends     BaseClafer
                implements IDontSubstituteByDefault
                           ICommentable

  instance can be root: false
  alias: <no alias>
  short description: clafer

  properties:
  isAbstract : boolean

  children:
  superClafer  : SuperClaferRef[0..1]
  children     : IElement[0..n]
  multiplicity : AbstractCardinality[0..1]
  groupCard    : AbstractCardinality[0..1]
  ref          : ClaferRef[0..1]
  initializer  : ClaferInit[0..1]

  references:
  << ... >>
```

Figure 7.3: The structure aspect of the Clafer concept

```
concepts constraints Clafer {
  can be child <none>

  can be parent
    (childConcept, node, childNode, operationContext, link)->boolean {
      if (link == linkNode/Clafer : multiplicity/) { return childConcept == conceptNode/RangeCardinality/ ||
          childConcept == conceptNode/NumberCard/ || childConcept.isSubConceptOf(ExplicitCardinality); }
      if (link == linkNode/Clafer : groupCard/) { return childConcept == conceptNode/RangeCardinality/ ||
          childConcept.isSubConceptOf(GroupCardinality); }
      return true;
    }
```

Figure 7.4: The constraints aspect of the Clafer concept

Table 7.1: The syntactic components of clafer with the corresponding MPS representations

Component	Mandatory	Syntax	MPS representation
<abstract>	no	abstract	isAbstract : **boolean**
<group cardinality>	no	xor or mux opt <int literal>..<int literal>	GroupCard RangeCard
<name>	yes	string	name : **string**
<super>	no	: <name>	SuperClaferRef
<reference>	no	-> <set expression> -≫ <set expression>	ClaferRef
<multiplicity>	no	? * + <int literal> <int literal>..<int literal> <int literal>..*	ExplicitCard NumberCard RangeCard
<initializer>	no	= <set expression>	ClaferInit

7.2 Architecture DSL

The ClaferMPS structure was inspired by the modular design of mbeddr, where various modularization and composition techniques are used to reduce complexity and improve reuse of domain specific languages. In Architecture DSL, we split the abstract and concrete syntax of architecture elements into multiple languages [36]. Currently, the Architecture DSL consists of two languages. The first one, **org.clafer.architecture.core**, defines generic architecture concepts, diagram

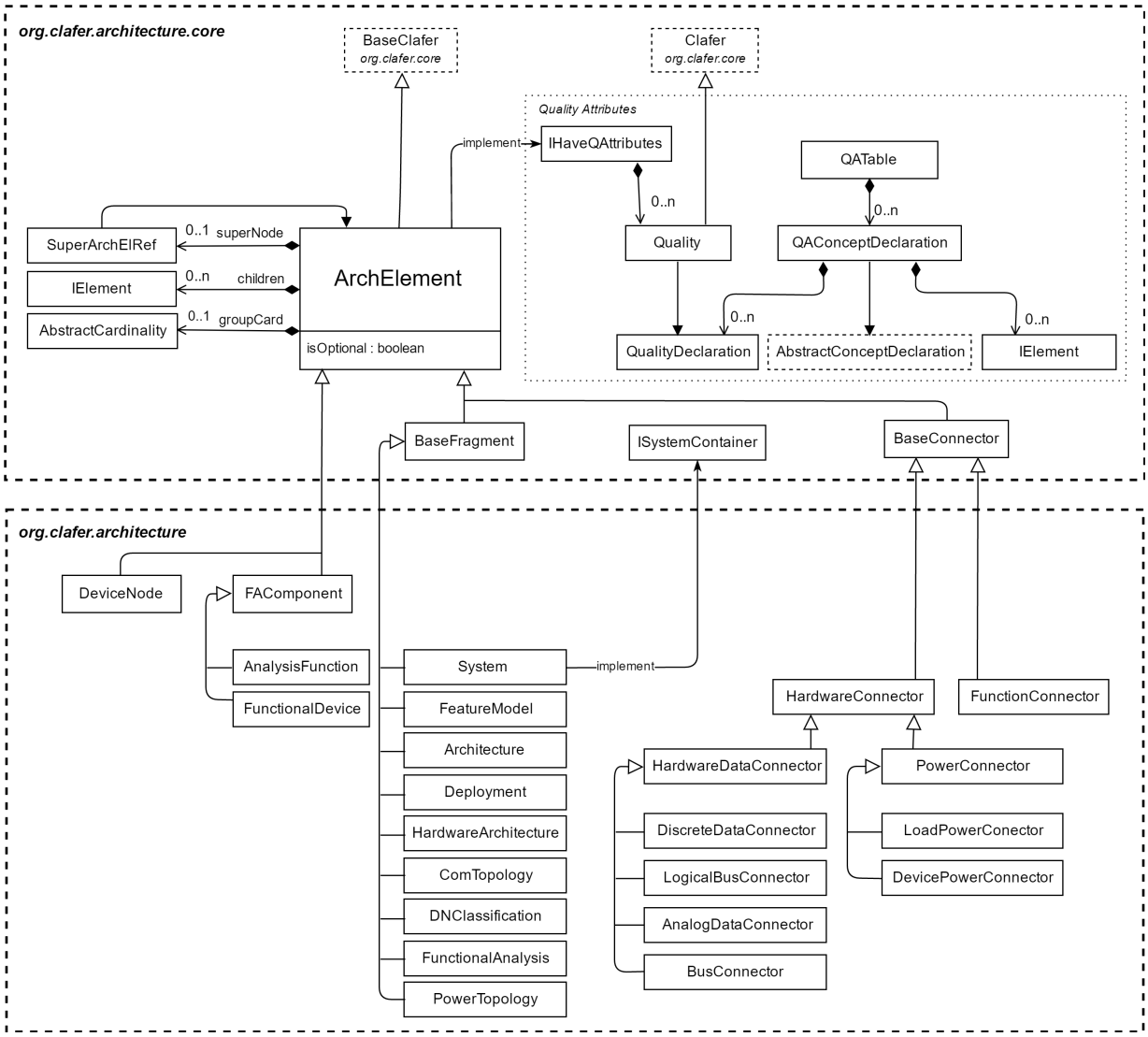


Figure 7.5: The abstract syntax of the Architecture DSL. Dashed boxes represent concepts from other languages

editor components, quality attributes, and various methods needed for building architecture DSLs on top of Clafer. The second language, **org.clafer.architecture**, is essentially an MPS implementation of the reference model discussed in Section 2. This separation allows for extending concepts defined for a particular domain, and also building new architecture DSLs for other domains, without modifying the core functionality. A simple example of such an architecture

extension is given in Section 9.2

7.2.1 Structure

Figure 7.5 shows the structure of the Architecture DSL. The central concept in the core language is `ArchElement`, which represents a generic architecture element. This concept is similar to a clafer in that it may have a super type, multiplicity, group cardinality, and nested `IElements`. However, the reference model imposes some additional restrictions. First, only `ArchElement` can be used as super type. Second, only optional multiplicity is available for architecture elements (a Boolean property `isOptional`). Finally, there is no separation between abstract and concrete elements. By default, any `ArchElement` element is an abstract clafer. In order to create the concrete model, users should wrap their architectures with a top-level `System` container that implements `ISystemContainer`.

The Architecture DSL comes with three types of elements. The first type is just a regular node (such as device, analysis function or feature), which inherits directly from the `ArchElement` concept and defines, for example, a device or function. The second type is represented by a `BaseFragment` concept. A fragment is a container that wraps nodes and connectors within a particular layer. Each layer has its own type of fragments (e.g. `FeatureModel`, `Architecture`, `Deployment`, etc.), and each fragment can contain only a specific subset of concepts. These hierarchy rules are specified by the reference model ([31], Chapter 5), and can be encoded using MPS constraints. For example, Figure 7.6 shows the Constraints aspect for a device node classification (DNC). Each DNC can be defined only inside the hardware architecture, other DNC, or as a top-layer fragment. Moreover, DNC can contain only device nodes, nested DNCs, plain clafers, or constraints.

Finally, `BaseConnector` defines a supertype for all communications between nodes, such as simple connectors, which connect only two nodes, and buses, which connect two or more nodes.

These abstract constructs determine basic functionalities of architecture elements; any subconcepts defined in the **org.clafer.architecture** language inherit their aspects such as editors or constraints. The only mandatory property needed to be specified is an alias, which represents the given concept in the textual notation and the completion menu. However, we can also extend or override other aspects, if needed. For example, each fragment (e.g. `FeatureModel`, `FunctionalAnalysis`, etc.) has its own Constraint aspect, which specifies concepts that can be instantiated within the given fragment.

```
concepts constraints DNClassification {
  can be child
    (childConcept, node, link, parentNode, operationContext)->boolean {
      parentNode.isInstanceOf(HardwareArchitecture) || parentNode.isInstanceOf(DNClassification) ||
          parentNode.isInstanceOf(ClaferModule);
    }

  can be parent
    (childConcept, node, childNode, operationContext, link)->boolean {
      if (link == linkNode/ArchElement : contents/) { return node.childConceptIsClafer(childConcept) ||
          childConcept.isSubConceptOf(DNClassification) || childConcept.isSubConceptOf(DeviceNode); }
      true;
    }
```

Figure 7.6: An Example of the Constraints aspect for Device Node Classification

```
<default> editor for concept FunctionConnector
  node cell layout:
    [- # ArchElement_GroupCard # # ArchElement_Optional # # ArchElement_Definition # from % sourceExpr % to
       % targetExpr % ?^[- ^ deployed to ^(- % deployedTo % /empty cell: no deployment -) -] # ArchElement_Body # -]

  inspected cell layout:
    # ArchInspector #
```

Figure 7.7: `ArchElement` editor declaration

7.2.2 Editors

In MPS, a concept can have more than one editor [40], and a model can be edited using different projections of the given concept. Each of the multiple editors may have a tag called an `editor hint`, and by setting hints in the editor window, users can switch between notations corresponding to these tags. In the Architecture DSL, models can be edited in a textual notation or as diagrams, as discussed in Section 6. Figure 7.7 shows an example of an MPS editor declaration for a connector's textual notation. The syntax consists of reusable "snippets" called *editor components* (surrounded by #). Some examples of Architecture DSL editor components are shown in Figure 7.8. These components define basic syntax elements for architecture constructs and can be used in custom editors. For example, Figure 7.9 shows an editor for `DeviceNode`, where we added an additional cell representing a list of possible types of the given device node.

In addition to a textual notation, we also defined a graphical projection to represent architectures as diagrams (discussed in Section 6.6). The graphical projection is based on the mbeddr diagram language, which provides additional editor cells for diagrams creation. Figure 7.10 shows an example of the `ArchElement` graphical editor. Here, in the top left corner, we specify a hint (`architectureDiagrams`) this editor is assigned to. By adding or removing this hint via the MPS context menu, users can render their models either textually or graphically.

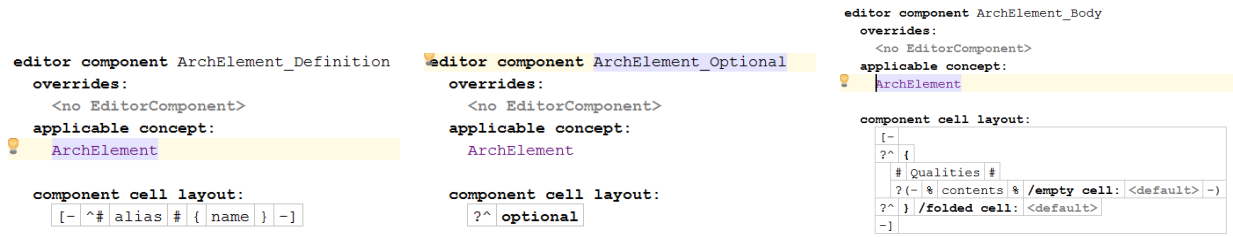


Figure 7.8: Examples of MPS editor components

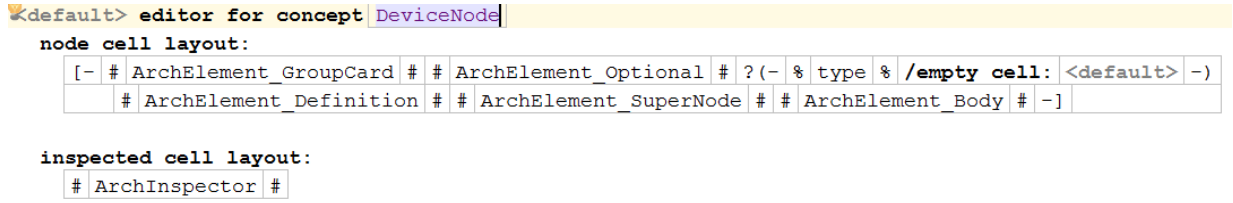


Figure 7.9: `DeviceNode` editor declaration

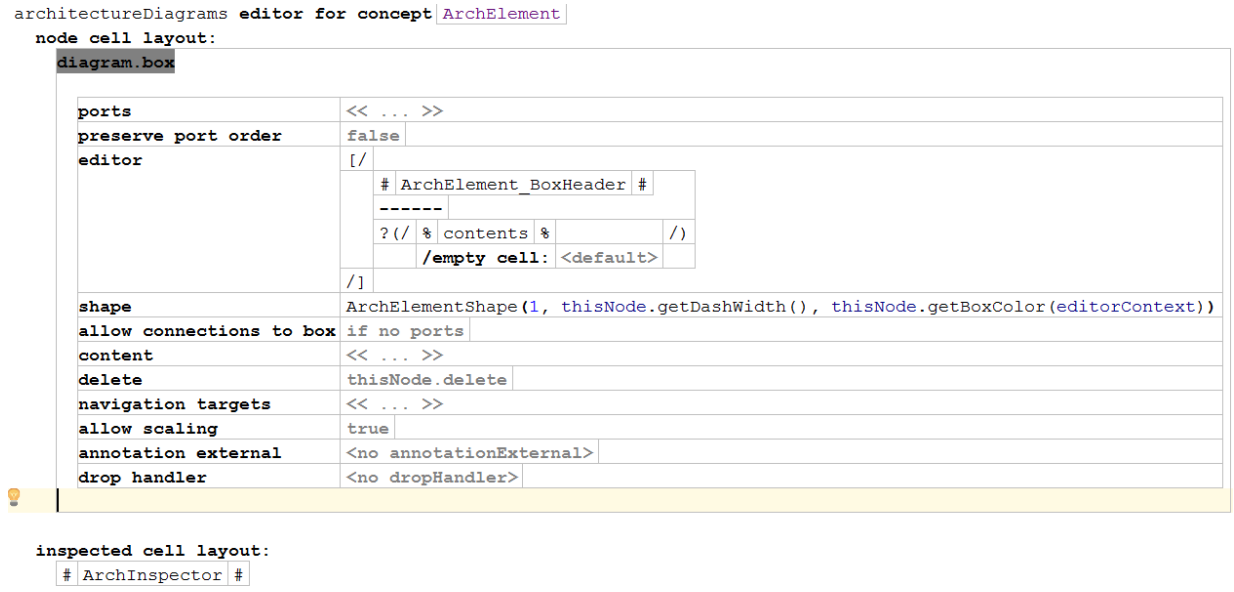


Figure 7.10: `ArchElement` graphical projection

7.2.3 Generators

The Architecture DSL constructs are transformed into plain clafers using model-to-model transformations. The generation process has been discussed in Section 6.7. In this section, we provide additional implementation details about model-to-model transformations.

MPS generators consist of *mapping configurations* and *generator templates*. The mapping configuration controls the generation process and contains generator rules, pre/post-processing scripts, generator parameters, etc. Generator rules are used to assign templates to language elements. For the Architecture DSL, we use reduction rules and weaving rules. The *reduction rule* is an in-place transformation, which replaces the input node with the rule's result. The *weaving rule* is used to insert additional content into the output model at a specified location. Additionally, we use pre-processing scripts to import the reference model to the output model. The reference model is just a predefined Clafer module, which contains abstract clafers (the full Clafer encoding for the reference model can be found in Appendix A).

Figure 7.11 shows the `reduce_ArchElement` template, which generates a clafer body from an architecture element. The part enclosed in `<TF .. TF>` is called the *template fragment*. This code replaces the `ArchElement` during the transformation. Any code outside template fragments is used as a context and does not participate in the generation process. The template fragment includes syntax components that are common for all elements (such as name, multiplicity, group cardinality, super clafer, etc). Next, using weaving rules, we add additional concept-specific content to the output clafer. For example, Figure 7.12 shows a weaving template for a bus connector. The template creates constraints that specify bus types (LIN, CAN, etc.) and reference targets for endpoints. These constraints are specific to bus connectors and cannot be shared with other elements. Similarly, other architecture elements can be transformed into plain clafers.

```
template   reduce_ArchElement
input      ArchElement

parameters
<< ... >>

content node:
_module_                                                    imports     Reference_Model
model   org.clafer.architecture.core.generator.template.main

_superClafer_
  _qaAttribute_ -> integer

<TF ArchElement [ $MAP_SRC$[ $COPY_SRC$[xor] $[_claferName_] $IF$[: ->$[_superClafer_]] $IF$[?]
                    $IF$[$LOOP$[this.->$[_qaAttribute_] = $MAP_SRC$[$COPY_SRC$[10]]]]
                    $LOOP$[$COPY_SRC$[_Content_ ]]
                ] ] TF>
```

Figure 7.11: The template for creating a clafer from an `ArchElement`. Transformation macros (e.g. `$IF$`, `$COPY_SRC$`, etc) replace dummy content with the code generated by the transformation based on the input node. Each macro has a number of properties and functions for computing the node that replaces the dummy code

```
_busConnector_ : BusConnector
  <TF [ $IF$[this.endpoint in ($INSERT$[_deviceNode_])] ] TF>
  <TF [ $IF$[$INSERT$[this.type]] ] TF>
```

Figure 7.12: The weaving template for a bus connector.

Chapter 8

Evaluation

The main goal of our work is to make the modeling and reasoning power of Clafer accessible to practitioners. To evaluate and improve ClaferMPS, we performed the following exploratory case study. The objectives of the evaluation are to O1) obtain feedback on the usability of the DSL and tool support, O2) demonstrate expressiveness of the DSL with respect to the case studies in the automotive body domain, and O3) demonstrate support for modeling and analysis tasks, such as modular validation. First, we take two existing automotive system architectures, power window and door locks, which were previously modeled independently in plain Clafer [30].

The models for power window and door locks contain approximately 600 and 900 lines of Clafer and they encode 203,753,368 and 2,028 variants, respectively. Table 8.1 shows the number of reference model elements found in each of the models along with the number of elements which have *presence variability*, meaning the element could be present or not, indicated in parenthesis.

Next, we asked the modeler (to whom we refer to as "the evaluator"), who previously created both models in plain Clafer, to recreate them in ClaferMPS and record his experience; the raw and detailed notes (40 pages) are available for the record [21]. The evaluator first modeled a single-door power window system, then he generalized it to a two-door system, and then he modeled the door-locks system. Finally, we discussed the notes with the evaluator, analyzed them, and extracted the main observations, which we present here. The evaluator raised issues, reported bugs, made observations, and provided requirements. Some of the bugs and requirements were subsequently implemented in an iterative approach.

Case study completeness Overall, the evaluator was able to model both case studies in ClaferMPS completely, generate equivalent but slightly different plain Clafer model when compared to the original model, and perform the same kinds of analyses using the Clafer toolchain as before [27, 30].

Table 8.1: Number of reference model elements and deployment configurations for both Case Studies. The number in () is the number of elements having presence variability.

	Two Door Power Window	**Central Door Locks**
Features	6 (4)	7 (6)
Analysis Functions	6 (2)	3 (2)
Functional Devices	9 (2)	33 (15)
Deployment Configurations	4096	96
Function Connectors	7 (4)	33 (18)
Device Nodes	10 (3)	21 (14)
Discrete/Analog Connectors	18 (18)	34 (30)
Bus Connectors	2 (1)	2 (1)

Graphical projection The evaluator frequently used the graphical projection to validate connections and he actually discovered wrong connections once. The evaluator found the graph view to be beneficial whenever references are used. Also, the graphical projection was useful for showing containment and ownership. However, the current graphical projection has a few shortcomings: the automatic layout sometimes requires manual rearrangement, and the evaluator could not view only a few selected elements because the projection always displays the entire module. The evaluator provided many observations about the advantages as well as suggestions for improving the graphical projection, including the ability to visualize a selected subset of elements.

In MPS, the evaluator divided the original single-file plain Clafer models into many modules, which was essential when working with such large models. Additionally, smaller modules make the graphical projection more useful and usable. The built-in "jump to definition" mechanism of MPS supports navigation across the modules.

Modeling, debugging, verification, and validation workflow In ClaferMPS, the evaluator relied more on the Architecture DSL to create a more correct-by-construction model because the DSL enforces the proper structure and checks for typical errors during editing; this made the creation of the model faster in ClaferMPS. However, through the use of the Architecture DSL, the evaluator still created an invalid model, initially, by forgetting to assign some quality attributes and setting references to invalid targets. ClaferMPS helped with debugging, and finding such mistakes, because the evaluator no longer had to manually comment out fragments of a large model as ClaferMPS automatically generates code for every module and its imports, with or without the

quality attributes. This allowed for testing each layer in isolation as well as testing the module logic while omitting quality attributes. In some situations, however, the evaluator still had to comment out the unused fragments of the reference model. For example, in order to test the functional architecture layer in isolation, the evaluator had to comment out the `deployedTo` reference, which induces a dependency on hardware architecture. As a result, we have implemented the separation of the deployment from the other layers and weaving of the deployment when needed during code generation; it has reduced the need for commenting out as above but can be improved further in the future to not require commenting out any portions of the model or reference model. Finally, the evaluator set up partial test modules which contain only a partial system and a subset of layers, which allowed testing the individual layers in isolation. These partial test modules allowed for testing and verifying logic associated with a specific layer of the system.

Autocomplete The evaluator ranked autocomplete as the top feature of ClaferMPS because it prevents naming mistakes and helps in correctly selecting nested elements based on their type and the rules of the reference model. Although autocomplete could also be provided for plain Clafer, it would not be able to interpret nesting constraints.

Inconsistencies between the reference model used in both case studies and reference model evolution Chronologically, ClaferMPS was developed after the first version of the power window case study and the Architecture DSL was based on the reference model from that case study. The door locks case study was developed later and subsequently the power window case study was revised. In our evaluation, we not only observed that the reference models between the two case studies were slightly different, but also that the Architecture DSL was initially outdated. Eventually, we made all three reference models consistent.

This demonstrates the typical organizational problem which occurs when people apply a supposedly common reference model but are free to adjust it slightly in every project: the organization cannot easily enforce the consistent application of the reference model. By encoding the reference model in the DSL, providing limited extensibility, and enforcing domain-specific rules, the DSL ensures the consistent application of the reference model.

On the other hand, having a DSL creates the typical "schema migration" problem when a reference model evolves and the user models must be co-evolved consistently with the DSL. We observed this when we updated the DSL to be consistent with the reference model used in the power window case study. The architectural models became broken and the evaluator had to manually redo these broken parts. In practice, this problem is usually mitigated by versioning the reference model and providing migration scripts.

Required knowledge of Clafer The evaluator stated that using the Architecture DSL requires basic understanding of object-orientation and navigation between objects by following the

references. Building and using advanced models, such as the ones in our case studies, requires the ability to write propositional logic constraints and navigating among objects. Familiarity with constraint languages such as OCL, Alloy, or Clafer is very helpful to be able to create non-trivial architectural models. While a user could model an E/E architecture in ClaferMPS without a good understanding of Clafer, knowledge of Clafer is needed for debugging the models or creating ones with interesting variants.

Threats to Validity

A threat to the internal validity of our exploratory evaluation is that some of the development of ClaferMPS was performed in response to the evaluator's bug reports, issues, and requirements. This has not introduced any bias since the design of the DSL was originally based on the evaluator's case studies and the iterative process allowed completing the evaluation and ensuring that the DSL actually covers the entire scope of the case studies.

A threat to the external validity of our evaluation is that it was performed by a single person, who is an expert Clafer user, and therefore the observations cannot be generalized. It is possible that non-expert users of Clafer who are familiar with the reference model would not be capable of modeling the two case studies in the Architecture DSL. However, the evaluator was a novice user of MPS and his observations are likely to be valid for other users. In the future, we are planning to conduct a more extensive evaluation with many users with diverse backgrounds.

Chapter 9

Architecture DSL Extensions

As has been discussed before, each DSL is designed for a concrete scope, which makes domain models more concise and easier to understand for domain specialists. On the other hand, the strength of DSLs is also their weakness, because they need adaptations to make them usable in slightly different domains. In case when a domain is beyond the scope of a DSL, it becomes very hard or even impossible to express domain models in terms of the given DSL. In this case, the DSL needs to be extended or composed with other DSLs to cover the new domain. Thanks to the modular nature of MPS languages, we can decompose our DSL into a set of smaller languages, as has been demonstrated in Section 7. This decomposition allows for modifying only the relevant part of the syntax and seamlessly integrating new languages into the system. To demonstrate ClaferMPS's extensibility we consider another case study, which is slightly different from the reference model. We also show how the DSL can be adjusted to the new scope by extending existing languages and integrating new ones.

9.1 Autonomoose Project

The Autonomoose Project[1] is dedicated to implementing autonomous driver software from scratch. The main goal of this project is to create a platform to enable research on topics related with autonomous driving in Canada. The autonomous driver systems includes complex hardware (such as radar, lidar, gps, cameras, vehicle state sensors, etc.) and software (perception, mission execution, etc.) architectures. These architectures are built using the Robotic Operational System

[1] http://www.autonomoose.net/

(ROS), which is a framework for writing robot applications (discussed in Section 9.2). In this section, we discuss applicability of ClaferMPS to supporting modeling of such types of architectures.

9.1.1 Challenges

The Autonomoose Project faces a number of challenges. First, there are no prescriptive modeling tools available for the ROS framework. Currently, modelers have to use general-purpose tools (such as Excel spreadsheets, generic drawing applications, etc.) to plan and describe ROS systems. Moreover, since models are informal, there is no way to do the analysis of architectures. Next, since models are not automatically synchronized with implementation, they are often outdated. Finally, there is no convenient way to evolve the model. Usually, models are evolved by copying and modifying. As a result, there is no a single model, and developers have to work with multiple versions of the same system.

9.2 ROS

The Robot Operating System[2] (ROS) is an open-source framework for writing robot software. It provides libraries and tools, such as device drivers, visualizers, hardware abstractions, package managers, and more. The ROS framework aims to simplify the creation of complex robotic software on a wide variety of platforms. A ROS system can be decomposed into a set of independent nodes connected with each other using a publish/subscribe messaging model. In next section, we introduce basic concepts used for building communication graphs in ROS.

9.2.1 Concepts

The ROS implementation includes the following fundamental concepts:

- *Nodes*: Nodes are processes that perform computation. In ROS, a robot control system typically comprises many nodes, and each node performs a specific task, such as controlling the wheel motors, performing path planning, detecting collisions, etc. Nodes must have a unique name in the system to communicate with each other without ambiguity.

[2] http://www.ros.org/

- *Messages*: Nodes communicates with each other using messages. A message is a data structure that is defined by a set of data field descriptions and constant definitions. Each field consists of a type and a name, separated by a space. Fields types can be primitive types (e.g. integer, float, string, boolean, etc.), arrays, names of message descriptions defined on their own, and the special "Header" type, which includes additional metadata. Listing 9.1 shows a message example, which reports the status of an individual component of the robot. ROS uses simple .msg text files to store message definitions. Each message definition has a type defined by its name using ROS naming conventions: package name + / + .msg file name.

- *Topics*: Nodes exchange messages over named buses called topics. Topics use a publish/-subscribe mechanism to transmit messages. A single topic can have multiple concurrent publishers and subscribers, and a single node can publish and/or subscribe to multiple topics. Each topic is strictly typed by a message that can be published to it.

Listing 9.1: ROS Message Example

```
# Possible levels of operations
byte OK = 0
byte WARN = 1
byte ERROR = 2
byte STALE = 3

byte level # level of operation enumerated above
string name # a description of the test/component reporting
string message # a description of the status
string hardware_id # a hardware unique string
KeyValue[] values # an array of values associated with the status
```

9.2.2 ClaferMPS applicability

The ROS ecosystem is big and includes a variety of tools, libraries, and conventions. Obviously, covering all these aspects requires a lot of effort. However, particular tasks are not overly complex and can be improved using existing ClaferMPS facilities. In this thesis, we consider modeling and visualizing of ROS nodes and communication among them.

There are several reasons, why we chose ROS diagrams as a case study for demonstrating ClaferMPS extensibility. First, a ROS communication graph is very similar to the Functional Analysis layer in the Architecture DSL. Essentially, analysis functions and functional devices act as nodes in ROS and describe the same functionality. The only difference between ROS graphs and functional analysis architectures is that the Architecture DSL does not have constructs

that describe the topic-based publish-subscribe model. Therefore, we added a new construct called ROS connector, which represents communications between nodes based on topics. Second, implementation of ROS concepts cannot be done without integrating ROS messages into ClaferMPS. Therefore, we had to implement the ROS messages definition language in MPS (for the remainder of this thesis we refer to it as *the message language*). We then use this language to demonstrate how non-Clafer entities can be seamlessly integrated into ClaferMPS projects. Moreover, MPS helps to significantly improve user editing experience when working with ROS messages by providing various IDE features. In particular, we implemented two extensions of the messages definition language (discussed in Section 9.2.3). The first one is message templates, which improves code reuse when working with messages. The second extension is bandwidth calculation, which helps modelers to perform a simple analysis of communications between nodes.

9.2.3 Overview of the ROS Messages Language

Messages The message language is essentially an MPS implementation of the Message Description Specification[3] used for describing messages. The message language supports all build-in types (e.g. bool, int8, float32, etc.), custom message types (names of Message descriptions defined on their own, such as "geometry_msgs/Vector3"), constants, and arrays (e.g. int8[ ], Point32[20], etc.). The message language also supports all IDE features provided by MPS, such as syntax highlighting, jumping to definition, and code completion. Additionally, message type names are automatically adjusted according to ROS naming conventions[4]. It means that the type name depends on the context and may be relative, if the corresponding message is defined in the same package (e.g. "Vector3"); otherwise it must be absolute (e.g. "geometry_msgs/Vector3"). In addition, the message language distribution includes predefined common messages[5] that are widely used in ROS, including messages for diagnostics, actions, geometry primitives, robot navigations, and sensors.

Topics Figure 9.1 shows an example of ROS topic declarations in the message language. For each topic, the user defines its name, type (a reference to the message that can be published to the given topic), a publishing rate (aka frequency) of the topic, and an optional text description.

Templates Templates allow developers to embed fields defined in one message into another one. Figure 9.2 shows an example of templates usage. First, a user creates a snippet, which is essentially a regular message (Fig. 9.2a). Snippets may contains any fields available for

[3] http://wiki.ros.org/msg
[4] http://wiki.ros.org/Names
[5] http://wiki.ros.org/common_msgs

Topics

Name	Type	Frequency	Description
/route_map	/anm_msgs/VehicleState	1	<no desctiption>
/vehicle_state	/anm_msgs/VehicleState	2	2D Pose: x, y, theta
/global_path	/nav_msgs/Path	4	Discrete global path with velocity profile included
/map	/nav_msgs/OccupancyGrid	10	occ grid, cost maps, etc
/local_path	/nav_msgs/Path	3	Custom message that contains a descritized path of points along with a descritized velocity profile
/emergency_stop	float32	1	Single instantaneous turning radius, the presence of the message indicates full brake
/control_commands	/anm_msgs/ControlCommands	15	Custom ROS message that clones polysync message
/lane_markings	/anm_msgs/LaneMarkings	1	Whatever is suitable for the subscriber

Figure 9.1: An Example of ROS topic declarations

message definitions including other templates. Next, the user instantiates this snippet in any other message by choosing a "template" item in the code completion menu (Fig. 9.2b). The template instance (a piece of code surrounded by square brackets) contains all fields from the snippet message. For each field in the template instance, the user can add custom values (e.g. `x`, `cell_width`, `cell_height`). Otherwise, default values will be used, if available (e.g. `y`, `z`).

Bandwidth Analysis As has been mentioned before, the reference model layers can be cross-cut by one or more perspectives including variability and quality attributes. When changing a domain, we might also need to consider new perspectives to cover a new scope. One such perspective for ROS architectures is topic bandwidth. The bandwidth of a topic can be calculated as a size of the message published to the topic multiplied by the topic's publishing rate. The message size is calculated as a sum of sizes of the message's fields (e.g. int8 has size 1 byte, float32[5] has size 20 bytes, etc.). The message automatically performs these calculations and show the result to users (Figure 9.2). This helps guide users in identifying and optimizing high bandwidth messages topics (e.g. using shared memory instead of TCP). Thus, by adding the new perspective, we adapted the Architecture DSL to the ROS framework and made architecture diagrams closer to actual ROS connections graphs.

```
M1
Size: 32 bytes
model   ca.uwaterloo.gsd.ros.example

float64 x = 0
float64 y = 0
float64 z = 0

float32 cell_width
float32 cell_height
/geometry_msgs/Point[] cells
```

(a) Template Message

```
M2
Size: 84 bytes
model   ca.uwaterloo.gsd.ros.example

Header header
/geometry_msgs/Vector3 vector

/example/M1
float64 x = 20
float64 y = 0
float64 z = 0
float32 cell_width = 10
float32 cell_height = 10
/geometry_msgs/Point[] cells
```

(b) Template instance

Figure 9.2: ROS template example.

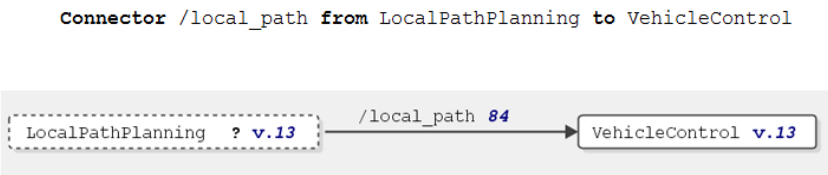


Figure 9.3: An Example of ROS Connector textual and graphical notations

ROS connectors The ROS connector is a simple construct representing topic-based communications among nodes in ROS graphs. It extends the `FunctionConnector` concept and depends on topics defined in the model. Figure 9.3 shows an example of the ROS connector textual and graphical notations. In order to instantiate the ROS connector, users specify a topic (`local_path`), a publisher (`LocalPathPlanning`), and a subscriber (`VehicleControl`). Additionally, the ROS connector shows the calculated bandwidth of the topic, as discussed above.

Thus, integrating the message language and adding ROS connectors to the Architecture DSL allows developers to define reusable message definitions, model and visualize ROS architectures, and estimate bandwidth of communications between nodes.

9.3 Milestones

Typically, a design of complex software/hardware architectures requires many iterations, so a model is constantly changing and developing due to feature additions, bug fixing, etc. In some cases, modelers might need to have access to previous "snapshots" of a system's model to compare them with each other and analyze the impact of changes on the system. One way to upgrade a model to the next version is to copy and modify it as a completely separate model. Obviously, it is inconvenient and hard to maintain, because instead of editing a single model, users have to work with a set of (partially different) designs of the same system.

In this section, we introduce another approach of model evolution based on versions. We extended the Architecture DSL by adding a new mechanism called *milestones*. A milestone acts as a filter that queries a model for a specific subset of nodes which satisfy some conditions. Typically, milestones are used as follows:

- For an arbitrary architecture element, a user can assign an integer attribute, which represents some version of the element (Figure 9.4a).
- Next, the user creates milestone configurations. For this purpose, we provide a table shown in Figure 9.4b. The milestone is a named boolean expression, which is used to query a model for one or more node versions. Additionally, the user may define a color used for highlighting nodes that satisfy the given milestone.
- Finally, the user activates milestones using a controller shown at the top left corner of Figure 9.5. The controller can contain any number of milestones defined in the milestone table. By toggling checkboxes in the controller, the user can show/hide different subsets of nodes and consequently obtain different variants of the model.

In MPS, the projection system always goes in one direction, from AST to concrete syntax, and never vise versa. This has two important consequences. First, the projection may be partial. This means that the concrete syntax shows only some parts of the model. Second, the projection system can also project additional concrete syntax that is not part of the original language. Since the concrete syntax is used only for presentation, such additional syntax does not confuse the system.

In the milestone extension, we use both of these approaches. First, we add versions and controllers using an MPS mechanism called *annotations*. An annotation is a metadata that can be attached to arbitrary node and can be shown with the concrete system of the attached node. Second, by switching active milestones in the controller, we can change the portion of the overall model that is presented to users. An advantage of this approach is that it does not modify

v.13 **Analysis Function** CommandChecking
v.13 **software Analysis Function** PerceptionServer
v.14 **software Analysis Function** LaneMarkingDetection

(a) Version annotations

Name	Condition	Color
Dev1	(>= v.11) & (<= v.13)	
Dev2	(= v.14)	

(b) Milestone table

Figure 9.4: Example using milestones.

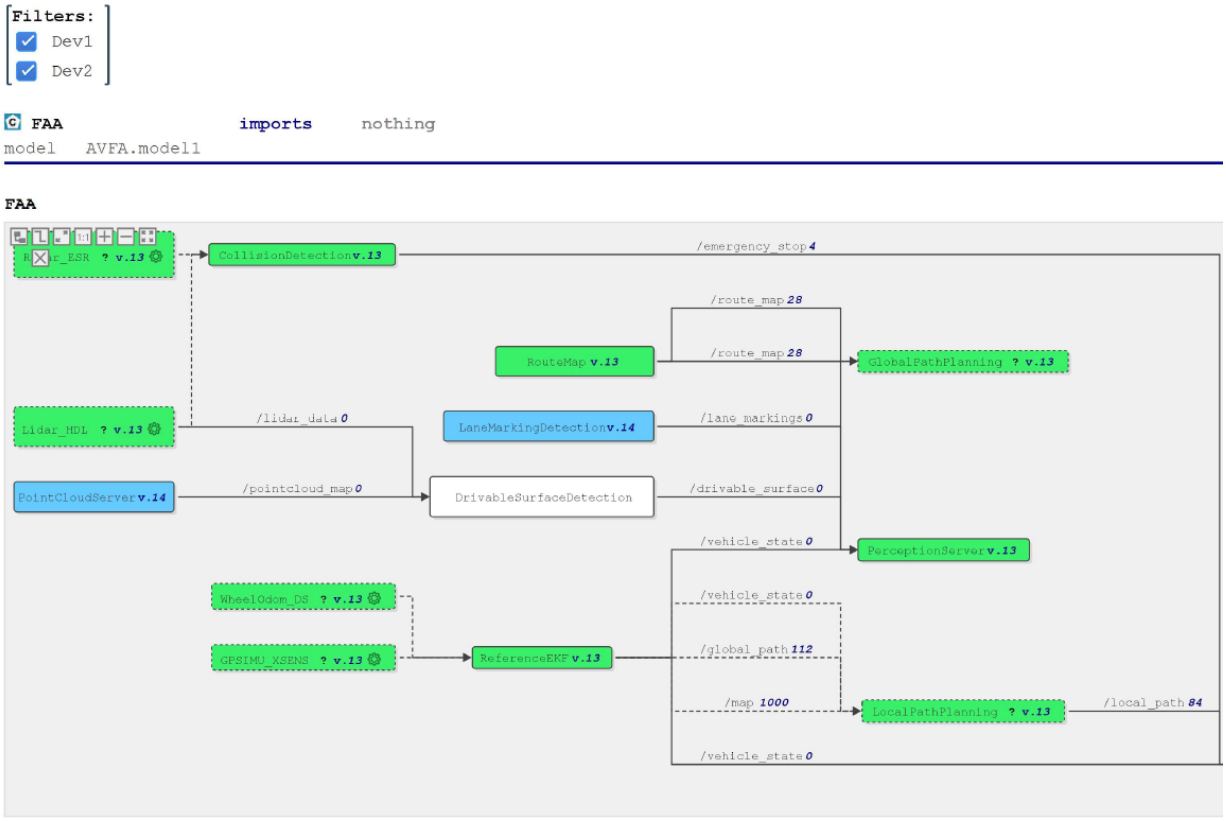


Figure 9.5: Example using a milestone controller

the AST. As a result, it is possible to use different milestone configurations in multiple editor windows. This allows users to compare different configurations of the same system side-by-side. For example, Figure 9.6 shows an MPS tool with two editor tabs. Each editor shows the same model, but with different sets of active milestones.

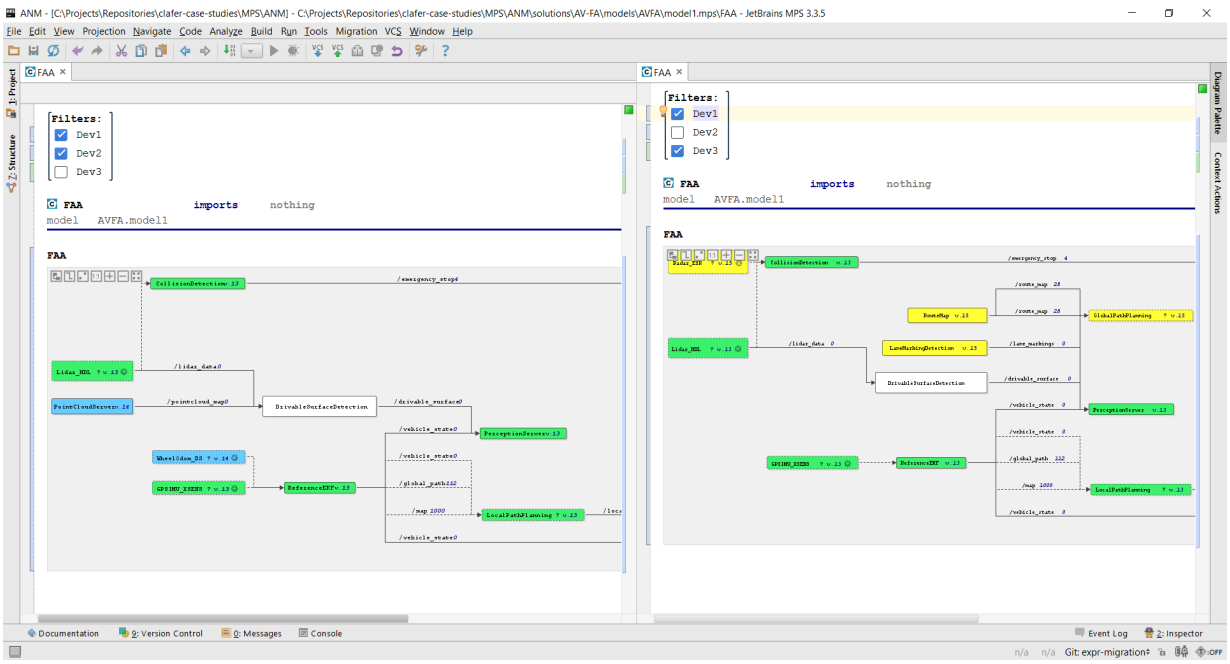


Figure 9.6: Example using two editors with different milestone configurations

Chapter 10

Related Work

Automotive architecture synthesis Aleti et al. surveyed over 180 works concerning architecture optimization in the domains of information systems and embedded systems [1]. The surveyed methods, along with other related works [44, 16, 26, 4, 18, 35], considered different design decisions or degrees of freedom (i.e., variability points) for hardware selection, deployment of software to hardware, task scheduling, redundancy allocation, communication topology design, hardware component placement, and wireharness sizing and routing. They also considered different design constraints such as memory capacity, functional dependencies, co-location restrictions, among many quality constraints such as mass, cost, and reliability. Lastly, these optimization works considered a number of different objectives such as performance, reliability, cost, mass, and energy consumption.

The majority of these works, however, only considered a handful of design decisions, constraints, and objectives; where as in our work, we can consider a design decision for each reference model component. Additionally, in our MPS models, we were able to reason about mass, parts cost, warranty parts cost, and latency as in [27, 30] since ClaferMPS is an extension of Clafer. In this work, we also consider decisions made about the features and their impact on the other layers of the system (functional and hardware) in E/E architectures which was first introduced by Murashkin [27].

Additionally, the works surveyed in [1] only consider the equivalent of the functional analysis architecture, device node classification, and the network buses in the communication topology. In other works that consider both the functional and hardware layers of the architecture as well as a graphical projection of the architecture, such as AF3 [3], OSATE [34], and PreeVision [32], they do not allow for expressing variability about almost any component in the model along with a supporting reasoner, as we do.

Language Workbenches The term language workbenches was coined by Martin Fowler in 2004 [15] to describe a class of tools for efficiently developing domain-specific languages. However, the class of tools has a longer history; early examples of language workbenches include the Synthesizer Generator [29] and the Meta Environment [23]. The latter is an editor for languages defined via SDF, a general parsing framework. More recent examples of languages are Rascal [24], Spoofax [20], Xtext[1] and MPS; for a systematic comparison see [12]. What sets MPS apart from most other language workbenches is its use of a projectional editor, which allows for the mix of graphical and textual notations and the syntactic extensibility mentioned earlier. The only other industry-strength projectional editor is the Intentional Domain Workbench [33]. In terms of syntactic flexibility, it has demonstrated diagrams and tables mixed with text. In terms of language extension and extension composition the available information is limited, since it is a commercial product. The other projectional language workbenches mentioned in [12] are not ready outside of small-scale experimental scenarios.

[1] http://eclipse.org/Xtext

Chapter 11

Conclusion

In this thesis, we presented the ClaferMPS system, a DSL-based approach for modeling automotive E/E architectures. We demonstrated how domain-specific extensions can be used to address challenges in modeling E/E architectures using Clafer.

We presented the design and implementation of an Architecture DSL for modeling automotive E/E architectures. The goal of the DSL is to make the reasoning power of Clafer accessible to practitioners by guiding them in the correct application of the reference model, minimizing the need for writing constraints, and automatically generating plain Clafer files that can be used with the existing Clafer toolchain. Additionally, we demonstrated extensibility of our approach by adapting the Architecture DSL to the ROS framework. In particular, we designed and implemented a language for editing ROS message definitions, and extended the existing DSL with additional ROS constructs.

This work opens up new possibilities in the design exploration of automotive architectures. As has been previously demonstrated in plain Clafer [27, 30], architects can now include design decisions and alternatives about any element in their architectural model, automatically synthesize candidate architectures to see the impact of their decisions, enrich the model with quality attributes and multi-objectively optimize the model to find the set of Pareto-optimal candidates and explore the tradeoffs among them. This work is also applicable to modeling automotive product-line architectures and synthesizing concrete architectures for products. We also believe that the ClaferMPS approach can be successfully used outside the E/E architecture domain.

In the future, we would like to address the remaining limitations and requirements uncovered by our evaluation, such as reference model slicing to eliminate the need for commenting out unused fragments of the reference model, separation of variability similar to the quality attributes and deployment, and integration of the instance generator and support for working with the

candidate architectures to provide a smooth workflow within MPS. We would also like to perform experimental evaluation with external users to assess the practicality of the approach and the required expertise in Clafer.

References

[1] 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, May 2013.

[2] Michał Antkiewicz, Kacper Bąk, Alexandr Murashkin, Rafael Olaechea, Jia Liang, and Krzysztof Czarnecki. Clafer tools for product line engineering. In *SPLC*, 2013.

[3] Vincent Aravantinos, Sebastian Voss, Sabine Teufl, Florian Hölzl, and Bernhard Schätz. AutoFOCUS 3: Tooling concepts for seamless, model-based development of embedded systems. *ACES-MB'15*, page 19, 2015.

[4] 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.

[5] Kacper Bąk, Zinovy Diskin, Michał Antkiewicz, Krzysztof Czarnecki, and Andrzej Wąsowski. Clafer: unifying class and feature modeling. *Soft. & Sys. Modeling*, pages 1–35, 2014.

[6] Hans Blom, Henrik Lönn, Frank Hagl, Yiannis Papadopoulos, Mark-Oliver Reiser, Carl-Johan Sjöstedt, De-Jiu Chen, Fulvio Tagliabò, Sandra Torchiaro, Sara Tucci, et al. East-adl: An architecture description language for. *Embedded Computing Systems: Applications, Optimization, and Advanced Design: Applications, Optimization, and Advanced Design*, page 456, 2013.

[7] Manfred Broy, Sascha Kirstan, Helmut Krcmar, Bernhard Schätz, and Jens Zimmermann. What is the benefit of a model-based design of embedded software systems in the car industry? *Software Design and Development: Concepts, Methodologies, Tools, and Applications: Concepts, Methodologies, Tools, and Applications*, page 310, 2013.

[8] DeJiu Chen, Lei Feng, TahirNaseer Qureshi, Henrik Lönn, and Frank Hagl. An architectural approach to the analysis, verification and validation of software intensive embedded systems. *Computing*, 95(8):649–688, 2013.

[9] Philippe Cuenot, DeJiu Chen, Sébastien Gérard, Henrik Lönn, Mark-Oliver Reiser, David Servat, Ramin Tavakoli Kolagari, Martin Törngren, and Matthias Weber. Towards improving dependability of automotive systems by using the east-adl architecture description language. In *Architecting dependable systems IV*, pages 39–65. Springer, 2007.

[10] Philippe Cuenot, DeJiu Chen, Sebastien Gerard, Henrik Lonn, Mark-Oliver Reiser, David Servat, Carl-Johan Sjostedt, Ramin Tavakoli Kolagari, Martin Torngren, and Matthias Weber. Managing complexity of automotive electronics using the east-adl. In *12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)*, pages 353–358. IEEE, 2007.

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

[12] Sebastian Erdweg, Tijs Storm, Markus Völter, et al. The state of the art in language workbenches. In Martin Erwig, RichardF. Paige, and Eric Wyk, editors, *Software Language Engineering*, volume 8225 of *LNCS*. Springer, 2013.

[13] Martin Erwig and Eric Walkingshaw. Semantics first! In *SLE*, pages 243–262, 2012.

[14] Rick Flores, Charles Krueger, and Paul Clements. Mega-scale product line engineering at general motors. In *Proceedings of the 16th International Software Product Line Conference - Volume 1*, SPLC '12, pages 259–268, New York, NY, USA, 2012. ACM.

[15] Martin Fowler. Language Workbenches: Killer-App for DSLs? *ThoughtWorks, http://www.martinfowler.com/ articles/languageWorkbench.html*, 2005.

[16] M. Glaß, M. Lukasiewycz, R. Wanka, C. Haubelt, and J. Teich. Multi-objective routing and topology optimization in networked embedded systems. In *SAMOS*, pages 74–81, 2008.

[17] Iris Groher and Markus Voelter. Aspect-oriented model-driven software product line engineering. In *Transactions on aspect-oriented software development VI*, pages 111–152. Springer, 2009.

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

[19] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, DTIC Document, 1990.

[20] Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench: Rules for declarative specification of languages and IDEs. In *OOPSLA*. ACM, 2010.

[21] Eldar Khalilov and Jordan Ross. Supplemental material for the paper 'modeling and optimizing automotive electric/electronic (E/E) architectures: towards making Clafer accessible to practitioners', May 2016. http://gsd.uwaterloo.ca/node/668.

[22] Eldar Khalilov, Markus Voelter, and Michal Antkiewicz. ClaferMPS source code repository. https://github.com/gsdlab/claferMPS/, Accessed: 2016-05-02.

[23] Paul Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering Methodology*, 2(2), 1993.

[24] Paul Klint, Tijs van der Storm, and Jurgen Vinju. EASY meta-programming with Rascal. In *GTTSE III*, volume 6491 of *LNCS*. Springer, 2011.

[25] Peter Liggesmeyer and Mario Trapp. Trends in embedded software engineering. *IEEE software*, 26(3):19–25, 2009.

[26] 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 Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 34(11):1730–1741, 2015.

[27] 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.

[28] Alexandr Murashkin, Michał Antkiewicz, Derek Rayside, and Krzysztof Czarnecki. Visualization and exploration of optimal variants in product line engineering. In *SPLC*, 2013.

[29] Thomas W. Reps and Tim Teitelbaum. The synthesizer generator. In *First ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*. ACM, 1984.

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

[31] Jordan Ross. Synthesis and exploration of multi-level, multi-perspective architectures of automotive embedded system. 2016.

[32] Jörg Schäuffele. E/E architectural design and optimization using PREEvision. Technical report, SAE Technical Paper, 2016.

[33] Charles Simonyi, Magnus Christerson, and Shane Clifford. Intentional Software. *SIGPLAN Not.*, 41(10), October 2006.

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

[35] 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.

[36] Markus Voelter. Language and IDE Development, Modularization and Composition with MPS. In *GTTSE 2011*, LNCS. Springer, 2011.

[37] Markus Voelter. *Generic tools, specific languages*. TU Delft, Delft University of Technology, 2014.

[38] Markus Voelter. Preliminary experience of using mbeddr for developing embedded software. In *Tagungsband des Dagstuhl-Workshops*, page 73, 2014.

[39] Markus Voelter, Arie van Deursen, Bernd Kolb, and Stephan Eberle. *Using C language extensions for developing embedded software: a case study*, volume 50. ACM, 2015.

[40] Markus Voelter and Sascha Lisson. Supporting Diverse Notations in MPS' Projectional Editor. *GEMOC Workshop*.

[41] Markus Voelter, Daniel Ratiu, Bernd Kolb, and Bernhard Schaetz. mbeddr: Instantiating a language workbench in the embedded software domain. *ASE*, 20(3):339–390, 2013.

[42] Markus Voelter, Daniel Ratiu, Bernhard Schaetz, and Bernd Kolb. mbeddr: an extensible c-based programming language and ide for embedded systems. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, pages 121–140. ACM, 2012.

[43] Markus Voelter, Jos Warmer, and Bernd Kolb. Projecting a modular future. *Software, IEEE*, 32(5):46–52, 2015.

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

APPENDICES

Appendix A

Source Code for the Reference Model

```
abstract System
abstract FeatureModel
abstract Architecture
abstract FunctionalAnalysis
abstract HardwareArchitecture
abstract DeviceNodeClassification
abstract CommTopology
abstract PowerTopology
abstract Deployment

// Some generic "types" of Clafer's. Some types don't have properties but
// are rather used for readability for a user
abstract Feature

abstract FunctionalAnalysisComponent
   deployedTo -> DeviceNode
   xor implementation
      hardware
         [latency = baseLatency]
         [deployedTo.type in (EEDeviceNode, SmartDeviceNode)]
      software
         [latency = baseLatency*deployedTo.speedFactor]
         [deployedTo.type in SmartDeviceNode]
   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 = SmartDeviceNode | EEDeviceNode | PowerDeviceNode

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 (PowerDeviceNode, EEDeviceNode)) => (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 = SmartDeviceNode]
   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 = SmartDeviceNode]
   bus -> BusConnector 2
   gatewayTransferTimePerSize -> integer // [us/byte]
   [transferTimePerSize = gatewayTransferTimePerSize + sum(bus.transferTimePerSize)]
   [length = 0]
   [mass = 0]
   [cost = 0]
```