Modeling and Optimizing Automotive Electric/Electronic (E/E) Architectures: Towards Making Clafer Accessible to Practitioners

Eldar Khalilov, Jordan Ross, Michał Antkiewicz,
Markus Völter*, and Krzysztof Czarnecki

University of Waterloo, Canada *Independent/ITEMIS, Germany

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 paper, 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 it. We 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.

Keywords: architecture, modeling, optimization, synthesis, language engineering, domain-specific language, DSL, Clafer, Meta-Programming System, MPS

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 capturing the crosscutting concerns, including variability and quality. We present such a reference model in Section 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 [1], and we introduce architectural modeling using Clafer in Section 3. 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 (Section 4). Our implementation relies on the JetBrains MPS language workbench [2], whereby we implemented Clafer as an MPS language and the Architecture DSL as an extension of Clafer in MPS.

We present the design of the Architecture DSL and how it addresses the challenges of applying plain Clafer to architectural modeling in Section 5. We evaluate our work by using the DSL to model two existing architectures of two automotive subsystems which were previously modeled in plain Clafer [3]. 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. We present the key observations and discussion in Section 6. We briefly summarize the related work in Section 7, and conclude the paper in Section 8.

2 A Reference Model for E/E Architecture Modeling

In this work, we use the reference model illustrated in Fig. 1, which is an adaptation of the EAST-ADL [4] (details in [3]). The model is *multi-layer* and it prescribes dividing the architecture into a feature model, a functional architecture, and a hardware architecture. The feature model contains *user-facing features*, such as *express up* and *pinch protection* in a power window system. These features are then implemented using functions, which are subsequently deployed onto hardware (the block arrow).

The model is also *multi-perspective*: it supports multiple cross-cutting concerns, including variability and quality. Variability crosscuts all layers of the architecture. For example, an optional feature (e.g., *express up*) is implemented by functions and hardware which also have to be optional as they are not needed when the feature is not selected.

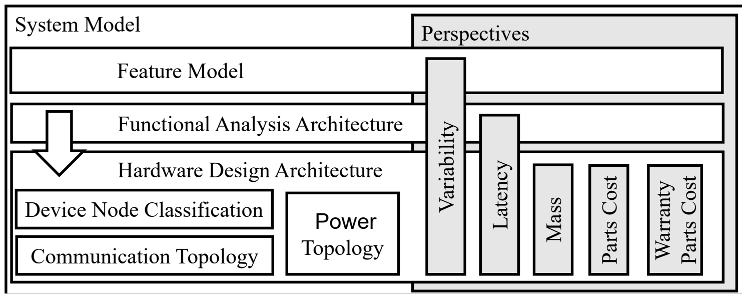


Fig. 1. An automotive E/E system architecture reference model. The block arrow denotes the deployment of the functional analysis architecture to the hardware design architecture.

Furthermore, there may exist alternative ways of realizing the feature as different functions (e.g., different techniques of pinch detection), as well as alternative ways of deploying the functions onto hardware. Similarly, 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 only applies to the hardware design architecture.

3 E/E Architecture Modeling in Clafer

Clafer [1] has been successfully applied in several automotive architecture case studies [5,3]. Here, we first briefly introduce the modeling and reasoning workflow when using Clafer and next, we summarize the main challenges of using plain Clafer.

Clafer is a lightweight, general-purpose, textual, structural modeling language. In Clafer, a model consists of *clafers*[1]. The name "clafer" comes from the words class, feature, and reference because a clafer provides modeling capabilities of all these language constructs. A clafer is like a *class* in that it can have instances, it can contain other clafers which represent attributes, references, and contained classes, and it can inherit the children from other clafers. A clafer is like an *attribute* and a *reference* in that its instances can point to primitive values (e.g., integers) or instances of clafers. A clafer is like a *feature* in that it has multiplicity restricting how many instances of that clafer are allowed per instance of its parent (`1` - exactly one, `?` - at most one, `*` - any number, etc.). A clafer is like a *feature group* in that it has group cardinality restricting how many instances of its children are allowed (`mux` - at most one, `xor` - exactly one, `or` - at least one, etc.). Clafer also provides a powerful constraint language (first-order relational logic) and means of stating multiple optimization objectives (e.g., minimize cost, maximize performance).

Furthermore, Clafer is supported by a set of tools [6], 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 [7] 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 [5,3].

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

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

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.

This causes some challenges for model creators and model readers related to applying and recognizing the modeling idioms needed for architectural modeling. Thus, to make the modeling and reasoning power of Clafer available to practitioners who are not Clafer experts, we implemented an Architecture DSL [8] which encodes the reference model concepts and which guides the users in correctly and consistently applying the reference model rules. We first present the overview of the implementation of the DSL in Section 4 followed by detailed design which addresses the challenges of using plain Clafer in Section 5.

4 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. We implemented the DSL using the Meta Programming System (MPS) [2] language workbench.

Meta Programming System (MPS) As a language workbench [9], 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; no parsing is involved. This allows MPS to support a wide range of notations [10] and various ways of language composition [11]. 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 [12,2].

Components of ClaferMPS Fig. 2 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 [6,7].

On the MPS side (left), we build on top of MPS and use some utilities of mbeddr [12] 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

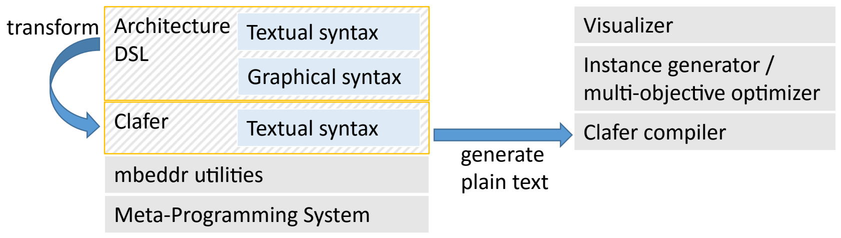


Fig. 2. ClaferMPS (left) and plain Clafer (right) tool stacks

to leverage the existing Clafer toolchain. We refer to both the Clafer implementation in MPS and the Architecture DSL collectively as "ClaferMPS".

5 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.

5.1 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 1.1 shows the feature modeling concepts *feature model* and *feature* encoded as abstract clafers.

Listing 1.1. Feature modeling concepts defined in plain Clafer

```
abstract FeatureModel
abstract Feature
```

Listing 1.2. 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 1.2 (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 [3]. 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 it is specific to the reference model.

In addition to being correctly nested, reference model concepts must also be constrained properly. For example, Listing 1.3 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 1.3. 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 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 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 1.4 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. 3 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.

Listing 1.4. A valid and an invalid function connector model example

```
WinSwitch : FunctionalDevice
WinArbiter : AnalysisFunction
  [latency = 10]
WinControl : AnalysisFunction

// valid connector
winReq : FunctionConnector
  [sender = WinArbiter]
  [receiver = WinControl]

// underconstrained (invalid) connector
localWinReq : FunctionConnector
```

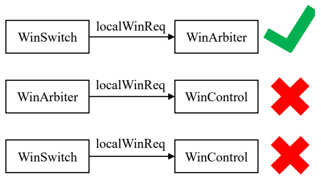
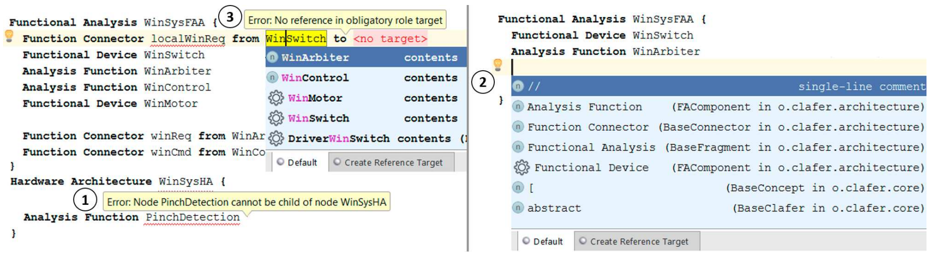


Fig. 3. Instances generated from Listing 1.4

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 4 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. 4. If the user copy/pastes an element into the wrong context, the error will be presented as shown in Fig. 4, ①.


```
Functional Analysis WinSysFAA {
  Function Connector localWinReq from WinSwitch to <no target>
  Functional Device WinSwitch
  Analysis Function WinArbiter
  Analysis Function WinControl
  Functional Device WinMotor

  Function Connector winReq from WinAr
  Function Connector winCmd from WinCo
}
Hardware Architecture WinSysHA {
  Analysis Function PinchDetection
}
```
Error: No reference in obligatory role target

```
WinArbiter          contents
WinControl          contents
WinMotor            contents
WinSwitch           contents
DriverWinSwitch contents (
```

Error: Node PinchDetection cannot be child of node WinSysHA

```
Functional Analysis WinSysFAA {
  Functional Device WinSwitch
  Analysis Function WinArbiter

}
```

```
//                                                    single-line comment
Analysis Function     (FAComponent in o.clafer.architecture)
Function Connector  (BaseConnector in o.clafer.architecture)
Functional Analysis  (BaseFragment in o.clafer.architecture)
Functional Device     (FAComponent in o.clafer.architecture)
[                                (BaseConcept in o.clafer.core)
abstract                          (BaseClafer in o.clafer.core)
```

Fig. 4. ClaferMPS functional analysis example.

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. 4, 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 1.3; 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, ⑦).

5.2 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 1.2 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.

5.3 Quality Attributes

To evaluate quality of the 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 door locks system might.

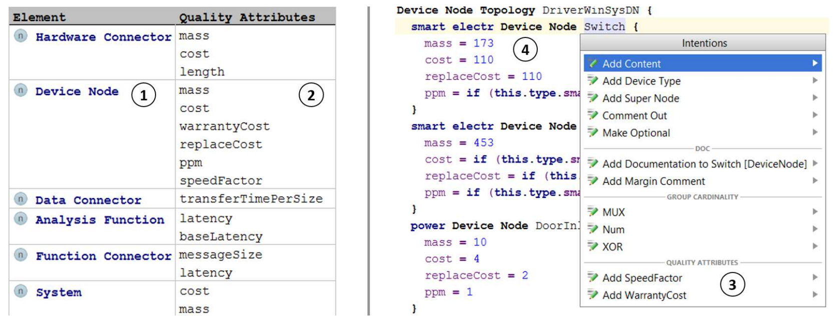
In Clafer, quality attributes can be added to the reference model by nesting a clafer under the component type as shown on line 3 of Listing 1.3. Then, in the definitions of concrete components, the values can be defined using constraints as shown on line 3 of Listing 1.4. The challenge with this approach 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. 5 whereby users define one or more *integer-valued* quality attributes ② for the chosen architectural concepts ①. 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 ③, ④. 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 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 5.7, we describe how the generation process supports this functionality.

5.4 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



Element	Quality Attributes
Hardware Connector	mass cost length
Device Node ①	mass ② cost warrantyCost replaceCost ppm speedFactor
Data Connector	transferTimePerSize
Analysis Function	latency baseLatency
Function Connector	messageSize latency
System	cost mass

```
Device Node Topology DriverWinSysDN {
  smart electr Device Node Switch {
    mass = 173
    cost = 110
    replaceCost = 110
    ppm = if (this.type.sma
  }
  smart electr Device Node
    mass = 453
    cost = if (this.type.sr
    replaceCost = if (this
    ppm = if (this.type.sma
  }
  power Device Node DoorInl
    mass = 10
    cost = 4
    replaceCost = 2
    ppm = 1
  }
```

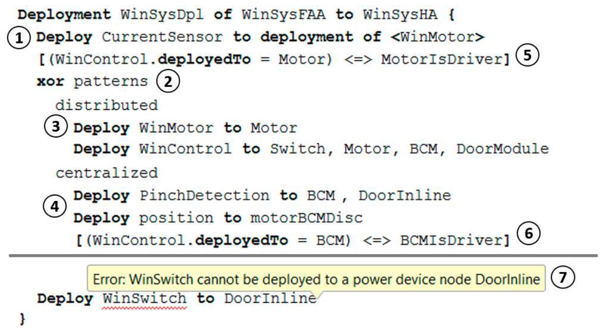
Fig. 5. User-defined quality attribute declarations for the architectural concepts (left). An intention menu for assigning values of quality attributes to model elements (right).

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 occurrence 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 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 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 Clafer 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, 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 for working with the proposed extension before formally implementing it as a DSL in MPS.



```
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
     Deploy PinchDetection to BCM , DoorInline
  ④ 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
}
```

Fig. 6. 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).

5.5 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 5.7.

5.6 Presentation

The Clafer compiler can generate a static graphical representation of a model which shows the inheritance hierarchy and references as shown in Fig. 7. This graphical representation is complementary to the textual syntax which emphasizes clafer nesting; however, it is not suitable for visualizing architectures.

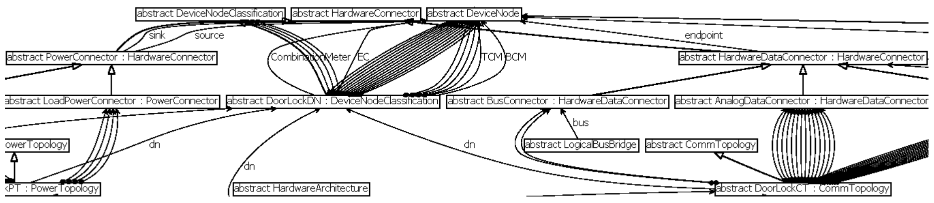

abstract DeviceNodeClassification
abstract HardwareConnector
abstract DeviceNode
sink
source
endpoint
abstract PowerConnector : HardwareConnector
CombinationMeter
EC
TCM
BCM
abstract HardwareDataConnector : HardwareConnector
abstract LoadPowerConnector : PowerConnector
abstract DoorLockDN : DeviceNodeClassification
abstract BusConnector : HardwareDataConnector
abstract AnalogDataConnector : HardwareDataConnector
bus
owerTopology
dn
dn
abstract LogicalBusBridge
abstract CommTopology
dn
PT : PowerTopology
abstract HardwareArchitecture
abstract DoorLockCT : CommTopology

Fig. 7. Snippet of the graph for door locks generated by Clafer compiler

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 connections between different elements to ensure that their model matches what they intended. Figure 8 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*[2]. 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.

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

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

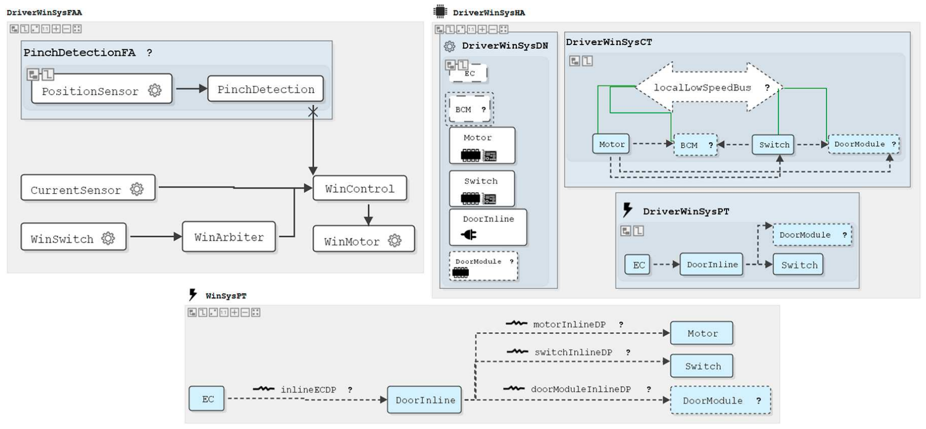


Fig. 8. ClaferMPS Architecture DSL Diagrams

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.

5.7 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 [5]. Next, users perform multi-objective optimization and impact and trade-off analyses [6,3].

Furthermore, the modelers often validate the model logic without the quality attributes, which requires commenting them out (cf. Section 5.3).

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 9 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 1.1 and 1.3, 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

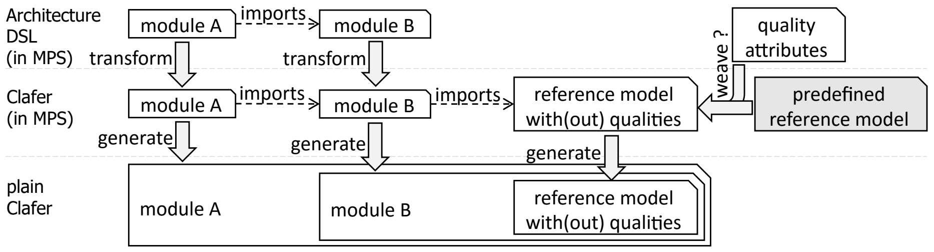


Fig. 9. Plain Clafer generation process

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.

6 Evaluation

The main goal of our work is 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 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 by the second author [3].

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.

Next, we asked the second author (to whom we refer to as "the evaluator") to recreate both models in ClaferMPS and record his experience; the raw and detailed notes (40 pages) are available for the record [13]. 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 completely model both case studies in ClaferMPS, 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 [5,3].

Graphical projection The evaluator frequently used the graphical projection to validate connections and he has actually discovered wrong connections once. In fact, the graph view was 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 observed that not only the reference models between the two case studies were slightly different but also 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.

7 Related Work

Aleti et al. surveyed over 180 works concerning architecture optimization in the domains of information systems and embedded systems [14]. The surveyed methods, along with other related works [15,16,17,18,19,20], 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, 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 [5,3] 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 [5].

Additionally, the works surveyed in [14] 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 [21], OSATE [22], and PreeVision [23], they do not allow for expressing variability about almost any component in the model along with a supporting reasoner, as we do.

8 Conclusion

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. This paper reports on the progress towards that goal.

This work opens up new possibilities in the design exploration of automotive architectures. As has been previously demonstrated in plain Clafer [5,3], 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.

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. Bąk, K., Diskin, Z., Antkiewicz, M., Czarnecki, K., Wąsowski, A.: Clafer: unifying class and feature modeling. Soft. & Sys. Modeling (2014) 1–35
2. Voelter, M., Warmer, J., Kolb, B.: Projecting a modular future. Software, IEEE **32**(5) (2015) 46–52
3. Ross, J.: Case studies on E/E architectures for power window and central door locks systems (May 2016) http://gsd.uwaterloo.ca/node/667.
4. 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.
5. Murashkin, A.: 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.
6. Antkiewicz, M., Bąk, K., Murashkin, A., Olaechea, R., Liang, J., Czarnecki, K.: Clafer tools for product line engineering. In: SPLC. (2013)
7. Murashkin, A., Antkiewicz, M., Rayside, D., Czarnecki, K.: Visualization and exploration of optimal variants in product line engineering. In: SPLC. (2013)
8. Khalilov, E., Voelter, M., Antkiewicz, M.: ClaferMPS source code repository https://github.com/gsdlab/claferMPS/, Accessed: 2016-05-02.
9. Fowler, M.: Language Workbenches: Killer-App for DSLs? ThoughtWorks, http://www.martinfowler.com/ articles/languageWorkbench.html (2005)
10. Voelter, M., Lisson, S.: Supporting Diverse Notations in MPS' Projectional Editor. GEMOC Workshop
11. Voelter, M.: Language and IDE Development, Modularization and Composition with MPS. In: GTTSE 2011. LNCS. Springer (2011)
12. Voelter, M., Ratiu, D., Kolb, B., Schaetz, B.: mbeddr: Instantiating a language workbench in the embedded software domain. ASE **20**(3) (2013) 339–390
13. Khalilov, E., Ross, J.: 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.
14. Aleti, A., Buhnova, B., Grunske, L., Koziolek, A., Meedeniya, I.: Software architecture optimization methods: A systematic literature review. IEEE Transactions on Software Engineering **39**(5) (May 2013) 658–683
15. Voss, S., Eder, J., Schaetz, B., eds.: Scheduling Synthesis for Multi-Period SW Components
16. Glaß, M., Lukasiewycz, M., Wanka, R., Haubelt, C., Teich, J.: Multi-objective routing and topology optimization in networked embedded systems. In: SAMOS. (2008) 74–81
17. Lin, C.W., Rao, L., Giusto, P., D'Ambrosio, J., Sangiovanni-Vincentelli, A.L.: 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) (2015) 1730–1741
18. Biondi, A., Di Natale, M., Sun, Y.: Moving from single-core to multicore: Initial findings on a fuel injection case study. Technical report, SAE Technical Paper (2016)
19. Hamann, A.: Iterative Design Space Exploration and Robustness Optimization for Embedded Systems. Cuvillier (2008)
20. Streichert, T., Glaß, M., Haubelt, C., Teich, J.: Design space exploration of reliable networked embedded systems. Journ. on Systems Architecture (2007) 751–763
21. Aravantinos, V., Voss, S., Teufl, S., Hölzl, F., Schätz, B.: AutoFOCUS 3: Tooling concepts for seamless, model-based development of embedded systems. ACES-MB'15 (2015) 19
22. Software Engineering Institute: OSATE, version 2 http://osate.github.io/.
23. Schäuffele, J.: E/E architectural design and optimization using PREEvision. Technical report, SAE Technical Paper (2016)