APPEL - AGILA ProPErty and Dependency Description Language

Christoph Grimm[1], Frank Wawrzik[1], Alexander Louis-Ferdinand Jung[2], Konstantin Lübeck[2], Sebastian Post[1], Johannes Koch[1], and Oliver Bringmann[2]
[1]TU Kaiserslautern, Germany
[2]Eberhard Karls Universität Tübingen, Germany

Abstract

We give an overview of the language APPEL, the "AGILA Property and Dependency Description Language". It is part of the cloud-based tool AGILA that supports agile development methods. The language allows us to structure and document the knowledge about system-wide dependencies in a formal, textual form. APPEL models can be uploaded to the cloud, where they are used as a knowledge-base for continuous verification and validation, from early specification to run-time verification. We describe syntax, semantics, and demonstrate its application for predicting the performance of hardware/software systems in the context of the GENIAL! project.

1 Introduction

Modeling and simulation of executable models has its limitation once executable models are not available, or of different domains, or require excessive effort, e.g. in terms of simulation time. This is the case for the early project phases, where models still have to be created, starting from the elicitation of requirements to the formalization using e.g. SysML. In this context, it is crucial to acquire and consider knowledge; indeed, knowledge acquisition and its systematic application in an organization has been shown to be a key factor for commercial success of innovations [1]. A problem of SysML is that it is made for modeling experts, which often causes projects to fail due to lack of support by domain-experts [2].

In this paper, we propose a domain-specific, textual modeling language. Its purpose is to lower the hurdle for domain-experts to document, formalize and share their knowledge. The language permits to describe possible design alternatives, its performances, properties, constraints and the dependencies between them in a declarative way. It targets the following improvements:

- closeness to natural language,
- interactive applicability as a script language, and
- integrated means for modeling uncertain properties.

Properties can be seen as an abstract representation of system specifications that summarize its intended behavior, without having to describe details such as concurrency, or signals. Property specification languages, e.g. PSL (IEEE 1850), give a relation between temporal signals and properties. APPEL targets a higher level of abstraction, where properties and constraints are used for system specification, and the overall consistency of specified properties and constraints with the context from the environment, legislation, previous experiences, or simple natural laws.

In the following, we discuss related work and give an overview of the application of APPEL. In Section 2 we give an overview to the underlying ontological model and the semantics. In Section 3 we describe the language APPEL, and in Section 4 we give an example of the use of APPEL in the GENIAL! project.

1.1 State of the art and related work

Popular languages for the high-level specification of systems are UML [3], SysML [4], and OCL [5]. While UML and SysML allow users to specify systems by diagrams, and to exchange these diagrams via various textual representations (e.g. OMG XMI, UMLDI, or Mof2Text), the textual representations focus on the exchange of diagrams for tools, and are not intended to be a human-readable modeling language. SysML v2 [6] is a work in progress that also includes a textual modeling language. OCL complements UML and SysML by means to specify constraints on classes and objects.

However, the combination of SysML v2 and OCL is a language for specification of concrete designs, and not necessarily for the representation of knowledge. An extension in that direction is given by the OMG-ODM metamodel that permits the description by semantic triples. Compared to the combination of SysML v2, OCL, and the OMG-ODM metamodel, APPEL as a single, near-natural language, offers a lower entry-hurdle for non-modeling experts, and provides support for the specification of probabilistic uncertainties, which OCL does not.

For the representation of knowledge, OWL ([7], Web Ontology Language) is popular; a comparison between SysML and OWL is given in [8]. Its semantic triple structure positions OWL closer to natural languages. OWL is often used to close the gap between natural language specifications and formal models, e.g. in [9–11]. The mapping of a language to an ontology has also been done in Guizzardi's multi-level theory [12].

While OWL in many cases, e.g. [9–11] successfully permits to bridge the gap to near-natural language specifications, it lacks support for hard, numerical constraints and mathematical dependencies that are offered by APPEL.

As a potential use-case, we demonstrate the very early evaluation of embedded automotive systems (without being limited to that); it supports in particular the modeling of knowledge of functions, possible logical architecture, technical architectures and possible HW/SW mappings. A potential analysis can then be done by constraint propagation methods, e.g. to get very early estimations of performance properties. However, this is based on expert knowledge, not on simulation as e.g. in Eclipse's AMALTHEA/APP4MC [13].

1.2 Overview and context

The language APPEL is developed in the context of the tool AGILA. AGILA targets the knowledge-based analysis of mechatronic and electronic systems, from the early requirements elicitation to operation at a very high level. Its central element is a graph data base that stores semantic triples and that The APPEL compiler generates these semantic triples. A semantic triple is a set of three entities that represents a statement in the form of subject–predicate–object, e.g. *[A] car has wheels*. APPEL and AGILA furthermore permits triples with numbers and (in-)equations, e.g. *[A] car has 4 wheels*.
For analysis, we use a constraint propagation mechanism in the tool AGILA, based on the AADD-framework described in [14]. Alternatively, an external reasoner can be used on the semantic triples. The implementation of these tools is a work in progress, while the APPEL compiler that generates the semantic triples and uploads them into the data base is finished as described in Sections 3 and 4.

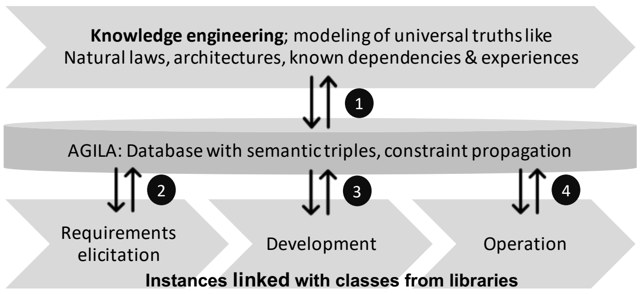

Knowledge engineering; modeling of universal truths like
Natural laws, architectures, known dependencies & experiences
1
AGILA: Database with semantic triples, constraint propagation
2
3
4
Requirements elicitation
Development
Operation
Instances linked with classes from libraries

Figure 1 Use of APPEL in AGILA.

APPEL is intended for use as a modeling language in the use cases (1) - (4) that are also shown in Figure 1:

(1) Knowledge engineering: It is used for modeling "universal truths" such as natural laws, taxonomies of known functions and architectures of general value, but as well performance models, equations or methods for choosing parameters of variants; we refer to such models as *classes*; collections of classes are a knowledge base, short *library*.

(2) Requirements elicitation: Specification of *instances* with concrete, required properties. Collections of such instances are a *design*; APPEL links instances to its classes and enables consistency checks by AGILA's constraint propagation.

(3) Development: Revision of specification and design decisions; in an agile development process, changes of a property can be announce quickly. The impact of the changes can be observed continuously via the propagated constraints.

(4) Operation: During operation, runtime-verification might be a use-case that we do not target in this paper: after deployment, monitors would have to compute properties based on sampled signals.

Focus of this paper is in particular the modeling language for application in the steps (1), (2) and (3). The purpose of these models is the very early check of the consistency of the requirements each other and the knowledge base by constraint propagation.
The constraint-propagation is for use cases (2) and (3) is implemented in the tool AGILA. It is based on a semi-symbolic approach using AADD and BDD introduced in [14, 15], but extended with the ability to propagate constraints in a bi-directional way. For the case that an inconsistency is detected, e.g. a requirement that cannot be satisfied, an error is reported; see Section 2.2.4.
The runtime verification (4) is subject of future work.

2 Basic vocabulary and consistency

The APPEL language consists of three parts: first, the basic vocabulary. It follows the way how ontologies are designed. It can be considered as a top-level ontology from which all further classes can be derived. It could also be implemented by a UML profile for modeling very abstract mechatronic systems. Second, properties and dependencies. They are inspired by OCL and provide means to specify Boolean and arithmetic dependencies on the elements of mechatronic systems. Third, complementary documentation and administrative information such as pictures, PDF files, users and access rights etc. that is not in focus of this paper.

2.1 Basic vocabulary

The basic vocabulary and language constructs of APPEL are aligned with the Genial Basic Ontology (GBO) that is based on the Basic Formal Ontology (BFO [16]), and leans on the ISO 26262 vocabulary:

- Element – An element is the base class of design or library elements. An element has a collection of properties (see below). An element can also contain natural-language description or complementary files, e.g. drawings, or video recordings.

- Function – A function is an element that specifies an abstract objective in a declarative way. The objective of the function is described in the field description in natural language.

- Component – A component is an element that implements a function by arbitrary means.

- Processor – A component that is a hardware part which executes software.
- Software – A component that is a software element or software unit which runs on a processor.

Note, that the above classes act as base classes in APPEL models and are intended to offer a low entry hurdle, while the BFO and ISO 20626 offer more precise definitions, see Section 3.3. APPEL allows a developer to describe elements using relations and properties. The following relations are pre-defined:

- hasElement – describes the hierarchical decomposition of an element (Function, Component, etc.) into further elements. The hasElement relation can be labeled with an integer range that describes a possible number of elements.
- isA – describes the inheritance relation between two classes.
- isInstanceOf – links an instance with the related class.
- isImplementedBy – links a function with components that implement it (inverse relation: implements).
- isExecutedBy – links a software with the processor on which it runs (inverse relation: executes).
- hasProperty – specifies a property of an element.

The relations can be used to specify relations between classes or instances of elements. The above basic vocabulary is defined more formally in the GENIAL! Basic Ontology (see Section 3.3). This ontology gives the language precise, formal semantics for application in reasoning. Using the vocabulary, we can give descriptions with semantic triples like:

```
LowPassFilter isA Filter
LowPassFilter implementsFunction Interpolation
Filter1 isInstanceOf LowPassFilter
```

2.2 Properties and dependencies

Elements have a collection of properties. A property is a variable that has an unknown value, and optionally a unit. For this unknown value, kind, type, and optional domain restriction, and an optional dependency expression give further information or constraints.

2.2.1 Kind and type of properties

We distinguish properties of the kind Quantity, Requirement, and Performance. The kind gives its domain restrictions and dependencies different consistency semantics as described below.

Quantity The declaration of a quantity introduces a variable of one of the following types:

Real: The value is from the reals. A unit and/or dimension can be specified.

Int: The value is from the natural numbers. We do currently not consider units as meaningful here.

Bool: The quantity can take a value from $\{true, false\}$.

String: The quantity is a string; strings are not used for constraint propagation beyond checking of equality.

Semantics: A quantity declaration introduces a variable that takes an (unknown) value from the set defined by the type with an optional unit.

Requirement A requirement is a predicate that shall evaluate to $true$ for a valid design. Hence, it is formally equivalent to a quantity of type Bool. However, the user interaction in case of inconsistencies can be handled in a different way by tools for visualization, e.g. to show that the specification is not consistent in this point.

Performance A performance is an indicator for a system performance that is subject to e.g. optimization. We assume it as a quantity of type Real with an additional annotation that announce whether it shall be maximized or minimized. This allows users to specify e.g. Key Performance Indicators.

2.2.2 Domain restrictions

A domain restriction directly introduces constraints for a property. This constraint models the uncertain information that we have on the unknown value. It can be given by:

- An interval, where e.g. upper and lower bounds are floating-point representations that gives a safe inclusion of the unknown value (For Int, Real).
- Probabilistic properties: either the mean value, variance and skewness, or the mean value and a confidence interval (Real), or the discrete probability (Bool).

Semantics: the domain restriction restricts the value of a property to those that are between upper/lower bounds, or it gives it the semantics of a random variable from a process with the given statistical properties.

Note, that for an implementation of the language, an accurate representation of values is not required. We consider it as sufficient to be able to give sound abstractions thereof. To give an example from our proof-of-concept implementation: we represent the unknown real value of a quantity of type Real by safely rounded floating point numbers: π is represented by an interval $[\pi - ulp, \pi + ulp]$, where ulp is the unit of least precision of its floating point representation. Overflow and other special cases (e.g. division by zero) are handled via symbols for the ranges Reals (all reals) or RealsOrNaN (a Real number, or a NaN result).

The domain restriction allows users the specification of a value range. An example for the specification of an interval and a probabilistic variable that lies in the interval [500..1000] with a confidence of 6 and a mean value of 750 is:

```
Filter1 hasProperty fc: Real(200..220) [Hz]
Filter1 hasProperty a0: Real(500..1000, 3*sigma, 750)
```

2.2.3 Dependency specification

The dependency specification of a property describes a relation between a property and other properties. The relation can be specified by a Boolean (and, or, not) or arithmetic (+, -, *, /) expression. Furthermore, pre-defined functions include $sqr, sqrt, exp, ln$ functions.
Semantics: for the (unknown) value of a property, the dependency specification must hold.
Note, that comparison operations ($>, <, =, >=, <=$) convert Real/Int values into a Boolean value that can be used in Boolean expressions. The ITE function can (also) select two Real value representations from a Boolean value representation, and thereby permits to model the dependency of Real values from Boolean values. Furthermore, rounding functions convert from Real values to Int values.
A probabilistic property p is converted by

- $interval(p, confidence)$ to an interval, and
- $conf(p, interval)$ to the respective probability.

2.2.4 Consistency

APPEL models have no operational semantics; its semantics is purely denotational, given by the declarations of properties. The property kind gives information on the semantic intention of dependency and domain restriction of the property:

- A quantity is a property in which dependency and domain restrictions must be satisfied; violations are not possible, e.g. as it models a natural law.
- A requirement is a property in which dependency and domain restrictions must be satisfied for a valid design.
- A performance is a property in which dependency and domain restrictions must be satisfied, and whose value is subject to optimization.

This information can be used for various purposes by tools. For example, one can check models for consistency, i.e. if there are requirements or properties that contradict each other. A model is considered as consistent, if there potentially exists at least a single combination of values for all properties such that

- all properties fulfill its domain restrictions, and
- all dependency specifications are satisfied.

A model in which we can show that a property (incl. domain restriction, dependency) is not satisfiable is called *inconsistent*.

2.2.5 Inheritance

For the inheritance (`isA`) and instantiation (`isInstanceOf`) relations, APPEL follows the Liskov principle [17]. Hence, subclasses are related to superclasses such that a subclass can be used instead of the superclass:

- subclasses inherit the properties from their superclass, with the same types,
- domain restrictions must be subsets of the domain restriction of the superclass,
- subclasses implement (in the sense of giving implements relations) at least the superclass functions.

Note, that in addition, the dependency expressions shall be free effects that violate the Liskov principle. Hence, the dependency of a superclass must result in a safe inclusion of all dependencies of its subclasses. More complicated violations could be introduced in particular by multiple inheritance. In order to not complicate the modeling – in the end we target domain-expert users, and not modeling experts – multiple inheritance and refinement of probabilistic properties are not supported.

3 APPEL Syntax

3.1 Lexical elements and scope

APPEL uses space characters, tabulators and carriage return/line break as white space elements. APPEL is case-sensitive, hence `isA` and `isa` are two different tokens.

Comments Comments start with "//"; the rest of the line is considered as a comment. We do not ignore comments, but keep them as description in the AGILA Database (see Figure 1). All comments before a statement, and in the same line are considered as documentation for it.

Number literals Simple number literals are defined like Double resp. Int values in usual programming languages.

Keywords Keywords of APPEL are written in the syntax definition in quotes. To improve readability of semantic triples, we allow the keywords `hasProperty / Property`, `hasPerformance / Performance`, `hasRequirement / Requirement` as alternative lexemes.

Identifiers and their scope An identifier is a globally unique reference to an artefact: a package (a library or a design), an element of a package, or a property. It consists of names that are separated by dots. Names start with a letter (a-zA-Z) and are followed by a letter or a digit (0-9) or an underscore (_). A name must not be equal to a keyword.

- The name of a package is a single name, e.g. `circuits` that is not used by any other package.
- The name of an element is unique in a design; there exists no other element in a design with the same name. It consists of the name of the library or design, and the name of the element, separated by a dot, e.g. `circuits.SigmaDeltaADC`.
- An identifier of a property (of an element) is unique in that element, and for example the identifier `circuits.SigmaDeltaADC.bitwidth` refers to

the property `bitwidth` in the element `SigmaDeltaADC` in the package `circuits`.

Artefacts can always be accessed by its identifier. If a name is used, the last used package and element are used to extend a name to the identifier.
Alternatively, elements and properties can be accessed by giving a path across the relations given in Section 2, starting from a current element `it`.
For example,

```
["it"] (relationNAME "->" elementNAME)+
```

refers to an element, and

```
["it"](relationNAME "->" elementNAME)+ "." propertyNAME
```

refers to a property.

3.2 Syntax

In the following we give an overview of the syntax of APPEL, partially in an extended BNF notation. Rules start with a small letter, terminals resp. literals are written in capital letters, e.g. NUMBER, NAME. We furthermore give some simple examples of the syntax. APPEL models are structured in packages that are either libraries or designs. Libraries hold classes that model generally valid facts, e.g. natural laws or known architecture variants. Designs hold instances that must be consistent with the facts from the libraries. Each package has a name (NAME) and consists of a collection of triples:

```
package :- ("Library" | "Design") NAME
triple*
```

In the following we explain informally the syntax of triples.

Inheritance and instantiation The most important triple permits the inheritance of classes from another class, or to create an instance.

```
NAME ("isA"|"isInstanceOf") className
```

As a running example, we create a library that holds some general axioms about circuits; the dots indicate that we need to complete the running example in the following:

```
Library runningExample
  LowPass iaA Function
  ...
  RCLowpass isA Component
  ...
```

Alternatively, we allow for the pre-defined classes Function, Component, Processor and Software the equivalent syntax `ClassNAME NAME`, e.g.:

```
Library runningExample
  Function LowPass
  ...
  Component RCLowpass
```

Hierarchical de-composition The hierarchical de-composition can be modeled by triples of the form:

```
["It" | elementNAME] "hasElement"
      NAME ":" [NUMBER[ ".." NUMBER]] elementClassNAME
```

The `hasElement` relation has an optional cardinality. It specifies that the subject consists of a range *number..number* or a constant number of elements from the class elementClassName. We can now express some relations for the running example, e.g.:

```
RCLowpass isA Component
  It hasElement R1: Circuits.R
  It hasElement C2: Circuits.C
```

Note, that the logical architecture as the de-composition of functions and features into subfunctions in a similar way.

Logical, technical and HW/SW architecture The technical architecture (mapping functions and features to components) can be modeled with the predefined relations implements/executes and its inverse relations from the GBO:

```
["It" | elementNAME]
    "implements" | "isImplementedBy" |
    "executes" | "isExecutedBy" )
  elementTarget ("," elementTarget)*
```

For the running example, the following would state that and RC low pass implements a low pass function:

```
RCLowpass isA Component
  ...
  It implements LowPass
```

Other possible implementations could be digital filters, software implementations, etc. In a similar way, the predicates `executes / isExecutedBy` can be used to model the HW/SW architecture at a very high level.

Properties Properties can be of the kind quantity, performance, or requirement. In deviation to the classical semantic triples, they have additional elements.
Quantities have a type, and optionally a domain restriction, a unit, and a dependency expression:

```
quantity :- ["It" | "elementNAME"]
            "[has]Property" NAME ":"
              type [domainRestriction]
              ["[" unit "]"]
              ["=" expression]

type :- "Real" | "Int" | "Bool" | "String"

domainRestriction :-
  "(" (const ",")* const ".." const ")"
  | "(" [const ","] "TRUE"  ")"
  | "(" [const ","] "FALSE" ")"
```

In the production quantity, `unit` is an expression that consists of unit names, e.g. `m/s2`; we don't describe the unit system of APPEL/AGILA here further.
`type` and `domainRestriction` describe the property's domain, where const is a production that can be evaluated to a number. Unlike in most modeling languages, we allow

properties and literals to hold uncertain values, specified by ranges, sets, probabilities. Hence, `domainRestriction` is given by

- an interval specified by `(lb..ub)`, where lb, ub are Double resp. Integer literals (for Real, Int type)
- the probabilistic properties confidence, skewness, and confidence interval bounds (for Real)
- TRUE or FALSE (for Bool)
- TRUE or FALSE with a parameter that specifies the probability of the values TRUE resp. FALSE.

Also, `expression` is an arithmetic or Boolean expression. In the expression, the usual operators are allowed; in addition, pre-defined constants (PI, e), and functions (sqrt, sqr, exp, power2, ite). Of particular relevance is the availability of a function that search for elements of a particular class: findElement(className) and hasElement(className). This function search – in the current library or design – an element that is either a class itself or an instance of the class.
Note, that the intended analysis is constraint propagation, e.g. by symbolic methods. Hence, we allow instead of literals in expression also the combination of a type with domain restriction:

```
literalOrType :- NUMBER | "TRUE" | "FALSE" | STRING
               | type [domainRestriction]
```

With the above constructs, we can further extend the running example with a property f_c that depends on the values of components R_1, C_1 that have an accuracy of 10% with a confidence of 2σ:

```
RCLowpass isA Component
  It hasElement R1: Circuits.R
  It hasElement C1: Circuits.C
  It implements LowPass
  It hasFile "schematic.spice"
  // f_c is the corner frequency.
  It hasProperty f_c: REAL(0.0 .. 1000.0) [kHz]  =
    1/(2*PI*R1.value*C1.value*Real(sigma2, 0.9..1.1))
  // Chip area; we neglect interconnect.
  It hasProperty area: REAL(0.0 .. 1000.0) [um2]
        = R1.area + C1.area
```

Performances and requirements have an implicitly known type (Real resp. Bool), hence:

```
performance :- "Performance" NAME
               [domainRestriction]
               ["=" "MIN" | "MAX" "(" expression")"]

requirement :- "Requirement" NAME
               [domainRestriction]
               ["=" expression]
```

The performance statement gives an expression that is subject to minimization (MIN) or maximization (MAX). The requirement statement gives a Boolean expression that must evaluate to true if no other domain restriction (e.g. to false) is specified. In the running example, we add some requirements:

```
RCLowpass isA Component
  ...
  It hasRequirement not(hasElement(Digital.clock))
  It hasPerformance area = MIN( ... )
  It hasRequirement maxArea = (area < 1.0 cm2)
```

The example library gives a very simple example how to create a library that is used as a knowledge base of general facts and dependencies that can be use to validate the specification of a concrete design very early development during specification and architecture exploration. The running example might be used to very quickly check the feasibility of e.g. a filter specification:

```
Design exampleDesign
  ...
  LowPass1 isInstanceOf RCLowpass
    It hasProperty f_c: Real(10.0 .. 11.0) [kHz]
    It hasRequirement maxArea = (area < 10.0 [um2])
```

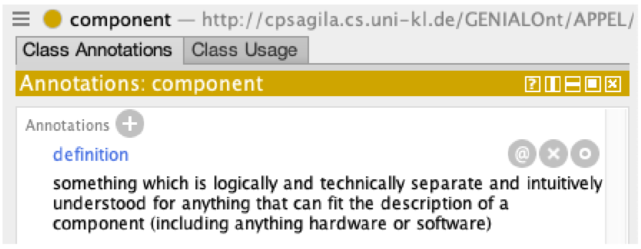
With further dependencies in the elements R, C, e.g. a constraint network can estimate the chip-area, based on the corner frequency f_c, the architecture (RCLowpass), and estimations for the size of R_1, C_1. If there is no value pair for R_1, C_1 that satisfies the requirement maxArea and yields a apecified corner frequency, the design is inconsistent.
Chapter 4 gives a comprehensive example for HW/SW systems.

3.3 Mapping to the GBO

A proof-of-concept implementation of APPEL has been implemented that can either process complete files or be used interactively. After sufficient information has been collected, the compiler calls the REST API of AGILA and uploads the element into the AGILA data base.
In AGILA, elements are assigned to the pre-defined classes of the GENIAL! Basic Ontology for HW/SW Systems (GBO). The GBO is a top-level ontology. It acts as an overall ontology where things/classes/concepts are classified and defined more precisely. The GBO defines classes

- in line with the ISO 26262 vocabulary, e.g. differentiating precisely between component, hardware part, software unit, software component, and
- extending the BFO [16] that introduces precise definitions of classes such as function, continuant, quality.

A domain expert using APPEL who is supposed to contribute expert knowledge from his expertise might not have this understanding of the definitions in ISO26262 or BFO. For him, APPEL provides a low entry hurdle, while a knowledge engineer, if necessary is intended to classify the ontology in more detail. Nevertheless, APPEL suggests a reasonable initial classification for a model-expert driven classification into the GBO and the BFO [16] classes that act as top-level ontology and provide an initial formal apparatus for reasoning.
To illustrate the difference between the APPEL vocabulary and the definitions in the GBO, we show two terms with its definitions used in the knowledge base. Figure 2 shows the definition of component, which is more general and follows the intuition.


component — http://cpsagila.cs.uni-kl.de/GENIALOnt/APPEL/

Class Annotations | Class Usage

Annotations: component

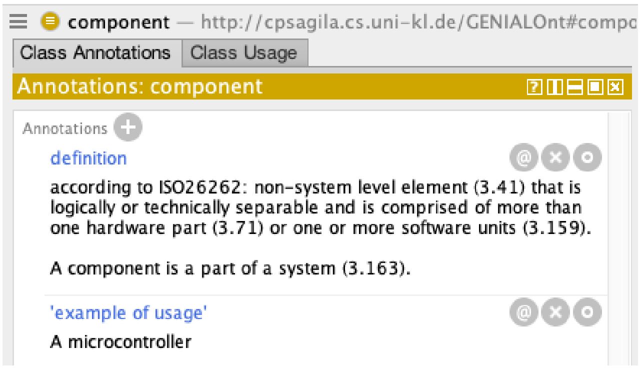
Annotations

definition

something which is logically and technically separate and intuitively understood for anything that can fit the description of a component (including anything hardware or software)

Figure 2 APPEL vocabulary definition of Component.

On the other hand, the definition of component in ISO 26262 in Figure 3 requires the understanding of a modeling expert (e.g. an ontologist). Besides others, the understanding of granularity (order in the hasParts resp. hasElements hierarchy), of other definitions (e.g. what a system is) are aspects to consider when classifying this element.


component — http://cpsagila.cs.uni-kl.de/GENIALOnt#compo

Class Annotations | Class Usage

Annotations: component

Annotations

definition

according to ISO26262: non-system level element (3.41) that is logically or technically separable and is comprised of more than one hardware part (3.71) or one or more software units (3.159).

A component is a part of a system (3.163).

'example of usage'

A microcontroller

Figure 3 ISO 26262 definition of Component.

4 Application in GENIAL!

The GENIAL! research project aims at the creation of an innovation roadmap for microelectronics in the automotive industry. Innovative functions are becoming the driving force behind the development of current and especially future mobility. For the realization of many of those innovative functions a large number of hardware and software components have to be combined: Starting at different sensors which provide a variety of data which, after initial preprocessing, will be used in multiple stages such as perception, prediction and control. These stages can either run in parallel or sequentially. All those steps create a functional network starting at the sensor and ending at the actor. To aid the modeling process of such a complex system function in an early development stage, APPEL can be used

- for creating a (generally valid) knowledge base that includes in particular models of the estimated, future properties and performances, and
- for modeling various future architectures, and getting very quick evaluations of its properties and performances,

which will be demonstrated briefly in this section.

4.1 Modeling expert knowledge

In an APPEL library, generally valid knowledge about HW/SW architectures is modeled by classes that describe the possible design space of innovative HW/SW systems. We focus on innovative HW/SW systems such as neural networks and sensor data processing. The modeling strategy is as follows:

- functions are used to describe high-level features, their properties, and their requirements,
- processors and components are used to model available hardware, that implement a function or execute a software, and
- software is used to model software components and -units that implement a function.

4.1.1 Overall HW/SW library

The overall library is work in progress and structured as follows: For example, we give functions to preprocess data, or to compute deep neural networks (DNNs). For the functions, we give relevant properties that must also be provided by the respective implementation (similar to interfaces in programming languages):

```
Library HwSw

  // Functions, subfunctions, possible realization
  (...)
  preprocess isA Function  (...)
  computeDeepNeuralNetwork isA Function (...)
    Property (...) // more properties, see below
    (...)
```

Furthermore, we provide possible implementations of the functions, e.g. a HW or SW implementation, as well as HW components that execute SW.

```
  // Implementation variants for functions
  DeepNeuralNetworkSW isA Software
    It implements computeDeepNeuralNetwork
    It hasElement [1..INF] DeepNeuralNetworkLayer

  DeepNeuralNetworkLayer isA Software
    Property C: Int
    Property C_w: Int
    (...) // see below

  // Some processors, e.g. a HW accelerator
  DeepNeuralNetworkHW isA Processor
    It executes DeepNeuralNetworkSW
    Property fclk: Real(0 .. 1e9) [Hz]

  UltraTrail isA DeepNeuralNetworkHW
    Property fclk: Real(2.5e5 .. 4e5) [Hz]
    (...) // see below
```

Note, that APPEL does not provide specific means for modeling elements for ports and signals. Generally, we consider them as part of a more refined design that would be realized later within the development process.
However, to model timing constraints, effect chains or communication bottlenecks, basic classes for communication are provided as a library that includes port, channel, wire, air as means for communication. This library just gives a basic high-level framework and can be extended (which was not needed for the example).

4.1.2 Modeling the performance of a neural network hardware accelerator

The neural network hardware accelerator UltraTrail was introduced by Palomero et. al. [18] and consists of a MAC array to execute convolutional computations and an output processing unit (OPU) for applying additional activation and pooling operations to the output of the MAC array. There are several memories (WMEM, BMEM, FMEM0-2) connected to the MAC array and the OPU acting as input, output, and temporal storage for the neural network being processed. Figure 4 shows the system architecture of the UltraTrail accelerator.

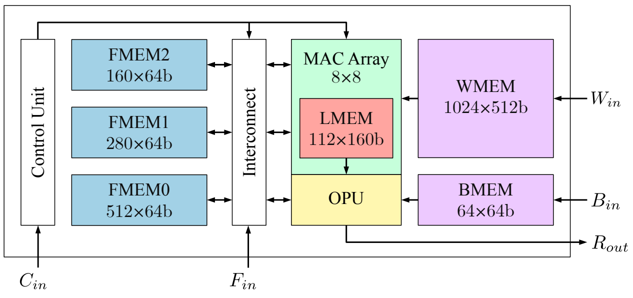


Figure 4 Overview of UltraTrail system architecture [18].

The MAC array consists of 64 MAC units in an 8 × 8 grid. Each MAC unit is capable of an 8-bit by 6-bit multiplication and a subsequent 8-bit by 8-bit addition. The whole MAC array is implemented as a combinational circuit which allows for 64 MACs/cycle. The OPU connected to the MAC array is also implemented in a fully combinational fashion and therefore only takes one clock cycle to process. A schematic overview of the MAC array is presented in Figure 5.

The authors propose a cycle accurate analytical timing model for mapping neural networks (software) onto the UltraTrail accelerator (hardware). In which each neural network layer is described as a 6-tuple $l = (C, C_w, K, F, s, p)$ which can be directly implemented as a `Software DeepNeuralNetworkLayer` class in APPEL with the following properties:

```
Property C: Int (1 .. 56)     // num. input channels
Property C_w: Int (1 .. 127) // width of input channel
Property K: Int (1 .. 56)     // num. of output channels
Property F: Int (1 .. 15)     // filter width
Property ns: Int (1 .. 7)     // stride
Property p: Bool              // padding
```

The equations from [18] describe the timing behaviour of the accelerator when computing a neural network and can be modeled as APPEL properties of the `Processor UltraTrail` class:

```
Property s: Int = power2(ns)
Property C_w_hat: Int                          // eq. 8
    = ITE(p, C_w + 2*floor(F/2), C_w)
Property a_w: Int = floor((C_w_hat - F)/s + 1)// eq. 9
Property C_wb: Int = floor(F/2)                // eq. 10
Property a_pb: Int                             // eq. 11
    = ITE(p, floor((C_wb - 1)/s + 1), 0)
Property MAC_notb: Int                         // eq. 12
    = sum_i(0, a_pb - 1, floor(F/2) - s * i)
```

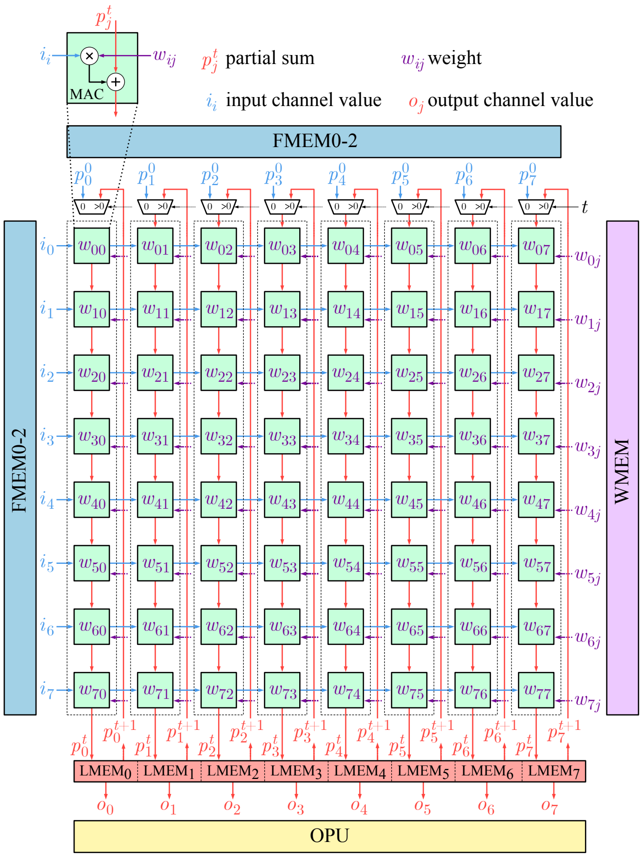

p_j^t
i_i
w_{ij}
MAC
p_j^t partial sum
w_{ij} weight
i_i input channel value
o_j output channel value
FMEM0-2
p_0^0 p_1^0 p_2^0 p_3^0 p_4^0 p_5^0 p_6^0 p_7^0
t
i_0 i_1 i_2 i_3 i_4 i_5 i_6 i_7
w_{00} … w_{77}
w_{0j} w_{1j} w_{2j} w_{3j} w_{4j} w_{5j} w_{6j} w_{7j}
WMEM
p_0^t p_0^{t+1} … p_7^t p_7^{t+1}
LMEM$_0$ LMEM$_1$ LMEM$_2$ LMEM$_3$ LMEM$_4$ LMEM$_5$ LMEM$_6$ LMEM$_7$
o_0 o_1 o_2 o_3 o_4 o_5 o_6 o_7
OPU

Figure 5 Schematic view of the MAC array [18].

```
Property Fw: Int = a_w * s + F - s             // eq. 13
Property C_we: Int = Fw - C_w - C_wb           // eq. 14
Property a_pe: Int                             // eq. 15
    = ITE(p, floor((C_we - 1)/s) + 1, 0)
Property MAC_note: Int                         // eq. 16
    = sum_i(0, a_pe-1, floor(F/2)-s*i-(C_wb-C_we))
Property t_l: Int                              // eq. 17
    = 1 + ceil(C/8) * ceil(K/8)
    * (a_w*F - MAC_notb - MAC_note)
```

To predict the runtime $T(\mathbb{L})$ in clock cycles of all neural network layers $l \in \mathbb{L}$ the runtimes $t(l)$ have to be summed up

$$T(\mathbb{L}) = \sum_{l \in \mathbb{L}} t(l)$$

For each neural network layer a concrete instance of `Software DeepNeuralNetworkLayer` is instantiated with the corresponding layer parameters as properties. To obtain the runtime of a neural network on the UltraTrail accelerator in seconds $\tau(T(\mathbb{L}))$ the following APPEL property has to be evaluated:

```
Property tau: Real [s] = T_L/fclk
```

4.2 A voice-control functional network

As one moves forward to the development of a concrete system, APPEL can be used to model and analyze concrete designs at a very high level:

- Requirements and intended functions are modeled by APPEL functions with properties and constraints.

- The logical architecture (breakdown of function) to sub-functions is modeled by a structural decomposition of these functions.
- The technical and HW/SW architecture (mapping of functions to components) is modeled by the relations `implements` resp. `isImplementedBy` and `executes` resp. `isExecutedBy`.

In particular, the analysis targets to get a very early performance estimation: the number of clock cycles for a SW unit running on e.g. an UltraTrail Processor.
A complex system-function could for example be a voice-control interface in a car with which it would be possible to open the trunk when standing in front of the car, start music playback or the car itself. However, the different components of this system-function can be implemented in several ways.

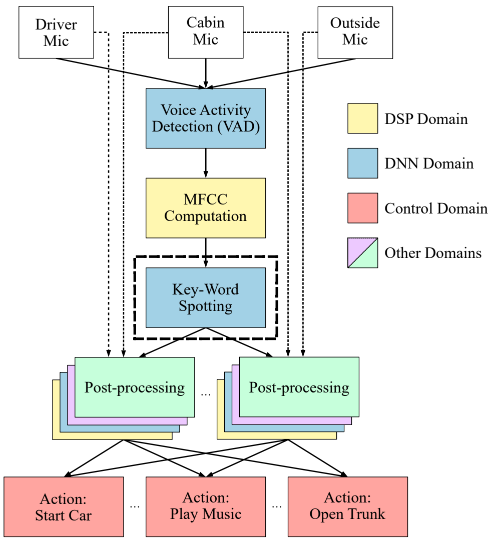


Figure 6 Voice-control functional network.

The functional network of this system-function is illustrated in Figure 6. Here some of the abstract functions have already been mapped to more specific algorithm domains. It starts by initially detecting if there is actually a voice present in the audio stream coming from different microphones in the car and not just background noise from traffic and pedestrians on the street. This step is called Voice Activity Detection (VAD) and will be performed by a deep neural network (DNN). If there is indeed a voice to be heard the recording of the raw audio data is started which then needs to be pre-processed with algorithms from the DSP domain. During this step the Mel Frequency Cepstral Coefficients (MFCC) are computed which serve as an input to the next step: the key-word spotting (KWS) using again a DNN. Only after the key-word has been detected the actual voice command will be recorded, post-processed and the corresponding actions are carried out.
In the following we provide an incomplete APPEL model of this system-function:

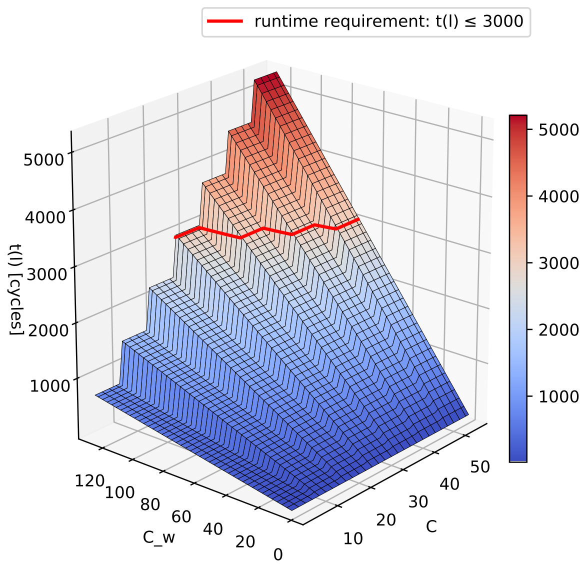
```
Design VoiceControl

  (...)
```

```
VoiceActivityDetection isA Function  (...)
  It isImplementedBy DeepNeuralNetworkVAD, (...)

AudioPreProcessing isA Function  (...)
  It isImplementedBy MFCCComputation, (...)

KeyWordSpotting isA Function  (...)
  It isImplementedBy DeepNeuralNetworkKWS, (...)
(...)

DeepNeuralNetworkVAD isInstanceOf DeepNeuralNetworkSW
  Property (... Properties refined from library ...)

MFCCComputation isInstanceOf DigitalSignalProcessingSW
  Property (... Properties refined from library ...)

DeepNeuralNetworkKWS isInstanceOf DeepNeuralNetworkSW
  Property (... Properties refined from library ...)
(...)
```

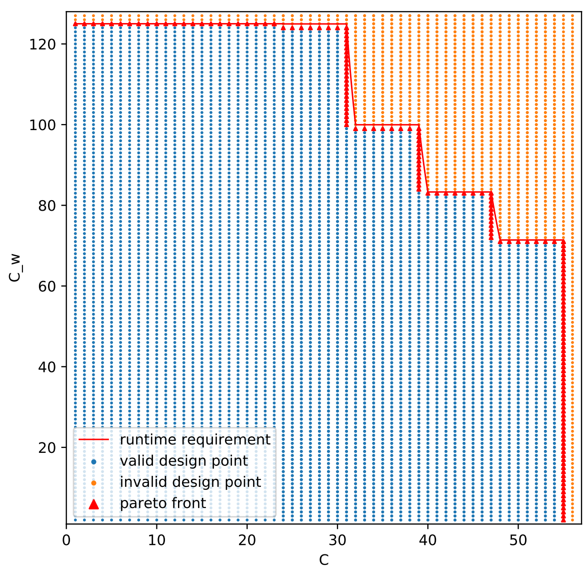
Furthermore, those software components have to be mapped onto a hardware component. Obviously, the automotive domain comes with utterly different requirements than voice controlled smart assistants that have a stable internet connection and wall power. The whole process needs to be performed on site as, apart from privacy concerns, it might not be possible to use a server based speech-to-text system due to an unreliable internet connection. Moreover, it needs to be highly energy-efficient as to not drain the car's battery. Following those requirements the DNN used for key-word spotting is mapped onto an instance of the ultra low-power AI hardware accelerator UltraTrail as described in [18]:

```
(...)
ultraTrail1 isInstanceOf UltraTrail

DeepNeuralNetworkKWS isInstanceOf DeepNeuralNetworkSW
  It isExecutedBy ultraTrail1
  It hasElement LayerConvExt_0
  (...)
  It hasElement LayerConvExt_4

LayerConvExt_0 isInstanceOf DeepNeuralNetworkLayer
  Requirement runtime = (t_l <= 3000)
  Property C: Int(1 .. 56)
  Property C_w: Int(2 .. 127)
  Property K: Int = 16
  Property F: Int = 3
  Property s: Int = 1
  Property p: Int = 0
(...)
```

In a design model, the constraint network of AGILA uses all properties and its dependencies to compute new upper and lower bounds (actually, safe inclusions thereof) for all properties. For this purpose, it uses AADD and BDD for symbolic computation (see [14]) on a network of dependencies generated by the APPEL parser.
Once the model is loaded, properties can be restricted to ranges or also set to concrete values. This permits an interactive, very early analysis of system performances based on available knowledge – there the library and estimation formula from Section 4.
In the given example the parameters `C` and `C_w` of the `LayerConvExt_0` layer are intervals taken from [18]. `C` is the number of network layer input channels and `C_w` represents the width of each input channel. Given the timing



(a) Runtime $t(l)$ for varying `C` and `C_w`.



(b) Design points for runtime requirement $t(l) \leq 3000$.

Figure 7 Runtime evaluation of the `LayerConvExt_0` on the UltraTrail accelerator.

requirement `runtime` $t(l) \leq 3000$ of `LayerConvExt_0` a design space for the execution on the UltraTrail accelerator can be obtained through an external tool. Figure 7(a) shows this design space together with the requirement `runtime`. In Figure 7(b) all possible design points are shown and classified into valid, invalid, and pareto front. From all valid design points a selection based on other external requirements can be achieved.

5 Discussion and Conclusion

APPEL is a formal, textual modeling language that allows designers to model the inheritance, decomposition, and properties and its dependencies of a system. Libraries in APPEL can be seen as an ontology with a domain-specific top-level ontology for modeling mechatronic, HW/SW and Analog/Mixed-signal knowledge and designs. Libraries support the documentation and semantification of knowledge throughout the product life cycle, whereas designs can be evaluated using this knowledge, which allows very early estimates for system properties. The expressivity and application of the language has been demonstrated by the example of a knowledge base for the estimation of runtimes of future HW/SW systems.
Compared with the SysML v2 textual notation in combination with OCL, the readability by non-modeling experts, and the integration of model and constraint propagation – are the main difference.
As a compact example, we give a simple Motorcycle model that is a specialization of a Vehicle, has a property mass and consists of an engine and two wheels (at least in this simple model). In SysML:

```
  block Motorcycle :> Vehicle {
    value mass : ScalarValues::Real;
    part eng : Engine;
    part wheels: Wheel [2 .. 2];
  }
```

In addition, we assume that the mass is limited to values between 0 and 500 kg via OCL, e.g.:

```
context Motorcycle inv: self.mass >=0;
context Motorcycle inv: self.mass <= 500;
```

For comparison, Appel integrates the SysMLv2 model and the OCL constraints:

```
Motorcycle isA Vehicle
  It hasElement Engine
  It hasElement wheels: 2 Wheel
  It hasProperty mass: Real(0 ..500) [kg]
```

The model is arguably closer to natural language. Furthermore, each line can be extended to a semantic triple as done above, if one adds the optional `it` or the name to the triple. This permits the interpreted application and interactive work where a user can interactively make additions or changes to a model, or explore the impact of possible refinements.
The language itself does not support access rights and version management for traceability. This is currently done in the AGILA backend, where also the constraint propagation is done to check consistency which is not in the scope of this paper.

Future Work There are several strains of research that build on top of APPEL. The first one is the development of domain-specific modeling libraries in the domains

- Future HW/SW Systems that include also functions for predicting performances of future realizations, based on e.g. Moore's law.

- Sensors, converters, and Analog/Mixed-Signal systems.
- Mission profiles.

Second, we are continuously improving APPEL and the AGILA backend with its constraint network that is used for the analysis. Current work includes in particular the improved support for probabilistic properties in the constraint network. Furthermore, we work on improving the support for representation of requirements in designs and its relations to component instances, in particular to also enable tracing of these relations.

Acknowledgement The authors acknowledge the financial support by the Federal Ministry of Education and Research of Germany and the European Commission in the GENIAL! project and Arrowhead Tools project.

References

[1] C. W. Soo, D. Midgley, and T. M. Devinney, "The process of knowledge creation in organizations," *SSRN Electronic Journal*, 2002. [Online]. Available: https://ssrn.com/abstract=376080

[2] N. Regnat, "Why SysML does often fail – and possible solutions," in *Modellierung 2018, Lecture Notes in Informatics*. Gesellschaft für Informatik, 2018.

[3] "Object Management Group, Unified Modeling Language," 2017. [Online]. Available: https://www.omg.org/spec/UML/2.5.1/PDF

[4] "Object Management Group, Specification of SysML 1.4." [Online]. Available: http://www.omg.org/spec/SysML/1.4/PDF

[5] "Object constraint language, v2.4," 2014. [Online]. Available: https://www.omg.org/spec/OCL/

[6] SysML v2 Submission Team, "Introduction to the SysML v2 Language – Textual Notation." [Online]. Available: https://drive.google.com/drive/folders/1VMirOt7aQHyG912eQJETXU606WkAdgVw

[7] "World Wide Web Consortium, Web Ontology Language (OWL)," 2012. [Online]. Available: https://www.w3.org/OWL/

[8] H. Graves, "Integrating SysML and OWL," in *Proceedings of the 6th International Conference on OWL: Experiences and Directions - Volume 529*, ser. OWLED'09. Aachen, Germany, Germany: CEUR-WS.org, 2009, pp. 117–124. [Online]. Available: http://dl.acm.org/citation.cfm?id=2890046.2890059

[9] E. Alkhammash, "Formal modelling of owl ontologies-based requirements for the development of safe and secure smart city systems," *Soft Computing*, vol. 24, no. 15, pp. 11 095–11 108, 2020. [Online]. Available: https://doi.org/10.1007/s00500-020-04688-z

[10] S. J. Körner and T. Brumm, "Improving natural language specifications with ontologies," in *Proceedings of the 21st International Conference on Software Engineering & Knowledge Engineering (SEKE'2009), Boston, Massachusetts, USA, July 1-3, 2009*. Knowledge Systems Institute Graduate School, 2009, pp. 552–557.

[11] H.-J. Happel and S. Seedorf, "Applications of Ontologies in Software Engineering," in *International Workshop on Semantic Web Enabled Software Engineering (SWESE'06)*, Athens, USA, November 2006. [Online]. Available: http://fparreiras/papers/AppOntoSE.pdf

[12] P. João, J. Almeida, F. Musso, V. Carvalho, C. Fonseca, and G. Guizzardi, "Preserving multi-level semantics in conventional two-level modeling techniques," 08 2019.

[13] R. Höttger, H. Mackamul, A. Sailer, J.-P. Steghöfer, and J. Tessmer, "App4mc: Application platform project for multi- and many-core systems," *it - Information Technology*, vol. 59, 11 2017.

[14] C. Zivkovic, C. Grimm, M. Olbrich, O. Scharf, and E. Barke, "Hierarchical verification of AMS systems with affine arithmetic decision diagrams," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 10, pp. 1785–1798, Oct. 2019.

[15] C. Grimm and M. Rathmair, "Dealing with Uncertainties in Analog/Mixed-Signal Systems," in *Proceedings of the 54th Annual Design Automation Conference, {DAC} 2017, Austin, TX, USA, June 18-22, 2017*, 2017, pp. 35:1–35:6. [Online]. Available: http://doi.acm.org/10.1145/3061639.3072949

[16] R. Arp, B. Smith, and A. D. Spear, *Building Ontologies with Basic Formal Ontology*. The MIT Press, 2015.

[17] B. H. Liskov and J. M. Wing, "A behavioral notion of subtyping," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 6, p. 1811–1841, Nov. 1994. [Online]. Available: https://doi.org/10.1145/197320.197383

[18] P. P. Bernardo, C. Gerum, A. Frischknecht, K. Lübeck, and O. Bringmann, "UltraTrail: A configurable Ultralow-Power TC-ResNet AI Accelerator for Efficient Keyword Spotting," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 4240–4251, 2020.