Modeling Architectures of Cyber-Physical Systems

Evgeny Kusmenko[1], Alexander Roth[1], Bernhard Rumpe[1,2], and Michael von Wenckstern[1(✉)]

[1] Software Engineering, RWTH Aachen, Aachen, Germany
{kusmenko,roth,rumpe,vonwenckstern}@se-rwth.de
[2] Fraunhofer FIT, Aachen, Germany
http://www.se-rwth.de
http://www.fit.fraunhofer.de

Abstract. Cyber-physical systems (CPS) in automotive or robotics industry comprise many different specific features, e.g., trajectory planning, lane correction, battery management or engine control, requiring a steady interaction with their environment over sensors and actuators. Assembling all these different features is one of the key challenges in the development of such complex systems. Component and connector (C&C) models are widely used for the design and development of CPS to represent features and their logical interaction. An advantage of C&C models is that complex features can be hierarchically decomposed into subfeatures, developed and managed by different domain experts. In this paper, we present the textual modeling family MontiCAR, **Mo**del**i**n**g** and **T**est**i**n**g** of **C**yber-Physical **Ar**chitectures. It is based on the C&C paradigm and increases development efficiency of CPS by incorporating (i) **component and connector arrays**, (ii) **name and index based autoconnections**, (iii) a **strict type system** with **unit** and **accuracy support**, as well as (iv) an advanced **Math language** supporting **BLAS operations** and **matrix classifications**. Arrays and their autoconnection modes allow an efficient way of modeling redundant components such as front and rear park sensors or an LED matrix system containing hundreds of single dimmable lights. The strict type system and matrix classification provide means for integrated static verification of C&C architectures at compile time minimizing bug-fixing related costs. The capabilities and benefits of the proposed language family are demonstrated by a running example of a parking assistance system.

1 Introduction

Development of Cyber-Physical Systems (CPSs) rises domain specific challenges that are rarely present in other software engineering disciplines such as enterprise applications and web development. These challenges mainly originate from steady interactions of such systems with the real world through imperfect sensors and actors while being exposed to complex environments and physical laws.

[KRRvW17] E. Kusmenko, A. Roth, B. Rumpe, M. von Wenckstern:
Modeling Architectures of CyberPhysical Systems.
In: Modelling Foundations and Applications (ECMFA'17). Springer International Publishing, 2017.
www.se-rwth.de/publications/

Germany's industrial de facto standard to address these challenges is the exida®/BMW SMArDT[1] approach, which consists of four layers: *object of reflection* (textual requirements and use cases), *logical layer* (functionality modeled by abstract C&C models and underspecified activity diagrams), *concrete technical concept* (deterministic C&C models and C code), and *realization* (e.g., ECUs, CAN-BUS, Flexray, and timing). C&C modeling strengths on the logical layer comprise the ability to describe architectures by components executing computations and information flows modeled via connectors between their interfaces. The paradigm focuses on software features and their logical communication. Due to hierarchal component decomposition, large and complex systems can be developed by different stakeholders in a divide and conquer manner. Prominent examples of C&C languages - used in both academia and industry - are Simulink [28] and LabView [20].

A C&C modeling approach should be easy to use and let the developer focus on the functionality of the system likewise it should reduce the error-proneness in the design phase. To fulfill these demands, we derived a set of requirements for a language to model the logical layer of CPS from a series of automotive, embedded and CPS projects. Today, these requirements are addressed by the intersected features of currently existing C&C modeling approaches rather than by one unified solution. Hence, this paper presents the MontiCAR language family, a textual modeling DSL based on the C&C paradigm. MontiCAR incorporates a strict type system with an integrated unit support allowing developers to work with physical quantities in a type-safe manner and liberating them from unit checks and conversion. MontiCAR types have a value range and a resolution to account for limited operating areas and accuracies of the system components.

Since many CPS tasks can be solved by mathematical models, an advanced math language is an integral part of the MontiCAR language family. Additionally, to guarantee system properties and to increase the performance of the generated code, we introduce a matrix type system that tracks matrix size, matrix elements' type, and algebraic properties. It is used at compile-time to infer variable properties of the computation results for simulation purposes and to choose the best internal data representation (e.g., full or sparse matrix storage).

The main contributions of this paper are: (C1) **a comprehensive comparison of different C&C concepts needed for modeling embedded software**, (C2) **high-Level modeling of dataflows including its semantics**, and (C3) **new concepts for component reuse based on arrays and efficient connector descriptions**.

This paper is structured as follows. First, a running example is described in Sect. 2, which is used to motivate the requirements explained in Sect. 3. Existing C&C modeling approaches are evaluated with respect to these requirements in Sect. 4. Based on these requirements, the following sections present the MontiCAR language family (Sect. 5) with a focus on EmbeddedMontiArc (Sect. 6) and MontiMath (Sect. 7).

[1] http://www.exida.pl/EnterTheDOOR/help/soley-generation.htm.

2 Running Example

A running example of a lightweight but incomplete driver assistance software system providing automated emergency braking and visual user feedback is depicted as a simplified C&C architecture in Fig. 1. Since this is a logical model, it does not exhibit any technical details such as assignments of components to concrete ECUs or transmission protocols. The `ParkingAssistant` component interface is defined by its in and out ports: In ports on the left hand side receive signals needed for component computations including the GPS position, speed, steering angle of the vehicle, as well as a port array for complex radar signals containing in-phases and quadrature components for object movement detection. In contrast, out ports on the right hand side represent the calculated results, i.e., user feedback for the dashboard and a `brakeForce` array controlling the car's four brakes.

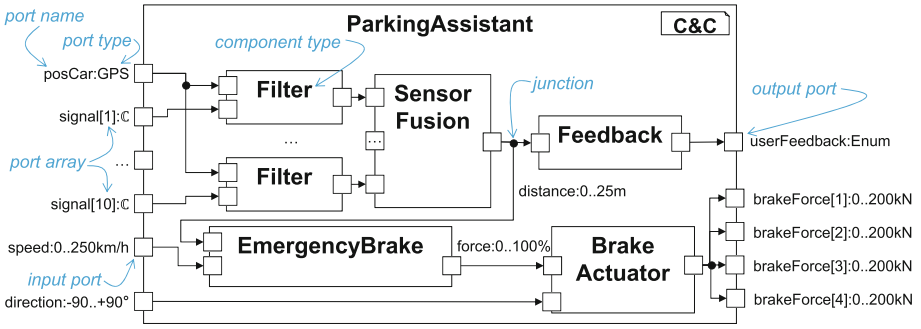

Fig. 1. C&C architecture of an Park Assistant component in automated vehicle.

The behavior of the `ParkingAssistant` (i.e., its concrete computation) is decomposed into several subcomponents each handling one specific task: filtering signals (`Filter`), fusioning sensor data (`SensorFusion`), calculating overall emergency brake effort (`EmergencyBrake`) depending on the distance, assigning concrete brake forces to each wheel (`BrakeActuator`) relative to the car's direction, and creating user feedback. The connectors, depicted by solid arrow lines, represent directed data flows between subcomponents.

A major concern in embedded systems is that most components only guarantee correct behavior for certain working conditions, e.g., radars are only able to detect obstacles within a certain area due to their physical nature. Models need to be enriched with such details to ensure verifiability and to enforce correctness at runtime. For example, `EmergencyBrake` uses radar measurements to execute its task, where it needs an obstacle detection within a predefined range having no blind spots between the sensors' beams. However, it is capable of processing input values outside this area, say all positive distances. The component might not have been designed to cope with unusual inputs such as negative distances.

Hence, it is essential to make supported ranges, operational areas, and accuracies of used types explicit in models so that they can be validated for system integrity and compatibility; e.g., it can be ensured that OEMs only use radars fulfilling the accuracy requirement of `EmergencyBrake` instead of choosing the cheapest one. Furthermore, explicit declarations of the ports' units are essential to make interconnections between ports describing different physical quantities, e.g., lengths and speeds, impossible and to provide automatic conversion between ports working with different unit prefixes such as meters and kilometers, or different systems of units, e.g., metric and imperial. Such static checks are usually not provided by existing C&C languages (cf. Sect. 4).

3 Requirements

From industry cooperations using CPS modeling, we derived the following requirements to solve the challenges introduced in Sect. 1:

- (R1) **Unit support.** In- and output ports should support (R1.1) *metric*, (R1.2) *imperial*, and (R1.3) *customized* units, such as pixel-per-inch.
- (R2) **Unit conversion.** Units should be automatically converted to SI units and prefixes should be resolved (R2.1) for port connections and (R2.2) in mathematical expressions.
- (R3) **Array support.** Redundancy in models should be avoided by supporting (R3.1) *port arrays* and (R3.2) *component arrays*. (R3.3) A convenient mechanism to interconnect and access ports and components should be supported.
- (R4) **Domain and Accuracy.** There is a need for concepts to model the domain, i.e., (R4.1) minimum, (R4.2) maximum, and (R4.3) resolution, of the values exchanged between components, and (R4.4) accuracies for sensor and actuator components. In addition, (R4.5) multiple domains with separate accuracies should be supported, e.g., high accuracy of a distance sensor is essential if the object is near but less important if the object is far away.
- (R5) **Static Analysis.** Theoretical concepts and tools to support *static analysis*, i.e., (R5.1) over- and (R5.2) underflow checks, (R5.3) division by zero, and (R5.4) detection of components in dead paths.
- (R6) **Reuse concepts.** (R6.1) A library concept for components and (R6.2) ports configurable over parameters is needed. Advanced reuse concepts such as (R6.3) *configuration parameters* and (R6.4) *generics* are required to allow modifications of component interfaces and behavior.
- (R7) **Matrix supports.** Discrete control systems are often described by matrix-vector expressions. To reduce error-proneness a type system should support (R7.1) static matrix dimension, (R7.2) units, and (R7.3) detection of domain incompatibilities, e.g., multiplying two 3×3 matrices having the domain $[0, 1]^{3\times3}$ must result in a $[0, 3]^{3\times3}$ matrix .
- (R8) **Differential equations.** Physical systems are often modeled by differential equations. A native support can facilitate the system design.

– (R9) **Acausal Modeling support.** Acausal modeling is needed to model systems where the behavior of each system's component depends on the global system architecture rather than having casual data flows with static component behavior. An example is a non-ideal voltage source whose output depends on the load connected.
– (R10) **Operational area.** It should be possible to define the operational area of a system, i.e., the constraints regarding sensor values in which the system is fully operational. For example, if multiple distance sensors are used, the operational area can be defined by at least two sensors detecting the obstacle.

4 Existing C&C Modeling Languages

This section compares the most important C&C modeling languages, listed in Table 1, according to the requirements in Sect. 3.

Most C&C modeling languages do not support units. In SystemC, unit support can be added by defining unit types as C++ preprocessor templates [23, p. 299]. In Simulink, units are used for documentation purposes only, e.g., in the bus editor. Partial unit support is available in xADL extending the meta-schema, Verilog AMS and VHDL AMS. The AMS extensions provides the `nature` concepts for defining a collection of attributes with own unit types [36, 3.6.1]. The unit annotation is passed to the simulator to check compatibility using *Units Value Rule*. Modelica, SysML, LabView, AutoSAR fully support SI units according to ISO 31-1992. Metric, imperial and customized units can be expressed as unit dimensions. Developers might need to provide transformations to SI units for own defined ones. MontiCAR uses the same unit meta model as defined in SysML 1.4 specification Sect. 8.6.4 to support units.

All languages having full SI unit support also allow unit conversions. Simple conversion (R2.1) is possible in AMS languages (Verilog AMS, VHDL AMS), since prefixes such as Pico, Micro are part of the number, e.g., `3 pA` still has Ampere as unit. Even though AMS languages support conversion of flow to potential and vice versa (using `disciplines`), they do not support more complex conversions such as kilometer per hour to miles per hour.

Support for port and/or component arrays (R3) is not present in Simulink, SysML, Marte, AutoFocus3, xADL, AutoSAR, LabView, MontiArc, Rapide, SADL, Scade, and TECS. Partial support for arrays is given in Modelica by using a class as a component. Classes can be instantiated as an array. This concept has been reused in MontiCar to support component arrays. Ptolemy supports arrays by using the Java array syntax and semantics, allowing only to define the number of array dimensions but not their concrete array size. Verilog supports one and two dimensional arrays using a similar syntax as Ptolemy. VHDL supports ranged and unconstrained arrays of defined types. SystemC allows to declare array sizes of ports and signals using a C-like syntax. UniCon supports fixed length arrays of simple types. WRIGHT supports multiple instances of pipes, which can be seen as arrays of the same connector.

Table 1. Comparison of C&C tools and standards (*), √: yes, P: partially, -: no

Architectural description language	Unit support (R1)	Unit conversion (R2)	Component/Port arrays (R3)	Domain (R4.1), (4.2)	Resolution (R4.3)	Static analysis (R5)	Configuration parameters (R6.3)	Generics (R6.4)	Matrix Support (R7)	Differential Equation (R8)	Acausal Modeling support (R9)	Operation Area (R10)
Simulink [28]	-	-	-	√	-	p	√	-	√	-	-	-
Modelica*[4,16]	√	√	P	√	√	P	√	√	√	√	√	-
SysML*[31]	√	√	-	P	P	-	√	√	-	-	-	-
Marte*[30]	√	√	-	P	P	-	√	√	√	√	-	-
AutoFocus3 [2]	-	-	-	√	-	√	√	-	-	-	-	-
xADL [11]	P	-	-	P	-	P	P	-	-	-	-	-
AutoSAR*[5,6]	√	√	-	√	√	P	√	-	-	-	-	-
LabView [20]	√	√	-	√	√	√	√	-	√	P	-	-
MontiArc [17]	-	-	-	√	√	P	√	√	-	-	-	-
MontiCar (this paper)	√	√	√	√	√	√	√	√	√	-	-	√
Ptolemy [14]	-	-	√	-	-	√	√	√	-	-	-	-
Verilog (AMS) [36]	√	P	√	√	√	√	√	√	-	P	√	-
VHDL (AMS) [3,15]	√	P	√	√	√	√	√	√	-	P	-	-
Rapide [25]	-	-	-	-	-	√	√	-	-	-	-	-
SADL [18]	-	-	-	-	-	√	√	-	-	-	-	-
Scade [13]	-	-	-	-	-	√	-	-	-	-	-	-
SystemC [34]	-	-	√	-	-	√	√	√	-	-	-	-
TECS [7]	-	-	-	-	-	√	√	-	-	-	-	-
UniCon [35]	√	-	√	√	√	√	√	-	-	-	-	-
WRIGHT [1]	-	-	√	-	-	√	√	-	-	-	-	-

Multiple languages provide support to specify the domain together with its resolution of a variable as required by (R4). Simulink allows to specify the minimum and maximum values for a signal and a resolution for fixed data types using the data type parameters slope and bias. LabView supports creation of custom scales, which can be linear, polynomial or table based. This facilitates flexibility for the creation of custom resolutions, ranges. However, for types with multi-

ple domains, only table scale is applicable. Modelica allows to specify attributes with a variable declaration. Both, reals and integers support a minimum and a maximum value. A `nominal` attribute can be used for automatic model scaling. AUTOSAR requires all of its integer types to have a constraint subnode to specify a scaling with minimum and maximum values.

SysML allows to create value types that are classifiers having the stereotype `«data type»` and the corresponding attributes, e.g., `min` and `max`. Using this data type allows setting minimum and maximum values for variables using it, e.g., in an object diagram. Alternatively, OCL constraints can be used to specify ranges and resolutions. Verilog-AMS natures concept supports type parameters such as `abstol`, and `max` absolute tolerance to define tolerances and allowed value ranges primarily for the VLSI domain. VHDL-AMS provides a similar approach but uses tolerances to specify the preciseness of approximations provided by numerical algorithms.

SysML, Marte have no concepts to detect over-, underflows, division by zero or unused components (R5). The Simulink Design Verifier detects division by zero. Over- and underflow checks are done at runtime. However, no compile-time verification is possible. Autofocus 3 uses model checkers (NuSMV/nuXmv [9]) to check variable ranges and find unreachable states. Since xADL is a modeling language without any semantic definition (it also has no denotational semantics introduced by a code generator), it only checks that connectors of models are correct. LabView programs are verified by translating their models to ACL2 solver expressions and formulate theorems that should be proved [21]. Finding duplicates (even semantical ones) in MontiArc is done using MontiMatcher [33]. MontiCAR uses the MontiMatcher framework and extends it with support for checking over-, underflow and divisions by zero by using the MontiMatcher's intermediate controlflow graph for backward compatibility checks. Structural crosscutting specification, C&C views, verification [26] as well as static consistency checks for extra-functional properties [27] are also available in MontiCAR. Ptomely is a Java-based event extension for modeling architectures. Therefore, static analysis tools (e.g., Cibai [24]) for Java programs can be used. Verilog models are checked by translating them to BLIF-MV [10] to perform symbolic verification[2]. In SystemC, type checking, control flow graph analysis, and verifying C pointers or static analysis in general can be done with SCOOT [8].

Enabling component reuse, e.g. of general library ones, can be decomposed into supporting configuration parameters and generics (R6). Simulink supports configuration parameters (`set_param` command), which can be defined arbitrarily. However, Simulink does not support general generics for modifying component interfaces. It only can be done partially, e.g. `Logical Operator` block allows to define the number of input ports. Modelica supports generic components and configuration parameters provided by the Modelica generic block (MBLOCK). SysML does not provide explicit language constructs to model variants but provides a profile mechanism to extend SysML with a concept for variant modeling, i.e., stereotypes. Stereotypes are also used in MARTE to define parameters and

[2] http://vlsi.colorado.edu/~vis/whatis.html.

generics. AutoFocus 3 supports parameters that can be used for configuration purposes but lacks support for generic components. By default xADL does not support configuration parameters and generics. However, because it uses XML as the base structure, the tooling supports partial configuration parameters by direct manipulation of the XML schema. In AutoSAR, parameters can also be used to configure components. However, it does not support generics for components. Similar holds for LabView. MontiArc supports concepts for configuration parameters and generic components, which are used in the MontiCAR language family. Being an extension of Java, Ptolemy supports configuration parameters and generics. Verilog, VHDL, Rapide, TECS, UniCon, WRIGHT, and SADL only support configuration parameters but no generics. Scade supports configuration parameters nor generics. Since SystemC is an extension of C++, it supports configuration parameters and adds support for generics.

Several languages implement native matrix support including MATLAB/Simulink as one of the most prominent examples, thereby, partially fulfilling (R7). However, MATLAB/Simulink neither provides a strict type system allowing for static checks required by (R7.1), and (R7.3), nor does it allow to specify units for the entries of a matrix. A far more elaborated matrix type system is provided by Modellica [16], which not only allows to define the element types and its dimensions but also units of the matrix elements allowing far more rigorous static checks. Other languages providing matrix support comprise MARTE [30] and LabView. The latter does not allow for the restriction of a matrix to a specified size and, hence, does not provide static checks. Instead, similarly to MATLAB/Simulink, matrices can grow dynamically and checks are only performed at runtime. In contrast, MontiCAR provides full matrix support with a strict type system and unit support with compile-time checks used in system verification. Furthermore, to our knowledge, MontiCAR is the only language using a matrix taxonomy to derive matrix properties for static checks.

Native support for differential equations (R8) is provided by Modelica and MARTE. With support for ordinary differential equations only, LabView partially fulfills this requirement. The same holds for Verilog and VHDL [29]. All other languages do not fulfill requirement (R8).

In Modelica, acausal modeling (R9) is done using flow ports and declarative equations[3]. In Verilog-AMS, acausal modeling (e.g.y Kirchhoff's Flow Law and Potential Law [36, Figs. 1–3]) is done using signal-flow systems. Since MontiCAR is designed for modeling on the logical layer of the SMArDT methodology, there is no need for modeling flow properties, e.g., current or voltage flows. Therefore, Modelica's acausal modeling concept has not been integrated to keep the language syntax slim.

From the overview in Table 1, it can be derived that none of the analyzed modeling languages support definition of an (R10) operational area, e.g., to guarantee a correct behavior for a limited set of environmental conditions. In MontiCAR, we integrated OCL/P to allow for the definition of such constraints.

[3] https://www.openmodelica.org/images/docs/Modelica-and-OpenModelica-overview-Peter-Fritzson-120328.pdf.

5 MontiCAR Modeling Family

As was shown in Sect. 4, existing modeling languages fail to provide all necessary means for type-safe and verifiable modeling of cyber-physical systems. Therefore, we present MontiCAR, a modeling language family developed against real industrial requirements gathered in Sect. 3. The structure of the complete MontiCAR modeling family is shown in Fig. 2. In this section we give a short description for each family member.

The base language used by all the other language members is `NumberUnit`. It contains rules to parse complex numbers, e.g., `2 - 4i` or rational numbers with and without units, e.g., `-3/7 m/s^2`, `0.35`, `1N`. `OCL/P` [32] is a Java-based OCL derivative to formulate constraints such as `brakePedalPressed implies vehicleAcceleration < 0 m/s^2`, i.e., the acceleration should be negative if the brake pedal is pressed. The syntax of `MontiMath` is very similar to the one of MATLAB except that it forces all its variables to be typed. This language will be introduced in more depth in Sect. 7. The `Type` language allows the definition of enumerations and C-like structures. An example is provided in Fig. 3. Lines 1–2 define a struct type for GPS coordinates aggregating the scalars latitude and longitude. Pay attention to the type of the two primitive

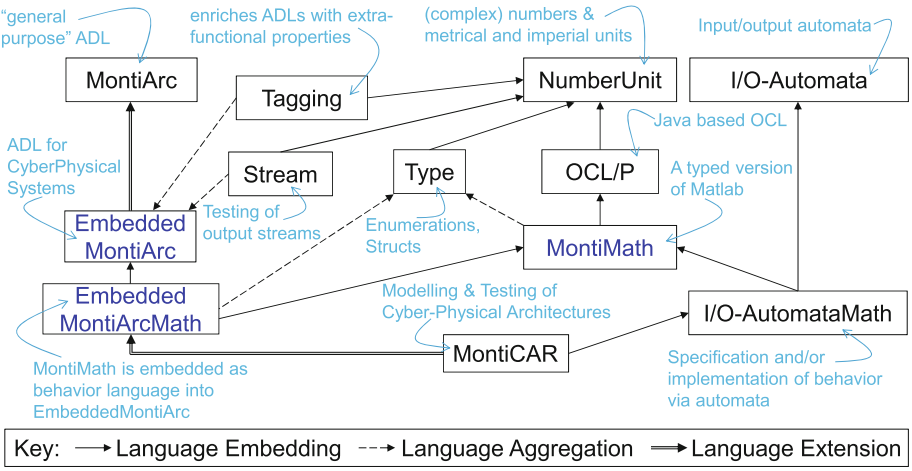

Fig. 2. MontiCAR modeling family.

```
min. value  stepsize  max. value                                   Type
1 struct GPS { Q(-90°:0.001°:90°) latitude;
2              Q(-180°:0.001°:180°) longitude; }   structure type with two attributes
enumeration
                Q=rational, C=complex, Z=whole numbers
3 enum UserFeedback { GREEN | YELLOW | ORANGE | RED }
4 type N1 = Z(1:oo);   defines type Natural numbers going from 1 to ∞
```

Fig. 3. Define new port types

struct members. We are not interested in the concrete realization of the scalars such as `int` or `float` on our level of abstraction. Instead, we specify, that each of the two members is a rational number, denoted by the letter `Q` according to the set of rational numbers $\mathbb{Q}$. The latitude coordinate can only take values from $-90°$ to $+90°$ whereas a valid longitude must lie between $0°$ and $180°$. We incorporate this range specification in brackets thereby declaring a new type. Moreover, GPS sensors only have a limited resolution. This is declared through the resolution parameter, here taking a value of $0.001°$. An accuracy can also be added to ranges; e.g. `Q(-90°:0.001°:90°)` $\pm$ `0.02°` says that values between $-90°$ to $+90°$ have an accuracy, normally distributed noise, of $0.02°$. Making such constraints explicit will later help us identifying incompatible components, e.g., sensors not providing the required signal resolution. The concrete type implementation is delegetated to the compiler allowing the system designer to focus on the functionality. In line 3 we use enumerations to declare a type in MontiCAR. It can take one of the four possible color values the feedback LED of a driver assistance system can emit. The core language of our family is `EmbeddedMontiArc` which extends the general purpose Architecture Description Language (ADL) `MontiArc` [17] used for modeling web and cloud services in a C&C like manner. `EmbeddedMontiArc`, explained in detail in the next section, supports in contrast to its base language `MontiArc` additionally port and component arrays, and it overwrites the type system of `MontiArc` in order to provide unit support and integrate the `Type` language. The `Tagging` language [27] enables the developer to enrich `EmbeddedMontiArc` models with extra-functional properties allowing for semantic consistency checks on C&C architectures. The `EmbeddedMontiArcMath` language enriches `EmbeddedMontiArc` models with the possibility to specify the behavior for atomic components by embedding `MontiMath` syntax which will be demonstrated in Sect. 7. `Stream` models, based on the stream theory of Broy and Rumpe [22], allow the definition of ordered sequences of input values for C&C input ports and the expected output sequences for all output ports to faciliate unit and integration testing of C&C models. Note that due to language aggregation `Stream` models have knowledge about the `EmbeddedMontiArcMath` models, but not vice versa. This allows deploying productive C&C models later-on without their respective test models. `I/O-Automata` is a language for describing behavior by finite automata. It provides internal variables, states and transitions pointing from a source to a target state. On activation, the automaton goes into the first start states for which the guard conditions are satisfied by the variables provided at the input ports. Moreover, transitions produce output values according to their defined output-port-assignment expressions and activate the automaton's target state. `I/O-AutomataMath` embeds the `MontiMath` language for describing guard conditions and output assignments using the Math syntax into the `I/O-Automata` language. `MontiCAR` extends `EmbeddedMontiArcMath` language allowing both Math and I/O-Automata behavior descriptions.

Since presenting the entire MontiCAR modeling family is out of the scope of this paper, the next sections will focus on the two most interesting family

members: `EmbeddedMontiArc` for describing CPS features and their interaction as C&C models and the `MontiMath` language for defining the features' behavior.

6 EmbeddedMontiArc

Based on our example introduced in Sect. 2 this section shows how the architecture of embedded and cyber-physical systems can be modeled with EmbeddedMontiArc language belonging to the MontiCAR language family.

The EmbeddedMontiArc model for the running example is given in Fig. 4. Line 1 defines the main component, having the `ParkAssistant` type and the derived instance name `parkAssistant`. Similar to Java's convention, all component types start with a capital letter and all component instances with a small one. Lines 2–5 define `ParkAssistant`'s in- and ll.6–7 out ports. An advantage of EmbeddedMontiArc's port arrays, `Z brakekForce[4]`, over MontiArc's solution with one port having a data type array, `Integer[] brakeForce`, is that each port in the port array can be wired-up individually. A port definition has the following structure: *direction*, can be `in` or `out` indicating incoming or outgoing data flow, *port type*, see paragraph **Type** in Sect. 5, *port name*, a small letter Java variable name, and an 1-dimensional *array size* (squared brackets). If the direction is missing such as in l.3, then the one of the previous port definition, here in l.2, is taken. The default value for missing array size is one.

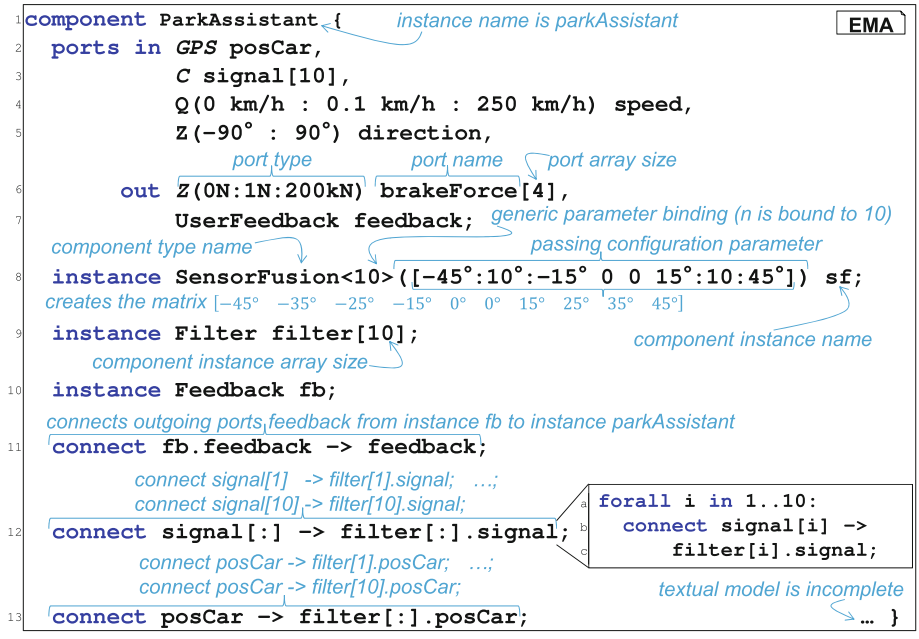

Fig. 4. Textual EmbeddedMontiArc model of `ParkAssistant`

All `ParkAssistant`'s subcomponent instances are listed in ll.8–10, each starts with the `instance` keyword, followed by a component type, instance name, an optional one dimensional array specifier. Note that in MontiArc which is the base language of EmbeddedMontiArc, the keyword `instance` is also `component`. But this ambiguity, `component` for component definitions and component instances, lead to confusion. `ParkAssistant` is decomposed into one `sf` of type `SensorFusion` (l.8), ten `filter` (l.9), one `fb` of type `Feedback` (l.10) instances. The singletons of type `EmbergencyBrake` and `BrakeActuator` in Fig. 1 are skipped in Fig. 4 for presentational reasons. `SensorFusion` component type has a generic parameter binding (in guillemot brackets) specifying the number of input signals needed for fusion, and a configuration parameter passing (in round brackets), specifying sensors' tilt angles. This configuration parameter is a 1×10 vector defined in a MATLAB-like syntax. The difference between a generic and a configuration parameter is that the primer changes the component's interface (has any impact on ports), whereas the latter has no influence on the interface and is only needed for the component behavior.

Lines 11–13 demonstrate the concrete syntax for interconnecting ports of subcomponent instances. While l.11 connects two ports using standard MontiArc syntax, in l.12 we have an EmbeddedMontiArc style interconnection of a port array with an array of components. Thereby, the colon notation, a short-form of `1:end`, selects all entities of the array and connects each entity to a corresponding entity on the right hand side separated by the `->` operator. Instead of the colon syntax, the `forall` syntax shown inside the box in ll.a–c can be used.

7 MontiMath Language

MontiMath is a mathematical matrix based behavior modeling language for MontiCAR. It is mainly inspired by MathJS[4], which supports matrices, units and rational numbers allowing one to solve linear equations exactly. MontiMath is based on MATLAB syntax. Since CPS in automotive and robotic domain mostly describe safety critical systems, MontiMath has - in contrast to existing matrix based languages, such as Modelica, Maple, MATLAB and MathJS - a very strict type system to minimize runtime errors. This type system includes **unit**, **dimension**, and **element ranges** information. Furthermore, it keeps track of **algebraic matrix properties** based on [19].

For variable assignments and matrix expressions MontiMath detects the following errors at compile-time: **Matrix Property Errors** occuring when a matrix violates the defined properties, e.g., diagonal, positive (semi)definite, lower/upper triangular, invertible, symmetric, or hermitian. For example, `diag inv Q^{3,3} A = diag([0km 1Km 2cm])` declares a rational 3×3 *diag*onal and *inv*ertible matrix variable `A`, being initialized with a diagonal matrix having values 0 km, 1 km and 2 cm on the main diagonal. This assignment results in a compiler error due to having a value of 0km on the main diagonal and thereby violating the *inv*ertible property. **Dimension Errors**

[4] http://mathjs.org.

occur during matrix assignments, e.g., `Q^{1,3} = [1 2]` and matrix infix operations, e.g., (element-wise) power, (element-wise) multiplication, summation and equation solving. **Unit Errors** occur when units of matrices are not compatible, e.g., in summation `[10cm 7cm] + [7kg 9kg]` and assignments. **Out-of-Bounds Erros** occuring by direct indexing of a non-existing matrix position. **Range Errors** occuring when an element in the matrix is not inside the allowed range or if a rational number is given but only integers are allowed.

Lines 6–11 in Fig. 5 show how MontiMath is applied in the EmbeddedMontiArc `SensorFusion` component to specify its behavior. Lines 7* and 8* are not part of the model but are displayed to show how the in and out ports of EmbeddedMontiArc are adapted to MontiMath matrix variable declarations. Line 9 defines a rational $n \times n$ matrix allowing values between `0` and `1`. The `*` operator in `cos*(tilt)` denotes that the cosine function is applied element-wise on `tilt`, returning the vector `[cos(tilt(1,1)), ..., cos(tilt(1,n))]`. The `diag` function creates a diagonal matrix with the elements of `tilt` on its main diagonal; the result is assigned to `facMatrix`. Line 11 multiplies the `distance` vector with this `facMatrix` resulting in a $1 \times n$ column vector with its minimum value assigned to the output port `mergedDistance`. Assume, in a previous software evolution step l.10 was `sin*(diag(tilt))` and now needs to be replaced by `cos*` due to a change in the sensors' relative coordinate system. Since `sin(0°) = 0` the element-wise sine of a diagonal matrix is again a diagonal matrix; replacing the sine with a cosine however results in a MontiMath compiler error due to `cos(0°)` $\neq$ `0` making the result non-diagonal. Without the

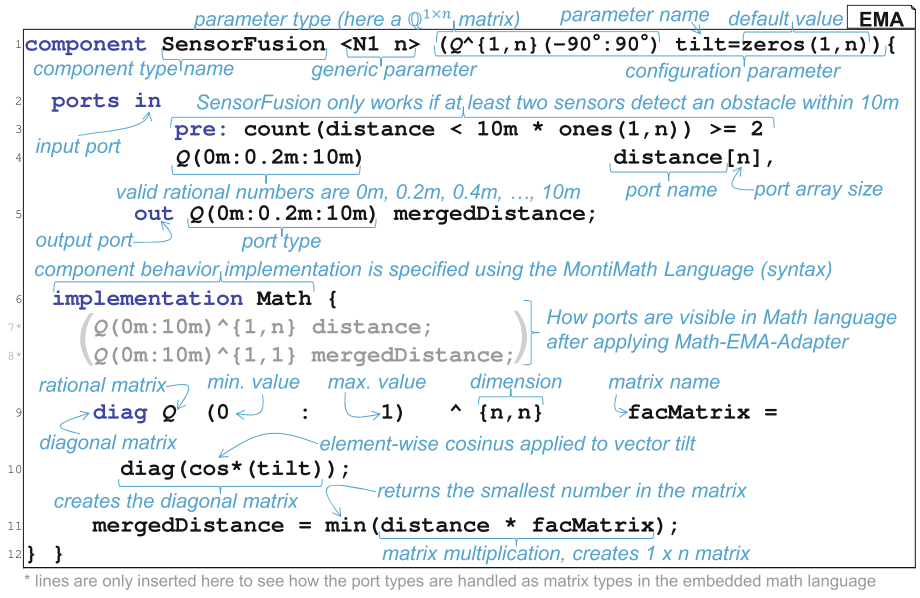

Fig. 5. SensorFusion component with its behavior defined in MontiMath

strong type system and meaningful error messages such a bug may be detected later during tests or at runtime thereby increasing development costs.

Line 2 in Fig. 6 shows the OCL/P pre-/post condition used by MontiMath's type checker to find out that `cos(0°)=1` and therefore not zero, and ll.5–7 defines the `diag` matrix property in OCL/P requiring all off-diagonal values to be zero which is violated by the expression `cos*(diag(tilt))`. Using the `implies` keyword in l.5 we require that the `diag` property guarantees the `lowerTriangular` and `upperTriangular` properties to be true. This correctness of this specification is then proven by the Z3 [12] solver. Furthermore, already existing matrix properties can be reused via (multiple) inheritance, i.e., the `diag` property could have been defined by `matrix-property<N1 n> diag Q^{n,n} y extends upperTriangular & lowerTriangular`, thereby saving l.7. Lines 8–13 show two other important concepts of MontiMath: Operator overloading based on matrix properties, and function overloading based on matrix dimensions. The first one is used to define more efficient algorithms for special matrix types, as it is done for adding two diagonal matrices in ll.9–11 where only the diagonal elements of the two matrices are added reducing the computational complexity for matrix addition from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$ for diagonal matrices. Calling the function defined in l.12, `diag(a)` in l.10 returns a vector containing the matrices' diagonal elements; while the outer `diag` having a row vector as its argument creates a diagonal matrix by invoking the function defined in l.13. The decision which function to invoke is based on

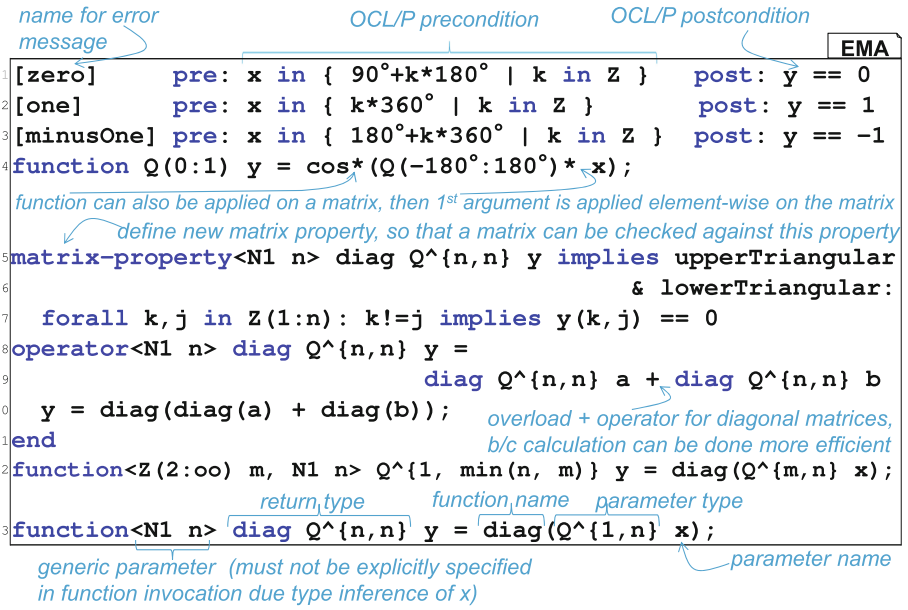

Fig. 6. Function definition in math language (";" after function definition is here used to omit the body implementation of the function.)

the matrix dimensions. For this reason the type of the first generic parameter in l.12 cannot be `N1`, and therefore it is `Z(2:oo)`. Otherwise the compiler cannot infer which overloading of the `diag` function (ll.12–13) should be invoked in case of a row vector.

The concept of overloading functions based on matrix dimensions is taken from MATLAB. In contrast, our approach makes the overloading with two functions having different generic types explicit. Moreover, in l.13 in Fig. 6, the return type has the return type name `y`, since MontiMath supports, similar to MATLAB, multiple return values and therefore each return value has always a name to differ between multiple return values.

8 Conclusion

We presented the modeling language family MontiCAR for the design of cyber physical system encorporating requirements we derived in multiple case studies. The core of the language family is the architecture description language EmbeddedMontiArc. This core language was extended by a stricter typing than is usually known from modern languages. It includes a unit system, value domain and resolution support. Furthermore, the math language MontiMath is an integral part of the language family which allows for efficient component behavior descriptions. The simple Type language facilitates the aggregation and reuse of data packages without overwhelming the user with unnecessary features such as pointers and inheritance. Finally, a stream language allows a straightforward definition of input and output data flows for the definition of test cases.

The language family was evaluated on a simplified example from the automotive domain. Thereby, it was shown how the integrated language concepts support the system developer helping him to focus on the functionality of the system instead of implementation details.

A large scale evaluation is subject of ongoing and future work. This evaluation comprises the modeling of driver assistance systems and autonomous vehicle components such as a sensor fusion, a planning system, a controller unit, and a wireless communication system as well as the integration of these modules into a working software architecture. To validate the resulting system, a series of use cases needs to be defined and simulated in a virtual environment.

Acknowledgements. This research was supported by a Grant from the GIF, the German-Israeli Foundation for Scientific Research and Development, and by the Grant SPP1835 from DFG, the German Research Foundation.

References

1. Allen, R.J.: A formal approach to software architecture. Technical report (1997)
2. Aravantinos, V., Voss, S., Teufl, S., Hölzl, F., Schätz, B.: AutoFOCUS 3: tooling concepts for seamless, model-based development of embedded systems. In: ACES-MB (2015)

3. Ashenden, P.J.: The Designer's Guide to VHDL, vol. 3. Morgan Kaufmann, San Francisco (2010)
4. Association, M., et al.: Modelica Language Specification. Linköping, Sweden (2005)
5. AUTOSAR: layered software architecture. Technical report 053 (4.3.0), AUTOSAR (2016)
6. AUTOSAR: modeling guidelines of basic software EA UML model. Technical report 117 (4.3.0), AUTOSAR (2016)
7. Azumi, T., Yamamoto, M., Kominami, Y., Takagi, N., Oyama, H., Takada, H.: A new specification of software components for embedded systems. In: ISORC (2007)
8. Blanc, N., Kroening, D., Sharygina, N.: Scoot: a tool for the analysis of SystemC models. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 467–470. Springer, Heidelberg (2008). doi:10.1007/978-3-540-78800-3_36
9. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv symbolic model checker. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 334–342. Springer, Cham (2014). doi:10.1007/978-3-319-08867-9_22
10. Cheng, S.T., York, G., Brayton, R.K.: Vl2mv: A compiler from verilog to blif-mv. HSIS Distribution (1993)
11. Dashofy, E.M., van der Hoek, A., Taylor, R.N.: A highly-extensible, xml-based architecture description language. In: Conference on Software Architecture (2001)
12. Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). doi:10.1007/978-3-540-78800-3_24
13. Dormoy, F.X.: Scade 6: a model based solution for safety critical software development. In: ERTS 2008, pp. 1–9 (2008)
14. Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity-the Ptolemy approach. Proc. IEEE **91**(1), 127–144 (2003)
15. Texas Instruments of Electrical and Electronic Engineers: Standard VHDL language reference manual. IEEE Std (1988)
16. Elmqvist, H., Mattsson, S.E., Otter, M.: Modelica-a language for physical system modeling, visualization and interaction. In: Proceedings of the 1999 IEEE International Symposium on Computer Aided Control System Design, pp. 630–639. IEEE (1999)
17. Haber, A.: MontiArc - Architectural Modeling and Simulation of Interactive Distributed Systems. Shaker Verlag (2016)
18. Herbert, J., Dutertre, B., Riemenschneider, R., Stavridou, V.: A formalization of software architecture. In: FM (1999)
19. Horn, R.A., Johnson, C.R.: Matrix Analysis. Cambridge University Press, Cambridge (2012)
20. Instruments, N.: BridgeView and LabView: G Programming Reference Manual. Technical report 321296B–01, National Instruments (1998)
21. Kaufmann, M., Kornerup, J., Reitblatt, M.: Formal verification of LabVIEW programs using the ACL2 theorem prover. In: ACL2 (2009)
22. Klein, C., Rumpe, B., Broy, M.: A stream-based mathematical model for distributed information processing systems-the SysLab system model. In: Formal Methods for Open Object-based Distributed Systems (1997)
23. Lemke, J.: C++-Metaprogrammierung: Eine Einführung in die Präprozessor-und Template-Metaprogrammierung (2016)

24. Logozzo, F.: Cibai: an abstract interpretation-based static analyzer for modular analysis and verification of Java classes. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 283–298. Springer, Heidelberg (2007). doi:10.1007/978-3-540-69738-1_21
25. Luckham, D.C., Vera, J.: An event-based architecture definition language. IEEE Trans. Softw. Eng. **21**(9), 717–734 (1995)
26. Maoz, S., Ringert, J.O., Rumpe, B.: Verifying component and connector models against crosscutting structural views. In: ICSE (2014)
27. Maoz, S., Ringert, J.O., Rumpe, B., von Wenckstern, M.: Consistent extra-functional properties tagging for component and connector models. In: ModComp (2016)
28. Mathworks: simulink user's guide. Technical report R2016b, MATLAB & SIMULINK (2016)
29. Nikitin, P.V., Shi, C., Wan, B.: Modeling partial differential equations in VHDL-AMS (mixed signal systems applications). In: SOC (2003)
30. OMG: UML profile for MARTE: modeling and analysis of real-time embedded systems. Technical report Version 1.1, OMG Group (2011)
31. OMG: OMG systems modeling language (OMG SysML). Technical report. Version 1.4, OMG Group (2015)
32. Rumpe, B.: Modeling with UML: Language, Concepts, Methods. Springer, Heidelberg (2016)
33. Rumpe, B., Schulze, C., von Wenckstern, M., Ringert, J.O., Manhart, P.: Behavioral compatibility of simulink models for product line maintenance and evolution. In: SPLC (2015)
34. IEEE Computer Society: IEEE standard for standard systemc® language reference manual (2012)
35. Chromatography software: UNICORN 5.0 - User Reference Manual. Technical report 03–0014-90 (2004)
36. Accellera Systems Initiative: Verilog-AMS Language Reference Manual. Technical report 2.4.0, Accellera Systems Initiative standards (2014)