2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)

[HKR21a] M. Heithoff, E. Kusmenko, B. Rumpe:
Synchronous Execution Semantics for Component & Connector Models.
In: S. Bin Abid, R. Oliveira, I. Ober, editors, Companion Proceedings of MODELS 2021. Workshop MoDeVVa, 264-273, ACM/IEEE, Oktober 2021
www.se-rwth.de/publications/

Synchronous Execution Semantics for Component & Connector Models

Malte Heithoff, Evgeny Kusmenko, Bernhard Rumpe
Chair of Software Engineering, RWTH Aachen University, Aachen, Germany, {heithoff,kusmenko,rumpe}@se-rwth.de

***Abstract*—Component-and-connector languages such as Simulink and LabView are widely used in research and industry to model complex technical systems from a dataflow-based perspective. In safety-critical domains such as automotive and avionics, where human lives depend on the correctness of the software, a clear modeling language semantics leaving no room for ambiguities is crucial. In this paper, we present a synchronous component execution model enabling us to model embedded systems in a precise way guaranteeing that the system is executable within a limited amount of time and has a bounded discretization error for all possible inputs.**

***Index Terms*—MDSE, CBSE, CPS, execution semantics, code generation**

I. Introduction

In engineering disciplines such as automotive, avionics, and control, systems are often designed using the component-and-connector (C&C) paradigm, where functional blocks encapsulate side effect-free behavior and connectors make the dataflows between the interfaces of these blocks explicit. The hierarchical (de)composition principle of C&C models facilitates the organization of complex systems, allowing dedicated teams to work on specialized subcomponents and to integrate their results into a large architecture. Examples of architecture description languages (ADLs) implementing this paradigm are Simulink, LabView, and Architecture Analysis & Design Language (AADL), to name a few. Although the basic principles of C&C ADLs often seem straightforward at first sight, the concrete meaning of a C&C model depends on the language used. Often C&C languages provide only a vague idea of their semantics and leave room for interpretations, which can be fatal in safety-critical domains, where human lives depend on the correctness of the software. During the design of a semantics for a C&C-based language there is a multitude of nuances and interpretation variants to consider. What is more, the desired behavior may differ from to domain to domain. For instance, in distributed systems, an asynchronous, strongly causal semantics with delays and, possibly, message losses is often the way to go. On the other hand, for the design of algorithms which are meant to be executed on a single CPU, a synchronous, weakly causal modeling language such as Simulink is preferable.

The main contribution of this work is the definition of a FOCUS-based [1] and Simulink-inspired weakly synchronous execution semantics for a cyclic execution of C&C models covering algebraic and differential loops. The execution model is implemented for and evaluated based on the C&C-based EmbeddedMontiArc (EMA) ADL.

This work was supported by the Grant SPP1835 from DFG, the German Research Foundation.

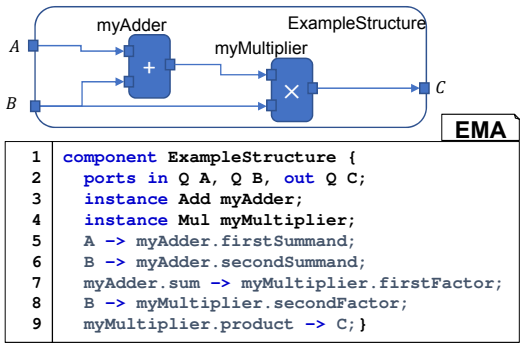

Fig. 1. An example C&C component structure and the corresponding textual EMA component definition computing the formula $C = (A + B) \times B$ in each execution cycle by interconnecting two atomic subcomponents.

II. Background

EMA is a textual C&C ADL designed with the needs of engineering domains such as automotive in mind [2], [3]. Due to its open-source availability and extensibility we are going to use EMA as the basis for the implementation and evaluation of the concepts presented in this paper. For this reason, we are going to present the foundations of EMA in this section.

Consider a simple application receiving the two numbers A and B as input and computing the output as $C := (A+B)\times B$ in each execution step. We can specify this behavior according to the C&C paradigm by using an adder and a multiplier component and interconnecting them accordingly. The graphical structure and the corresponding textual EMA representation are given in Figure 1.

The component header in L.1 starts with the keyword `component` and is followed by the component type name, which can be used to instantiate a component instance in other components, similarly to class names in object-oriented languages. If needed the component name can be followed by a list of component parameters which are used at component instantiation, similar to constructor parameters in Java or C++. The interface of an EMA component is defined by input and output ports, cf. L.2 in Figure 1. The port kind is set using the

978-1-6654-2484-4/21/$31.00 ©2021 IEEE
DOI 10.1109/MODELS-C53483.2021.00047

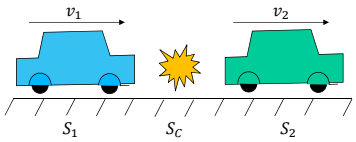

Fig. 2. The sketch shows two autonomous vehicle at their respective positions s_1 and s_2, driving at velocities v_1 and v_2. The autopilot of each vehicle needs to estimate whether and when there will be a collision under the assumption that the current velocities remain constant.

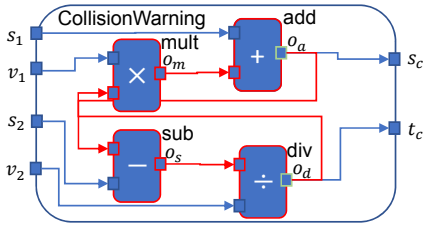

Fig. 3. A C&C architecture representing the algebraic equation system of the collision example as formulated in Equations (3) and (4).

keyword `in` or `out`. This is followed by the port's type and a unique port name. EMA uses an abstract type system, which is based on frequently used mathematical sets $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, and $\mathbb{C}$. The corresponding EMA types are `N`, `Z`, `Q`, `C`, representing natural numbers, integers, rationals, and Gaussian rationals (complex numbers with rational components), respectively. Additionally, the type `B` represents Booleans. These types are obviously for specification only. The concrete realization is delegated to the compiler and can vary, e.g. with respect to accuracy, depending on the application domain. A basic type can be combined with a unit of the physical quantity it represents and constrained by a range and step size, e.g. `Z(-3kg:2kg:3kg)` represents weights of -3,-1,1, and 3 kg. Furthermore, a basic type can be extended to a vector or a matrix by appending the dimensionality, e.g. `Q^{2,3}` represents a 2×3 matrix of rationals.

The two subcomponent instances of type `Add` and `Mul` are instantiated in L.3 and L.4 of Figure 1, respectively. The instance names can then be used to interconnect their ports using the connect operator `->` in L.5-9. For instance, in L.5 we connect the main component's input port `A` to the `firstSummand` port of the `myAdder` component. Consequently, inputs at `A` are automatically forwarded to `myAdder.firstSummand`.

The behavior of atomic components, i.e. components, which cannot be decomposed into subcomponents such as our adder and multiplier, can be specified in the implementation block of the respective component definition using the MATLAB-like matrix-oriented but strongly typed language MontiMath or as a deep neural network [4]–[6]. Furthermore, EMA provides a set of frequently used arithmetic and logical atomic components out of the box.

III. Running Example & Requirements

We are now going to introduce two running examples, which will serve as a basis for the requirements as well as the presentation of the main concepts. First, consider the example in Figure 2, where two vehicles drive on the same lane, i.e. there is only one spatial dimension. In cooperative automated driving, the autopilot software of a vehicle might need to compute the time until collision with another traffic participant and the collision position under the highly simplified but illustrative assumption that both vehicles maintain their current velocity and direction. We can formulate the problem as a linear algebraic equation system with the independent variables s_1, v_1, s_2, v_2 representing the current position and velocity of vehicle 1 and vehicle 2, respectively, and the variables of interest s_c and t_c representing the estimated collision position and time to be computed by the autopilot model:

$$s_c = s_1 + v_1 \cdot t_c \tag{1}$$
$$s_c = s_2 + v_2 \cdot t_c. \tag{2}$$

To be able to model the input/output behavior of this equation system, we need to solve it for s_c and t_c, respectively, which results in the following formulas:

$$s_c = s_1 + v_1 \cdot t_c \tag{3}$$
$$t_c = \frac{s_c - s_2}{v_2}. \tag{4}$$

Now, using the basic blocks for addition, multiplication, and division we can model the equation system as a C&C model, cf. Figure 3. Note that the interdependency of the two output ports from each other results in a structural loop in the C&C model. An ordered execution of the subcomponents in a procedural manner to compute the desired output as is done by Simulink is not feasible any more. The given model is more of a specification defining the input/output relationships, but abstaining from giving an explicit instruction how to realize these relationships. For code generation and execution the algebraic equation system spanned by this model needs to be solved. Since our target domains include safety-critical systems, a closed solution of the equation system needs to be found at compile-time (solving an equation system at runtime has the advantage that problems can be solved numerically if there is no closed form solution for the problem class; however, the amount of time needed to find a solution numerically can vary drastically and there is no guarantee that a solution exists for a given input at all, which can lead to undefined behavior). The input domain for which a solution exists needs to be derived at compile-time. In our one-dimensional example, a collision only takes place if $v_1 \neq v_2$. If $v_1 - v_2 = 0$, there is no collision and the ordinary equation does not have a solution. For this case we need to be able to define an alternative behavior. When using runtime solvers, an alternative behavior can also be used if the solver does not come up with a solution within a specified amount of time.

While algebraic systems are an important means to describe static relationships, differential equation systems are inevitable

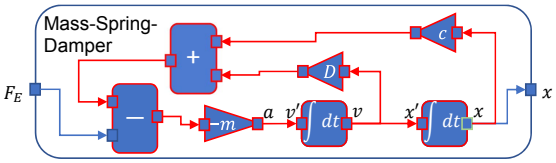

Fig. 4. A C&C architecture representing the ordinary differential equation system of a mass-spring-damper model, as formulated in Equations (5) to (8).

in dynamic systems and process modeling (in discrete systems: difference equation systems) and are, therefore, widely used in engineering. A prominent example is the mass-spring-damper model, which is used in various applications [7], [8] including automotive actuation mechanisms such as fuel injectors [9] as a basis for simulation and model predictive control (MPC). The formulation as an ordinary differential equation (ODE) is given as follows:

$$F_E - F_M - F_D - F_C = 0 \tag{5}$$
$$\Leftrightarrow F_E - a \cdot m - D \cdot v - c \cdot x = 0 \tag{6}$$
$$\Leftrightarrow F_E - x'' \cdot m - D \cdot x' - c \cdot x = 0 \tag{7}$$

with

$$a = v' = x'' = (D \cdot v + c \cdot x - F_E)/(-m). \tag{8}$$

The corresponding C&C model is given in Figure 4. Again, there is no linear dependency between the input and the output of the system. The interdependency between x and its derivatives requires a circular component structure. Note that the system has an inherently continuous formulation, as derivatives and integrals only exist in continuous domains (in our case the domain is time). Due to the discrete nature of software systems, this model does not have an exact software implementation. To realize this model in software it obviously needs to be discretized first. The way continuous models like this one are mapped to discrete approximations becomes part of the language semantics.

The above considerations result in the following set of requirements, which we are going to tackle in our concept presentation: **(R1) Unambiguity:** The meaning of a model must be unambiguous. The semantics must define exactly, how a valid model will map streams of inputs to output ports. The compiler must be able to implement the semantics of a given model up to some tolerances, e.g. emerging from time and value discretization. **(R2) Robustness:** It must be ensured at compile-time that a model can map given input values to output ports at any time. If a mapping is undefined for parts of the domain of an input variable, e.g. if a specific input value leads to a division by zero, this must be identified at compile-time and an alternative behavior must be specified by the developer. The language must ensure this. **(R3) Structural loops:** The modeling methodology must enable the developer to define circular component structures representing algebraic, differential, or difference equations. Such structures need to be identified and solved at compile-time. Optionally, it should be possible to postpone the solution of such equation systems to runtime. In this case it must be ensured that a) the solution remains within guaranteed tolerance bounds; b) the solution is available within a predefined amount of time *or* c) an alternative behavior is provided.

IV. Related Work

In this section, we are going to discuss the semantics of different C&C languages and frameworks and the ways the models are executed.

a) FOCUS: FOCUS is a formal specification methodology developed based on the stream theory [1]. It is particularly suited for modeling distributed safety-critical systems and uses the stream theory to describe the communication of components. Components are defined on infinite streams of messages M,I,O with $\mathrm{M}^{\omega} = \mathrm{M}^{*} \cup \mathrm{M}^{\infty}$. Each component is defined through a function

$$\mathrm{f} : \mathrm{I}^{\omega} \rightarrow \mathrm{O}^{\omega}. \tag{9}$$

We can also have several channels on a component (input ports and output ports)

$$\mathrm{g} : \mathrm{I}_1^{\omega} \times \mathrm{I}_2^{\omega} \times \mathrm{I}_3^{\omega} \rightarrow \mathrm{O}_1^{\omega} \times \mathrm{O}_2^{\omega}. \tag{10}$$

For $(\mathrm{in}_1, \mathrm{in}_2, \mathrm{in}_3) \in \mathrm{I}_1 \times \mathrm{I}_2 \times \mathrm{I}_3$ the output of a component is then defined as $(\mathrm{out}_1, \mathrm{out}_2) = \mathrm{g}(\mathrm{in}_1, \mathrm{in}_2, \mathrm{in}_3)$.

With the introduction of time, FOCUS defines timed weakly causal and timed strongly causal components. In timed strongly causal components there is always a delay between the input and the output, while timed weakly causal components can also provide an instant mapping.

An implementation of FOCUS is given by AutoFOCUS 3, which supports the whole development process from requirement specification to deployment [10]. Although FOCUS provides a very formal approach, it does not fully fulfill our requirement for structural loops. FOCUS allows loops only in strongly causal systems, which means those loops are interrupted by some time delaying component.

b) MathWorks Simulink: Simulink [11] is a graphical C&C modeling framework based on MATLAB. It is widely used in industry and academia, particularly in engineering and is therefore close to our target domain. Simulink maintains an execution order for its subcomponents, which is computed at compile-time by flattening the hierarchical structure. The execution order of a C&C model manages the order in which C&C components are executed. A component is said to be direct-feedthrough if it computes outputs on the current input.

The semantic definition is mainly defined in the user manual [11] and reference web pages. However, the information is sometimes vague or even contradicting. For instance, the algebraic loops manual[1] states that nondirect-feedthrough blocks maintain a state variable. This is contradictory to the defintion

[1] https://de.mathworks.com/help/simulink/ug/algebraic-loops.html

given in integration considerations[2] stating that in nondirect-feedthrough blocks the value of the output signal does not depend on the value of the input signal in at least one function during the simulation. The latter matches our definition. We assume that this conflict was due to deprecated reference manual pages.

Mathworks makes the claim [12] that nondirect-feedthough blocks can appear anywhere at the start of the execution order. We show in a simple example why this is not true. Figure 5 shows an incorrect execution order which satisfies this claim. The delay block on the left needs to wait for the current output of the delay block on the bottom to update its state properly. Instead it appears in an arbitrary position 1. Hence, the semantics of Simulink is not always predictable and violates our unambiguity requirement (R1).

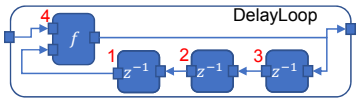

Fig. 5. Loop of *Delay* components/blocks

In contrast to FOCUS Simulink handles algebraic loops and interprets them as a form of equation systems expressible in a semi-explicit form and solves them at runtime using numeric solvers [13]. Simulink uses either *Trust Region* [14] or *Line Search* [15] for solving nonlinear equation systems. If the solver fails for one of the methods, a user can still try to use the other. Differential algebraic equations (DAEs) are solved with the method described by Shampine et al. [16]. It shows that every equation system built from a direct-feedthrough loop can be expressed in the semi-explicit form. It utilizes the underlying MATLAB suite which provides numerous ODE and DAE solvers adaptable to the current situation. If solving a loop fails, the system breaks at runtime, which violates our robustness and structural loops requirements (R2) and (R3). Furthermore, Simulink does not offer dynamic block reconfiguration to handle errors. If a runtime error such as failing to solve algebraic loop occurs, it is not possible to switch to an alternative execution model. Moreover, algebraic loops are only supported at simulation time and are not compiled to executable code.

c) Others: A feedback loop in LabVIEW using an *Algebraic Variable Function* is interpreted as a DAE as well. It uses a Backward Differential Formula (BDF) method to solve the DAE which supports error control. In addition, LabVIEW provides and calculates an execution order similar to Simulink. Similarly to Simulink, LabVIEW has a range of numerical ODE solvers that are able to control the error. Again, LabVIEW lacks the ability to recover from failures.

Furthermore, Shampine et al. prove a method for finding consistent initial conditions by applying a BDF step from $t - \Delta t$ [16]. This method is applicable to every DAE of order 1.

[2]https://de.mathworks.com/help/simulink/ug/integration-considerations-for-matlab-function-blocks.html

Mosterman et al. analyse artificial algebraic loops and come up with automated approaches to resolve those [17] . They range from executing the nonvirtual components several times to split up the execution function for each of the underlying subcomponents.

Our semantics is inspired by both FOCUS and Simulink, but focuses on fulfilling the three requirements defined earlier. The prototype implementation compiles code that satisfies these requirements.

V. Execution Semantics

C&C models for cyber-physical systems (CPSs) are built with the intent to design a dataflow which is processed by the components and directed by the connectors. This means we have data which enters the system (e.g. via sensors) and which is processed to produce an output (e.g. for actuators) for each time step. In this section we introduce a synchronous and weakly causal execution semantics for such models. The presented semantics is used in the EMA language family, the open source implementation is available on Github[1].

EMA semantics is defined on atomic components and the composition of such components for each time step. Here, the domain of time t is the discrete set $\{0, 1 \cdot \Delta t, 2 \cdot \Delta t, \dots\}$ with Δt being the system's sampling rate.

A component maintains input and output ports defining its interface and inner state variables. For an atomic component c we denote the set of input ports as c_{in}, the set of output ports as c_{out} and the set of state variables as c_{state}. For each input port $i_k \in c_{in}$ its value at a given time step t is $i_k(t)$ while $c_{in}(t)$ denotes the values for all input ports at time t. Same applies for the output ports $o_j \in c_{out}$ and state variables $s_l \in c_{state}$, respectively. We denote the respective domains as $I_k = \mathrm{dom}(i_k), O_j = \mathrm{dom}(o_j)$ and $S_l = \mathrm{dom}(s_l)$.

A component's behavior is defined through its implementation which sets the output in relation with the current input and current state. An implementation is a series of assignments which computes the output on the current input and updates the state variables. This defines a set of functions for both the calculation of the output port values and the calculation of the next state variable iteration. Both are computed on the inputs and current inner state at time step t. We define an output function $\lambda_c^{o_j}$ for each output port and an update function $\phi_c^{s_l}$ for each state variable s_k on the interface

$$\lambda_c^{o_j} : I_1 \dots I_m \times S_1 \dots S_n \to O_j \quad \forall o_j \in c_{out} \tag{11}$$

$$\phi_c^{s_l} : I_1 \dots I_m \times S_1 \dots S_n \to S_l \quad \forall s_l \in c_{state}. \tag{12}$$

Consequently, the values of the output ports and state variables at time t are

$$o_j(t) = \lambda_c^{o_j}(c_{in}(t), c_{state}(t)) \quad \forall o_j \in c_{out} \tag{13}$$

$$s_l(t + \Delta t) = \phi_c^{s_l}(c_{in}(t), c_{state}(t)) \quad \forall s_l \in c_{state}. \tag{14}$$

$\lambda_c^{o_j}$ and $\phi_c^{s_l}$ are deterministic functions which calculate the same output for the same set of inputs. Those can range from the simple implementation of an addition operation to the

[1]https://github.com/MontiCore/EmbeddedMontiArc/

highly complex definition of an MPC optimization [18]. Note that for the initial time step the inner state variables $s_l(t=0)$ have to be provided by the modeler. We can obtain these separated functions from the component's implementation block simply by executing it, setting the according values and then resetting the state variables. This way all functions operate on the same set of inputs and state and compute the corresponding result. That allows for more control over the system execution as well as for a more detailed definition of the mathematical semantics. For simplification, assuming we are only interested in the current time step we now drop the notation of time and interpret c_{out}, c_{in} and c_{state} as the variables at this point in time. Later, we will drop the notation of the corresponding port or state variable if there is only one present, e.g. λ_{comp} instead of λ^{o}_{comp}. We obtain the following notation:

$$o_j = \lambda_c^{o_j}(c_{in}, c_{state}) \quad \forall o_j \in c_{out} \tag{15}$$
$$s_l \leftarrow \phi_c^{s_l}(c_{in}, c_{state}) \quad \forall s_l \in c_{state}. \tag{16}$$

For instance, the component in Figure 6 is defined by

$$o_1 = \lambda^{o_1}_{comp}(i_1, i_2, i_3, s_1, s_2) \tag{17}$$
$$o_2 = \lambda^{o_2}_{comp}(i_1, i_2, i_3, s_1, s_2) \tag{18}$$
$$s_1 \leftarrow \phi^{s_1}_{comp}(i_1, i_2, i_3, s_1, s_2) \tag{19}$$
$$s_2 \leftarrow \phi^{s_2}_{comp}(i_1, i_2, i_3, s_1, s_2). \tag{20}$$

The output and update functions could be the mathematical implementations

$$\lambda^{o_1}_{comp} : o_1 = i_1 \cdot i_2 + s_1 \tag{21}$$
$$\lambda^{o_2}_{comp} : o_2 = \max(i_3, s_2) \tag{22}$$
$$\phi^{s_1}_{comp} : s_1 \leftarrow s_1 + 1 \tag{23}$$
$$\phi^{s_2}_{comp} : s_2 \leftarrow s_2 + 1. \tag{24}$$

When executing the system, we realize this semantics for atomic components by setting the current values to the input ports and executing the code for its behavior implementation. The new output values are set to the output ports and the component's inner state is updated. For the component *comp* shown above and input values $i_1 = 3, i_2 = 2, i_3 = 4$ and state variable values $s_1 = 1$ and $s_2 = 2$, this would lead to the execution shown in Figure 6.

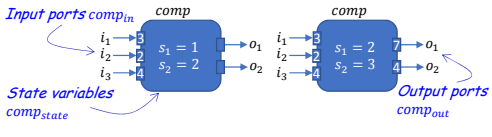

Fig. 6. Executing the behavior implementation of an atomar component with input ports i_1, i_2 and i_3, the output ports o_1 and o_2, the inner state variables s_1 and s_2 and their values.

We can refine the definition of the output and update function by acknowledging the fact that not all functions are dependent on all inputs. In fact, some output functions do not depend on any inputs. Since $\lambda^{o_1}_{comp}$ directly depends on the input values for i_1 and i_2, in order to be able to get valid results, the values for the two input ports must be present before computing. Such a function is called *direct-feedthrough*. In contrast, an output function $\lambda_c^{o_j}$ which is only dependent on its inner state values is called *nondirect-feedthrough*. Such output behavior of a component c can then be defined through a function on the interface

$$\lambda_c^{o_j} : S_1 \dots S_n \rightarrow O_j \tag{25}$$
$$o_j = \lambda_c^{o_j}(c_{state}). \tag{26}$$

Normally, components which are defined by *nondirect-feedthrough* functions compute their state update on the current input values. Examples for *nondirect-feedthrough* components are the `Constant` component or `Delay` component. The `Constant` component does not have an input port at all and always outputs the same value. The `Delay` component delays the input by one time step and hence only the update function is dependent on the current input. We will use this property later when defining the order in which the components need to be executed. The goal for our semantics of a component

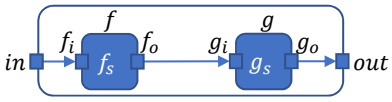

Fig. 7. Component composition of the two components f with input port f_i, output port f_o and state variable f_s and g with input port g_i, output port g_o and state variable g_s.

composition as seen in Figure 7 is an execution of the output and update functions which results in the computation for the current time step. In the mathematical sense, we can put all output functions in one equation system which needs to be solved at each time step. Every function introduces one equation with exactly one new variable, the output port. State variables are given at each time step and input port values can be derived with the connector equations:

$$out = g_o \tag{27}$$
$$g_o = \lambda_g^{g_o}(g_i, g_s) \tag{28}$$
$$g_i = f_o \tag{29}$$
$$f_o = \lambda_f^{f_o}(f_i, f_s) \tag{30}$$
$$f_i = in. \tag{31}$$

Equations (28) and (30) are introduced by the output functions and Equations (27), (29) and (31) are the connector equations. In general, for a component b the current value for an input port $b_{in_k} \in b_{in}$ can be derived by the equation built by an incoming connector $(a_{out_j}, b_{in_k}) \in \text{connectors}_{EMA}$ for which a_{out_j} is the source of b_{in_k}. For component g the values for the input port g_i can be derived by replacing it with the corresponding connected output port f_o leading to a more compact version of the equation system

$$out = g_o \tag{32}$$
$$g_o = \lambda_g^{g_o}(f_o, g_s) \tag{33}$$
$$f_o = \lambda_f^{f_o}(in, f_s). \tag{34}$$

The actual execution of the semantics on a component composition is realized by computing the components' output functions sequentially and transporting the output values over the connectors to the input ports of the target components. For instance, Figure 8 shows a single execution iteration of

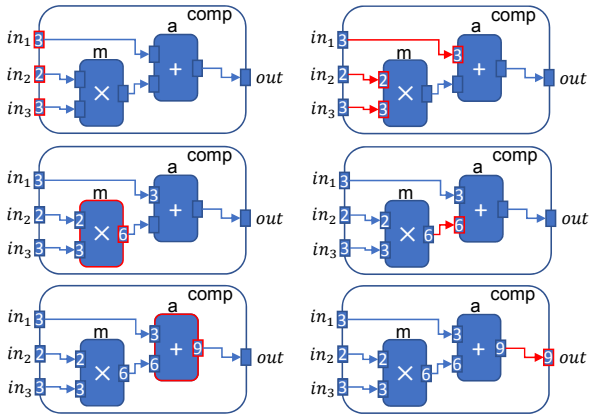

Fig. 8. Left to right, top to bottom: One execution iteration of the component composition of the addition component a and the multiplication component m with the input values $in_1 = 3$, $in_2 = 2$ and $in_3 = 3$ leading the the overall result $out = 9$.

a simple component composition of an addition component `a` and a multiplication component `m`. First, the input values $(3, 2, 3)$ are transported along the connectors to the input ports of `a` and `m`. Thereafter, all input values for `m` are present and λ_m can be executed. The result is used in λ_a which produces the value for output port out.

We can easily figure why the order of execution here is highly important and must not be mixed up. If we executed component `a` in step 2 instead of component `m` we run into errors. The value for the second input port is not set yet and we could only either assume a default value on that port or take the value of a previous execution iteration. The value for out would be independent of the result of `m` and the overall result would be wrong according to the semantics we defined earlier. To face this very problem we define an *execution order* consisting of the operations for transporting values along connectors, calculating a component's output with its output functions λ_c^{oj}, and calculating a component's next state variables with its update fuctions ϕ_c^{si}. To implement the semantics definition of atomic components and component compositions and to compute the correct results, the *execution order* has to satisfy the following conditions:

- Before executing the output function of a *direct-feedthrough* component, all input values of the current time step must be given.
- Before executing the update function of a component, all input values of the current time step must be given.
- The update function is executed after the component's output function.

The first and third condition are responsible for the correct execution of each atomic component output definition so that it computes on the input and inner state values of the current time step. The second condition is responsible for the correct execution of the update function on the current input values. Note that the output functions of a *nondirect-feedthrough* component can be executed at any time before the component's update functions. An execution order that satisfies those three conditions is correct with respect to the semantics defined earlier. Such an execution order is called valid. For the example above a valid execution order as seen in Figure 8 would be:

1 Transport the input values along the connectors.
2 Execute λ_m and transport the output value along the connector.
3 Execute λ_a and transport the output value along the connector to out.

An invalid execution order would be:

1 Transport the input values along the connectors.
2 Execute λ_a and transport the output value along the connector to out.
3 Execute λ_m and transport the output value along the connector.

A valid execution order solves the output function equation system. Whenever an output function is ready to be executed according to the execution order, all input values are present und thus the function computes the correct mathematical result. So for Figure 8 λ_s computes the correct system output. This is equivalent to substituting the equations into each other until there is only one equation for each output port of the overall system. For our system in Figure 8 this substitution would lead to the function

$$out = \lambda_a(in_1, \lambda_m(in_2, in_3)). \tag{35}$$

We can always calculate an execution order for cycle-free component systems by a simple tree traversal which computes a topological order. This is more complicated when observing a cyclic system. In our `CollisionWarning` system (Figure 3) we notice that there is a circular dependency of direct-feedthrough components which we call a direct-feedthrough loop. The components' output calculations are directly dependent on their own results for the same time step. The mathematical interpretation of the direct-feedthrough loop is the circular equation system

$$mult_o = \lambda_{mult}(v_1, div_o) = v_1 \cdot div_o \tag{36}$$
$$add_o = \lambda_{add}(s_1, mult_o) = s_1 + mult_o \tag{37}$$
$$sub_o = \lambda_{sub}(add_o, s_2) = add_o - s_2 \tag{38}$$
$$div_o = \lambda_{div}(sub_o, v_2) = \frac{sub_o}{v_2}. \tag{39}$$

If we tried to apply our substitution approach to this system, we would run into an infinitely long expression calling itself

recursively:

$$s_c = add_o \tag{40}$$
$$add_o = \lambda_{add}(s_1, mult_o) \tag{41}$$
$$add_o = \lambda_{add}(s_1, \lambda_{mult}(v_1, div_o)) \tag{42}$$
$$add_o = \lambda_{add}(s_1, \lambda_{mult}(v_1, \lambda_{div}(sub_o, v_2))) \tag{43}$$
$$add_o = \lambda_{add}(s_1, \lambda_{mult}(v_1, \lambda_{div}(\lambda_{sub}(add_o, s_2), v_2))) \tag{44}$$
$$\dots$$

By this example, we can easily see that in order to calculate λ_{add}, we first need to provide all input values, hence we first need to compute λ_{add} and thus we cannot calculate a valid execution order. This means that we cannot provide a valid execution semantics via the execution order for systems which contain a direct-feedthrough loop. We will discuss this in more detail in the next section.

In contrast, we actually can calculate a valid execution order on nondirect-feedthrough loops. Figure 9 shows the component system `SumUp` which successively sums up the input values. An input stream of $[1, 1, 1, \dots]$ leads to the output stream of $[1, 2, 3, \dots]$. The component `delay` outputs the current state and updates it afterwards with the current input value. This component is defined by the two behavior functions λ_d and ϕ_d:

$$\lambda_d : S_d \to O_d \qquad o_d = \lambda_d(s_d) = s_d \tag{45}$$
$$\phi_d : I_d \to S_d \qquad s_d = \phi_d(i_d) = i_d. \tag{46}$$

This is a nondirect-feedthrough component, it does not directly forward its input to its output. We can use this to break loops and calculate an execution order as we are able to directly solve the equation for sum by substituting

$$sum = \lambda_a(in, \lambda_d(s_d)). \tag{47}$$

and the corresponding execution order is

1 Transport the input values along the connectors.
2 Execute λ_d and transport the output value along the connector.
3 Execute λ_a and transport the output value along the connector to sum.
4 Execute ϕ_d.

Note that the overall execution of the `delay` component is split into the two steps of output and update separated by the output calculation of the `add` component. This independent execution of the two functions allows for resolving loops that contain nondirect-feedthrough components in contrast to the stiff compound of a single component execution. The execution order above fulfills the three rules and thus the value of sum is assigned the correct value for the current time step and the state values of all components are updated correctly with the correct input values.

We defined the mathematical semantics as well as the execution semantics for atomic components and component compositions and stated that we cannot provide an execution order for systems which contain a *direct-feedthrough* loop.

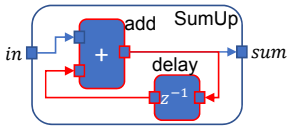

Fig. 9. The component system SumUp forms a nondirect-feedthrough loop with components `delay` and `add`. z^{-1} denotes a delay of one time step.

VI. Loop Solving

As discussed in the last section, we have no approach to compute an execution order for loops of direct-feedthrough components so that the execution produces an output which solves the model's spanned equation system. However, the problem of direct-feedthrough loops arises in the domain and to tackle RQ3 we need a methodology to handle it. The equation systems defined by a direct-feedthrough loop might have a solution and such a solution can be approached numerically at runtime. But this raises the following problems: **First**, if there is no solution to an equation system, what is the semantics for our component system? This problem arises for unsolvable equation systems like $x = 1 + x$ and equation systems that turn out to be unsolvable for some concrete values like our collision warning system which does not have a solution for crash location and time if both vehicles drive with the same velocity. **Second**, the numeric solver might fail to converge to a solution or the time to acquire an exact solution might take too long for real-time safety-critical systems. And **third**, the solution might not be exact but instead has an error margin. We need to address all three problems in order to define an execution semantics for component systems with direct-feedthrough loops.

First of all, whenever a direct-feedthrough loop occurs we cannot provide an execution semantics for which we know with certainty that it solves the mathematical semantics, with very few exceptions. What we might provide is an execution that approximates this semantics with some error margin, at best. Exceptions are equation systems which we can solve analytically at compile-time by replacing them with a loop-free component system. For example, the equations of the collision warning system (Equations (3) and (4)) form the problem of the intersection of two straight lines which has the analytical solution that is only dependent on the system's input ports:

$$t_c = \frac{s_2 - s_1}{v_1 - v_2} \tag{48}$$
$$s_c = s_1 + v_1 \cdot t_c. \tag{49}$$

Our compiler uses an analytical solver [19] which first attempts to come up with a solution for the equation system. If it is able to solve it, we can synthesize a component system from this solution at compile-time and directly insert it into the existing system as shown in Figure 10. This is done fully automatically since we can map every atomic operation from a solution term to a math component in EMA. If there are multiple solutions to choose from, we accept a random one and inform the modeler. An analytical solution is always loop-free and we therefore obtain a new system

complying with the original model, but for which we can provide an execution order. To perform an analytical solution

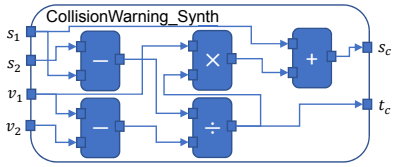

Fig. 10. Synthesized analytical solution for the collision warning system

approach the solver requires an explicit equation system but, in general, the behavior of components is not covered by a single assignment like $c = a + b$ as it is the case for components for simple arithmetic operations. In fact, most of the components seen in this field of research have a highly complex behavior implementation like an MPC controller. Accordingly, we cannot always resolve the output function $\lambda_c^{o_i}$ to a single term which makes analytical solving infeasible for such component systems. Only simple assignment chains are automatically resolved to a single term.

Nonetheless, a modeler, who builds a system with direct-feedthrough loops our solver cannot solve at compile-time and decides that the semantics can be approximated at runtime, should be provided with a solution. First, we need to analyze the kind of equation system defined by C&C models in more detail. As we stated earlier, each single output function computes the output port value as a variable dependent on current input port and inner state variable values. There is one exception to that: the integrator component sets the derivative of the output in relation with its input as $output' = input$. This is also the only component for which we provide a definition of integration, all other components are just custom numerical integration approximations. To bring this in a more readable form, we collect all differential variables in the variable vector y and all algebraic variables in the variable vector z. We replace all input ports for which the source port is in y or z by the corresponding variable via the connector equations. We can assume that all other input values are given at this time of the execution, thus we do not need to solve for them. This way the input values together with the current state variables form the configuration for the equation system for a given point in time. We replace each of the output functions by one that is dependent on y, z and the current time t. The dependency of t is mainly for the current configuration, but is also important for differential equations. With some reordering we obtain an equation system in the semi-explicit form [16]:

$$y' = f(t, y, z) \tag{50}$$
$$0 = g(t, y, z). \tag{51}$$

The semi-explicit form is the base representation for most numerical solvers and we can use it to decide what kind of solver is required depending on the kind we observe.

We display the equation system for the output behavior of the loop in the `CollisionWarning` system (Figure 3) in this form with $z = (o_a\ o_m\ o_s\ o_d)^T$ and $g = (g_{o_a}\ g_{o_m}\ g_{o_s}\ g_{o_d})^T$ as

$$0 = g_{o_a}(t, y, z) = o_a - \lambda^{o_a}_{add}(s_1(t), o_m) \tag{52}$$
$$0 = g_{o_m}(t, y, z) = o_m - \lambda^{o_m}_{mult}(v_1(t), o_d) \tag{53}$$
$$0 = g_{o_s}(t, y, z) = o_s - \lambda^{o_s}_{sub}(o_a, s_2(t)) \tag{54}$$
$$0 = g_{o_d}(t, y, z) = o_d - \lambda^{o_d}_{div}(o_s, v_2(t)). \tag{55}$$

The output equation system for the `Mass-Spring-Damper` (Figure 4) can be displayed in this form with $y = (x\ v)^T, z = (o_s\ o_a\ a\ o_D\ o_c)^T, f = (f_x\ \ f_v)^T$ and $g = (g_{o_s}\ g_{o_a}\ g_a\ g_{o_D}\ g_{o_c})^T$

$$x' = f_x(t, y, z) = v \tag{56}$$
$$v' = f_v(t, y, z) = a \tag{57}$$
$$0 = g_{o_s}(t, y, z) = o_s - \lambda^{o_s}_{sub}(o_a, F_E(t)) \tag{58}$$
$$0 = g_{o_a}(t, y, z) = o_a - \lambda^{o_a}_{add}(o_c, o_D) \tag{59}$$
$$0 = g_a(t, y, z) = a - \lambda^{a}_{gain_m}(o_s) \tag{60}$$
$$0 = g_{o_D}(t, y, z) = o_D - \lambda^{o_D}_{gain_D}(v) \tag{61}$$
$$0 = g_{o_c}(t, y, z) = o_c - \lambda^{o_c}_{gain_c}(x). \tag{62}$$

This way we transform each direct-feedthrough loop into an equation system in semi-explicit form. Based on this form we differentiate between the four categories

- Linear equation systems without differential equations,
- Nonlinear equation systems without differential equations,
- Ordinary differential equation systems (ODE),
- Differential algebraic equation systems (DAE).

A linear equation system emerges when y (and f) is empty and g resolves to a completely linear function. If g does not resolve to a linear function or is too complex to resolve to a linear function, the equation system falls into the nonlinear equation system category. We obtain an ODE when x (and g) is empty, and a DAE if both x and y are not empty.

The equation system shown in Equation (52) is linear and we already saw that we can solve it analytically. We know for sure that linear equation systems can always be solved by various methods like the *Gaussian Elimination* and our compiler is capable of doing so. Therefore we do not consider such systems in the further analysis. The analytical solver we use is capable of solving some problems of the other categories as well. For nonlinear equation systems there are solving techniques for certain problem classes as for example lower grade polynomials and known solutions to common problems. Some ODEs can be solved, e.g. the class of linear ODEs has well-studied solution approaches via the characteristic polynomial. DAEs are even harder to solve, but some can be reduced to ODEs. But in general, we cannot assume that we can provide an analytical solution to the last three categories and we need to be able to solve them with numerical solvers at runtime. Such solvers differ in their methods and accuracy depending on the problem class. Nonlinear solvers as [20] tackle nonlinear equation systems by iterative approaches like the *Newton root-finding algorithm* or minimization procedures

like *Line-Search* or *Trust-Region* [14]. To help the numeric solver a modeler can provide a good initial guess as a starting point. In EMA we model that along with the component initialization as seen in L.3 of Figure 11. Finding a numerical solution for an ODE (e.g. [21]) is the inital value problem

$$x' = f(x,t), \quad x(t=0) = x_0 \tag{63}$$

which can be approached for example with *Explicit Euler* or *Runge-Kutta*. The modeler needs to provide the initial values and can do so similarly to defining the initial guesses as shown in L.4. Approaches to solve a DAE (e.g. [22]) contain a Backward Differential Formula (BDF) [23] which forms a nonlinear equation system that needs to be solved. Again, a good initial guess is key, and also the initial values for the differential variables need to be provided. All those procedures

```
1  …                                                    EMA
2  instance Addition add{sum ~= 2};   Initial Guess
3  instance Integrator x{out = 4};
4  …                                  Initial Value
```

Fig. 11. A modeler can provide an initial guess for algebraic variables and initial values for differential variables

are well studied and the error margin can be estimated at each iteration step which allows for control over the error acceptance or accepted time overhead for the solving process. We use this to adress the third problem by forwarding the error margin to numerical solvers giving the control over it to the modeler while compiling.

The iterative procedures all have in common that the output functions are called repeatedly without updating the configuration or in our case the state variables. With our separation of output and update calculation we had the tools to implement the interface without much complexity.

Our compiler replaces all components in the loop with components whose behavior consists of the attempt to solve the equation system using numerical solvers. These components are now only dependent on the external inputs which makes some components obsolete since they are not relevant for the model's output. Figure 12 shows this process: components `mult` and `sub` are deleted and all input ports are forwarded to the two components `add` and `div`. Their behavior implementation is now to solve the equation system defined by the direct-feedthrough loop.

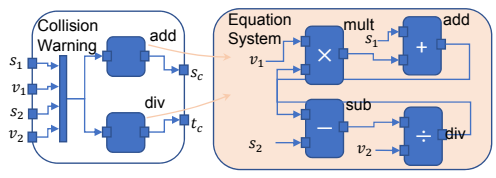

Fig. 12. The semantics of a direct-feedthrough loop is to solve the circular equation system representing this loop

As discussed above, a component might not be able to deliver an output due to different reasons. First, there might be no solution for the given inputs. Second, the computation might exceed the user's given time tolerance. Third, the iterative numerical approaches for solving nonlinear equation systems as well as DAEs require the search for roots ($f(x) = 0$) and for non-continuous functions and some other cases this might fail to converge. Along with the analytical solver, we provide a suite to automatically check whether an equation system is unsolvable or unsolvable for certain inputs with the Microsoft z3 SAT solver [24]. If these cases cannot be fully ruled out at compile-time, an error handling mechanism is required. Therefore, we introduce the notion of fallback components. A fallback component must have the same interface as the primary component it backs up. In case the execution of the primary component fails, the fallback component is used instead. To avoid delays, a fallback component can be executed in parallel to the actual component and discard the result if the primary component was executed successfully. We can define a chain of fallback components, i.e. if a component fails for a given input, the fallback successor of the chain is executed instead and so on, until an execution succeeds. Assume that $\mathcal{X}$ is the set of all possible inputs for a primary component's interface. $\mathcal{X}_i \subseteq \mathcal{X}$ is the set of all inputs which can be handled by the i-th fallback component in the fallback chain, where $i = 1$ refers to the primary component and $i+1$ is the component which is executed if i fails. In general, we must prove at compile-time that $\bigcup_i \mathcal{X}_i = \mathcal{X}$. Often, we can only determine a subset of $\mathcal{X}_i$ analytically at compile-time. Therefore, the last fallback component often needs to be chosen such that $\mathcal{X}_{last} = \mathcal{X}$, which can be accomplished easily by outputting a constant or only using basic arithmetic operations.

An example showing how fallback components are implemented in EMA is given in Figure 13. The component `CollisionSystem` has two subcomponents, one of type `CollisionWarning` and the other of type `CollisionAvoidance`. The former is the algebraic loop component of Figure 3. The rationale of the system is that the `CollisionWarning` component detects potential collisions and the `CollisionAvoidance` consumes the computed collision information to come up with an appropriate reaction. However, as we have seen, the algebraic loop has no solution if the velocities of the two vehicles are equal. Hence, the compiler requires the developer to provide a fallback component. This is specified in L.5 of Figure 13. For the component instance named `cw`, we declare the main type to be `CollisionWarning`. This type declaration is followed by a slash and the fallback component type, in this specific case: `NoCollision`. An arbitrarily long fallback chain can be defined using this syntax.

The fallback component approach is similar to the try-catch mechanism in languages like Java. To be able to ensure at compile-time that the last fallback component is executable if the primary components fail, its behavior implementation is restricted in EMA to constants, delays (to reuse the last output of the primary component), and arithmetic operations. In our concrete example, the `NoCollision` component outputs two

```
EMA
1  component CollisionSystem {
2    ports in Q s1, Q s2, Q v1, Q v2,
3          out Q brake, Q throttle, Q steering;
4    instance CollisionWarning / NoCollision cw;
5    instance CollisionAvoidance ca;
6    s1 -> cw.s1;
7    s2 -> cw.s2;
8    t1 -> cw.s1;
9    t2 -> cw.s2;
10   cw.sc -> ca.sc;
11   cw.tc -> ca.tc;
12   ca.brake -> brake;
13   ca.throttle -> throttle;
14   ca.steering -> steering;}
```

Alternative behavior for the case that the CollisionWarning component does not have a solution for given inputs

Fig. 13. The collision avoidance component has a collision warning subcomponent which contains a structural loop. A loop-free backup component is used if there is no solution of the algebraic equation for the given inputs or if the components takes too much time to compute the result at runtime. The alternative component type needs to exhibit the same interface as the original. The instance cw is polymorphic as we don't know what concrete component type is being executed.

large negative constants for the collision time `t_c` and the, for the sake of simplicity, one-dimensional collision position, indicating that there is no collision in the future and that the vehicles are driving at similar velocities.

VII. Conclusion

The C&C paradigm is widely used in various fields in research and academia through tools like Simulink or LabView. Unfortunately, these tools lack a clear and precise semantics definition. In this work, we developed a synchronous and weakly causal execution semantics for component models and demonstrated its feasibility by the example of the EMA ADL. The semantics is realized by a compile-time scheduling algorithm for linear component structures, but also supports algebraic loops and differential equation systems, which are indispensable in many engineering applications. This is realized by mapping component loops to loop-free equivalents. Model correctness and the valid input domain are verified at compile-time. Input sets without a solution are identified by the compiler and require fallback substitution components. The high-degree of compiler verification makes a modeling language using the presented semantics applicable to safety-critical systems, where runtime errors need to be avoided at all cost. A toolchain implementing the concepts presented in this work has been evaluated on examples featuring different kinds of algebraic and differential loops shown throughout the paper and hence constitutes a versatile C&C modeling methodology applicable to the design of complex systems.

References

[1] Manfred Broy and Ketil Stølen. *Specification and development of interactive systems: focus on streams, interfaces, and refinement*. Springer Science & Business Media, 2012.

[2] Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. Modeling Architectures of Cyber-Physical Systems. In *European Conference on Modelling Foundations and Applications (ECMFA'17)*, LNCS 10376, pages 34–50. Springer, July 2017.

[3] Evgeny Kusmenko, Bernhard Rumpe, Sascha Schneiders, and Michael von Wenckstern. Highly-Optimizing and Multi-Target Compiler for Embedded System Models: C++ Compiler Toolchain for the Component and Connector Language EmbeddedMontiArc. In *Conference on Model Driven Engineering Languages and Systems (MODELS'18)*, pages 447 – 457. ACM, October 2018.

[4] Evgeny Kusmenko, Sebastian Nickels, Svetlana Pavlitskaya, Bernhard Rumpe, and Thomas Timmermanns. Modeling and Training of Neural Processing Systems. In *Conference on Model Driven Engineering Languages and Systems (MODELS'19)*, pages 283–293. IEEE, September 2019.

[5] Nicola Gatto, Evgeny Kusmenko, and Bernhard Rumpe. Modeling Deep Reinforcement Learning Based Architectures for Cyber-Physical Systems. In *Proceedings of MODELS 2019. Workshop MDE Intelligence*, pages 196–202, September 2019.

[6] Evgeny Kusmenko, Svetlana Pavlitskaya, Bernhard Rumpe, and Sebastian Stüber. On the Engineering of AI-Powered Systems. In Lisa OConner, editor, *ASE19. Software Engineering Intelligence Workshop (SEI19)*, pages 126–133. IEEE, November 2019.

[7] Mark Nagurka and Shuguang Huang. A mass-spring-damper model of a bouncing ball. In *Proceedings of the 2004 American control conference*, volume 1, pages 499–504. IEEE, 2004.

[8] Ali Asadi Nikooyan and Amir Abbas Zadpoor. Mass–spring–damper modelling of the human body to study running and hopping–an overview. *Proceedings of the institution of mechanical engineers, Part H: Journal of engineering in medicine*, 225(12):1121–1135, 2011.

[9] D. Dyntar and L. Guzzella. Optimal control for bouncing suppression of cng injectors. *J. Dyn. Sys., Meas., Control*, 126(1):47–53, 2004.

[10] Vincent Aravantinos, Sebastian Voss, Sabine Teufl, Florian Hölzl, and Bernhard Schätz. AutoFOCUS 3: Tooling concepts for seamless, model-based development of embedded systems. *Joint proceedings of ACES-MB 2015–Model-based Architecting of Cyber-physical and Embedded Systems*, pages 19–26, 2015.

[11] Mathworks Inc. Simulink User's Guide. Technical Report R2016b, MATLAB & SIMULINK, 2016.

[12] Control and Display Execution Order https://de.mathworks.com/help/simulink/ug/controlling-and-displaying-the-sorted-order.html, 2020.

[13] Philip Rabinowitz. *Numerical methods for nonlinear algebraic equations*. Gordon & Breach Science Pub, 1970.

[14] David M Rosen, Michael Kaess, and John J Leonard. An incremental trust-region method for robust online sparse least-squares estimation. In *2012 IEEE International Conference on Robotics and Automation*, pages 1262–1269. IEEE, 2012.

[15] Jorge J Moré and David J Thuente. Line search algorithms with guaranteed sufficient decrease. *ACM Transactions on Mathematical Software (TOMS)*, 20(3):286–307, 1994.

[16] Lawrence F Shampine, Mark W Reichelt, and Jacek A Kierzenka. Solving index-1 DAEs in MATLAB and Simulink. *SIAM review*, 41(3):538–552, 1999.

[17] Pieter J. Mosterman and John Edward Ciolfi. Automated approach to resolving artificial algebraic loops, January 23 2007. US Patent 7,167,817.

[18] Ting Qu, Shuyou Yu, Zhuqing Shi, and Hong Chen. Modeling driver's car-following behavior based on hidden markov model and model predictive control: A cyber-physical system approach. In *2017 11th Asian Control Conference (ASCC)*, pages 114–119, 2017.

[19] A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, T. Rathnayake, S. Vig, B. E. Granger, R. P. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, M.J. Curry, A. R. Terrel, Š. Roučka, A. Saboo, I. Fernando, S. Kulal, R. Cimrman, and A. Scopatz. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3, January 2017.

[20] Andreas Wächter and Lorenz T Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical programming*, 106(1):25–57, 2006.

[21] Karsten Ahnert and Mario Mulansky. Odeint–solving ordinary differential equations in c++. In *AIP Conference Proceedings*, volume 1389, pages 1586–1589. American Institute of Physics, 2011.

[22] Ivan Korotkin. A simple but powerful c++ solver for differential algebraic equation (dae) systems, 06 2019.

[23] Simulink Solve Differential Algebraic Equations (DAEs) https://de.mathworks.com/help/matlab/math/solve-differential-algebraic-equations-daes.html, 2020.

[24] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337340, Berlin, Heidelberg, 2008. Springer-Verlag.