T U M

I N S T I T U T   F Ü R   I N F O R M A T I K

COLA – The Component Language

Stefan Kugele, Michael Tautschnig, Andreas Bauer, Christian Schallhart, Stefano Merenda, Wolfgang Haberl, Christian Kühnel, Florian Müller, Zhonglei Wang, Doris Wild, Sabine Rittmann, Martin Wechs

TUM-I0714
September 07

T E C H N I S C H E   U N I V E R S I T Ä T   M Ü N C H E N

TUM-INFO-09-I0714-0/1.-FI
Alle Rechte vorbehalten
Nachdruck auch auszugsweise verboten

©2007

Druck: Institut für Informatik der
Technischen Universität München

COLA – The Component Language

Stefan Kugele, Michael Tautschnig, Andreas Bauer, Christian Schallhart, Stefano Merenda, Wolfgang Haberl, Christian Kühnel, Florian Müller, Zhonglei Wang, Doris Wild, Sabine Rittmann, and Martin Wechs

Technische Universität München

Abstract In this paper we introduce the *co*mponent *la*nguage COLA for the design and development of embedded systems. We present the formal syntax and semantics of COLA which is based upon synchronous dataflow. Utilizing the abstraction provided by this paradigm, the designer is freed from implementation details and is able to focus on the core-functionality to be modeled and implemented. Due to the well-founded semantics of the language, it is possible to establish an integrated development process, the artifacts of which can be formally reasoned about and are dealt with in automated tools such as model checkers or model-based test case generators.
Since COLA has been developed in the context of an industrial cooperation, the main rationale in the design of COLA is its industrial applicability: Thus, the language must provide a well-defined semantical foundation which is accessible to today's tools and techniques and which is also able to capture today's industrial applications. To this end, we extend the dataflow approach with automata to model operating modes concisely as well as with a hierarchical decomposition to facilitate reuse efficiently. We designed a textual and an equivalent graphical representation of COLA models to meet the need of our industrial collaborators. Finally, we demonstrate the viability of our approach with a case study from the automotive domain where we design an adaptive cruise control (ACC) unit.

1 Introduction

COLA is a *synchronous dataflow language* for describing complex software systems, such as automotive or avionic control systems, in terms of hierarchic components with a graphical and textual syntax. It offers modeling concepts known in a similar manner from other industry standards, such as the Unified Modeling Language (UML) [1] or MATLAB/Simulink [2], but combines these with a rigorous formal semantics.

Richer COLA concepts are built from a minimal set of core primitives that make models easy to use, process, and understand from a user's point of view (where, for the sake of argument, the user can be a human or a third-party toolkit, such as a model checker [3], for instance). This paper introduces these core concepts, and defines the formal syntax and semantics of COLA.

Being a synchronous formalism, COLA follows the *hypothesis of perfect synchrony* [4]. Basically, this asserts that computations and communication occur instantly in a system, i. e., take no time. Components in a synchronous dataflow language then operate in parallel, processing input and output signals at discrete instants of time. This discrete uniform time-base allows for a deterministic description of concurrency by abstracting from concrete implementation details, such as physical bus communication, or signal jitter. Real-time requirements which are inherent in many industrial applications are then expressed with respect to the uniform time-base.

For COLA, this abstract computational model also facilitates the formal validation and verification of systems. For instance, COLA systems can be automatically model checked for safety and liveness properties, or used for the automatic generation of test cases, required to test the actual implementations of systems on hardware.

The key concept of COLA is that of a *unit* which caters for composition of elements, but may also define the actual behavior of COLA *systems*. Units can be composed hierarchically, or occur in terms of *blocks* that define the basic (arithmetic) operations of a system.

Each unit has a set of typed *ports* describing the interface, which amount to the *signature of the unit*, and which are categorized into input and output ports. Units can be used to build more complex components by building a *network* of units and by defining an interface to such a network. The individual connections in a network of (sub-) units are called *channels* and connect an output port with one or more suitably typed input ports.

In addition to the hierarchical structure of networks, COLA provides a decomposition into *automata* (i. e., finite state machines, similar to Statecharts [1]). If a unit is decomposed into an automaton, each state of the automaton is associated with a corresponding sub-unit, which determines the behavior in a particular state. This definition of an automaton is therefore well-suited to partition complex networks of units into disjoint *operating modes* (cf. [5,6,7]), whose respective activation depends on the input signals of the parent unit. Therefore modes are also a suitable device to capture the hybrid nature of many control systems, because discrete state changes can be modeled and continuous elements associated to them (in terms of the corresponding units). An example for the usage of units, automata and other COLA elements is given in Section 2.

1.1 The synchronous approach to software and systems engineering

Dataflow languages are an increasingly popular tool for the description of embedded control software. Established CASE-tools like MATLAB/Simulink use dataflow networks for the description of complex automotive systems, and subsequent code-generation [8]. The synchronous subset of these languages, however, caters for a modeling paradigm of such systems where concurrency and parallelism are explicable in a deterministic manner. Model-based design and development approaches to embedded control software based on this paradigm

have been described, e. g., in [5,9]. Both articles give a good overview on the subject.

In the synchronous paradigm, computation and communication of systems is assumed to occur infinitely fast, i. e., without delay. Moreover, a uniform discrete time base synchronizes all system events on so-called clock "ticks". Systems produce and process results in accordance with this time base. However, to cater for multi-rate sampling and to express real-time requirements of control systems (e. g., deadlines), most synchronous approaches known from the literature provide dedicated sampling operators for the derivation of additional clocks. Clocks are then defined with respect to a base clock, giving the entire design a well-defined semantics. Examples include [10,11,12]. Especially, Lucid Synchrone [12], was one of the first approaches to extend the synchronous paradigm towards a higher-order type system known from functional languages, and to express programs that would be expressive and at the same time executed synchronously.

Another well-known synchronous approach, but instead of a functional character using an imperative character, is Esterel [13]. Esterel systems differ, in that the behavior is defined in a *reactive* manner, rather than functionally based on the dataflow relations alone.

Various efficient implementations of synchronous languages in the form of textual (e. g., Esterel, LUSTRE [11], SIGNAL [14], and FOCUS [15]), or graphical languages (e. g., AUTOFOCUS [16]) exist. In fact, when restricted to use only discrete modeling blocks, MATLAB/Simulink also adheres to the synchronous languages paradigm [17].

For an example of an efficient implementation scheme of synchronous dataflow programs cf. [18]. Their work is based on the *CAN-bus* system as it is used in present-day automobiles. The authors present the constraints under which distribution of synchronous programs via an event-triggered medium is possible, and give means to determine worst-case scenarios concerning the resynchronization of systems. Moreover, with GALS (globally asynchronous, locally synchronous) [19] there exists another wide-spread approach to implement synchronous systems in a distributed and non-synchronous environment. GALS can be understood as having "synchronous islands" sitting and communicating in an asynchronous environment. For COLA and related systems this would usually require the use of FIFO-queues (first in, first out) to channel communication. Experiments with this type of setup have been carried out, e. g., in the context of the Esterel programming language [20], and formal analyses performed in the works of [21], and by means of model checking in [22].

Since COLA is an inherently synchronous formalism, the language benefits immediately from these deployment schemes, and distributed COLA systems can be distributed under the well-known constraints accordingly. Note that for the remainder, we focus mostly on the syntax and semantics of COLA, rather than its formal deployment properties in a distributed setup. With this discussion, however, we would like to point out that deployment of synchronous systems is a well-understood and researched topic in the literature.

This paper's case study of an adaptive cruise control system makes a convincing case of how embedded control systems can directly benefit from our approach; that is, we combine a UML-like graphical description catering for different views and model aspects (e. g., depending on the stake-holders of a system), and a formal semantics that is required to verify system properties (e. g., for systems certification).

Existing approaches are not yet able to fully bridge the gap between industrial process support and formally verifiable designs. Either the according tools lack formal semantics that would allow users to perform dedicated reasoning steps [23,24], or the tools' focus rests solely on the semantics, often disregarding important requirements such as scalability, traceability of design changes, or versioning.

Efforts close to our approach are made by the Metropolis project [25], with a focus on hardware/software codesign. The Architecture Analysis and Design Language (AADL) [26] also offers well-defined models aimed at industrial applications. Unlike COLA, however, neither Metropolis nor AADL support modeling down to the implementation level within a single formalism. This may be required for functional verification.

1.2 Organization

The remainder of this paper is structured as follows. The following section gives a short introduction to the concepts of COLA using a case study taken from the automotive domain. In Section 3 the modeling elements of COLA are briefly introduced and the required basics on data types within COLA are given. Further, a syntax of the language constructs used in COLA is provided in Section 4. Next, Section 5 details on the formal semantics of the introduced syntactical elements of COLA. The graphical representation of COLA systems, as used in the example following, is defined in Section 6. We conclude with an overview of ongoing work around COLA.

2 Case study: an adaptive cruise control (ACC)

This section gives an intuitive introduction to COLA on a case study taken from the automotive domain. The modeled adaptive cruise control (ACC) enables cars to keep the speed at a value set by the driver, while maintaining a minimum distance to the car driving ahead (cf. [27]).

When implementing such control systems, usually an informal specification in natural language is given. In the following paragraph, we provide a possible excerpt of such a document that explains the relevant characteristics.

2.1 Informal specification

The intended functionality includes the possibility to turn the ACC on and off. If the device is turned off, the motor speed set by the user is forwarded to the

engine control without any modification. The display shows “off” to indicate the current ACC state. By engaging the ACC, the speed and distance regulation are activated. This includes the measurement and comparison of the pace set by the user and the actual measured car velocity. If the desired user speed s_{user} differs from the actual speed s_{act}, the value for the motor control is corrected by $(s_{user} - s_{act})/20$. This results in a speed correction of 5 percent of the difference between actual and desired speed. This regulation is used as long as no object is detected within 35 meters ahead of the car. If the distance falls below this threshold, the actual speed is continuously decreased by 5 percent. The minimum distance allowed constitutes 15 meters. If the actual distance is lower, the car should perform an emergency stop. After either reducing speed or coming to a halt the ACC should speed up the car smoothly if the obstacle is out of the critical region again.

A correct operation of the described system is crucial for safety reasons. It is thus desirable to formally verify certain properties of the implementation, according to its specification. This requires a formal semantics of the modeling language. On the other hand, control systems, such as the ACC, are commonly described using graphical modeling languages. COLA, as described in this paper, satisfies both requirements.

2.2 The COLA model of the adaptive cruise control system

The implementation of this functionality is affected by the options available on the target platform. The employed controller offers buttons for control actions to the programmer. Further three sensor interfaces are present. The buttons are used to set the desired user speed, and a push button is used to implement switching the ACC on and off. The speed of the car is computed using a rotation sensor, whereas distance to obstacles ahead is measured using an ultrasonic device. In the COLA model, interfaces to hardware are marked by naming the blocks DEV_A_ for actuators and DEV_S_ for sensors.

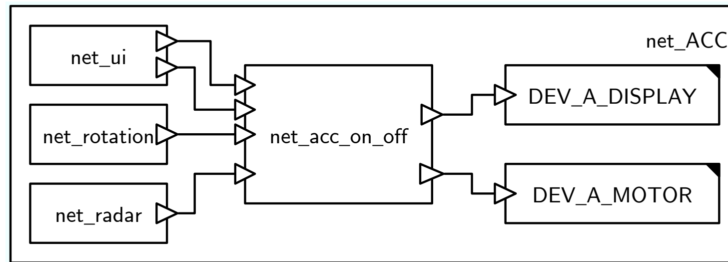


Figure 1. ACC top-level view

The top-level network of the ACC model is shown in Figure 1. As required for a COLA system (see Definition 3 on page 11), this network has no ports and

all sensors and actuators used are included in the network. The main components are the user interface (net_ui and the display DEV_A_DISPLAY), the speed computation (net_rotation), the distance sensing (net_radar), the connection to the engine (DEV_A_MOTOR), and the main control code (net_acc_on_off). Rectangles with a black triangle in the upper right corner mark blocks that are not further refined, whereas all other rectangles are networks. For example, consider the decomposition of net_radar in Figure 2.

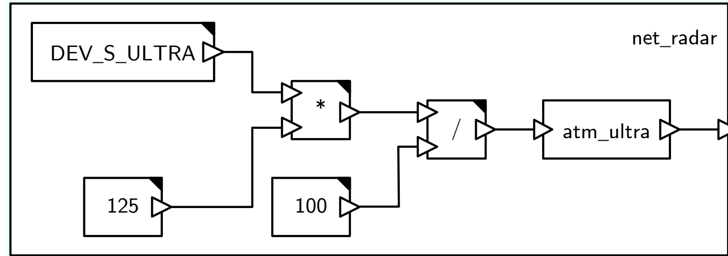


Figure 2. net_radar: distance sensing

The network is implemented by constants and basic arithmetic operations on data provided by the ultrasonic hardware (DEV_S_ULTRA). The interface of the network consists of a single output port, whereas all sensor specific data manipulation is encapsulated within this network. The characteristics of the employed hardware require some further computation, performed within an automaton (atm_ultra), which is displayed in Figure 3.

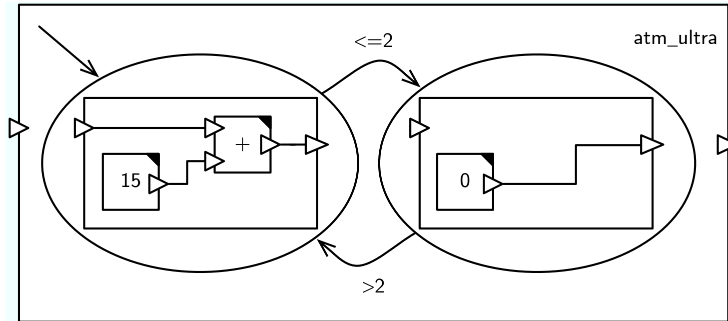
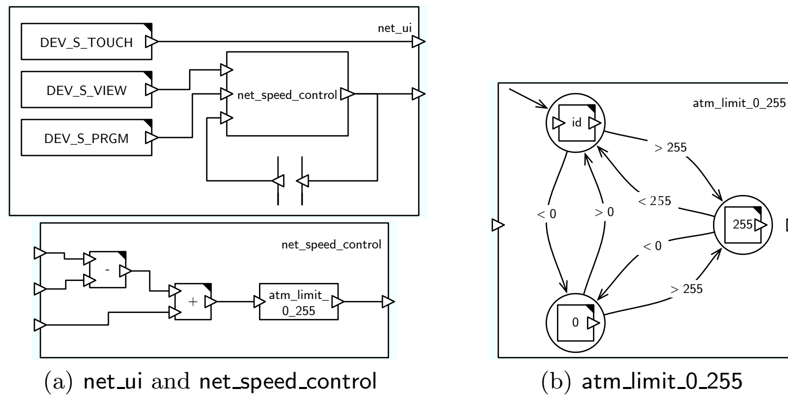


Figure 3. atm_ultra: input-specific computation

Depending on whether the data provided by the ultrasonic sensor and the arithmetic postprocessing is greater than 2 or not, a constant value of 15 must be added, or 0 is returned, respectively. This function is implemented using an

automaton with two states that describe the respective behavior. For brevity, in Figure 3 the implementing units are drawn inside the states.

The following figures complete the model of the adaptive cruise control. By generating code and compiling for the target platform, the application can be deployed and used immediately.



(a) net_ui and net_speed_control (b) atm_limit_0_255

Figure 4. net_ui and its refinements

In Figure 4(a) the user interface is detailed further. Apart from the blocks—which correspond to the buttons on the embedded system—DEV_S_TOUCH, DEV_S_VIEW, and DEV_S_PRGM, the network net_speed_control and a delay are shown. The latter is denoted by two vertical bars. The network net_ui facilitates a feedback loop to allow the user to increment and decrement the previously selected speed.

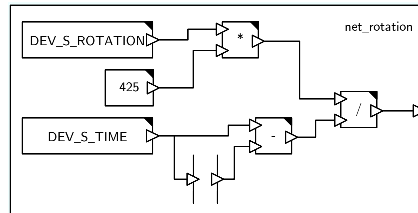


Figure 5. net_rotation

The network net_rotation shown in Figure 5 implements the speed measurement. Abstractedly speaking, it computes a function $\frac{\Delta rot \cdot 425}{\Delta t}$. The constant factor of 425 is specific to the platform as it, among others, depends on the diameter of the wheels.

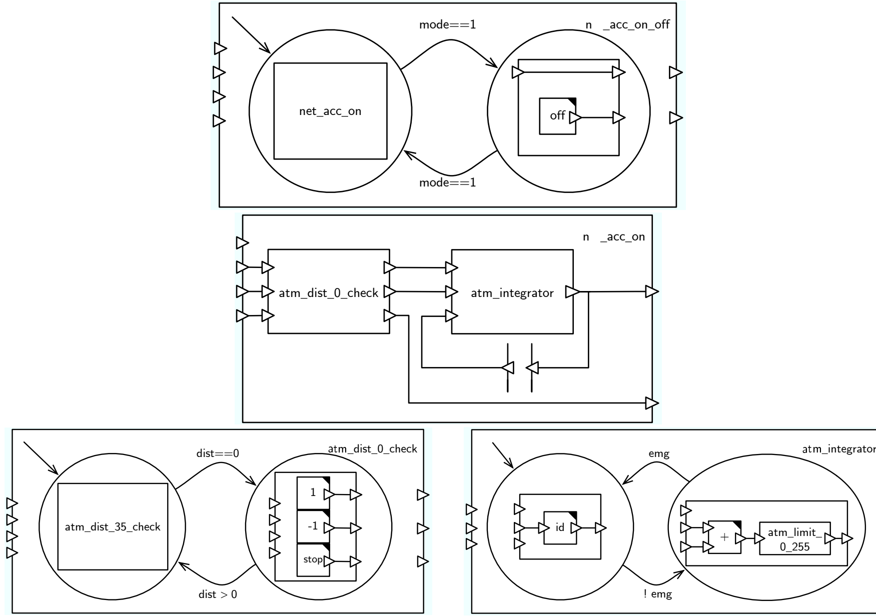


Figure 6. net_acc_on_off and further refinement

Figure 6 includes the network atm_integrator. This network reuses the automaton atm_limit_0_255, which is displayed in Figure 4(b). Note that this automaton has already been used within net_speed_control.

The effective control, which determines the behavior whenever the cruise control is switched on, is shown in Figure 7. It satisfies the functional requirements that were expressed in natural language in Section 2.1.

In the following sections, the formal interpretation of such systems will be given. Furthermore, a textual representation of COLA will be given, where we again refer to parts of this case study.

3 Modeling formalism

A COLA *system* is composed of *units*. Each unit has a set of input and output ports and describes a relation on input and output values. Units appear as basic *blocks*, forming the atomic building elements of a COLA system, or are *composed* forming networks and automata. In the latter case units allow for combination and integration of elements into more complex units. All kinds of units are possibly connected by *channels* attached to the input and output ports.

In the following all modeling concepts are formally defined and further explained. Right before a notation of types is introduced to give way to a sound definition of the syntax and the semantics of COLA.

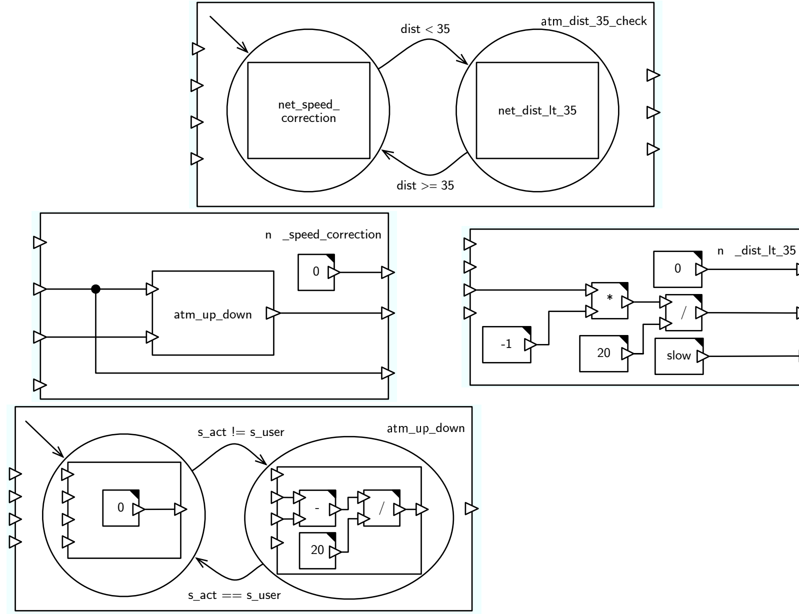


Figure 7. The cruise control

3.1 Types

In order to validate fundamental static compatibility of connected units, their communication endpoints, i. e., input and output ports, are statically *typed* such that *type compatibility* can be checked. However, for this paper the details of type compatibility checking are not considered and therefore only the rudiments of a type system are introduced to describe units and all subsequently defined terms properly. For a proper introduction to types in programming languages consult, e. g., [28].

It is assumed that there exists a universe of types, $\mathbb{T}$, such that the assumptions described next are consistent. These types may include basic types such as Boolean or Float, but also complex types. With each type a domain of values is associated, denoted by $\mathsf{dom}(t)$ for some $t \in \mathbb{T}$.

To express type compatibility, a subtype relation forming a lattice of types in $\mathbb{T}$ is required. Let the subtype relation be denoted by $\sqsubseteq$. This and the above requirements are met by, e. g., [29]. The type system described there can be applied to COLA immediately.

3.2 Units, signatures, and ports

The basic, yet abstract, computing element within COLA is a *unit*. Each unit defines a relation on a set of input and output values. The valid domains are implied by the types of the input and output *ports*. Together they syntactically

describe the interface, called the *signature*. To allow a proper definition of units, ports and signatures must be formally specified first.

The static typing outlined above is implemented by associating a type with each port. Otherwise, a port is solely an identifier that must be unique within the unit. Note that the discrimination of input and output ports is only derived from the signature, which also defines the order of the ports. We denote the resulting pair of an identifier a and the port type t by $a : t$.

Using this ordering the type of the relation that is computed by the unit is defined as the Cartesian product of all types of the input and output ports. The domain of the relation is implied by the input ports, and the range is inferred from the output port types.

Definition 1 (Signatures and ports). *Let* $P_{in} = \langle a_1 : t_1, \ldots, a_k : t_k \rangle$ *and* $P_{out} = \langle a_{k+1} : t_{k+1}, \ldots, a_n : t_n \rangle$ *with* $k, n \in \mathbb{N}$ *be (ordered) lists of* ports*, such that all identifiers* $a_1, \ldots, a_n$ *are pairwise distinct. A* signature σ *is written as* $\sigma = (P_{in} \rightarrowtail P_{out})$*. Then,* P_{in} *denotes the input and* P_{out} *the output ports. For the ease of use, the following notations are defined:*

- $\mathsf{in}(\sigma) = (a_1, \ldots, a_k)$ *and* $\mathsf{out}(\sigma) = (a_{k+1}, \ldots, a_n)$
- $\mathsf{type}(a_i) = t_i$
- $\mathsf{dom}(\mathsf{in}(\sigma)) = \mathsf{dom}(\mathsf{type}(a_1)) \times \ldots \times \mathsf{dom}(\mathsf{type}(a_k))$
- $\mathsf{dom}(\mathsf{out}(\sigma)) = \mathsf{dom}(\mathsf{type}(a_{k+1})) \times \ldots \times \mathsf{dom}(\mathsf{type}(a_n))$

Based on these preliminaries, units can be defined syntactically. Every unit has a name that allows referring to it, a signature, a classifier, and an implementation. The identifier of the unit must be unique among all units to allow the identification within the set of all units that build up a COLA system. This is a formal requirement for instantiation, as discussed in Section 3.3. The classifier specifies the type of the implementation, because the effective description of *functional blocks* ("fblock"), *timing blocks* ("tblock"), *networks* ("network") and *automata* ("automaton") is different. Depending on the classifier, the implementation may consist of a relation only, but may also be a specification of composite units as described in the following sections.

Definition 2 (Unit). *A* unit u *is a tuple* $u = \langle n, \sigma, c, I \rangle$*, where* n *is its name,* σ *is a signature,* c *is a classifier and* I *is its implementation.*

In the course of this section, the syntactical structure of each possible implementation will be discussed. However, to properly define the implementations of all kinds of units, first COLA systems must be described.

3.3 Systems

A COLA *system* is a set of units with pairwise distinct identifiers. Out of these, one is selected as the distinguished root unit of the hierarchy. It must not have any input or output ports as there is no means of providing data to the root unit or reading from it. Thus a system may also include a description of the environment of the object that the modeling focuses on.

Definition 3 (System). *A* system *is a pair $\langle u, U \rangle$ with a set of units, U, and an identifier, u. It must hold that*

1. *each unit identifier within U is unique, and*
2. *there exists a unit $\langle n, \sigma, c, I \rangle \in U$ with $n = u$ and no ports, i. e.,* $\mathsf{in}(\sigma) = \mathsf{out}(\sigma) = \emptyset$.

Next, types of nested units are explained. In the context of component based systems, hierarchical nesting is referred to as *composition*. The hierarchy of COLA elements is built by instantiating units from the collection of units provided by the system and thereby composing new elements. Note that the only initial elements to build such a set are blocks as defined at the end of this section.

3.4 Composite units

Within the hierarchy, units other than blocks are composed of further units. In COLA, two principles to structure hierarchies of units are employed:

- First, a hierarchically structured unit can consist of a (dataflow) network of further units, which are connected by channels.
- Second, a unit can contain an automaton. For example, a brake controller might have the states "car is moving" and "car halted." Then, for each state of this automaton, a further unit is given, which computes the output of the overall unit as long as the automaton remains in this state.

Networks and channels. If a unit is classified as a *network*, the implementation of this unit contains further units (or sub-units) which are connected by *channels*. Apart from the interconnection, channels also facilitate the mapping between input and output ports of the network and ports of the contained sub-units. The intention behind channels is the propagation of values within a network of units. Informally, each channel transfers the data values from its input, called the source, to its outputs, referred to as destinations. We distinguish four valid ways of using channels:

- Channels between sub-units that connect a single output port to one or more input ports.
- Connections between an input port of the network and one or more input ports of contained sub-units, or connections between an output port of a sub-unit and one or more output ports of the network.
- Direct connections from an input port of the network to one or more output ports of the network.
- Channels that connect a sub-unit's output port to one or more output ports of the network *and* to one or more input ports of other sub-units.

To sum up, we can say that on the one hand sources can either be output ports of sub-units or input ports of networks, on the other hand destinations can either be input ports of sub-units or output ports of networks.

We further refer to the single port that provides input as the *source* of the channel. On the contrary, the ports that allow gathering data from the channels are called *destinations* of the channel.

To ensure *compatibility*, it is in all cases required that the type of the source port is a subtype of the type of any connected destination.

Definition 4 (Channel). *A* channel *is a triple* $c = \langle a, s, \{d_1, \ldots, d_k\}\rangle$ *with port identifiers* $s, d_1, \ldots, d_k$ *for* $k \geq 1$ *and an identifier* a*. Then,* $d_1, \ldots, d_k$ *are the destinations of the channel and* s *forms the source of the channel. To guarantee type compatibility,* $\mathsf{type}(s) \sqsubseteq \mathsf{type}(d_j)$ *must hold for any* $1 \leq j \leq k$*. To refer to the source of the channel, the function* $\mathsf{src}(c) = s$ *is provided. Similarly, the set of destinations is obtained using* $\mathsf{dest}(c) = \{d_1, \ldots, d_k\}$*.*

Given a set of units and interconnecting channels, a network can be described. The resulting network *instantiates*, thereby *reuses*, (sub-)units and adds channels as its implementation. Thereby it forms a new element that is then added to the set of units of the system.

The instantiation is formally best described by a mapping inst from a set of identifiers to the set U of the system $\langle u, U\rangle$. Note that each of the identifiers must be unique within this set of sub-units. Further note that the mapping is specific to each network, as only within that the identifiers are guaranteed to be unique. Instantiation defined in this way also allows multiple uses of the same unit within a single network.

There are several requirements channels and ports of sub-units must satisfy in order to form a syntactically correct network. Note that type compatibility already follows from the definition of channels.

- First, to guarantee dataflow within the network, any input port of the sub-units must be a destination of some channel. As such, channels describe data dependencies within a network and thus requirements on the order of evaluation. The details thereof are given in the semantics in Section 5.3. It shall be noted that such an order can only be obtained if there are no cycles, unless a cycle contains at least one delay component (see Definition 8).
- Second, all output ports of the network must be supplied with data. To this end, each of them must be the destination of some channel.
- Third, the identifier of any port of the network or the sub-units must be contained in at most one channel.
- Finally, to prevent recursion, any referenced sub-unit must be already available in the collection U of the system $\langle u, U\rangle$.

Definition 5 (Network). *A* network *is a unit* $\langle n, \sigma, \text{network}, I\rangle$ *with an implementation* $I = \langle\{u_1, \ldots, u_k\}, \mathsf{inst}, C\rangle$*, with* $k \in \mathbb{N}$*. The total function* $\mathsf{inst} : \{u_1, \ldots, u_k\} \to U$ *maps a set of identifiers to units contained in the collection* U *of a* COLA *system* $\langle u, U\rangle$*.* C *is a set of channels.*

Valid port identifiers: *As channels must refer to port names unique within the network, the ports of instantiated units are addressed as* $u_i.a_j$ *for some port* a_j

of a unit instantiation identifier u_i. Whenever channels refer to the input or output ports of the network the unit name is omitted.

The syntactic requirements on networks are formally described as follows.

1. Dataflow: *For any identifier $x \in \{u_1, \ldots, u_k\}$ that yields the instantiation of a unit $\mathsf{inst}(x) = \langle n', \sigma', c', I' \rangle$ and any input port identifier $p \in \mathsf{in}(\sigma')$ there must be a channel $c \in C$ such that $p \in \mathsf{dest}(c)$.*
2. Cycle validity: *Assume that there exists a subset of the sub-unit identifiers, $\{u'_0, \ldots, u'_l\} \subseteq \{u_1, \ldots, u_k\}$, such that, together with a set of channels, they form a cycle. That is, for all $0 \leq i \leq l$ and $j = i + 1 \bmod l + 1$ there exists an output port p of the unit instantiated from u'_i, an input port q of the unit instantiated from u'_j, and a channel $c \in C$ with $\mathsf{src}(c) = u'_i.p$ and $u'_j.q \in \mathsf{dest}(c)$. If this is the case, then there must be at least one $0 \leq d \leq l$ such that $\mathsf{inst}(u'_d) = \langle n', \sigma', \mathrm{tblock}, I' \rangle$, i. e., u'_d instantiates a delay.*
3. Output presence: *For all output port identifiers $p \in \mathsf{out}(\sigma)$ of the network a channel $c \in C$ with $p \in \mathsf{dest}(c)$ must exist.*
4. Channel validity: *Channels must be connected. That is, for each channel $c \in C$ the source $\mathsf{src}(c)$ must resolve to a valid port identifier as described above, and $\mathsf{dest}(c) \neq \emptyset$ must consist of valid port identifiers as well. Furthermore, if $p = \mathsf{src}(c)$ and $p = \mathsf{src}(c')$, or $p \in \mathsf{dest}(c)$ and $p \in \mathsf{dest}(c')$, then $c = c'$ must hold.*

Orthogonal to the dataflow description using networks, control flow is described using automata. We make use of Moore automata [30], i. e., output (generated by contained dataflow descriptions) is provided by states, independent of the transition taken to reach the state.

Automata If a unit A is an *automaton*, then the implementation of this unit is a finite automaton with transitions guarded by predicates. The output is defined by the unit associated with each of the states. This is then equivalent to substituting one of the states for the automaton. Thus, the states are also referred to as *operating modes*. Each state of the automaton represents such an operating mode and each such state is associated with its own dedicated and possibly differing behavior in terms of a sub-unit.

Therefore, in each such unit A, there is one sub-unit which is enabled, namely the sub-unit which corresponds to the enabled state of the automaton. The question arises how to handle the passive sub-units, whether they should be reset on reactivation, whether they should completely freeze, or whether they should continue their computation while their outputs are ignored. As laid out in Definition 13, we follow the freeze approach as we found it to be the most natural one for our application domain. That is, only the unit corresponding to the selected state performs its computation, while all others retain their current system state.

Type correctness of the output of any enabled state is guaranteed by demanding that the signatures of A and each of the sub-units are equivalent.

Definition 6 (Automaton). *A unit* $\langle n, \sigma, \text{automaton}, I\rangle$ *with an implementation* $I = \langle Q, \mathsf{inst}, q_0, \Delta\rangle$ *is an* automaton*, where*

- $Q = \{q_0, \ldots, q_k\}$ *is the finite set of state labels. As with networks, each of which is associated with a unit using a total function* $\mathsf{inst} : Q \to U$ *for a system* $\langle u, U\rangle$*.*
- $q_0 \in Q$ *is the initial state,*
- $\Delta \subseteq Q \times \mathsf{dom}(\mathsf{in}(\sigma)) \times Q$ *is the transition relation.*

For syntactical compatibility of the automaton with each state $q \in Q$ *it must hold that* $\mathsf{inst}(q) = \langle n_q, \sigma_q, c_q, I_q\rangle$ *implies* $\sigma_q = \sigma$ *for all* $q \in Q$*.*

In practice, the transition relation is given by a set of predicates over the inputs, which is again expressible using COLA networks.

3.5 Blocks

A *block* provides some basic computational functionality which is not further decomposed. Thus they are the leaves of the tree formed by the hierarchy of a COLA system. Following the syntactic description in Definition 7, a block comes with a signature and a relation that maps the values at the input ports to values at the output ports. Further, it is distinguished between *timing blocks* and *functional blocks*. Functional blocks are used to define the behavior, whereas timing blocks at the current stage may be used to influence the succession of data. With the advent of a clock calculus for COLA there will also be operators to modify the order and frequency of the execution of units.

Definition 7 (Functional block). *A* functional block *is formalized as a unit* $\langle n, \sigma, \text{fblock}, I\rangle$ *with an implementation* I *that is a relation. It associates each input with a set of output values, i. e., the type of* I *is specified by*

$$I \subseteq \mathsf{dom}(\mathsf{in}(\sigma)) \times \mathsf{dom}(\mathsf{out}(\sigma))$$

Examples of functional blocks include the arithmetic operations $+, -, \times, /$, Boolean connectives like $\vee, \wedge, \neg$, or comparison operators $(=, <, >, \leq, \geq)$.

The only timing block that is defined here is the *delay*. Further timing operators will be discussed in the COLA clock calculus, which is subject to another document.

The idea of a delay is to provide means for retaining a value for a single time unit, thereby supplying a low-level realization of variables as found in high-level programming languages. The detailed semantics are described in Definition 12. To guarantee proper operation from the beginning, the delay must be initialized with a constant value c. The mapping computed by the delay is described by the delay relation, parametrized by the default value. It is of the following type:

$$\mathsf{delay}_c : \mathsf{dom}(t) \to \mathsf{dom}(t) \quad \text{for } t \in \mathbb{T}, c \in \mathsf{dom}(t)$$

Definition 8 (Delay). *A* delay *is a unit* $\langle n, \sigma, \text{tblock}, I\rangle$ *with a signature* $\sigma = (\langle i : t\rangle \rightarrowtail \langle o : t\rangle)$ *and the relation* $I = \mathsf{delay}_c, c \in \mathsf{dom}(t)$ *as its implementation.*

3.6 Unique identifiers for instantiated units

Within COLA, unique identifiers are used for referring to instances. Unique names are generated in a recursive manner following the hierarchy of composition by descending from the root unit to blocks that cannot be further decomposed.

Definition 9 (Unique identifier). *The* unique identifier *of an instance of a unit $x = \langle n, \sigma, c, I \rangle$ is inductively defined as follows:*

- *If x is the root unit of a system $\langle u, U \rangle$, i. e., $n = u$, then its unique identifier is root.*
- *For a network x with unique identifier net_id, the unique identifiers of the contained sub-units $\{u_1, \ldots, u_k\}$ are defined to be net_id.u_i for any $u_i \in \{u_1, \ldots, u_k\}$.*
- *For an automaton x with unique identifier atm_id, the unique identifiers of its states $\{q_0, \ldots, q_k\}$ are defined to be atm_id.q_i for any $q_i \in \{q_0, \ldots, q_k\}$.*

These three cases are sufficient to uniquely identify all unit instances of a system since functional blocks as well as timing blocks either occur as the root unit or are contained in one of the composed units.

4 Syntax

In this section a textual representation of COLA models is described in terms of an EBNF, compliant with the ISO standard as defined in [31].

4.1 Libraries and systems

As laid out in Definition 3, a COLA system is a pair $\langle u, U \rangle$ with a set of units U and an identifier u that distinguishes the unit which implements the system to be modeled by using the other given units. To facilitate reuse, the COLA language supports structuring the set of given units. Therefore COLA introduces libraries and inline definitions within a unit.

A library contains a set of reusable COLA units. For further structuring sub-libraries are also allowed. Since a library is marked as used in a dedicated context by an `include`-statement, all units contained in the library itself and its sub-libraries can be used. Libraries themselves can contain an `include`-statement which indicates that all units defined within this library and sub-libraries can use the units defined in the library given by the `include`-statement. Inline defined units can only be used within the defining unit.

Since libraries, inline definitions and `include`-statements only cater for further structuring of the defined units, these concepts are omitted in the COLA modeling formalism (Section 3) and semantics (Section 5). There, every defined unit can be used instead.

Since the defined units are captured by libraries and inline definitions, besides an identifier a COLA system just consists of the implementing root unit.

```
Library = “library” Identifier “{” { “include” Reference “;” }
          { Library } { Unit } “}” ;
System  = “system” Identifier “{” Unit “}” ;
```

4.2 Units

The unit, which is formally described in Definition 2, is the central modeling concept of COLA. Its implementation can be either a block, a network, or an automaton. As defined in the EBNF-rule *BasicUnitHeader*, every unit has an identifier, a signature and optional properties.

Identifier and signature are mandatory except for units used as guards, which describe the predicates of the transition relation, or behaviors of automata. In this case, identifiers can be omitted because guards and behaviors are not intended to be reused and thus do not need to be referenced. The guard's and behavior's signature can be be left out, as it is defined by the signature of the surrounding automaton. For behaviors the signature is exactly the same as the one of the automaton. For guards only the input ports are the same as those of the automaton; the output ports of a guard are defined by exactly one port with the type `Bool` and the name “`guard`”.

For automata and networks the *ExtendedUnitHeader* is defined. Again, the `include`-statement allows the indication of library usage. The units of the library can then be used in the implementation. Additionally, units can be defined inline. These units are only accessible by the defining unit and its other inline defined sub-units. As mentioned above, the inline definitions are omitted in the COLA modeling formalism and semantics.

```
Unit               = Block | Network | Automaton ;
BasicUnitHeader    = [ Identifier ] [ Signature ] “{” { Property } ;
ExtendedUnitHeader = BasicUnitHeader { “include” Reference “;” }
                     { “inline” Unit } ;
```

Signatures and ports According to Definition 1 a signature consists of input and output ports. Note that an empty list of input and/or output ports is possible. Thus `( -> )` denotes the empty signature. A port is defined by its identifier and its datatype.

```
Signature = “(” [ Port { “,” Port } ] “->” [ Port { “,” Port } ] “)” ;
Port      = Identifier “:” Datatype ;
```

Datatypes and literals Presently only the three datatypes `Bool, Int,` and `Real` are allowed. Records and tuples are currently under development and will

be subject of a continuative work. The corresponding literals are also defined below.

$$
\begin{array}{ll}
\mathit{Datatype} & = \text{“}\texttt{Bool}\text{”} \mid \text{“}\texttt{Int}\text{”} \mid \text{“}\texttt{Real}\text{”} \ ; \\
\mathit{Literal} & = \mathit{BooleanLiteral} \mid \mathit{IntegerLiteral} \mid \mathit{RealLiteral} \ ; \\
\mathit{BooleanLiteral} & = \text{“}\texttt{true}\text{”} \mid \text{“}\texttt{false}\text{”} \ ; \\
\mathit{IntegerLiteral} & = [\ \text{“}\texttt{-}\text{”}\ ]\ \mathit{Digit}\ \{\ \mathit{Digit}\ \} \ ; \\
\mathit{RealLiteral} & = [\ \text{“}\texttt{-}\text{”}\ ]\ \mathit{Digit}\ \{\ \mathit{Digit}\ \}\ \text{“}\texttt{.}\text{”}\ \mathit{Digit}\ \{\ \mathit{Digit}\ \} \ ;
\end{array}
$$

Properties Properties allow an additional (partial) description of the unit's behavior by a SALT [32] formula and/or natural language. A detailed description of properties will be given in an additional paper.

$$
\begin{array}{ll}
\mathit{Property} & = \text{“}\texttt{property}\text{”}\ (\ \mathit{TextProperty} \mid \mathit{SaltProperty}\ )\ \text{“}\texttt{;}\text{”} \ ; \\
\mathit{TextProperty} & = \{\ \mathit{UniCodeChar} - \text{“}\texttt{;}\text{”}\ \} \ ; \\
\mathit{SaltProperty} & = ?\ \text{a SALT formula}\ ? \ ;
\end{array}
$$

Blocks As described in Section 3.5 a block can not be further decomposed. In principle, blocks are used to represent four kinds of primitive units:

- primitive stateless functions, like `add` or `sub`
- the delay operator `pre`
- legacy units
- physical connections modeled by sources and sinks

Even though blocks are not further decomposed, properties can be optionally specified.

$$
\mathit{Block} = \text{“}\texttt{block}\text{”}\ \mathit{BasicUnitHeader}\ \text{“}\}\text{”} \ ;
$$

Automata Automata are formally defined in Section 3.4. According to that, the behavior of an automaton is described by a set of states and corresponding transitions. Exactly one of those states must be indicated as `initial`. For each state, the behavior is defined by a unit. All transitions which end at this state are defined by a `from`-statement including the corresponding guard unit. The guard unit with its Boolean output implements the transition relation described in Definition 6.

$$
\begin{array}{ll}
\mathit{Automaton} & = \text{“}\texttt{automaton}\text{”}\ \mathit{ExtendedUnitHeader}\ \mathit{State}\ \{\ \mathit{State}\ \}\ \text{“}\}\text{”} \ ; \\
\mathit{State} & = [\ \text{“}\texttt{initial}\text{”}\ ]\ \text{“}\texttt{state}\text{”}\ \mathit{Identifier}\ \text{“}\{\text{”} \\
 & \quad \text{“}\texttt{transitions}\text{”}\ \text{“}\{\text{”}\ \{\ \mathit{Transition}\ \}\ \text{“}\}\text{”}
\end{array}
$$

```
              "behavior" Unit "}" ;
Transition = "from" Reference "guard" Unit ;
```

Networks According to Definition 5 a network is implemented by units connected via channels. Once a unit is specified (in a library or inline) it can be used in multiple network implementations. Due to this fact the decomposition of networks is implemented by instances of units. For better readability, COLA provides a function-like notation for network implementations. Thus a network consists of a set of assignments. In order to be able to reference port-to-multiport channels, which are channels connected to more than one destination, within the assignments, they have to be declared explicitly by a `channel`-statement. We call these channels "named channels".

A reference on the left side of an assignment points either to an output port of the network or to a named channel. The right side of an assignment may be a reference to an input port of the network or a named channel. It is not allowed that both sides of an assignment are references to named channels. In case of a constant input the right side is a literal. Note that in the COLA modeling formalism a constant is seen as a COLA block. The third alternative for the right side of an assignment is a unit instance and therefore the output ports of the instantiated unit are connected to the left side of the assignment. The number of references on the left side must correspond to the number of output ports of the instantiated unit. The input for the instantiated unit is given by a list of arguments within parentheses. The notion looks like a function call and the channels are implicitly defined. For better readability some important COLA blocks are written in infix and prefix notation respectively.

```
Network      = "network" ExtendedUnitHeader
               { "channel" Identifier ";" } { Assignment ";" } "}" ;
Assignment   = ( Reference | ( "(" Reference "," Reference
               { "," Reference } ")" ) ) ":=" Argument ;
Argument     = Reference | Literal | UnitInstance ;
UnitInstance = ( Reference "(" [ Argument { "," Argument } ] ")" )
               | InfixFcts | PrefixFcts ;
InfixFcts    = "(" Argument ( "+" | "-" | "*" | "/" | "=" |
               "<=" | ">=" | "<" | ">" | "&" | "|" ) Argument ")" ;
PrefixFcts   = "(" "!" Argument ")" ;
```

4.3 Identifiers and references

In general, (simple) identifiers are not be globally unique. Based on simple identifiers and based on this grammar we obtain canonical identifiers, which are unique by construction. As described in Section 3.6, a canonical identifier is built up by concatenating the identifiers of all nodes on the path in the hierarchy from the root to the node to identify, separated by a dot. For referencing, canonical

identifiers can always be used. The simple identifier can be used instead if the canonical identifier of the referencing element and that of the referenced element are the same except for the last identifier in the paths.

$$Identifier = (\ BasicChar \mid \text{“_”}\ )\ \{\ BasicChar \mid Digit \mid \text{“_”}\ \}\ ;$$
$$Reference = Identifier\ \{\ \text{“.”}\ Identifier\ \}\ ;$$

4.4 Basic alphabet

Text strings in COLA may use the unicode character set.

$$\begin{aligned} UniCodeChar &= \text{? a printable unicode character ?}\ ; \\ BasicChar &= (\ \text{“A”} \ldots \text{“Z”}\ ) \mid (\ \text{“a”} \ldots \text{“z”}\ )\ ; \\ Digit &= (\ \text{“0”} \ldots \text{“9”}\ )\ ; \end{aligned}$$

4.5 Example

In the following the network net_radar, as already displayed in Figure 2 on page 6, with all of its sub-units, including atm_ultra from Figure 3, coded in the above defined COLA syntax is shown:

```
network net_radar ( -> dist:Int ) {
  automaton atm_ultra ( in:Int -> out:Int ) {
    initial state sPlus15 {
      transitions {
        from state s0 guard network ( in:Int -> guard:Bool )
        { guard := (in > 2); }
      }
      behavior network ( in:Int -> out:Int )
      { out := (in + 15); }
    }
    state s0 {
      transitions {
        from state sPlus15 guard network ( in:Int -> guard:Bool )
        { guard := (in <= 2); }
      }
      behavior network ( in:Int -> out:Int )
      { out := 0; }
    }
  }
  source block DEV_S_ULTRA ( -> ultra:Int ) {}
  dist := atm_ultra(((DEV_S_ULTRA() * 125) / 100));
}
```

5 Semantics

In COLA, as a synchronous dataflow language, it is assumed that operations start at the same instant of time and are performed simultaneously with respect to data dependencies. As denoted in Definition 10, the computation of the system over time can be subdivided into discrete steps, called *ticks*, and the execution is performed in a stepwise manner over the discrete uniform time-base. The data dependencies are implied by the employed channels. At each step a unit emits new values to the channels connected to its output ports. These values become available immediately for ports connected to the reading side of the channel. The semantics of a COLA system is described by an infinite run of the system. In order to define this system behavior, we first consider basic concepts common to all units.

5.1 Units

The semantics of a unit $\langle n, \sigma, c, I \rangle \in U$ with a unique identifier x of a COLA system $\langle u, U \rangle$ is described by a relation op_n. In order to define it properly, the state of a unit—which intuitively resembles to a snapshot of its internal memory—denoted by $state[x]$ has to be defined first.

The vector $state$ stores the state of all unit instances of the system $\langle u, U \rangle$. It is indexed, denoted as $[\cdot]$, by unique instance identifiers. As detailed below, in case of composite units, $state[\cdot]$ may again refer to entries in the vector corresponding to the instances of sub-units.

State of a unit The state of a unit $\langle n, \sigma, c, I \rangle$ with unique identifier x is described by the vector entry $state[x]$. Its domain depends on the unit's classifier c and the implementation I. In order to distinguish *stateful* and *stateless* units, we first have to clarify both terms: A *stateless* unit's output *only* depends on its input values. *stateful* means that a unit's output value may also depend on the unit's history, i. e., on previous computations. Based on this definition, we can specify the state of a unit in the following way:

– If $\langle n, \sigma, \text{fblock}, I \rangle$ is a functional block with a unique identifier x, then for any tick holds: $state[x] = \emptyset = \mathsf{dom}(state[x])$. A functional block can implement basic arithmetic operations, Boolean connectives, comparison operators as well as constant functions. All these operations and functions do not depend on past calculations and are therefore stateless.
– For a unit implementing a delay $\langle n, \sigma, \text{tblock}, \mathsf{delay}_c \rangle$ with unique identifier x, the state is a value from the domain of the port types: $state[x] \in \mathsf{dom}(\mathsf{in}(\sigma)) = \mathsf{dom}(\mathsf{out}(\sigma)) = \mathsf{dom}(state[x])$. In the first tick, the state of the delay is the default value (see Definition 8) $c \in \mathsf{dom}(\mathsf{in}(\sigma))$, i. e., $state[x] = c$. The delay copies the current input into the internal state and emits its previous internal state. The initial value is the first emitted value, whereas subsequently, the incoming stream of values is reproduced as output, delayed for a single tick.

- The state of a network $\langle n, \sigma, \text{network}, \langle N, \mathsf{inst}, C\rangle\rangle$ with the unique identifier x is a tuple of all unit states contained in $N = \{u_1, \ldots u_k\}$ with $k \in \mathbb{N}$:

$$state[x] = (state[x.u_1], \ldots, state[x.u_k])$$
$$\mathsf{dom}(state[x]) = \mathsf{dom}(state[x.u_1]) \times \ldots \times \mathsf{dom}(state[x.u_k])$$

- Finally, the implementation of a unit $\langle n, \sigma, \text{automaton}, \langle Q, \mathsf{inst}, q_0, \Delta\rangle\rangle$ with a unique identifier x as an automaton has to be considered. As introduced in Definition 6, each state q_i, $0 \leq i \leq k$ where $k+1$ is the number of states in the automaton, is itself realized by a unit. The state of a unit implementing an automaton is thus defined to be the enabled state $q_e \in \{q_o, \ldots, q_k\}$, which equals q_0 at the first tick, in combination with the state of all automaton states q_i.

$$state[x] = (q_e, state[x.q_0], \ldots, state[x.q_k])$$
$$\mathsf{dom}(state[x]) = Q \times \mathsf{dom}(state[x.q_0]) \times \ldots \times \mathsf{dom}(state[x.q_k])$$

Utilizing this description of the state of a unit, the *semantic relation* op_n, where n refers to the name of the unit, may be defined for any unit $\langle n, \sigma, c, I\rangle$ with unique identifier x. It describes all—possibly non-deterministic—valid transitions of a unit that can occur in a single tick. Because we assume perfect synchrony, computations and communication between different units take no time. Disregarding automata for a moment, for each instantiated unit of the COLA system, one of its possible transitions is chosen at each tick. In case of an automaton, however, this only applies to the enabled state, whereas all other states, and the entire hierarchy below the respective sub-units, do not perform any transitions. In consequence, the system is evaluated exactly once at each tick.

As the concept of a unit is a general one and not associated with a specific behavior, op_n is first described as a subset of tuples of internal state, input values, a new internal state, and output values.

$$\mathsf{op}_n \subseteq \mathsf{dom}(state[x]) \times \mathsf{dom}(\mathsf{in}(\sigma)) \times \mathsf{dom}(state[x]) \times \mathsf{dom}(\mathsf{out}(\sigma))$$

In the following the definition of op_n is explained in detail for blocks and composite units. We further describe the semantics in terms of an interpreter of COLA systems, because this approach naturally lends itself to later attempts of code generation.

To algorithmically express port valuations, for each unit instance uniquely identified by x a vector val_x is defined, with

$$val_x \in \mathsf{dom}(\mathsf{in}(\sigma)) \times \mathsf{dom}(\mathsf{out}(\sigma)).$$

It is indexed by port names from σ, i. e., $val_x[p] \in \mathsf{dom}(\mathsf{type}(p))$ for some $p \in \mathsf{in}(\sigma) \cup \mathsf{out}(\sigma)$. The values of the entries of val_x are set and read from in the algorithms described below.

5.2 Semantics of systems

As described in Algorithm 2, a COLA system $\langle u, U \rangle$ is evaluated in a recursive manner starting at the unit identified by u. We thus define the semantic relation, op_S, of a COLA system $S = \langle u, U \rangle$ in terms of the semantic relation of the root unit $\langle u, \sigma, c, I \rangle \in U$, which is the unit corresponding to the identifier u. Then, $\mathsf{op}_S = \mathsf{op}_n$.

To algorithmically obtain the unit with this identifier, FindRootUnit (see Algorithm 1) iterates over all units contained in the COLA system. The resulting unit instance is then the root instance of the COLA system.

The evaluation is performed until all enabled unit instances have been evaluated once by EvaluateUnit, which is described in Algorithm 3.

Algorithm 1 FindRootUnit($\langle u, U \rangle$)

```
1: foreach (⟨n, σ, c, I⟩ ∈ U) do
2:     if (n = u) then
3:         return ⟨n, σ, c, I⟩
```

Using FindRootUnit, Algorithm 2 first selects the root unit from the set of all units, U, and then runs the system. Each cycle of the while-loop corresponds to one tick. Note that no input needs to be consumed here and no output is generated, as the root unit of a COLA system has no ports. However, the internal state may be modified.

Algorithm 2 EvaluateColaSystem($\langle u, U \rangle$)

```
1: rootInstance = FindRootUnit(⟨u, U⟩)
2: while (true) do
3:     EvaluateUnit(rootInstance, root)
```

The way a unit is evaluated in each tick of the run is different for basic blocks and composite units that were introduced in Section 3. Algorithm 3 thus calls the respective functions, which are defined below, depending on the classifier of the unit $\langle n, \sigma, c, I \rangle$ with unique identifier x at hand. Each function returns the results of the evaluation of the unit.

Definition 10 (Semantics of a system). *The semantics of a system* $S = \langle u, U \rangle$ *is given by the semantic relation of the unit* $\langle u, \sigma, c, I \rangle \in U$*. That is,* $\mathsf{op}_S = \mathsf{op}_u$*. A* tick *is defined as a single evaluation of* op_S *for a given internal state.*

Following the top-down manner of evaluation of units, first the semantics of networks is given. As networks strongly depend on delays, these are considered next. Finally, automata and functional blocks are described.

Algorithm 3 EvaluateUnit($u = \langle n, \sigma, c, I \rangle, ui$)

1: **if** ($c =$ fblock) **then**
2: **return** EvaluateFunctionalBlock(u, ui)
3: **else if** ($c =$ tblock) **then**
4: **return** EvaluateTimingBlock(u, ui)
5: **else if** ($c =$ network) **then**
6: **return** EvaluateNetwork(u, ui)
7: **else if** ($c =$ automaton) **then**
8: **return** EvaluateAutomaton(u, ui)

5.3 Networks

To reason about a set of interconnected units, first the impact of channels, which were described in Definition 4, on the semantics shall be discussed. First, a channel implies that at each tick the value at its source equals the value present at the destination ports. Second, as consequence of this, channels effect the order of evaluation of the units contained in a network, because input values to a unit must be known prior to its evaluation. Thus a unit providing data to a channel has to be evaluated before any other unit which is connected to the destination side of the channel.

To reason about these data dependencies in the network, let $G = (V, E)$ be the directed graph spanned by the network $\langle n, \sigma, \text{network}, \langle \{u_1, \ldots, u_k\}, \mathsf{inst}, C \rangle \rangle$ in the following way. Each source-destination pair of a channel forms an edge in the graph, whereas the vertexes correspond to the sub-units. Thus, $V = \{u_1, \ldots, u_k\}$, and $E = \{(s, d) \mid \exists c \in C \ : \ s.p = \mathsf{src}(c) \wedge d.p' \in \mathsf{dest}(c)\}$, where $s.p$ denotes port p of the sub-unit $s \in \{u_1, \ldots, u_k\}$, $d.p'$ respectively, as detailed in Definition 5 on page 12.

The required evaluation order, which is a topological sorting of the vertexes, may be obtained by a breadth-first traversal of the graph of a network [33]. While this is straightforward in case of an acyclic graph, networks with data cycles must be looked at more closely. Despite their incoming channels, delays also constitute valid starting points of the traversal, because the output of these blocks may be read from without prior knowledge of the input. Their output only depends on the—known—internal state (see Definition 12).

We thus modify the graph such that there are no edges corresponding to outgoing channels of delays:

$$E' = \{(s, d) \mid \exists c \in C \ : \ s.p = \mathsf{src}(c) \wedge d.p' \in \mathsf{dest}(c) \\ \wedge (\mathsf{inst}(s) = \langle n', \sigma', c', I' \rangle \Rightarrow c' \neq \text{tblock})\}$$

Furthermore, a single starting point, v_0, of the traversal is defined as an additional vertex, i. e., $V = \{u_1, \ldots, u_k, v_0\}$. Consequently, edges from v_0 to all vertexes $u_i \in \{u_1, \ldots, u_k\}$ with an indegree of 0, denoted by $\mathsf{deg}^+(u_i) = 0$, must be added. These vertexes correspond to units without input ports or units that only depend on inputs to the network or input from delays; all of them may be evalu-

ated immediately. We then define $E = E' \cup \{(v_0, x) \mid x \in \{u_1, \ldots, u_k\} \wedge \mathsf{deg}^+(x) = 0\}$.

The evaluation order corresponds to a breadth-first traversal of the graph G, which is acyclic for a COLA network as any data cycle there must contain at least one delay. The acyclic graph canonically extends to a preorder $\preceq$ on the vertexes in that

$$v \preceq v' \Leftrightarrow (v, v') \in E \vee \exists v'' : (v, v'') \in E \wedge v'' \preceq v'.$$

Consequently, the evaluation of a network, as described in Algorithm 5, must proceed such that for any pair of sub-units u_i and u_j with $u_i \preceq u_j$, u_i is evaluated before u_j. While performing the evaluation, output port valuations must be forwarded to input ports, as implied by the channels. To facilitate the latter, the helper function getDestPorts is defined in Algorithm 4.

Algorithm 4 getDestPorts(C, p)

▷ Returns all destination ports connected to port p by a channel in C

1: $result = \emptyset$
2: **foreach** ($c \in C$) **do**
3:     **if** ($p = \mathsf{src}(c)$) **then**
4:         $result = result \cup \mathsf{dest}(c)$
5: **return** $result$

Definition 11 (Semantics of a network). *Let a unit $\langle n, \sigma, \text{network}, I\rangle$ with an implementation $I = \langle\{u_1, \ldots, u_k\}, \mathsf{inst}, C\rangle$, $k \in \mathbb{N}$, and a unique identifier x be a network. Further let $\mathsf{in}(\sigma) = (a_1, \ldots, a_l)$ and $\mathsf{out}(\sigma) = (b_1, \ldots, b_m)$ with $l, m \in \mathbb{N}$ be the input and output port names.*

The semantic relation op_n is defined in terms of the semantic relations of the sub-units. Assume that, for any $1 \leq j \leq k$, $\mathsf{inst}(u_j) = \langle n_j, \sigma_j, c_j, I_j\rangle$ with $\mathsf{in}(\sigma_j) = (a^j_1, \ldots, a^j_{l_j})$ and $\mathsf{out}(\sigma_j) = (b^j_1, \ldots, b^j_{m_j})$ with $l_j, m_j \in \mathbb{N}$. Let s_i with $1 \leq i \leq k$ be defined to be $state[x.u_i]$, further, for $1 \leq i \leq k$, let $s'_i \in \mathsf{dom}(state[x.u_i])$.

$$\begin{aligned}
&((s_1, \ldots, s_k), (i_1, \ldots, i_l), (s'_1, \ldots, s'_k), (o_1, \ldots, o_m)) \in \mathsf{op}_n \Leftrightarrow \\
&\quad \forall j \in \{1, \ldots, k\}\ \exists(i^j_1, \ldots, i^j_{l_j})\ \exists(o^j_1, \ldots, o^j_{m_j}) \ : \\
&\quad (s_j, (i^j_1, \ldots, i^j_{l_j}), s'_j, (o^j_1, \ldots, o^j_{m_j})) \in \mathsf{op}_{n_j} \wedge \forall c \in C\ \forall d \in \mathsf{dest}(c) \ : \\
&\qquad\qquad\qquad\qquad \textit{channel from an input to an output port of } u \\
&\qquad (\exists p \in \{1, \ldots, l\}\ \exists q \in \{1, \ldots, m\} \ : \\
&\qquad\quad \mathsf{src}(c) = a_p \wedge d = b_q \Rightarrow i_p = o_q) \vee \\
&\qquad\qquad\qquad\qquad \textit{channel from an input of } u \textit{ to an input port of } u_j \\
&\qquad (\exists p \in \{1, \ldots, l\}\ \exists q \in \{1, \ldots, m_j\} \ : \\
&\qquad\quad \mathsf{src}(c) = a_p \wedge d = u_j.a^j_q \Rightarrow i_p = i^j_q) \vee
\end{aligned}$$

channel from an output of u_j to an output port of u

$$(\exists p \in \{1,\ldots,l_j\}\ \exists q \in \{1,\ldots,m\}\ :$$
$$\mathsf{src}(c) = u_j.b^j_p \wedge d = b_q \Rightarrow o^j_p = o_q) \vee$$

channel from an output of u_j to an input port of $u_{j'}$

$$(\exists p \in \{1,\ldots,l_j\}\ \exists q \in \{1,\ldots,m_j\}\ \exists j' \in \{1,\ldots,k\}\ :$$
$$\mathsf{src}(c) = u_j.b^j_p \wedge d = u_{j'}.a^{j'}_q \Rightarrow o^j_p = i^{j'}_q)$$

Algorithm 5 outlines the basic procedure to evaluate a network. In this algorithm, **choose** is used to select an item from a set of items fulfilling a given property. **choose** works in a non-deterministic fashion, i. e., if there are more than one items with the demanded property, one item is non-deterministically picked and returned. For an effective implementation, e. g., for code generation, it is essential to resolve this non-determinism by choosing arbitrarily.

Given a COLA system it is guaranteed that **choose** always works on a non-empty set, because by Definition 5 on page 12, requirement *cycle validity*, there must be at least one delay in a cycle.

Two further technical details must be well considered for Algorithm 5. First, as defined above, val is specific to each unit instance, and thus denoted as $val_{x'}$ for a unique identifier x'. Second, as detailed in Definition 5, port identifiers used in channels are of the form $u_i.p$ for a sub-unit identifier u_i and a port name p, and of the form p, if the port is an input or output port of the network. This scheme is also used to obtain the unit identifier from a port name (see, e. g., lines 10–11).

5.4 Timing Block

As discussed above, delays—which are the only timing blocks considered in this paper—are fundamental when evaluating networks. Intuitively, they are a representation of memory, which is initialized with its default value. The semantics is as follows: At the first evaluation of the delay, i. e., at the first tick, the default value is written to the output and the input is written as the new internal state. In all further steps, the internal state is written to the output and again the input is stored in the internal state.

Recall that the type of the internal state equals the type of the input and output port, and also that of the default value. The semantic relation op_n then exchanges the previous state and the input value in that it stores the latter as the new state and sets the output to the previous state.

Definition 12 (Semantics of a delay). *Let $\langle n, \sigma, \text{tblock}, \mathsf{delay}_c\rangle$ be a delay with unique identifier x and $t = \mathsf{type}(\mathsf{in}(\sigma))$. Further, let $state[x] = m$, with $m \in \mathsf{dom}(t)$, denote the current state of the delay, which equals c at the first tick, i. e., the first emitted value is c. Subsequently, all delay outputs $i' \in \mathsf{dom}(t)$ are input values $i \in \mathsf{dom}(t)$ that were delayed for exactly one tick. Then, op_n is defined as follows:*

$$\forall m, i, m', i' \in \mathsf{dom}(t)\ :\ (m, i, i', m') \in \mathsf{op}_n \Leftrightarrow (i = i' \wedge m = m')$$

Algorithm 5 $\mathsf{EvaluateNetwork}(\langle n, \sigma, \text{network}, \langle \{u_1, \ldots, u_k\}, \mathsf{inst}, C \rangle \rangle, ui)$

```
Require: out(σ) = (b_1, ..., b_m)
                    ▷ Propagate values of input ports to the ports of connected sub-units
 1: foreach (c ∈ C : src(c) ∈ in(σ)) do
 2:     ports = dest(c)
 3:     foreach (d.p ∈ ports) do
 4:         ports = ports \ {d.p}
 5:         val_{ui.d}[p] = val_{ui}[src(c)]
                    ▷ Propagate values of input ports to output ports, if applicable
 6:     foreach (p ∈ ports) do
 7:         val_{ui}[p] = val_{ui}[src(c)]
                    ▷ Propagate the current values of delays to ports of connected sub-units
 8: foreach (u ∈ {u_1, ..., u_k} : inst(u) = ⟨n', σ', tblock, delay_c⟩) do
                    ▷ Get the name of the output port to build a valid port identifier in line 10
 9:     p' = out(σ')
10:     foreach (d.p ∈ getDestPorts(C, u.p')) do
11:         val_{ui.d}[p] = state[ui.u]

12: units = {u_1, ..., u_k}        ▷ Set of unit identifiers contained in the network
                                   ▷ Evaluate all sub-units in an appropriate order
13: while (units ≠ ∅) do
14:     choose u such that u ∈ units ∧ ∄u' ∈ units : u' ⪯ u
15:     units = units \ {u}
16:     result = EvaluateUnit(inst(u), ui.u)
17:     foreach (p ∈ out(σ')) do
18:         val_{ui.u}[p] = result[p]          ▷ Set values of the output ports
                                               ▷ Propagate values to connected units
19:         ports = getDestPorts(C, u.p)
20:         foreach (d.p' ∈ ports) do
21:             ports = ports \ {d.p'}
22:             val_{ui.d}[p'] = val_{ui.u}[p]
                    ▷ Propagate values to output ports, if applicable
23:         foreach (p' ∈ ports) do
24:             val_{ui}[p'] = val_{ui.u}[p]
25: return (val_{ui}[b_1], ..., val_{ui}[b_m])
```

While $\mathsf{EvaluateTimingBlock}$ in Algorithm 6 describes the semantic relation in terms of an interpreter, the evaluation of delays within networks first requires writing the output to connected units. This step can be performed without prior knowledge of the input and is executed at the beginning of Algorithm 5.

Algorithm 6 $\mathsf{EvaluateTimingBlock}(\langle n, (\langle i : t\rangle \rightarrowtail \langle o : t\rangle), \mathsf{delay}_c\rangle, ui)$

1: $result = state[ui]$
2: $state[ui] = val_{ui}[i]$
3: **return** $result$

5.5 Automata

The semantics of units which are implemented as automata is based on the state transition relation of the automaton and the evaluation of the unit associated with the enabled state. For selecting the next state the definition of the state transition relation possibly takes into account the inputs of the unit containing the automaton. Whenever a state and a respective unit is enabled, it is acting on behalf of the overall unit. All passive sub-units are frozen, i. e., their state is preserved until they become enabled again. This could have been dealt with in other ways, e. g., the disabled sub-units could be reset on activation or even continue their operation while discarding their outputs.

Given an automaton $\langle n, \sigma, \text{automaton}, \langle Q, \mathsf{inst}, q_0, \Delta\rangle\rangle$, assume q_l to be the current state and $q_{l'}$, with $0 \leq l, l' \leq k$, to be another state. A transition is enabled if and only if there exists a $\delta \in \Delta$ with $\delta = (q_l, i, q_{l'})$ for some $i \in \mathsf{dom}(\mathsf{in}(\sigma))$. The required evaluation of the transition relation takes place before any respective sub-unit is evaluated.

Let $o \in \mathsf{dom}(\mathsf{out}(\sigma))$ be an output of the automaton. Then, $(s, i, s', o) \in \mathsf{op}_n$ is a tuple in the relation op_n of the automaton with the state vectors $s = (q_l, s_0, \ldots, s_k)$ and $s' = (q_{l'}, s'_0, \ldots, s'_k)$ where the latter represents the state vector of the next state.

Let op_{n_l} and $\mathsf{op}_{n_{l'}}$ be the relations for the current and the next automaton state realized by the units $\langle n_l, \sigma_l, c_l, I_l\rangle$ and $\langle n_{l'}, \sigma_{l'}, c_{l'}, I_{l'}\rangle$ for any $l, l' \in \{0, \ldots, k\}$. Then we distinguish the following two cases:

- If the automaton remains in the current state, i. e., $l = l'$ then the relation $(s_l, i, s'_l, o) \in \mathsf{op}_{n_l}$ holds. Further, $s_j = s'_j$ for all $j \in \{0, \ldots, k\}, j \neq l$, i. e., only the currently enabled sub-unit is changes its state and produces output.
- On the other hand, if a transition is taken, i. e., $l \neq l'$ then it is required that $(s_{l'}, i, s'_{l'}, o) \in \mathsf{op}_{n_{l'}}$ and that $s_j = s'_j$ for all $j \in \{0, \ldots, k\}, j \neq l'$.

Algorithm 7 figures out the behavior of an automaton.

Definition 13 (Semantics of an automaton). *Let $\langle n, \sigma, \text{automaton}, I\rangle$ be an automaton with an implementation $I = \langle \{q_0, \ldots, q_k\}, \mathsf{inst}, q_0, \Delta\rangle$, $k \in \mathbb{N}$.*

Algorithm 7 $\mathsf{EvaluateAutomaton}(\langle n, \sigma, \text{automaton}, I\rangle, ui)$

Require: $\mathsf{in}(\sigma) = (a_1, \ldots, a_l)$
Require: $I = \langle\{q_0, \ldots, q_k\}, \mathsf{inst}, q_0, \Delta\rangle$
Require: $state[ui] = (q_e, state[ui.q_0], \ldots, state[ui.q_k])$
1: **if** $(\exists q \in Q\ :\ (q_e, (val_{ui}[a_1], \ldots, val_{ui}[a_l]), q) \in \Delta)$ **then**
2:     **choose** q **such that** $q \in Q\ :\ (q_e, (val_{ui}[a_1], \ldots, val_{ui}[a_l]), q) \in \Delta$
3:     **return** $\mathsf{EvaluateUnit}(\mathsf{inst}(q), ui.q)$
4: **else**
5:     **return** $\mathsf{EvaluateUnit}(\mathsf{inst}(q_e), ui.q_e)$

For any $d, e \in \{0, \ldots, k\}$, the semantic relation op_n is defined in terms of the semantic relation of the enabled state, q_e. Without loss of generality assume $d \leq e$.

$$((q_d, s_0, \ldots, s_d, \ldots, s_e, \ldots, s_k), i, (q_e, s_0, \ldots, s_d, \ldots, s'_e, \ldots, s_k), o) \in \mathsf{op}_n$$
$$\Leftrightarrow (d = e \vee ((q_d, i, q_e) \in \Delta)) \wedge (s_e, i, s'_e, o) \in \mathsf{op}_{n_e}$$

with $\mathsf{inst}(q_d) = \langle n_d, \sigma_d, c_d, I_d\rangle$ *and* $\mathsf{inst}(q_e) = \langle n_e, \sigma_e, c_e, I_e\rangle$.

With this semantics, the state transition relation is evaluated first to select the next enabled state. Thus it is possible that the sub-unit which is enabled during the first tick does not coincide with the initial state of the automaton.

5.6 Functional Blocks

In case of functional blocks op_n is canonically obtained from the implementing relation I that is specific to each functional block $\langle n, \sigma, \text{fblock}, I\rangle$ with unique identifier x. As these blocks are stateless, any pair $(u, v) \in I$ corresponds to a tuple $(\emptyset, u, \emptyset, v) \in \mathsf{op}_n$, because $state[x] = \emptyset$.

Note that the algorithmic description given in Algorithm 8 has to cope with the possible non-determinism involved in the relation I, where **choose** as outlined above is used again.

Definition 14 (Semantics of a functional block). *For a functional block $b = \langle n, \sigma, \text{fblock}, I\rangle$ the semantic relation op_x is defined as follows:*

$$(\emptyset, u, \emptyset, v) \in \mathsf{op}_n \Leftrightarrow (u, v) \in I$$

Assume $|\mathsf{in}(\sigma)| = l$ and $|\mathsf{out}(\sigma)| = m$. On input $(i_1, \ldots, i_l)$ to unit b, EvaluateFunctionalBlock returns $(o_1, \ldots, o_m)$ such that $((i_1, \ldots, i_l), (o_1, \ldots, o_m)) \in I$.

6 Visualization

This section introduces graphical representations of the syntactic elements of COLA as defined above.

Algorithm 8 EvaluateFunctionalBlock($\langle n, \sigma, \text{fblock}, I \rangle$, ui)

Require: $\text{in}(\sigma) = (a_1, \ldots, a_l)$
1: **choose** $result$ **such that** $((val_{ui}[a_1], \ldots, val_{ui}[a_l]), result) \in I$
2: **return** $result$

6.1 Unit

A unit is denoted by a box labeled with the identifier of the unit. The input and output ports belonging to the unit, as defined by the signature, are represented by small triangles drawn on the border line of the unit, resembling to an arrowhead. Input ports are triangles pointing into the rectangle, while output ports point outwards. In Figure 8(a) a unit $\langle n, \sigma, c, I \rangle$ with $\sigma = (P_{in} \rightarrowtail P_{out})$ is depicted. The unit has three input ports $\text{in1}, \text{in2}, \text{in3} \in P_{in}$ and two output ports $\text{out1}, \text{out2} \in P_{out}$. The ports are annotated with their respective implementation data types.

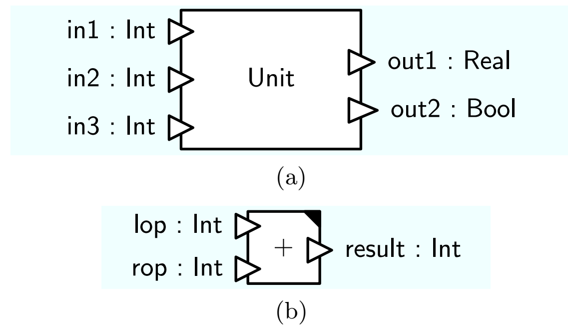


Figure 8. (a) An arbitrary COLA unit. (b) Functional block (adder) with two input and one output port.

If a unit cannot be further decomposed into sub-units, this is indicated by drawing a black triangle in the right upper corner of the unit. Figure 8(b) depicts an example of such a block.

6.2 Delay

A specialization of a unit is the delay. A delay is depicted by two vertical lines drawn in parallel. It has exactly one input and output port, represented by the triangles on the border. Its default value is written centered above the two lines. Figure 9 shows a delay with type information.

6.3 Network

In Section 3.4 composed units were introduced. One possibility of composing such a non-primitive unit is a network of units. The example in Figure 10 shows

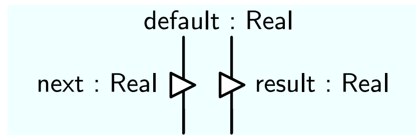


Figure 9. A COLA delay with a typed input and output port and a default value of the same type.

a network with the three units Unit1, Unit2, Unit3 and a delay. Each output

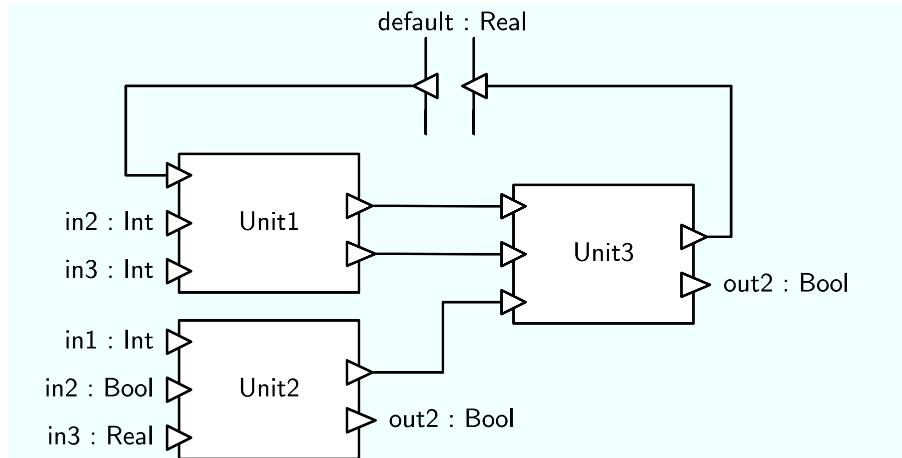


Figure 10. A network of units

port of Unit1 is connected to an input port of unit Unit3 via channels (for ease of readability, the identifiers of connected ports are omitted). The unconnected input ports shown in the example are the input ports of the network, and thus connected to the ports of the unit containing the network. The unconnected output ports, on the other hand, may stay unconnected or can be connected to the output ports of the containing network. The exact definition is given in Definition 5.

6.4 Automaton

Besides networks, units can be implemented by an automaton as specified in Definition 6. The visual representation of automata is similar to that used in other modeling languages. The automaton's states are depicted by ellipses containing the states' identifiers. Transitions between states are arrows annotated with predicates which are Boolean expressions for the transitions' guards. The initial state is marked by an arrow leading into the respective state. Figure 11

shows an automaton with its two states State1 and State2 and two corresponding transitions annotated with their guards. State1 is marked as the initial state.

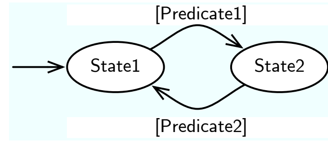


Figure 11. A COLA automaton with two states

The details of the visualization of decomposition are not discussed here. It could be realized in several ways but basically it is a substitution of a unit by its contained network or automaton.

7 Conclusions and ongoing work

This paper gave a formal and graphical description of COLA, the component language for the development of embedded control software. Consequently, the formal syntax, its synchronous dataflow semantics, and its visualization have been discussed and examplified on a case study.

Systems in COLA are synchronous dataflow networks consisting of hierarchically composed units, whose individual behaviors are defined by further units and/or automata. Real-time requirements of the system are expressed with respect to a discrete uniform time base.

However, a detailed *clock calculus* for COLA is not subject of this paper. A clock calculus facilitates the expression of real-time requirements, such that different signal frequencies and activation times can be expressed (and checked) in a convenient manner. COLA as described in this paper can be extended with such a calculus, in a straightforward manner. Still, the exact constraints imposed by the calculus and the methodological implications on the systems described by COLA are in the process of being evaluated.

Together with an appropriate tool-chain, COLA is currently applied in a case study to model an adaptive cruise control. To verify certain safety properties, model checking is used, and C code is to be generated automatically. Using a specifically crafted middleware, the system is deployed onto an experimental embedded platform to prove practical applicability and seamless integration without any manual modifications.

Despite already being usable, the formalism described in this document lacks some concepts that we found to be necessary when performing the described and several other case studies. Namely these are blocks describing input- and output interfaces, called *sources* and *sinks*, parametrization of units, and *legacy blocks* to interface with existing source code. Parametrization is used to describe constants and is required for the delay blocks.

These extensions are currently being integrated into the COLA framework and will be described in a separate document. Furthermore, support for complex data types is being added to the language.

Acknowledgments We thank Manfred Broy, Reinhardt Hierl, and Helmut Veith for their invaluable comments on earlier versions of this paper.

References

1. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide. Addison-Wesley (1998)
2. The MathWorks Inc.: Using Simulink. (2000)
3. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge, Massachusetts (1999)
4. Benveniste, A., Caspi, P., Edwards, S.A., Halbwachs, N., Le Guernic, P., de Simone, R.: The synchronous languages 12 years later. Proceedings of the IEEE **91**(1) (2003)
5. Bauer, A., Broy, M., Romberg, J., Schätz, B., Braun, P., Freund, U., Mata, N., Sandner, R., Ziegenbein, D.: AutoMoDe — Notations, Methods, and Tools for Model-Based Development of Automotive Software. In: Proceedings of the SAE 2005 World Congress, Detroit, MI, Society of Automotive Engineers (2005)
6. : IEEE Std 830-1998: IEEE Recommended Practice for Software Requirements Specifications. Institute of Electrical and Electronics Engineers (1998)
7. Maraninchi, F., Rémond, Y.: Mode-automata: a new domain-specific construct for the development of safe critical systems. Science of Computer Programming **46**(3) (2003) 219–254
8. Stürmer, I., Weinberg, D., Conrad, M.: Overview of existing safeguarding techniques for automatically generated code. SIGSOFT Softw. Eng. Notes **30**(4) (2005) 1–6
9. Caspi, P., Curic, A., Maignan, A., Sofronis, C., Tripakis, S., Niebert, P.: From simulink to SCADE/lustre to TTA: a layered approach for distributed embedded applications. In: LCTES, ACM (2003) 153–162
10. Gautier, T., Guernic, P.L., Besnard, L.: Signal: A declarative language for synchronous programming of real-time systems. In: Proceedings of a conference on Functional programming languages and computer architecture, London, UK, Springer-Verlag (1987) 257–277
11. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data-flow programming language LUSTRE. Proceedings of the IEEE **79**(9) (1991) 1305–1320
12. Pouzet, M.: Lucid Synchrone, version 3. Tutorial and reference manual. Université Paris-Sud, LRI. (2006) Distribution available at: `www.lri.fr/~pouzet/lucid-synchrone`.
13. Berry, G., Gonthier, G.: The esterel synchronous programming language: Design, semantics, implementation. Science of Computer Programming **19**(2) (1992) 87–152
14. Le Guernic, P., Gautier, T., Le Borgne, M., Le Maire, C.: Programming real-time applications with Signal. Proceedings of the IEEE **79**(9) (1991) 1321–1336
15. Broy, M., Stølen, K.: Specification and development of interactive systems: Focus on streams, interfaces, and refinement. Springer-Verlag, New York (2001)

16. Broy, M., Huber, F., Schätz, B.: AutoFocus – Ein Werkzeugprototyp zur Entwicklung eingebetteter Systeme. Informatik Forschung und Entwicklung **13**(3) (1999) 121–134
17. Romberg, J.: Synthesis of distributed systems from synchronous dataflow programs. PhD thesis, Technische Universität München (2006)
18. Romberg, J., Bauer, A.: Loose Synchronization of Event-Triggered Networks for Distribution of Synchronous Programs. In: Proceedings of the 4th ACM International Conference on Embedded Software (EMSOFT), Pisa, Italy, Association for Computing Machinery (2004) 193–202
19. Chapiro, D.M.: Globally-asynchronous locally-synchronous systems. PhD thesis, Stanford University (1984)
20. Berry, G., Sentovich, E.: Embedding Synchronous Circuits in GALS-based Systems. In: Sophia Antipolis Forum on MicroElectronics (SAME 1998), Sophia Antipolis, France (1998)
21. Chakraborty, S., Mekie, J., Sharma, D.K.: Reasoning about synchronization in GALS systems. Formal Methods in System Design **28**(2) (2006) 153–169
22. Doucet, F., Menarini, M., Krüger, I.H., Gupta, R.K., Talpin, J.P.: A verification approach for GALS integration of synchronous components. Electr. Notes Theor. Comput. Sci. **146**(2) (2006) 105–131
23. France, R.B., Evans, A., Lano, K., Rumpe, B.: The UML as a formal modeling notation. Computer Standards & Interfaces **19**(7) (1998) 325–334
24. Amálio, N., Polack, F.: Comparison of formalisation approaches of UML class constructs in Z and Object-Z. In Bert, D., Bowen, J.P., King, S., Waldén, M.A., eds.: ZB. Volume 2651 of Lecture Notes in Computer Science., Springer (2003) 339–358
25. Balarin, F., Watanabe, Y., Hsieh, H., Lavagno, L., Passerone, C., Sangiovanni-Vincentelli, A.L.: Metropolis: An integrated electronic system design environment. IEEE Computer **36**(4) (2003) 45–52
26. Feiler, P.H., Lewis, B., Vestal, S.: The SAE avionics architecture description language (AADL) standard: A basis for model-based architecture-driven embedded systems engineering. In: Proceedings of the RTAS 2003 Workshop on Model-Driven Embedded Systems (MDES), Washington, DC (2003)
27. GmbH, R.B.: Kraftfahrtechnisches Taschenbuch. 26th edn. Vieweg (2007)
28. Cardelli, L., Wegner, P.: On understanding types, data abstraction, and polymorphism. ACM Computing Surveys **17**(4) (1985) 471–522
29. Kühnel, C., Bauer, A., Tautschnig, M.: Compatibility and reuse in component-based systems via type and unit inference. In: Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), IEEE Computer Society Press (2007)
30. Moore, E.F.: Gedanken-experiments on sequential machines. In Shannon, C.E., MacCarthy, J., eds.: Automata Studies, Princeton University Press (1956) 129–153
31. : ISO/IEC 14977:1996: Information technology — Syntactic metalanguage — Extended BNF. International Organization for Standardization, Geneva, Switzerland (1996)
32. Bauer, A., Leucker, M., Streit, J.: SALT—structured assertion language for temporal logic. In: Proceedings of the Eighth International Conference on Formal Engineering Methods. Volume 4260 of Lecture Notes in Computer Science. (2006) 757–775
33. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, Second Edition. The MIT Press and McGraw-Hill Book Company (2001)