From COLA Models to Distributed Embedded Systems Code

Wolfgang Haberl [*] Michael Tautschnig [†] Uwe Baumgarten [*]

Abstract—**Model driven development has become state of the art in embedded systems software design. To take the resulting models to the designated hardware platform, automated code generation is sought for. The code obtained thereby must match the semantics of the model as closely as possible.**

In this paper we present an approach to translate Component Language (COLA) models to C code in an automated manner. This process involves handling of target specific sensors and actuators, and distribution of software across multiple nodes. We therefore also describe our middleware layer, which is configured in an unattended manner.

Using a prototypical implementation of the toolchain, we validated our approach on a case study using LEGO® Mindstorms™, which shows all characteristics of embedded systems. This case study includes benchmarks comparing the automatically generated code to a hand-written version providing the same functionality.

Keywords: embedded systems, component-based models, automated code generation

1 Introduction

Today, more than 90% of all processors are part of embedded systems [1, 2]. These are integrated in laundry machines, medical systems, cars, and aircrafts, just to name a few. In this paper we focus on distributed embedded systems, built up from dozens or even hundreds of computing nodes, interconnected by various bus systems. Such systems contain or constitute life-critical electronic resources. Faults, of any kind, thus may be fatal. Even if not fatal, they bare large warranty costs for the designers and integrators of the product.

Model driven development (MDD) is well established as a means of tackling the complexity involved in designing such distributed embedded systems. As the large scale prohibits engineers from grasping the entire system at once, a hierarchy of abstractions is applied to attain manageability at each level. The Component Language (COLA) [3] was built to tackle the complexity of such large scale systems in a framework consistent across all layers of abstraction. COLA has a slender syntax, which caters for easy understanding, and it is defined by a rigorous formal semantics. At higher levels of abstraction then formal verification, e.g., using model checking, can be applied to guarantee conformance with requirements. These characteristics make COLA well-suited for development of mission-critical real-time systems.

The complexity of the modeled system not only necessitates proper abstractions, but also calls for automation to take a model to an executable object, and later to a functional integrated system. An automated translation reduces errors and further guarantees reproducible results, and thus improves overall quality. The process of translation is best compared to that of a software compiler. Given a functional model, usually as a piece of source code, a runnable entity is produced. A simple model of the system will comprise three layers: the software, an operating system, and the (single piece of) hardware. Compiler and linker will be given all constraints imposed by the operating system and the target hardware platform to obtain an appropriate piece of software. Apart from the straightforward translation, a fundamental job of today's compilers is optimization in terms of size and execution speed. Therefore, a set of rules is built into the compiler to obtain runnable entities optimized for a given platform and operation system.

In embedded systems, we will call this process *systems compilation*, since the compilation must be accomplished for the overall system model, where the involved software and hardware components may be of various types. The target platform to operate on likely invalidates several assumptions made at model level, or exposes properties that are not captured by functional/behavioral modeling. Furthermore, the distributed nature of the target hardware must be accounted for.

Here, both translation and optimization are by no means straightforward. As per translation, often a heterogeneous heap of models and requirements must be considered to obtain a valid runnable entity (cf. [4]). Further, in general a certain level of black magic performed by engineers is required to fit the software and hardware components onto the target platform.

[*]Institut für Informatik, Technische Universität München, 85748 Garching, Germany, {haberl, baumgaru}@in.tum.de

[†]Institut für Informatik, Technische Universität Darmstadt, 64289 Darmstadt, Germany, tautschnig@cs.tu-darmstadt.de

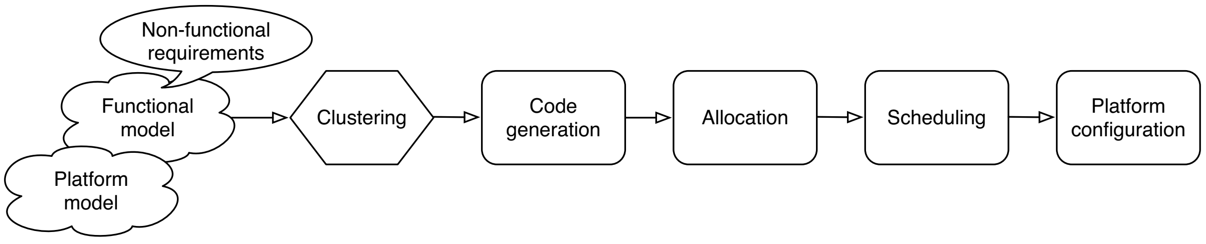

Figure 1: Systems compilation steps

In this paper, we focus on automation of the translation steps, and consider local optimizations only. Additional system-wide optimizations based on this framework are discussed in [5]. The process of automated systems compilation is outlined in Figure 1. Using COLA, we need not handle a multitude of heterogeneous models, but rather benefit from the consistent modeling formalism and start with a single, behavioral, model. It contains a *clustering* (see Section 3.5), which defines the tasks that must be allocated to hardware nodes, which are captured by the platform model. Therefrom, and based on the formal semantics of COLA, we generate C code that may be compiled (using an ordinary software compiler) for a target platform. As part of systems compilation, task allocation and scheduling are computed. Based on this information, the configuration of the middleware layer, which handles all communication, is generated automatically. At this point, the obtained executables are tailored towards the specific platform and require no further manual intervention before effective deployment (flashing) to the target hardware system.

1.1 Related work

Being the favorite approach, there is extensive tool support for graphical modeling, e.g., MATLAB/Simulink, ASCET-SD, or SCADE, which is based on Lustre [6]. Furthermore, the Unified Modeling Language (UML) has become an industry standard that includes means for component-based modeling. Initially targeted at avionic systems, the Architecture Analysis and Design Language (AADL) [7] also offers well-defined models aimed at industrial applications.

In a tool-backed MDD process, however, not only behavioral modeling is sought for, but several views of the complete system must be distinguished. Following the nomenclature of Pretschner et al. [8], behavioral models form the *logical architecture*. The description of the target hardware platform and other non-functional requirements then comprise the *technical architecture*. The latter is usually excluded in the modeling formalisms mentioned above, or rather, focused on with a lack of description of the logical layer. This kind of lower abstraction is applied, e.g., in the Metropolis project [9] and the CARAT toolkit [10]. An approach aiming at both layers is that of the VERTAF framework [11].

COLA initially also focused on the logical layer. Meanwhile it was extended to include a hardware model and further means to describe the technical architecture in a consistent formalism. This enables the COLA based process to include allocation and scheduling of software to a given hardware platform, rather then just coding single tasks. Whenever a platform description is available, executable code may be generated from the behavioral (graphical) models. Examples of such compilers include TargetLink, Real-Time Workshop/Embedded Coder, ASCET-SC, and SCADE Drive. An overview and comparisons of these tools have been presented by Reuter [12] as well as Wybo and Putti [13].

1.2 Automated code generation

As included in the requirements expressed by Whalen and Heimdahl [14], ensuring correctness of the translation requires both the source and the target language to have a rigorous formal semantics. While this is hardly viable for the target language C, our source language COLA provides these. Consequently, we only rely on a subset of C and a template-like set of transformation rules such that preservation of behavior of the generator remains easy to check. At this point it should be noted that COLA has a slender syntax and thus requires only a small number of transformation rules. The elements of COLA are introduced in Section 2.

Correctness of a distributed system not only relies on the semantically correct transformation of parts of the model into C code, but on the interaction of tasks. A COLA model is intended to cover complete systems, and thus defines data dependencies between groups of tasks. A cluster is the COLA model representation of a task from the operating systems point of view. As such, a COLA model is partitioned into *clusters* after a first iteration of functional design. Code generation then must take care of inter-cluster communication (see Section 3.5) in a way independent of the allocation, which is performed as a subsequent step in systems compilation. This transparency is enabled by our middleware layer, which has been described in [15].

1.3 Organization

In the following we first give a short introduction to COLA. In Section 3 the basic rules for compiling such models to C code are described. The steps required to effectively run the generated code on a specific target platform are outlined in Section 4. The results on a case study are presented in Section 5, including considerations of the code efficiency. We conclude with an outlook on possible enhancements.

Throughout the paper we use parts of the model of the case study to illustrate the abstract transformation steps.

2 Overview of COLA

The key concept of COLA is that of *units*. These can be composed hierarchically, or occur in terms of *blocks* that define the basic (arithmetic) operations of an application.

Each unit has a set of typed *ports* describing the interface. These ports form the *signature of the unit*, and 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 of sub-units in a network are called *channels* and connect an output port with one or more suitably typed input ports (cf. [16]).

In addition to the hierarchy of networks, COLA provides a decomposition into *automata* (i. e., finite state machines, similar to Statecharts [17]). 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 that particular state. This definition of an automaton is therefore well-suited to partition complex networks of units into disjoint *operating modes* (cf. [18]), the activation of which depends on the input signals of the automaton.

The collection of all units forms a COLA *system*, which models the application, possibly including its environment. Such a system does not have any unconnected input or output ports as there would be no way to provide input to systems. For effective communication with the environment not describable within the functional COLA model, *sources* and *sinks* model connectors to the underlying hardware. Sources are the model representation of sensors, and sinks correspond to actuators of the used hardware platform.

Distributable software modules are specified by *clusters*, which correspond to tasks at the operating system level. Each cluster has a distinguished root unit. Whenever this is a network or an automaton, the respective sub-units become part of the same cluster. Consequently, a cluster cannot have any sub-clusters. Code generation then yields a C source file for each cluster.

2.1 Semantics

COLA is a synchronous data flow language. It is assumed that operations start at the same instant of time and are performed simultaneously with respect to data dependencies. 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. 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.

To retain data for a series of ticks, the concept of *delays* is introduced. These blocks model memory by saving the actual input value and providing the input of the previous tick of the global clock at the output port. At the first tick, where no prior input is available, a default value is used, which is part of the model.

2.2 Stateful vs. stateless units

In the course of computation a unit may act differently depending on its history. Such units are considered *stateful*. In COLA only delays and automata retain information of previous computations, whereas all other kinds of units are *stateless*. Note that an instance of a unit containing a stateful element becomes stateful as well. In code generation we must guarantee both, proper initialization of stateful elements, and communication of the state across a series of clock ticks.

3 Coding basic model elements

In the following we describe the transformation of COLA elements into C code. The mapping is exemplified presenting COLA diagrams and the according code snippets from our case study.

3.1 Units and signatures

For each unit found in the given system, the code generator creates a C function with an appropriate signature. As COLA units may carry more than one output port in their signature, the resulting code has to offer an equivalent concept. We decided to include a variable of appropriate type for each input and output port of the unit in the function's signature. The C types corresponding to the type names used in the model are therefor predefined in the code generator. All variables are passed as pointers i.e., following the call-by-reference paradigm. In case of input ports this saves memory when complex data types are used as the pointer requires only a fixed amount of memory independent of the dimension of the data type pointed to. Plus, for output ports, this is the only way to allow for multiple return values in C. In addition to these variables each signature of a stateful unit includes a struct named `unit_state`. This struct keeps the actual state of delays or automata. Details on the preservation of the actual state are given in Section 4.1. If neither the unit itself, nor any of its sub-units are stateful, the state struct is superfluous and thus omitted.

In Figure 2 the main network, i.e., the system diagram of the case study used throughout the paper is shown. As can be seen in the diagram, the system consists of twelve units. These are connected using channels, which forward data from the output ports to the designated input ports of the succeeding unit. Listing 1 shows the generated code for Figure 2. The shown parent unit, named `net_ACC`, carries no ports, hence the signature of the according C function is empty. In the following we will explain the construction of the function body for this network.

3.2 Networks

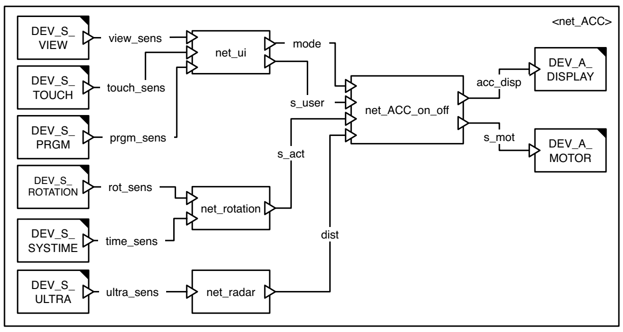

Figure 2: The `net_ACC` diagram

The body of a generated C function implements the related unit's behavior. For a network this means that the generated function for each sub-unit included in the network is called. Of course the sequence of calls has to preserve the order induced by semantics of the data flow. To do so, the set of sub-units provided for the network is searched for units not dependent on other units in the network. Each such unit can instantly be inserted in the resulting C code and removed from the set. This is true for all units which are connected to an input port of the parent unit, to an output port of a unit already coded, or to a unit which represents a constant value, a source or sink, or a delay. The described iteration over the set of sub-units is repeated until the set is empty. Then all sub-units are coded in a sequential order preserving the data flow semantics.

In Listing 1 the preservation of causality according to data flow semantics becomes apparent. The formal semantics for the evaluation order of networks is given in [3], Section 5.3. Looking at the example in Figure 2, six sensors indicated by the `DEV_S_` prefix in their names can be seen. The input data of these devices, sources in terms of COLA, are read from the middleware in lines 17 through 22. Further there are three sub-units `net_ui`, `net_rotation`, and `net_radar`, which can be evaluated independently. Thus they are called first in the resulting code, cf. lines 23 through 27 in Listing 1. The order of the three calls is arbitrary. Using their results, `net_acc_on_off` can be executed and then finally the actuators `DEV_A_DISPLAY` and `DEV_A_MOTOR` are written to. The resulting code is shown in lines 28 through 31 of Listing 1. The last two units are sinks, again indicating interaction with the hardware platform. Sources and sinks are detailed in Section 4.3.

In addition to the evaluation order, the listing shows how each channel connecting two sub-units is realized. A variable is used to pass data from one function call to the next. It is being written to by the ancestor unit and read from by the descendant one.

3.3 Automata

In COLA, automata provide the means of control flow. As described in [3], an automaton's behavior is implemented by the currently active state, which is a unit. For an implementation to determine the active state, first the current state must be known and then possible outgoing transitions must be evaluated. The former is stored in the `unit_state` struct, and transitions are implemented by a sequence of `switch ... case ...` and `if ... else ...` statements.

COLA automata describe general Moore-type finite automata [19], but the transformation to C code must yield deterministic behavior. In our current prototypical toolchain it is the responsibility of the modeler to employ only deterministic automata. Further efforts are put in an automated test for non-determinism while assuming data types with finite domains. Note that the problem is undecidable in case of unbounded data types (undecidability of equivalence).

In Figure 3 a COLA unit implemented by an automaton with two states is given. The states are named `atm_dist_35_check` and `net_emergency`. The actual state is changed either if the value of `dist` equals zero or is greater than zero and depending on the actual state. Listing 2 shows the code for the automaton given in Figure 3.

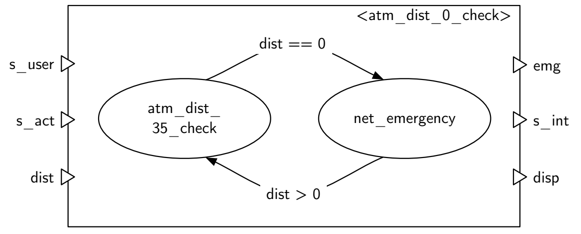

Figure 3: The `atm_dist_0_check` diagram

```
void net_acc()
{
  net_acc_state unit_state;
  int s_act_0;
  int dist_1;
  int mode_2;
  int s_user_3;
  int acc_disp_4;
  int s_mot_5;
  int rot_sens_6;
  int time_sens_7;
  int ultra_sens_8;
  int view_sens_9;
  int touch_sens_10;
  int prgm_sens_11;
  mw_restore_task_state(14, &unit_state);
  mw_receive(2, &rot_sens_6);
  mw_receive(1, &time_sens_7);
  mw_receive(3, &ultra_sens_8);
  mw_receive(5, &view_sens_9);
  mw_receive(4, &touch_sens_10);
  mw_receive(6, &prgm_sens_11);
  net_rotation200399(&(unit_state.state_rotation200399_num0), &rot_sens_6,
     &time_sens_7, &s_act_0);
  net_radar200400(&(unit_state.state_radar200400_num1), &ultra_sens_8, &dist_1);
  net_ui200398(&(unit_state.state_userinterface200398_num2), &view_sens_9,
     &touch_sens_10, &prgm_sens_11, &mode_2, &s_user_3);
  net_acc_on_off200401(&(unit_state.state_acc200401_num3), &s_user_3, &s_act_0,
     &dist_1, &mode_2, &acc_disp_4, &s_mot_5);
  mw_send(13, &s_mot_5);
  mw_send(12, &acc_disp_4);
  mw_save_task_state(14, &unit_state);
}
```

Listing 1: The `net_ACC` code

During code generation each state of an automaton is given a numeric id. In line 5 the automaton's state is used to decide on the transitions to check and the behavior to process subsequently. Here two states are coded, as can be seen in lines 7 and 19, represented by a case-switch based on the stored automaton state. In either case the guards for the outgoing transitions are evaluated. A guard is coded as a separate function returning a Boolean result. If one of the guards, called in lines 8 and 20, respectively, in our example evaluates to *true*, the transition is taken. Thus the automaton's state is changed and the behavior of the target state is executed as exemplified in lines 9 through 15 and 21 through 27, respectively. If in contrast the guards evaluate to *false*, the behavior of the actual state is processed. In our example this is shown in lines 16 and 28. As mentioned in Section 3.1, the state struct is omitted for stateless units. An example of this approach can be seen in the calls in line 16 and 24.

In Listing 2 the analogousness of the automaton's signature and the signature of its states is apparent. The variables passed to the automaton in line 1 are forwarded to the function calls in lines 12, 16, 24, and 28. The only variable differing is the `unit_state` as it is distinct for each unit.

3.4 Functional blocks and delays

Having only dealt with COLA elements allowing for hierarchical composition so far, we will now describe the coding of blocks. They form the base of the model representing elementary operations and marking the endpoint of the hierarchy. We distinguish functional blocks and timing blocks, i.e., delays. Strictly speaking, sources and sinks are blocks as well, but we defer their discussion to Section 4.3. The functional blocks allow for constants and arithmetic and Boolean operations, while timing blocks provide a means of retaining data over time.

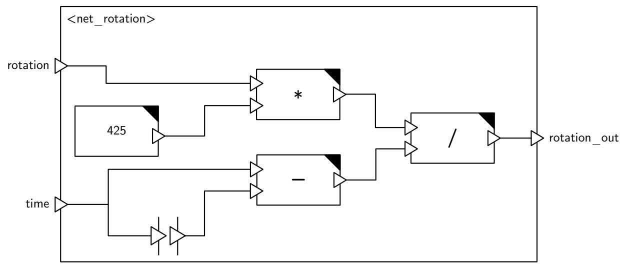

Figure 4: The `net_rotation` diagram

To avoid the function call overhead, all blocks are expanded inline instead of calling a function. An example

```
void dist_0_check200695(state_dist_0_check200695 *unit_state, int *s_user_in_0,
    int *s_act_in_1, int *dist_in_2, int *emg_out_0, int *s_int_out_1, int *disp_out_2)
{
  int guard_result;
  switch(unit_state->atm_state)
  {
    case 0:
      emergency_guard200747(s_user_in_0, s_act_in_1, dist_in_2, &guard_result);
      if(guard_result)
      {
        unit_state->atm_state = 1;
        atm_dist_35_check200787(&(unit_state->state_dist_gt_0_behavior200787_num1),
                  s_user_in_0, s_act_in_1, dist_in_2, emg_out_0, s_int_out_1, disp_out_2);
        break;
      }
      net_emergency200764(s_user_in_0, s_act_in_1, dist_in_2, emg_out_0,
                s_int_out_1, disp_out_2);
      break;
    case 1:
      dist_35_guard200730(s_user_in_0, s_act_in_1, dist_in_2, &guard_result);
      if(guard_result)
      {
        unit_state->atm_state = 0;
        net_emergency200764(s_user_in_0, s_act_in_1, dist_in_2, emg_out_0,
                  s_int_out_1, disp_out_2);
        break;
      }
      atm_dist_35_check200787(&(unit_state->state_dist_gt_0_behavior200787_num1),
                s_user_in_0, s_act_in_1, dist_in_2, emg_out_0, s_int_out_1, disp_out_2);
      break;
  }
}
```

Listing 2: Code for `atm_dist_0_check`

of coding blocks and delays is given in Listing 3. It shows the code generated for the diagram in Figure 4. As can be seen, all units in this diagram are marked with a black triangle in the upper right corner, indicating a functional block. A constant block with value 425 is given. It is multiplied with the value delivered by the port `rotation` and the result divided by the value calculated in the lower part of the diagram. The according code is given in line 4 of Listing 3. The delay included in Figure 4 is depicted by two parallel vertical lines, each of them carrying a port. This timing block functions as a one-step FIFO. It outputs the value given to it during the previous invocation of the network. Thus, in our example, the previous value of port `time` is delivered. Subsequently the data pending at its input port is stored for the next invocation. The two described working steps of the delay can be seen at the end of line 4 and in line 5 of the listing.

3.5 Clusters

To partition the model into distributable software units targeted at platforms built from a number of distinct nodes, COLA features the concept of clusters. Each cluster defines a software unit to be scheduled in the resulting system, i.e., a task. To allow for independent scheduling, a C code file has to be generated for each cluster. In Figure 5 a possible clustering of the ACC case study is given. The figure shows nine clusters, of which cluster `c7` contains the modeled functionality of the ACC, while the other clusters contain a sensor or actuator each. The clustering of hardware components denotes that these devices also rely upon the assignment of a communication address for use with the middleware. Further, the reading or writing of data from and to these devices has to be triggered similarly to the execution of application tasks.

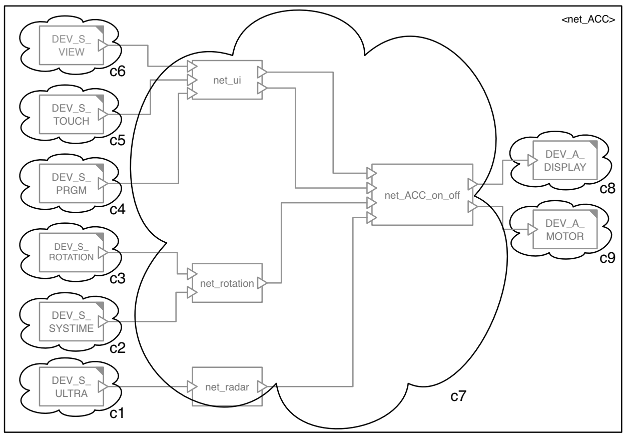

Figure 5: The clustered ACC model

```
void rotation200399(state_rotation200399 *unit_state , int *rotation_in_0 , int *time_in_1 ,
   int *rotation_out_2)
{
  *rotation_out_2 = ((*rotation_in_0 * 425) / (*time_in_1 - (unit_state->delay200513)));
  (unit_state->delay200513) = *time_in_1;
}
```

Listing 3: Code for `net_rotation`

4 Running the generated code

The presented code generator implements a part of a model driven system realization demanded for future embedded real-time systems [20]. Assuming the use of a standardized middleware enables for the use of our generator in combination with various platforms. Thus the effort for coupling the application to the underlying hardware is no more demanded to be done separately for every program, but is shifted to the implementation of the middleware. As this layer's interface remains unchanged for every application, the complexity of the application is decreased and the development resources saved can be used to develop a high-quality middleware.

When generating code for embedded hardware, the question of how to execute the code arises. First of all, of course, the code has to be compiled using a valid C compiler for the intended target platform. But there are other prerequisites for enabling the applicability of the code. One important point is data retention for stateful tasks. Furthermore the communication between modeled entities must be ensured, even when distributed to different computing nodes. Besides that, another point is the access of sensor and actuator values. In the following we will present our solutions to these problems using a middleware layer.

4.1 Preserving the actual state

In COLA we currently consider delays and automata as the only stateful units. A delay keeps its actual value from one invocation of the enclosing unit to the next. Analogously, an automaton maintains its actual state from one invocation to the next. In case of an automaton or network this is also true for all sub-units contained in any of their implementing units.

As we generate tasks for the target operating system, we need to find a means of initializing and storing data. We address this problem by defining a struct for every unit containing its state, which retains the values of all contained delays and each automaton's active state number. Further, instances of the respective structs of all sub-units of the unit are included. We have a struct for the top-level unit, i.e., the system, code is generated for. As stated before this struct contains nested structs for all stateful sub-units. During startup, the struct is initialized with default values specified in the model. Later on, this top-level struct is read from the middleware by the task at the beginning of every invocation, as shown in line 16 in Listing 1. The modified struct is written back to the middleware at the end of the task (line 32). The middleware maintains an appropriate amount of memory for every task.

4.2 Inter-cluster communication

Regarding today's embedded real-time systems, these are often distributed systems, consisting of dozens or more nodes interconnected using various bus systems. To enable the use of our approach in such large scale systems, communication links, i.e., channels specified in the COLA model, must be mapped to communication primitives between the nodes. For this purpose we employ our middleware. Its task is the mapping of logical addresses, which are in numeric format, to real hardware addresses of the underlying communication systems.

Figure 6 depicts the inter-cluster communication via our middleware. Task 1 allocated on node 1 sends a message with numerical address 42. All other tasks in the system can subsequently receive the datum using this address. The middleware distributes the datum locally as well as remote. The bus communication is realized using broadcasts, thus allowing an arbitrary number of nodes.

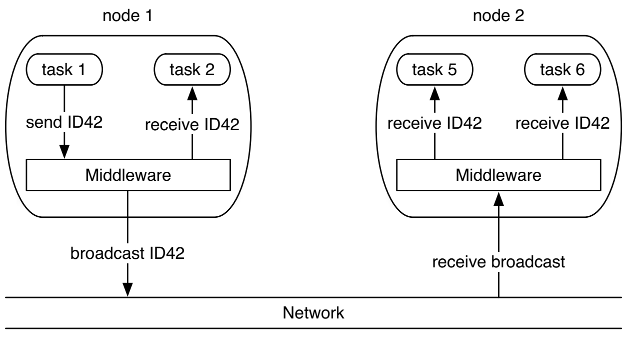

Figure 6: Inter-cluster communication via middleware

Besides retaining port connections, the synchronous semantics of the model have to be preserved. To guarantee the timely delivery of data, we chose a time division multiple access (TDMA) scheme for communication similar to the TTA, as presented by Kopetz [21]. Due to the avoidance of collisions in such a system, arrival times of data can be anticipated and the model's causality demands, as well as timing demands, are fulfilled.

Logical addresses for use by the middleware are generated using a dependency graph [22] covering all clusters of the modeled system. Each node of the graph representing a data buffer is given a numerical address. These addresses are later referred to in the middleware API calls, inserted by the code generator.

4.3 Interfacing sensors and actuators

Embedded systems in general require a vast amount of of software/hardware interaction. Thus, COLA as a domain-specific language, and consequently code generation, must cater for proper interfaces to sensors and actuators. The synchronous semantics of COLA require that all operations occur instantly, which must thus also be true for hardware interaction. Thus, if for example the model contains sources representing wheel sensors for all four wheels of a car, these sensors are read at the same instant, from the model's point of view, which of course is infeasible in any effective implementation.

To approximate the synchrony assumption in a best effort manner, hardware interaction performed in a batch style. That is, all sensors referenced in the model are read from before executing any application tasks. Consequently, all actuators are written to after execution of tasks has finished. This scheduling cycle is depicted in Figure 7.

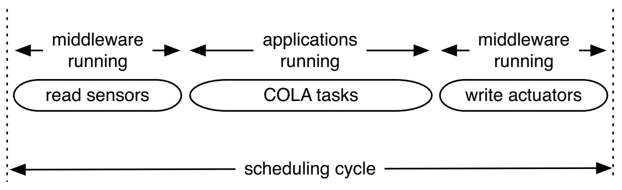

Figure 7: Hardware interaction within a scheduling cycle

Similar to inter-cluster communication, the generated code uses `mw_receive()` and `mw_send()` for any hardware interaction. As said before, the calls take the logical address, corresponding to the device in question in this case. The middleware decides on whether the address references a cluster or a piece of hardware according to its configuration. For details, please refer to our middleware description [15]. Examples of interfacing with the devices shown in Figure 2 are found in lines 17 through 22 and 30 plus 31, respectively, of Listing 1.

In contrast to regular COLA blocks, sources and sinks must always have exactly one port. An output port for sensors, delivering the sensor's values and an input port for actuators accepting the calculated value of the control loop algorithm. To avoid the possibility of race conditions, each sensor or actuator may only be inserted once into a COLA model. Since every device features just one port, this demands the developer to explicitly model the control of every concurrent hardware interaction.

5 Case study

To prove the practical viability of our approach, we did a case study using LEGO® Mindstorms™ controllers as hardware platform, equipped with the BrickOS[1] operating system. The demonstrator should realize the functionality of an adaptive cruise control (ACC) [23]. This is a control device for cars providing the functionality of keeping the car's speed at a value set by the driver, while maintaining a minimum distance to the car driving ahead. A picture of the demonstrator can be seen in Figure 8.

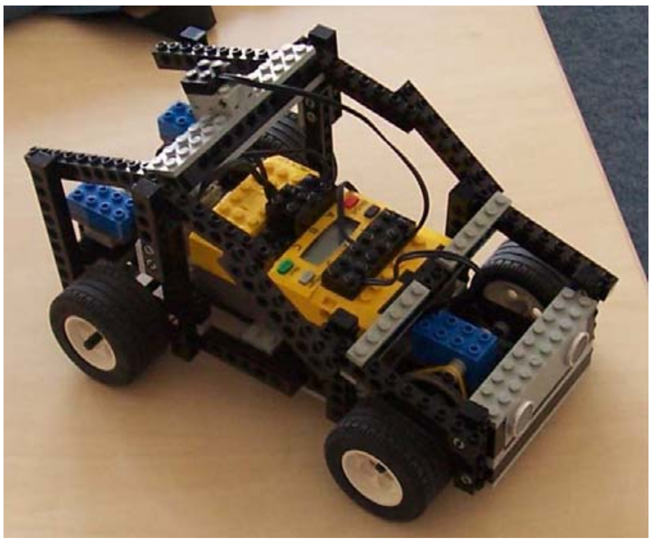

Figure 8: The Mindstorms demonstrator

The presented example is an imitation of the concerns and requirements of automotive design, and does not represent a real set of control algorithms for an actual product or prototype.

5.1 Functionality of the demonstrator

The intended functionality of the demonstrator 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 indicates 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 target speed 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

[1]http://brickos.sourceforge.net

detected within 35 centimeters ahead of the car. If the distance drops below this threshold, the actual speed is continuously decreased by 5 percent. The minimum distance allowed constitutes 15 centimeters. If the actual distance is lower, the car performs an emergency stop. After either reducing speed or coming to a halt, the ACC should speed up the car smoothly again, if the obstacle is out of the critical region.

5.2 The ACC COLA model

The implementation of this functionality is severely influenced by the hardware available. The used controller offers only two buttons available for control actions to the programmer. Further, three sensor and three actuator ports are present. For the demonstrator we use the two controller buttons for setting the desired user speed. A touch sensor is employed to switch the ACC on and off. The remaining two sensor ports are used to connect a rotation and an ultrasonic sensor. Two motors are connected to the actuator ports, while the third port was utilized to connect some brake lights.

The top-level network of the ACC COLA model is shown in Figure 2. This network has no ports as all sensors and actuators used for the ACC's operation are included in the network: the buttons `DEV_S_VIEW`, `DEV_S_PRGM`, the rotation sensor `DEV_S_ROTATION`, the system clock `DEV_S_TIME`, the touch sensor `DEV_S_TOUCH`, and the distance sensor `DEV_S_RADAR`. The actuators of the network are the controller's display `DEV_A_DISPLAY` and the motor control `DEV_A_MOTOR`. As described in Section 4.3 these blocks are coded as `mw_receive()` and `mw_send()` statements, e.g., the call to `DEV_A_MOTOR` would be transformed into `mw_send(13, &s_mot_5)` where `s_mot_5` is the calculated output for the motor speed. The address `13` is the numeric middleware address assigned for this actuator. The middleware forwards the receive and send calls to hardware driver calls provided by BrickOS, which in turn provide sensor values or modify some actuator state.

As this case study was intended to be executed on a single node, i.e., a mindstorms controller, we chose the clustering presented in Figure 5. All parts of the network to be implemented in software were put into a single cluster. As no distribution is possible using a single brick, the generation of a single task results in the fastest system possible for the given model. This avoids the—in this case—unnecessary middleware communication overhead of a distributed solution. Consequently the comparison of runtimes against a hand-written version of the ACC presented in Section 5.4 was facilitated, as the manual implementation was also realized as a single task. Thus the resulting runtimes are less influenced by OS and communication overhead, and the results rather depend on efficiency of the compared application code.

For a distributed system, a more fine-grained clustering might be favorable.

5.3 Code efficiency

Embedded systems typically provide limited amounts of memory and processing speed. Thus the code executed must be small and predictable in its memory usage. This applies to the amount of coded instructions as well as variables. To achieve this goal, the code generator uses call-by-reference wherever possible. As pointer variables have a fixed size, the memory footprint of each function call becomes independent of the data structures passed around. In case of complex data types, the usage of pointers may even reduce the amount of memory needed and lower the execution time.

Another way to save memory is the employment of reuse, i.e., the size of the generated code is kept small by making use of already coded components. Thus, if a unit appears multiple times in a given model, it is only once transformed into a C function. This function is then called every time the unit is referenced. Of course there is a separate `unit_state` struct for each unit instance, in case of a stateful unit.

5.4 Benchmarking the ACC code

When working with generated code, efficiency aspects surely play an important role. Using any valid tricks the programmer is aware of, hand-written code may be then considerably smaller and faster. Consequently, the benefit of automatic code generation rather is its deterministic result. If (to a certain extent) behavioral correctness of the model has been established using techniques such a (correct) code generator will produce equivalently correct code. Coding errors like unintentional casts, wrong pointer arithmetic and the like are avoided. Finally a major fraction of the potential performance drawbacks are negligible due to optimizations performed by the compiler.

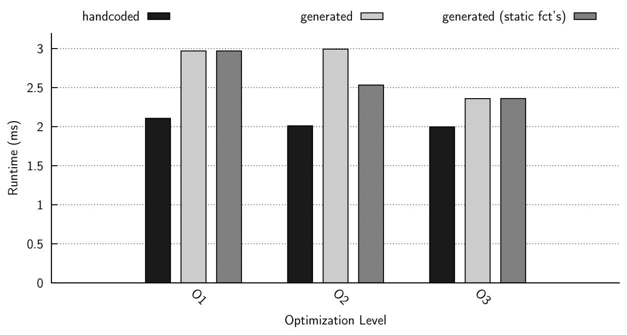

Figure 9: Benchmarking runtime results

To get an idea of the performance of the generated ACC code, we compiled it successively using the optimization levels O1, O2 and O3 of the GNU C compiler and checked the runtime results against the ones of a hand-coded version of the ACC. To give an impression of the quality of our manually coded version, we give the lines of code (LOC) as metrics. The hand-coded version fulfills the same functionality using 75 LOC, while the code generator produced 249 LOC for the modeled ACC. The resulting binaries were compared regarding their execution times. To minimize errors in the measurement of the running period induced by interrupts, context switches, etc., the ACC algorithm was called consecutively 100 times. We ran this benchmark 20 times for each optimization level. The averaged resulting times are given in Figure 9. The third bar in the diagram, named *generated (static fct's)*, indicates the values for a version of the generated code with all functions being declared static. As this allows the compiler to disregard the use of the functions from outside the binary, the functions calls can be replaced by their implementation. Thus there is no jump to fulfill and thus the execution time decreases. As can be seen in the diagram, this effect makes most impact in case of the O2 optimization level.

The measurements show that the execution time of our generated code isn't too far from the version implemented by hand. Especially when using the higher optimization levels of the compiler. There the generated code benefits even more than the alternative. At maximum optimization level, i.e., O3, the generated code even nearly reaches the execution times of the hand written version.

6 Conclusions

We presented a translation scheme of models given in the Component Language (COLA) to C code, and a prototypical implementation thereof. The formal semantics of COLA and its small number of syntactic elements allow for a transformation that fully retains the behavior of the model. The performance evaluation shows that, using an optimizing compiler, the execution time of the generated code only has an overhead of 20% when compared to a hand-crafted fully optimized version. We assume that such a low overhead immediately pays off as the generated code is guaranteed to realize the behavior of the model, which has undergone functional validation.

Future enhancements of the code generator will focus on generating more efficient code, i.e., using less memory and CPU time, and integrating more syntactic elements as the vocabulary of COLA is still growing. Recent additions to COLA include the introduction of multidimensional data types, called records, and the use of distinguished model elements representing sensor and actuator elements, namely sources and sinks. Additionally, the COLA standard library is intended to be filled with regularly used units, e.g., integrator, counter, saturation, PID-regulator, etc. The code for these units can be generated and saved for future code generation, thus avoiding to generate it more than once and speeding up the transformation.

Another ongoing project is the use of a model checker for verification of COLA models. To facilitate this, the model is transformed into Promela code and checked using the SPIN [24] model checker. The generator for Promela code is based on the implementation of the C code generator presented here.

References

[1] S. Schulz, J. W. Rozenblit, and K. Buchenrieder, "Multi-level testing for design verification of embedded systems," *IEEE Design & Test of Computers*, vol. 19, no. 2, pp. 60–69, 2002.

[2] M. Broy, "Automotive software and systems engineering (panel)," in *MEMOCODE*, pp. 143–149, 2005.

[3] S. Kugele, M. Tautschnig, A. Bauer, C. Schallhart, S. Merenda, W. Haberl, C. Kühnel, F. Müller, Z. Wang, D. Wild, S. Rittmann, and M. Wechs, "COLA – The component language," Tech. Rep. TUM-I0714, Institut für Informatik, Technische Universität München, Sept. 2007.

[4] T. A. Henzinger and J. Sifakis, "The discipline of embedded systems design," *IEEE Computer*, vol. 40, no. 10, pp. 32–40, 2007.

[5] S. Kugele, W. Haberl, M. Tautschnig, and M. Wechs, "Optimizing automatic deployment using non-functional requirement annotations," in *Proceedings of International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, Springer, 2008. To appear.

[6] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data-flow programming language LUSTRE," *Proceedings of the IEEE*, vol. 79, pp. 1305–1320, September 1991.

[7] P. H. Feiler, B. Lewis, and S. Vestal, "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)*, May 2003.

[8] A. Pretschner, M. Broy, I. H. Krüger, and T. Stauner, "Software engineering for automotive systems: A roadmap," in *Future of Software Engineering (FOSE '07)*, pp. 55–71, 2007.

[9] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. L. Sangiovanni-Vincentelli, "Metropolis: An integrated electronic system design environment.," *IEEE Computer*, vol. 36, no. 4, pp. 45–52, 2003.

[10] E. Bondarev, M. Chaudron, and P. H. N. de With, "CARAT: a toolkit for design and performance analysis of component-based embedded systems," in *Proceedings of the conference on Design, automation and test in Europe (DATE 2007)*, pp. 1024–1029, Mar. 2007.

[11] P.-A. Hsiung, S.-W. Lin, C.-H. Tseng, T.-Y. Lee, J.-M. Fu, and W.-B. See, "VERTAF: An Application Framework for the Design and Verification of Embedded Real-Time Software," *IEEE Transactions on Software Engineering*, vol. 30, no. 10, pp. 656–674, 2004.

[12] J. W. Reuter, "Analysis and comparison of 3 code generation tools," in *Proceedings of the SAE 2004 World Congress*, Society of Automotive Engineers, Mar. 2004.

[13] D. Wybo and D. Putti, "A qualitative analysis of automatic code generation tools for automotive powertrain applications," in *Proceedings of the 1999 IEEE International Symposium on Computer Aided Control System Design*, pp. 225–230, 1999.

[14] M. W. Whalen and M. P. E. Heimdahl, "An approach to automatic code generation for safety-critical systems," in *ASE*, pp. 315–318, 1999.

[15] W. Haberl, U. Baumgarten, and J. Birke, "A Middleware for Model-Based Embedded Systems," in *Proceedings of the 2008 International Conference on Embedded Systems and Applications, ESA 2008*, (Las Vegas, Nevada, USA), July 2008.

[16] C. Kühnel, A. Bauer, and M. Tautschnig, "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.

[17] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide.* Addison-Wesley, 1998.

[18] A. Bauer, M. Broy, J. Romberg, B. Schätz, P. Braun, U. Freund, N. Mata, R. Sandner, and D. Ziegenbein, "AutoMoDe — Notations, Methods, and Tools for Model-Based Development of Automotive Software," in *Proceedings of the SAE 2005 World Congress*, Society of Automotive Engineers, Apr. 2005.

[19] E. F. Moore, "Gedanken-experiments on sequential machines," in *Automata Studies* (C. E. Shannon and J. MacCarthy, eds.), pp. 129–153, Princeton University Press, 1956.

[20] A. Sangiovanni-Vincentelli and M. D. Natale, "Embedded system design for embedded automotive applications," *Computer*, vol. 40, no. 10, pp. 42–51, 2007.

[21] H. Kopetz, "The time-triggered architecture," in *ISORC '98: Proceedings of the The 1st IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, (Washington, DC, USA), p. 22, IEEE Computer Society, 1998.

[22] S. Kugele and W. Haberl, "Mapping Data-Flow Dependencies onto Distributed Embedded Systems," in *Proceedings of the 2008 International Conference on Software Engineering Research & Practice, SERP 2008*, (Las Vegas, Nevada, USA), July 2008.

[23] Robert Bosch GmbH, *Kraftfahrtechnisches Taschenbuch.* Vieweg, 26$^{\text{th}}$ ed., Jan. 2007.

[24] G. J. Holzmann, *The SPIN Model Checker : Primer and Reference Manual.* Addison-Wesley Professional, Sept. 2003.