Reliable Operating Modes for Distributed Embedded Systems

Wolfgang Haberl* Stefan Kugele*† Uwe Baumgarten*

*Institut für Informatik
Technische Universität München
Boltzmannstr. 3
85748 Garching b. München, Germany

†Fachbereich Informatik
Technische Universität Darmstadt
Hochschulstr. 10
64289 Darmstadt, Germany

{haberl, kugele, baumgaru}@in.tum.de

Abstract

Hard real-time embedded distributed systems pose huge demands in their implementation which must contain as few faults as possible. Over the past years, model-driven development and automatic code generation have proven to effectively reduce design faults in those systems. Still, models are mainly used for parts of the systems' functionality and most solutions do not address the generation of a whole system.

In this paper we will showcase an approach for code generation for entire systems. A crucial step is the semantically correct realization of operating modes defined in the model. If they are not changed synchronously, a distributed system will show unpredictable behavior. We will demonstrate how a reliable transition between operating modes, even for a distributed system, can be achieved. Our approach is exemplified using a case study we carried out recently.

1 Introduction

Embedded systems have become very popular over the past years. They exist in many forms, ranging from small sized consumer products like digital cameras or entertainment systems, to large-scale distributed systems like cars or airplanes. Besides their distributed nature, the latter ones pose higher demands in their correct implementation as they are mission-critical systems. In case of a failure, human lives are endangered. Our work, part of which is presented here, is focused on these hard real-time embedded systems.

Meanwhile model-driven development (MDD) is the state-of-the-art approach to come to grips with the complexity of embedded systems design. Using abstraction mechanisms, developers gain an insight only into those items necessary for their work, by hiding unnecessarily detailed information. Thus, the implementation is easier to survey, even more if the modeling language features a graphical syntax. If the employed language also has a mathematically defined semantics, the correctness and quality of the models described therewith, can be checked using verification techniques like model-checking or automated reasoning. To this end we developed COLA, the Component Language [16], for modeling distributed, embedded real-time systems. Around this language, a multitude of tools were implemented, proving the possible benefits when employing the described concepts.

A crucial step during MDD is the transformation of models into code. Often this is done coding the modeled system by hand, leading most certain to the introduction of human failures and thus compromising the correctness of the model. This can be avoided by using automated code generation. So, one of the tools implemented for our COLA-based development process is an automated code generator, as described in [9]. In this paper we introduce an extension of the presented work, dealing with the automatic generation of code for operating modes in a distributed system. As we will show, our approach ensures synchronous mode changes in a distributed system, thus allowing the definition of system wide operating modes without explicitly taking distribution into account, while modeling the system.

Compared to other tools commercially available, like MATLAB®/Simulink®, ASCET®, or SCADE Suite®, our approach possesses some key advantages. Compared to MATLAB/Simulink and ASCET the semantics of our language is formally defined, enabling as already mentioned automated checking and processing of systems modeled therewith. While SCADE Suite features a similar foundation, it is, as are MATLAB/Simulink and ASCET, targeted at the definition of partial functionality running on a single computing node, rather than for an entire distributed system. In contrast, COLA is suitable, and intended, for the specification of a distributed system as a whole. This is also true for all the tools implemented for a COLA based development process.

1.1 Operating Modes

When dealing with distributed systems, generating executable code for each computing node of the system is not sufficient. There is also the need for valid distribution and scheduling plans, which have to be calculated with respect to resource usage of the distributed software components.

The found scheduling plans especially have to consider *operating modes*. These modes group a set of functions, which shall be executed together to achieve a certain system behavior. Starting, landing, taxiing in case of an airplane or locked, running, ignition on/off in case of a car are examples for operating modes. Each operating mode triggers a set of tasks, which are distributed over the computing nodes forming the system. For modern cars, this may be up to 80 nodes, as presented by Broy in [4].

Using operating modes in embedded systems design offers two key advantages. First, it breaks up the complexity of a system into smaller pieces, making its specification easier. Second, an operating mode groups only those functions necessary in the actual situation. All other tasks are inactive in that mode. This leads to a more efficient resource usage, as the hardware platform can be chosen according to the most resource demanding operating mode. That platform may be still significantly less expensive than one which is capable of executing all functions of all operating modes in parallel.

A major difficulty of operating mode changes is the synchronous transition from one operating mode to the next in a distributed system. Just imagine the engine control not shutting down the motor, when the driver turns the key, because not all involved system nodes change their state to ignition off.

In the following, we will present our solution to this problem, using model-based code generation for the entire system. Thus not only single application tasks are transformed into C code, but also a main scheduling loop for each node of the hardware platform is created, implementing synchronous mode changes. Due to the automatic code generation concepts presented here, mode switches follow strictly the semantics of the defined COLA model. Hence, the resulting system is guaranteed to perform operating mode changes reliably.

1.2 Related Work

For SCADE, which is based on Lustre [10], a deployment concept for distributed embedded systems has been presented in [5]. Compared to COLA, this approach lacks a key concept: it does not offer the automatic deployment of operating modes, supporting a dynamic change of scheduling plans at runtime, although Lustre supports mode automata. Furthermore, unlike the COLA modeling process, no optimized automatic allocation is performed. Considering model-based engineering of embedded control software, Schätz proposes in [18] a clear separation of control- and data-flow models to avoid unnecessary complexity. Control-flow is used to specify modes of operation, whereas data-flow is used to define the mode's control task. In a similar way, COLA models are structured with respect to operating modes using mode automata. As an essential improvement—especially in the context of safety-critical systems—this paper describes a novel technique to generate executable code for complete systems, where operating modes are distributed over several computing nodes.

Another language with similar focus regarding the target systems is Giotto, as presented by Henzinger in [11]. The models specified therewith are also executed in a cyclic manner, similar to COLA models. An extension of Giotto towards distributed platforms has been described in [12], namely Distributed Giotto. Unlike COLA, Giotto defines the causal order of tasks and their resource requirements, but does not deal with specifying their implementation. Rather tasks are implemented by hand and the Giotto compiler guarantees the timely execution in a distributed system, given worst-case execution times and call frequencies for all tasks are known. COLA contrariwise defines the tasks implementation, which allows for verifying their implementation and calculating reliable execution times based on the designed model, as presented in [19].

Lately UML has become popular for modeling real-time systems. One approach for generating C code from UML models has been presented by Khan et al. in [13]. But compared to COLA, the current diagram types defined in UML do not provide enough information for generating the entire application code. Rather a framework consisting of some files with definitions of variables and functions can be derived. The rest of the implementation has to be carried out manually. Avoiding those error prone manual changes to the resulting code was one of the main reasons for using a data-flow language like COLA in our work. The information captured in a COLA model is sufficient for generating all of the code needed for a system.

1.3 Organization

The remainder of this paper is structured as follows: in Section 2 we give an introduction to the modeling language COLA which forms the base of our work. Following in Section 3 the concepts for calculating allocation and scheduling plans for a system modeled in COLA are presented. On the basis of these results the generation of code which preserves the model's semantics is shown in Section 4. Throughout the paper we use an autonomous parking system for cars as a case study. A brief description of the hardware platform used for the case study as well as the resulting system's functionality is given in Section 5. Finally we sum up our results in Section 6.

2 Overview of COLA

In the following we will give a brief introduction into the modeling constructs of COLA. COLA is a synchronous data-flow language, i. e., models are executed in a cyclic manner, following the time semantics described in Section 2.1. Details of COLA as well as its semantic foundation can be found in [16]. In addition to the software modeling constructs contained in COLA, the definition of the hardware plattform is facilitated. Specific information about the hardware modeling portion of COLA are explained in [15].

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, boolean, etc.) 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. Based on the formal semantics of COLA, type compatibility can be checked, as described in [17].

In addition to the hierarchy of networks, COLA provides a decomposition into *automata*, i. e., finite state machines, similar to Statecharts [3]. 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* [1], 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, including the interface to 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* embody connectors to the underlying hardware. Sources represent sensors and sinks correspond to actuators of the used hardware platform. via *channels*, dataflow is realized. A unit can be *automaton*. Delays are used to store values for a single execution cycle.

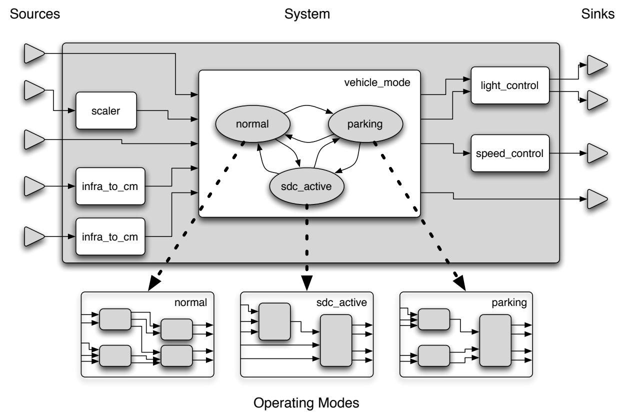

Figure 1. COLA model of the case study system

In Figure 1 a part of the COLA model of the case study used throughout the paper is shown. As depicted, the model features several sources and sinks to connect to the underlying hardware. The system is implemented at top-level using a network, which contains several sub-units. We are especially interested in the vehicle_mode automaton, whose three respective states, normal, parking, and sdc_active, form operating modes in the designed system, as we will detail later on. In the shown example, each operating mode is realized by a network, while an automaton also were a valid implementation.

The implementations of the units scaler, infra_to_cm, light_control, and speed_control are omitted for brevity in Figure 1.

2.1 Time Semantics

COLA is a synchronous data-flow language, i. e., it is assumed that computation and communication are infinitely fast. The term *synchronous* resembles to the hypothesis of perfect synchrony described by Berry and Benveniste [2] and which is assumed for the temporal semantics of COLA. According to this assumption the modeled 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.

Following the synchrony paradigm, the entire system is evaluated in logically zero time. Only data dependencies implied by the connecting channels, avoiding a unit to be evaluated before its inputs are available. At each tick 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 at their output port. At the first tick, when no prior input is available, a default value specified in the model is emitted.

Code generation maps the modeling construct of ticks to a time-triggered schedule. During each cycle of the schedule all software components of the modeled system are called. The length of the cycle is calculated using the deadlines specified in the model. If a mode change is detected, the actual schedule has to be adapted accordingly. Regarding the model time assumption this happens within one tick. Thus, in the real system the detection of a necessary mode change and the schedule modification have to be achieved in the same cycle of the time-triggered schedule. The code generator for the system dispatcher guarantees this requirement, as we will show in Section 4.3.

2.2 Clustering

Modeling system functionality with COLA results in a set of interconnected units, not taking any partitioning decisions into account. Defining a partitioning is done in a succeeding step. For this purpose, any unit of the model can be marked as defining a *cluster*. In this vein, the complete system has to be clustered, i. e., every unit has either to be marked to be a cluster or it is itself a sub-unit of an already clustered unit. Clusters are the model representation of distributable software components, i. e., tasks in the executable system.

Two types of clusters are distinguished, namely *mode clusters* and *working clusters*. As indicated by the term mode cluster, these clusters implement the transitions between operating modes. To this end, a mode cluster is built up of one or more automata as exemplified by the vehicle_mode automaton in Figure 1. Inside a mode cluster, further clusters may be defined. Usually, each state of a mode cluster's automaton is defined to be an individual (sub-)cluster, denoted as an *operating mode*. These clusters may be mode clusters themselves, which would mean they define sub-modes of the actual mode.

Alternatively, working clusters can be defined for a mode automaton's states. In contrast to mode clusters, a working cluster must not contain further clusters. Instead working clusters define the behavior of an operating mode. Code generation for working clusters has been described in [9]. In the following we will detail on the generation of code for mode clusters and their semantically correct execution. For this purpose, a synchronized timing for all computing nodes of the system is indispensable.

Considering our example in Figure 1, the vehicle_mode automaton is defined to be a mode cluster, while all other units depicted are coded as working clusters. Accordingly, the three states, normal, parking, and sdc_active are exclusively active, as they are distinct operating modes of our case study.

3 System Distribution

The overall functionality of the modeled system design is distributed onto the components of the hardware platform due to reasons of different nature: first of all, clusters may be allocated, i. e., placed on different computing nodes due to redundancy reasons.

Second, third-party suppliers providing both software and hardware may demand that clusters have to be placed onto their controllers because of being a one-stop shop.

Third, in some cases it is advantageous to place a cluster directly onto the controller, which is itself connected to sensors and actuators, providing and consuming data processed by the cluster. Thereby the execution time of that cluster may be minimized. Similarly other reasons to distribute a complete system model can be imagined.

3.1 Allocation

The problem of placing COLA clusters onto available computing nodes can be compared best with the VARIABLE SIZED BINPACKING problem, i. e., placing a finite set of weighted items onto a set of bins, each having a certain capacity. A similar problem, but in the context of system distribution considerably extended, is to determine an optimal placement of clusters onto the available computing nodes, taking non-functional requirements into account. Although, this problem is known to be **NP**-complete [7], i. e., there is probably no efficient (polynomial) procedure to solve the problem, an approach based on *Integer Linear Programming (ILP)* turned out to be feasible at least for systems of manageable size [15]. Since allocation, as the overall system generation process, is done *offline*, i. e., at design-time, enough computing resources are available. Therefore, we aim at getting the optimal rather than an approximated solution, if this is possible. The ILP approach provides the flexibility to quickly adopt the technique to the respective needs. In this regard, non-functional requirements like redundancy, costs, suppliers, and further aspects like power states, and processor architectures could be considered.

3.2 Dependency Analysis

COLA, as a data-flow language, provides the modeling concepts of networks representing data-flow and automata illustrating control-flow. Hereby a causal order between

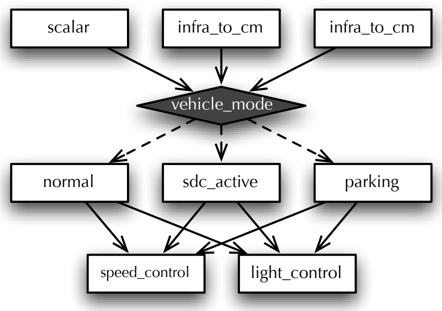

Figure 2. Cluster Dependency Graph for the running example

units is induced which is reflected in dependencies between clusters of the partitioned model. The consideration of these dependencies is crucial to maintain the COLA semantics. We introduced the concept of a *Cluster Dependency Graph (CDG)* in [14], visualizing the cluster dependencies.

3.3 Scheduling

Based on this dependency analysis, schedules for the complete system can be computed. Here, two aspects have to be emphasized: first, the CDG defines a relation $R \subseteq C \times C$ on the set of clusters C, i. e., $(c_i, c_j) \in R$ if and only if c_j depends on c_i. In this modality, a natural execution order is defined. Second, if a pair of cluster nodes (c_i, c_j) is not in this relation, they can be executed in an arbitrary order, including parallel execution. Figure 2 depicts a simplified CDG for the autonomic parking assistant used as a case study here.

In this example, $(\textsf{scalar}, \textsf{infra\_to\_cm})$ is not in the dependency relation R and therefore the respective clusters can be executed in parallel, if they were allocated onto different computing nodes. In contrast the execution of vehicle_mode has to be delayed until the results of the preceding working clusters scalar, infra_to_cm, and infra_to_cm are available since the following holds: $(\textsf{scalar}, \textsf{vehicle\_mode})$, $(\textsf{inftra\_to\_cm}, \textsf{vehicle\_mode})$, and $(\textsf{inftra\_to\_cm}, \textsf{vehicle\_mode})$ are contained in the relation R as they have dependencies.

By traversing the graph top-bottom, the following sets of schedulable clusters induced by the operating modes are obtained (mode clusters are underlined):

$$\begin{aligned} S_1 = \{&\textsf{scalar}, \textsf{infra\_to\_cm}, \textsf{infra\_to\_cm}, \underline{\textsf{vehicle\_mode}},\\ &\textsf{normal}, \textsf{speed\_control}, \textsf{light\_control}\}\\ S_2 = \{&\textsf{scalar}, \textsf{infra\_to\_cm}, \textsf{infra\_to\_cm}, \underline{\textsf{vehicle\_mode}},\\ &\textsf{sdc\_active}, \textsf{speed\_control}, \textsf{light\_control}\}\\ S_3 = \{&\textsf{scalar}, \textsf{infra\_to\_cm}, \textsf{infra\_to\_cm}, \underline{\textsf{vehicle\_mode}},\\ &\textsf{parking}, \textsf{speed\_control}, \textsf{light\_control}\} \end{aligned}$$

In all three sets, the working clusters before the (underlined) mode cluster are identical. Depending on the mode decision either normal, sdc_active, or parking is executed subsequent to the mode cluster. For this purpose, the schedule plans are changed at runtime according to the calculated mode. As indicated by the CDG, no cluster will be executed, before the result of the mode cluster is available. This assures that, if a mode change is necessary, all computing nodes will alter their scheduling plan simultaneously. On the strength of the presented *offline scheduling* approach, it is guaranteed, that if a feasible scheduling for all possible cluster sets exists, the generated system behaves exactly as defined by the found schedule. Notice, that the scheduling result not only contains information for a single computing node, but for the *complete* system. Scheduling plans for the single nodes are derived thereof.

In consideration of the fact that in principle cascades of mode nodes are possible, thus leading to an exponential (in the height of the CDG) number of schedulable cluster sets, the calculation of each schedule plan should be implemented efficiently. Therefore, we implemented a binary search algorithm based on a decision procedure—in this case the SMT-solver YICES [6]—to find for each set of clusters the optimal starting time. We define an optimal plan as the one, which invokes all clusters as early as possible during the schedule cycle (of length CT), leading to the shortest finishing time of all tasks in the distributed system.

For each set of clusters to schedule, S_1, S_2, and S_3 in the example, the procedure shown in Algorithm 1 is called. The idea of the outlined algorithm is to schedule all clusters as early as possible. Therefore, a binary search between the lower bound lb and upper bound ub, respectively, looking for the minimal satisfying value for mid is performed. Hence, the bounds are initially chosen according to equations (1) and (2).

$$lb = \sum_{p \in P} \sum_{i=1}^{|C(p)|} \left(\sum_{j=1}^{i} d_p(c_j) \right) \tag{1}$$

$$ub = \sum_{p \in P} \sum_{i=1}^{|C(p)|} \left(CT - \sum_{c \in C(p)} d_p(c) + \sum_{j=0}^{i-1} d_p(c_{|C(p)|-j}) \right) \tag{2}$$

Algorithm 1 Scheduler(int lb, int ub, Set $clusters$)

```
1: $last \leftarrow \emptyset$
2: while ($lb <= ub$) do
3:     $mid = lb + \left(\frac{ub-lb}{2}\right)$
4:     $result =$ isFeasible($clusters$, $mid$)
5:     if ($result = \emptyset$) then
6:         $lb \leftarrow mid + 1$
7:     else if ($result \neq \emptyset$) then
8:         $last \leftarrow result$
9:         $ub \leftarrow mid - 1$
10:    end if
11: end while
12: return $last$
```

On the one hand, equation (1) determines the lower bound. For each processing node $p \in P$ out of the set of all available nodes P, the task completion times are summed up for all clusters c_j allocated onto p, $1 \leq j \leq |C(p)|$, where $C(p)$ provides the set of those clusters allocated onto node p. This information is available since scheduling is performed after the optimal cluster placement has been computed. $d_p(c)$ determines the duration (worst case execution time) of cluster c on processing node p. If all clusters c_j, $1 \leq j \leq |C(p)|$, were sorted in *ascending* order with respect to their duration, i. e., $d_p(c_1) \leq d_p(c_2) \leq \ldots \leq d_p(c_{|C(p)|})$, and aligned at the beginning of the scheduling cycle, lb provides the minimal value of the sum of all task completion times on processor p.

On the other hand, equation (2) determines the upper bound. It is achieved by aligning all clusters c_j, $1 \leq j \leq |C(p)|$ in an *descending* order with respect to their execution times, i. e., $d_p(c_{|C(p)|}) \geq \ldots \geq d_p(c_2) \geq d_p(c_1)$ on the respective processing node p at the end of the scheduling cycle. Similarly, ub is calculated as the sum of all task completion times. Figure 3 illustrates the idea for four clusters $c_1, \ldots, c_4$ with worst case execution times $d_p(c_1) = 10, d_p(c_2) = 20, d_p(c_3) = 30$, and $d_p(c_4) = 40$, respectively for a schedule cycle of length $CT = 150$. Using formulae (1) and (2) we obtain:

$$lb = 10 + 30 + 60 + 100 \qquad = 200$$
$$ub = 90 + 120 + 140 + 150 \qquad = 500$$

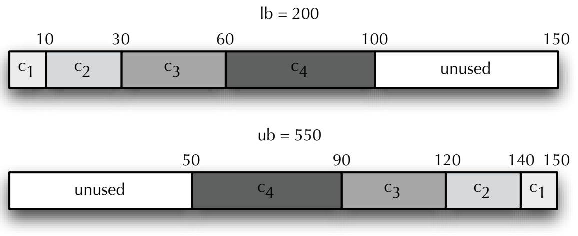

Figure 3. Initial determination of the bounds

Since the initial bounds do not take any task dependencies into account, they really define strict bounds and describe both best and worst case scenarios.

At each step of Algorithm 1 the method isFeasible($clusters$, mid) is called to check whether there is a feasible solution or not (cf. line 4), which then considers dependencies. In this vein, the earliest possible placement taking allocation and data-flow dependencies into account is accomplished. The function isFeasible() generates an input file for the YICES SMT-solver, which in turn checks whether there is an feasible schedule plan for the value mid and the given clusters. An annotated excerpt from the generated file is given in Figure 4. First, the used variables are declared and initialized, then the basic bounds for the cluster invocation and latest completion times are set. Next, an assertion is given, expressing that the sum of all task completion times has to be less or equal than the value mid. Afterwards, two cases are distinguished: first, a pair of independent clusters allocated onto the same ECU is considered. They can be executed in an arbitrary order. Second, two dependent clusters are considered. Their placement is not of importance.

In Figure 5, the allocation of clusters onto processing nodes (ECU 1, ECU 2, and ECU 3) and their respective starting times are depicted exemplarily for the mode parking. At the beginning of each scheduling cycle sensors are read and at the end actuators are written, respectively. In between, there is time for evaluating those clusters realizing the actual functionality. As a prerequisite for allocation and scheduling, the resource requirements for each cluster on a respective processing unit have to be known. Therefore, a technique described by Wang et al. [19] is used.

If a mode other than parking is chosen by the mode cluster, either the subsequent clusters of the schedule depicted in Figure 5 could have different starting times, or even a different set of clusters might be activated.

Code generation for working and mode clusters described in the following Sections 4.1 and 4.2 is completely decoupled, i. e., neither an allocation information nor the schedule plans are necessary. By that an arbitrary distribution of clusters onto computing node is facilitated. During generation of the dispatcher code for each node, finally, the results of allocation and scheduling are used to trigger the clusters at the starting time, defined by the schedule.

4 Code Generation

To allow for an automatic generation of distributed code, several different types of generators are needed. The code

```
;; Strating and ending times are integer as well as
;; the duration of a task; in this case the duration of
;; task 1 is set to 10 ms
(define start_task_1::int)
(define end_task_1::int)
(define duration_task_1::int 10)
;; similar for all others

;; Tasks can only be started after the sensor
;; reading phase
(assert (>= start_task_1 30))
(assert (>= end_task_1 30))
;; similar for all others

;; A task completion can only be after the start and the
;; duration
(assert (= end_task_1 (+ start_task_1 duration_task_1)))
;; similar for all others

;; All tasks have to be finished befor the actuator
;; writing phase begins
(assert (<= end_task_1 120))
;; similar for all others

;; Check if the sum of all task completion times is less
;; or equal than the value mid (in this case 993)
(assert (<= (+ (+ ... (+ end_task_1 end_task_2)...)
              end_task_n) 993))

;; CASE I  ;; (task_1, task_2)
(assert
 (or
  (and (> start_task1 start_task_2)
       (not (> start_task_2 end_task_1)))
  (and (> start_task_2 end_task_1)
       (not (> start_task_1 end_task_2)))
 )
)
;; similar for all others

;; CASE II  ;; (task_4, task_3)
(assert (> start_task_4 end_task_3))
;; similar for all others

(check)
```

Figure 4. Excerpt from the generated input file in the YICES format

used for a mode cluster differs from that of a working cluster. Thus two different code generators are used for clusters. Moreover, the calculated schedule has to be transferred into code for execution on a concrete platform, which demands

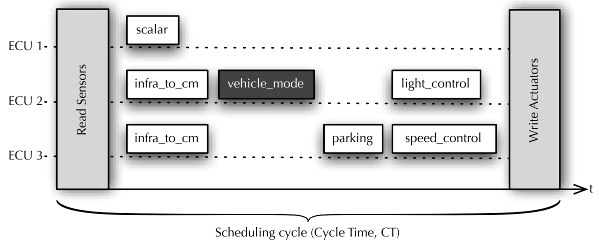

Figure 5. Schedule cycle

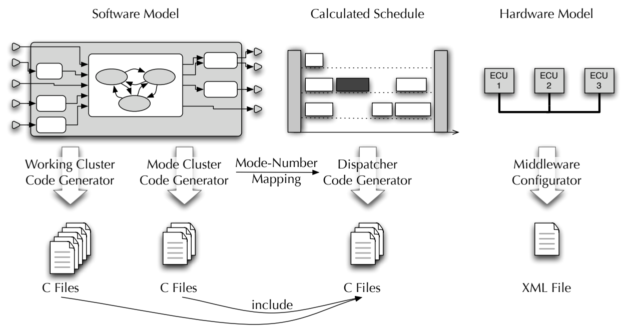

Figure 6. Code generators and their artifacts

for a third code generator. As we employ a middleware for transparent communication, which has been introduced in [8], there is a need to configure the platform according to the actual allocation. This is, again, done by a separate code generator, thus leading to four code generators overall.

For each cluster in the model one file of C code is generated, no matter if it is a working or a mode cluster. Furthermore, using the results calculated by the scheduling algorithm described in Section 3.3, a single file of C code is generated for every computing node. The latter file embodies the dispatcher and contains a loop calling all clusters at the point in time defined as starting time for each cluster by the scheduler. As indicated in Figure 6, the code files generated for mode and working clusters are hence referenced by the dispatcher. Finally the hardware model is used as input for platform configuration.

We will describe the execution of each of these generation steps next. All described concepts have been implemented using the Eclipse development plattform. Our prototypical implementation includes, amongst other tools, a meta-model for COLA whose API is used for accessing modeled systems' designs and a graphical editor for definition of those designs.

4.1 Working clusters

Working clusters implement the modeled system's actual behavior in an operating mode. They may be distributed over the available computing nodes of the hardware platform using an arbitrary mapping, just restricted by the hardware resources available. This degree of freedom regarding the allocation is achieved by communicating all the input and output values using the system's middleware. To this end, each channel connecting two clusters in the model is represented by a middleware address. The interconnected clusters, which are executed as tasks at runtime, then use this address in their communication primitives.

Besides inter-cluster communication, the middleware also serves as a data storage for the tasks' internal states.

```c
void vehicle_mode() {
   switch(unitstate->vehicle_mode_state) {
      case 0:
         if((! (steering_control == 0))) {
            decision = 2;
            unitstate->vehicle_mode_state = 2;
            break;
         }
         if(((steering_control == 0) && ((mode_control == 1) && (! emergency_stop)))) {
            decision = 1;
            unitstate->vehicle_mode_state = 1;
            break;
         }
         if(((mode_control == 0) || emergency_stop)) {
            decision = 2;
            unitstate->vehicle_mode_state = 2;
            break;
         }
         decision = 0;
         break;
      case 1:
         ...
      case 2:
         ...
   }
}

void mode2074738() {
   mw_restore_task_state(&stateVal, sizeof(stateVal), 110);
   mw_receive(&mode_control, sizeof(mode_control), 118);
   mw_receive(&speed_control, sizeof(speed_control), 130);
   mw_receive(&steering_control, sizeof(steering_control), 131);
   mw_receive(&distance_front, sizeof(distance_front), 115);
   mw_receive(&distance_front_right, sizeof(distance_front_right), 132);
   mw_receive(&distance_right, sizeof(distance_right), 136);
   mw_receive(&distance_back, sizeof(distance_back), 126);
   mw_receive(&axle_rotation, sizeof(axle_rotation), 123);
   mw_receive(&parking_ready, sizeof(parking_ready), 103);
   mw_receive(&emergency_stop, sizeof(emergency_stop), 101);
   mw_receive(&last_vehicle_mode, sizeof(last_vehicle_mode), 125);
   vehicle_mode();
   mw_send(&decision, sizeof(decision), 111);
   mw_save_task_state(&stateVal, sizeof(stateVal), 110);
}
```

Listing 1. The vehicle_mode **cluster code**

Thus each cluster is assigned an address, where it may save the activated states of its automata and recent values of contained delays.

We have presented the details of code generation for working clusters in [9]. Calling the correct set of working clusters for the actual operating mode is carried out by the dispatcher.

4.2 Mode clusters

As mentioned before, a mode cluster may, in contrast to working clusters, contain other clusters. Mode clusters decide upon which set of working clusters to execute in the actual operating mode and when to change that mode. Thus mode clusters are located from top level down to the first definition of a contained working cluster in the model hierarchy. Accordingly, the code generator for mode clusters starts working at the very top of the model and generates code for all sub-units, which do not define a cluster themselves. If a unit is found which defines a cluster, code generation stops. The discovered unit must, per language definition, be the implementation of an automaton's state, i. e., an operating mode. According to the type of cluster detected, it is again coded as a separate mode or a working cluster.

The mode cluster's duty is to decide upon the active mode, depending on its input values. Based on the calculated mode, it outputs a numeric value which is mapped to that mode. Thus, each mode is mapped to a distinct number

```
 1  infra_to_cm20792_init();
 2  rotation_2002502_init();
 3  sdc_active206638_init();
 4  speed_control2072689_init();
 5  normal206613_init();
 6  parking206663_init();
 7
 8  while(run) {
 9     mw_global_time(&time_start);                          //store the actual global time
10     rt_task_resume(&task_sensor);                         //first read the sensors
11     rt_task_resume(&task_send);                           //resume the send task
12     mw_global_time(&time_actual);
13     rt_task_sleep(78000000 - (time_actual - time_start));     //wait till 78ms
14     infra_to_cm20792();
15     rt_task_resume(&task_send);                           //resume the send task
16     ...
17     mw_global_time(&time_actual);
18     rt_task_sleep(121000000 - (time_actual - time_start));    //wait for mode decision at 121ms
19     mw_receive(&mode_decision, sizeof(mode_decision), 111);
20     if(mode_decision == 0) {
21        mw_global_time(&time_actual);
22        rt_task_sleep(122000000 - (time_actual - time_start)); //wait till 122ms
23        parking206663();
24        rt_task_resume(&task_send);                        //resume the send task
25        ...
26     }
27     else if(mode_decision == 2) {
28        ...
29     }
30     else if(mode_decision == 1) {
31        ...
32     }
33     mw_global_time(&time_actual);
34     rt_task_sleep(170000000 - (time_actual - time_start));
35     rt_task_resume(&task_actuator);                       //wait to run actuator task
36  }
```

Listing 2. A node's dispatcher code

by the code generator. The generated mapping is used during coding of the dispatcher to start the mode and working clusters in question. This mapping is used during generation of the dispatcher as given in Section 4.3.

In Listing 1 a shortened example for the code generated for a mode cluster can be seen. This sample belongs to the vehicle_mode automaton depicted in Figure 1. The mode is called using its main function defined in line 28. During the cluster's execution first the actual state, which includes the active mode, is read from the middleware. The corresponding middleware call is shown in line 29 of the listing. Then the values for the input ports are read, as shown in lines 30 through 40. The automaton responsible for deciding upon the active operating mode is called in line 41. Based on the previous state, c.f. line 2, the outgoing transitions are checked. If one of the guards in lines 4, 9 and 14 evaluates to true, the state of the vehicle mode automaton changes and the emitted decision value is set accordingly. For brevity, the code in Listing 1 only shows the implementation for one of the automaton's modes and omits variable declarations.

The mode decision is communicated to the underlying middleware layer, and thus to all computing nodes in the system, using the API call shown in line 42. Finally, the updated mode cluster state is stored in the middleware, which is achieved by the code in line 43.

4.3 Dispatcher

As stated before, it is the dispatcher's purpose to start the clusters allocated to that node, the dispatcher is executed on. Our concept envisions the use of a non-preemptive scheduler, which is assigned the tasks to execute by the dispatcher. For our demonstrator however, we had to rely on a preemptive scheduler, as the employed operating system Xenomai does not offer a non-preemptive mode. The solution to this problem was to use semaphores to achieve the desired non-preemptive behavior.

Extracts of the dispatcher for one of the demonstrator's nodes are given in Listing 2. In lines 1 through 6 the init functions of all clusters allocated to the node are called. This triggers the initial states for all automata and delays contained in the clusters. Then in line 8 the scheduling cycle loop is started. At its beginning the global time is read from the middleware by the function call in line 9. It is stored in a variable and used to calculate waiting periods between the executed tasks, according to the calculated schedule. Examples for these sleeps can be found in lines 13, 18, 22, and 34. Instances of called clusters are located in lines 14 and 23. The other clusters are omitted for brevity in Listing 2. As shown in Figure 5, the scheduling cycle starts with sensor readings and ends with actuator writes. The according calls are shown in lines 10 and 35 of Listing 2.

So far we have only dealt with working cluster calls. The special thing about mode clusters is that they are executed on one node of the system and their result influences the dispatching of all nodes of the system. In the example shown in Listing 2 the branching between different operating modes, and thus different scheduling plans, can be seen in lines 20, 27, and 30. These statements use the mapping of modes to numerical values, defined during mode code generation described in the preceding section. Please note that the actual mode value is read from the middleware in line 19, because the according mode cluster is run on another node. Obviously, the middleware address `111` has to be used here as well as in the code generated for the mode cluster to communicate the actual operating mode. For reference see line 42 of Listing 1 where the mode result is written to the middleware.

Because of the globally defined scheduling for all computing nodes of the system and the global time available from the employed middleware, synchronous mode switches can be guaranteed. Every node in the systems waits for the reception of the calculated mode value according to the pre-defined schedule. Thus this value is always up-to-date.

4.4 Middleware Configuration

To allow for transparent communication via our middleware, a configuration for the nodes has to be given. This configuration states information about the sensors and actuators connected to each node. Furthermore, for each cluster allocated onto a node, the middleware addresses read from and written to are included. Thus the memory needed for input and output data can be allocated.

As indicated in Figure 6 the configuration is done using an XML file. The middleware loads this file at startup and reads the information suitable for the current node. We explained the operation of the middleware in detail in [8].

5 Case Study

To prove the viability of our approach, we implemented a case study using the concepts and tools described before. The idea was to build a model car, featuring a function also available in real cars. We decided to implement an autonomous parking system based on several distance sensors. Additionally, the system should be controllable manually via a Bluetooth connection to a cell phone, and it should initiate an emergency stop when reaching a given minimum distance to obstacles.

The model car was equipped with three Gumstix® microcomputers connected by an Ethernet network. Our middleware was employed on top of the network for data exchange and clock synchronization services. Xenomai served as operating system for the nodes. Distances were measured using two infrared and one supersonic sensor. Bluetooth was used as another input, connected to the cellular phone remote. The model car's motor and steering, indicator, reversing, and breaking lights were the actuators of the system. The mentioned sensors and actuators were connected to different computing nodes, thus posing the need for synchronous communication in the system.

Further the system should be separated into three operating modes, namely normal, parking, and sdc_active. The normal mode leaves control to the user, which can modify speed and direction using the remote control. parking makes the model car run in parallel to a wall and search for a gap of sufficient size to park. If such a gap is found, the model automatically starts parking using a predefined curve. In sdc_active mode, the car drives parallel to a given wall and adjusts its distance to all convexities found. Mode changes are triggered using the remote control, thus implicitly altering the clusters executed on all microcomputers.

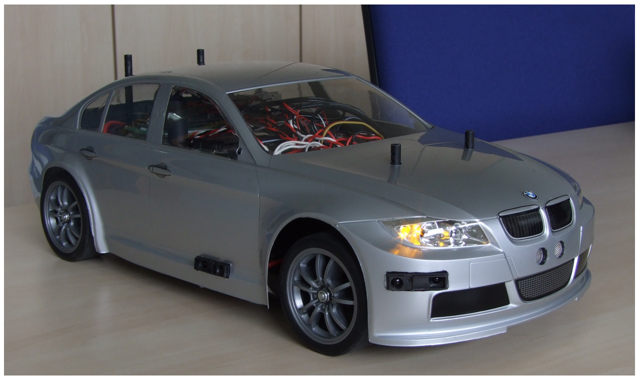

Figure 7. Developed case study

After specification of the software and hardware models, we were able to generate code for the specified software components as well as the middleware configuration files. Past cross-compilation of the code, the system behaved correctly without the need of any manual coding. We were able to change between driving, parking, and side distance mode without any issues. Hence the operating mode changes were communicated and executed correctly on all involved computing nodes.

6 Conclusions

In this paper, we presented an approach for the automatic generation of distributed embedded systems. The employment of operating modes offers a desirable partitioning of the systems functionality, by means of easier modeling as well as efficient resource usage. The modeling constructs of mode automata allows for an easy to understand definition of the different operating modes.

COLA features the concept of mode automata and enables for their specification using a graphical syntax. By using COLA with its clearly defined semantics, the automated checking of model characteristics is made possible, leading to a less faulty design. Further, automatic allocation and scheduling is facilitated. The results of those steps are used to generate executable code that preserves the correctness of the modeled system and the synchronous change between specified operating modes on an actual distributed platform.

The viability of this approach was pointed out by realizing a case study based on the implemented tools.

References

[1] 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.

[2] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. In *Readings in hardware/software co-design*, pages 147–159. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

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

[4] M. Broy. Automotive software and systems engineering (panel). In *MEMOCODE*, pages 143–149, 2005.

[5] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From simulink to SCADE/lustre to TTA: a layered approach for distributed embedded applications. In *LCTES*, pages 153–162. ACM, 2003.

[6] B. Dutertre and L. de Moura. The yices smt solver. Tool paper at http://yices.csl.sri.com/tool-paper.pdf, August 2006.

[7] D. K. Friesen and M. A. Langston. Variable sized bin packing. *SIAM J. Comput.*, 15(1):222–230, 1986.

[8] 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.

[9] W. Haberl, M. Tautschnig, and U. Baumgarten. From COLA Models to Distributed Embedded Systems Code. *IAENG International Journal of Computer Science*, 35(3):427–437, Sept. 2008.

[10] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[11] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: A time-triggered language for embedded programming. In *EMSOFT*, pages 166–184, 2001.

[12] T. A. Henzinger, C. M. Kirsch, and S. Matic. Composable code generation for distributed giotto. In *LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 21–30, New York, NY, USA, 2005. ACM.

[13] M. U. Khan, K. Geihs, F. Gutbrodt, P. Gohner, and R. Trauter. Model-driven development of real-time systems with uml 2.0 and c. *Model-Based Methodologies for Pervasive and Embedded Software, International Workshop on*, 0:33–42, 2006.

[14] S. Kugele and W. Haberl. Mapping Data-Flow Dependencies onto Distributed Embedded Systems. In H. R. Arabnia and H. Reza, editors, *Proceedings of the 2008 International Conference on Software Engineering Research & Practice, SERP 2008*, volume 1, pages 272–278. CSREA Press, July 2008.

[15] S. Kugele, W. Haberl, M. Tautschnig, and M. Wechs. Optimizing automatic deployment using non-functional requirement annotations. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation*, volume 17 of *CCIS*, pages 400–414. Springer, 2008.

[16] 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. Technical Report TUM-I0714, Institut für Informatik, Technische Universität München, Sept. 2007.

[17] 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.

[18] B. Schätz. Model-based engineering of embedded control software. In *MBD-MOMPES '06: Proceedings of the Fourth Workshop on Model-Based Development of Computer-Based Systems and Third International Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, pages 53–62, Washington, DC, USA, 2006. IEEE Computer Society.

[19] Z. Wang, A. Sanchez, and A. Herkersdorf. Scisim: a software performance estimation framework using source code instrumentation. In *WOSP '08: Proceedings of the 7th international workshop on Software and performance*, pages 33–42, New York, NY, USA, 2008. ACM.