Towards Modularized Verification of Distributed Time-Triggered Systems*

Jewgenij Botaschanjan[1], Alexander Gruler[1], Alexander Harhurin[1], Leonid Kof[1], Maria Spichkova[1], and David Trachtenherz[2]

[1] Institut für Informatik, TU München,
Boltzmannstr. 3, D-85748, Garching bei München, Germany
{botascha,gruler,harhurin,kof,spichkov}@in.tum.de
[2] BMW Group Research and Technology,
Hanauer Strasse 46, D-80992, München, Germany
David.Trachtenherz@bmw.de

Abstract. The correctness of a system according to a given specification is essential, especially for safety-critical applications. One such typical application domain is the automotive sector, where more and more safety-critical functions are performed by largely software-based systems.
Verification techniques can guarantee correctness of the system. Although automotive systems are relatively small compared to other systems (e.g. business information systems) they are still too large for monolithic verification of the system as a whole.
Tackling this problem, we present an approach for modularized verification, aiming at time-triggered automotive systems. We show how the concept of tasks, as used in current automotive operating systems, can be modeled in a CASE tool, verified and deployed. This results in a development process facilitating verification of safety-critical, real-time systems at affordable cost.

1 Introduction

Together with the growing functionality offered by today's distributed reactive systems, the associated complexity of such systems is also dramatically increasing. Taking into account that the vast majority of the functionality is realized in software, the need for appropriate design and verification support becomes obvious.

A prime example for this trend is the current situation in the automotive domain. Here, a premium class car contains up to 70 electronic control units (ECUs) which are responsible for all kinds of applications: infotainment (like navigation and radio), comfort (power windows, seat adjustment, etc.), control of technical processes (motor control, ABS, ESP), and much more. Consequently, the amount of associated software is enormous – with the tendency to further increase in the future.

* This work was partially funded by the German Federal Ministry of Education and Technology (BMBF) in the framework of the Verisoft project under grant 01 IS C38. The responsibility for this article lies with the authors.

With the trend going towards drive-by-wire, the software becomes responsible for safety-critical functions, like steer-by-wire and brake-by-wire. The state-of-the-art method of quality assurance, namely testing, is not sufficient in the case of safety-relevant functions: Testing can solely show the absence of bugs in a finite number of standard situations. However, it can never *guarantee* the software correctness. Formal verification is a better choice in this case, as it can *guarantee* that the software satisfies its specification.

Unfortunately, current verification techniques for reactive systems suffer from some problems: Firstly, in order to prove the correctness of a system, both the application logic itself as well as its infrastructure (operating system and communication mechanisms) have to be verified. This results in an overall verification effort which cannot be mastered by verifying the system as a whole.

Secondly, there is no continuous verification technique: While current CASE tools typically used for automotive software development (like MATLAB/Simulink [1], Rose RT [2], AutoFOCUS [3]) allow modeling of the functionality and structure of a real-time system, they do not provide an explicit deployment concept. However, without deployment support it makes no sense to verify properties on the application model, since they do not necessarily hold after deployment.

To tackle these problems we introduce a task concept for the model-based development of distributed real-time systems, which allows modularized verification while preserving verified properties for the model after deployment. Together, this results in a continuous methodological support for development of verified automotive software.

We show the feasibility of our concepts on a case study. We demonstrate that embedding of tasks into a realistic environment, such as a time-triggered bus and a time-triggered operating system, does not violate the verified properties.

The remainder of this paper is organized as follows: Section 2 introduces the case study used as a continuous example throughout the whole paper. Sections 3 and 4 present the deployment platform (FlexRay Communication Protocol [4] and OSEK-time OS [5]) and the CASE tool AutoFOCUS [3], used to specify the case study as a task model. Sections 5 and 6 are the technical core of the paper: They show how the tasks should be constructed in order that they are deployable without any loss of verified properties. Section 7 gives an overview of related work and, finally, Section 8 summarizes the whole paper.

2 Case Study: Emergency Call (eCall)

To demonstrate the introduced ideas we use an automated emergency call as a running example throughout this paper. According to the proposal by the European Commission [6], such an automated emergency call should become mandatory in all new cars as of 2009. The application itself is simple enough to be sketched in a few paragraphs, but it still possesses typical properties of automotive software. By this we mean that it is a safety-critical application distributed over several electronic control units (ECUs), whose correct functionality not only depends on the correctness of the application itself but also on the correctness of a real-time OS and a real-time bus.

We model the eCall as a system consisting of 3 sub-systems, namely: a GPS navigation system, a mobile phone, and the actual emergency call application. External information (e.g. the crash sensor, the GPS signals) is considered to be a part of the environment. According to [6], these components interact as follows: The navigation system sends periodically the vehicle's coordinates to the emergency call application so that it always possesses the latest coordinates. The crash sensor sends periodically the current crash status to the emergency call application. If a crash is detected, the emergency call application initiates the eCall by prompting the mobile phone to establish a connection to the emergency center. As soon as the mobile phone reports an open connection, the application transmits the coordinates to the mobile phone. After the coordinates have been successfully sent, the application orders the mobile phone to close the connection. The emergency call is finished as soon as the connection is successfully closed. If the radio link breaks down during the emergency call, the whole procedure is repeated from the initiation step.

3 Deployment Platform

In order to master the inherent complexity of automotive systems, industry came up with a number of standards, based on the *time-triggered paradigm* [7]. They allow realization of distributed systems with predictable time behavior, and thus can be considered as an appropriate deployment target for safety-critical real-time systems.

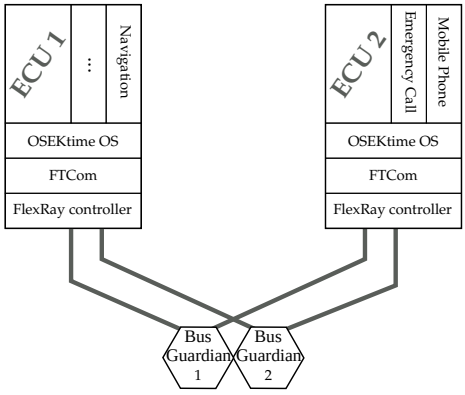

Fig. 1. Target Deployment Platform Architecture

In a time-triggered system actions are executed at predefined points in time. In particular, using time-triggered operating systems (VxWorks [8], QNX [9], OSEKtime [10]), the execution of application processes is statically scheduled, and by applying time-triggered communication protocols (TTP/C [11], TTCan [12], FlexRay [13]), the communication schedule becomes static as well. Further on, time-triggered communication protocols provide, using time synchronization, a global time base to the

distributed communication partners. By this a combination of time-triggered OS and network allows realization of a deterministic system behavior with guaranteed response times.

The target deployment platform of the presented work is a network of ECUs connected by a FlexRay bus with a multiple-star topology and with OSEKtime OS running on every node (see [14] for details). Fig. 1 shows a possible deployment of the three tasks from the eCall study on two ECUs.

OSEKtime. OSEKtime OS is an OSEK/VDX [10] open operating system standard of the European automotive industry [5]. The OSEKtime OS supports static cyclic scheduling. In every round the dispatcher activates a process at the point of time specified in the scheduling table. If another process is running at this time, it will be preempted until the completion of the activated process. OSEKtime also monitors the deadlines of the processes. In the case of deadline violation an error hook is executed.

FTCom [15] is the OSEKtime fault-tolerant communication layer that provides a number of primitives for interprocess communication and makes task distribution transparent. Messages kept in FTCom are uniquely identified by their IDs. For every message ID FTCom realizes a buffer of length one. Application processes can send or receive messages with certain IDs using communication primitives offered by FTCom. However, they are not aware of the location of their communication partners, i.e. whether they communicate locally or through a bus.

FlexRay. FlexRay [4,16,17] is a communication protocol for safety critical real-time automotive applications, that has been developed by the FlexRay Consortium [13]. It is a static time division multiplexing network protocol and supports fault-tolerant clock synchronization via a global time base. The drifting clocks of FlexRay communication partners are periodically synchronized using special sync messages.

The static message transmission mode of FlexRay is based on *rounds*. FlexRay rounds consist of a constant number of time slices of the same length, so called *slots*. A node can broadcast its messages to other nodes at statically defined slots. At most one node can broadcast during any slot. The latency of every message transmission is bounded by the length of one slot.

A combination of time-triggered OS and bus allows synchronization of the computations and communication. This can be done by synchronizing the local ECU clock with the global time delivered by FlexRay bus and by setting the length of the OSEKtime dispatcher round to be a multiple of the length of FlexRay round. A unit of computation is then also a FlexRay slot.

4 Logical Model

AutoFOCUS is a CASE tool for graphical specification of reactive systems. The tool has a formal time-synchronous operational semantics. With its graphical specification language AutoFOCUS supports different views on the system, namely structure, behavior, interaction and data-type view. This section gives a very short introduction to AutoFOCUS. A more detailed description of AutoFOCUS and its diagram types can be found in [18].

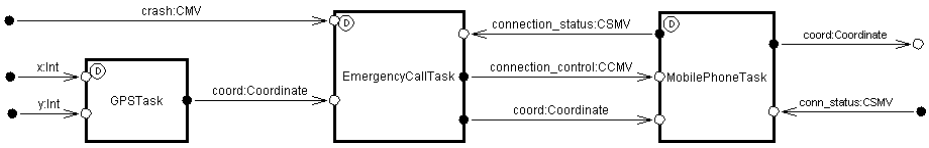

Fig. 2. The Component Architecture of the eCall Study

Structural View: System Structure Diagrams (SSDs). A system in AutoFOCUS is modeled by a network of components C, denoted as rectangles (cf. Fig. 2, where the task model of the eCall application is shown). The communication interface of a component $c \in C$ is defined by a set of typed directed *ports* P_c. There are *input* and *output* ports, represented by empty and solid circles, respectively. The components communicate via typed channels, which connect input and output port pairs of matching types. An output port can be a source of several channels, while an input port can be a sink of at most one. A component can be refined by a further component network. This results in a hierarchical order of SSDs. The leaf components in the hierarchy have an assigned behavior, described by *State Transition Diagrams*.

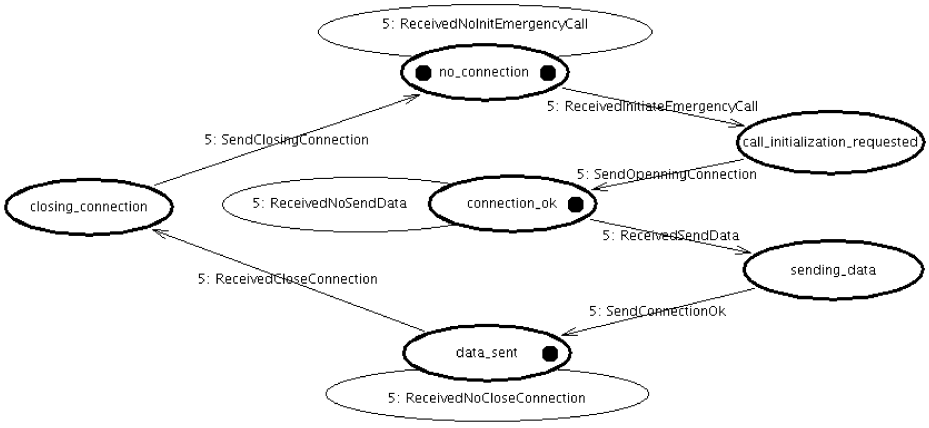

Fig. 3. State Transition Diagram of the *Mobile Phone* Logic Component

Behavior View: State Transition Diagrams (STDs). The behavior of a leaf component is defined by an extended I/O automaton. It consists of a set of *control states*, *transitions*, and *local variables* (cf. Fig. 3). Black dots on the left of the states denote initial states, black dots on the right denote idle states[3]. ("5" on the transition labels denotes the transition priority and is unimportant in the context of our application.)

[3] Idle states will be defined below, see Sect. 5.2.

An STD automaton $\mathcal{A}$ is completely defined by its set of control states S, the set of local variables V, the initial state $s_0 \in S$, and the transition relation δ. A transition, denoted by

$$(s_1 \xrightarrow{pre:in:out:post} s_2) \in \delta(\mathcal{A}), \quad \text{for } s_1, s_2 \in S,$$

consists of four elements:

- pre, transition precondition (a boolean expression referring to automaton's local variables and the ports of the corresponding component)
- in, input pattern that shows which message must be available on which port in order that the transition is triggered
- out, output pattern that shows which message is written to which port as a result of this transition
- $post$, transition postcondition (a boolean expression referring to automaton's local variables and the ports of the corresponding component)

A transition can be triggered in the state s_1 if the input ports specified in the input pattern in have received the necessary input messages and the precondition pre is satisfied. The transition outputs data to output ports specified in out, changes the local variables according to the postcondition $post$, and puts the automaton into the state s_2.

For example, the transition with label ReceivedNoCloseConnection from Figure 3 has the form:

```
(x != close_connection):
connection_control?x:
connection_status!data_sending_ok,idle!Present:
```

It is fired if it gets a signal different from close_connection on port connection_control and sends the data_sending_ok signal through port connection_status and the Present signal via port idle. The postcondition is empty.

Communication Semantics. AutoFOCUS components perform their computation steps simultaneously, driven by a global clock. Every computation step consists of two phases: First, a component reads values on the input ports and computes new values for local variables and output ports. After the time tick new values are copied to the output ports, where they can be accessed immediately via the input ports of connected components and the cycle is repeated. This results in a *time-synchronous* communication scheme with buffer size one.

5 Deployment

An automotive system is distributed over several communicating electronic control units (ECUs). In the early design stages it is simpler to model application logic without considering timing properties and future deployment. Pure functionality modeling results in systems that have to be deployed manually and after deployment can show wrong behavior or violate timing constraints.

In this section we motivate and describe the AutoFOCUS Task Model (*AFTM*), which will be used for (semi-)automatic verification of time–triggered systems. Here, we introduce a framework enabling modeling and deployment of tasks. The framework architecture ensures that the properties verified for the logical model are preserved after deployment. This results in a verified automotive system, provided that the infrastructure (OS/bus) is verified. For these purposes the framework [14] for automotive systems with the formalized and verified OSEKtime OS and the FlexRay protocol can be used. This verification framework is developed in the automotive part of the Verisoft project [19,20].

5.1 Design Process

The AutoFOCUS Task Model is designed to be an integral part of a model-based development process. The prerequisite for this process is a formal specification of the desired system. The envisioned process produces the following artifacts:

- The AutoFOCUS model (see also Section 4) represents the application logic (functionality) without incorporating any deployment information. It is the basis for the AutoFOCUS task model.
- The AutoFOCUS Task Model is an extension of the AutoFOCUS model, targeting at future deployment to a time-triggered platform. This model is verified against the given specification.
- C0 code[4] is generated from the AFTM. This code is the basis for the estimation of the worst case execution times (WCETs) that are needed for scheduleability analysis (see Section 6.1). WCETs can be estimated, given a compiled C program and the processor the program runs on [23]. The behavioral equivalence between C0 code and the automaton it was generated from must be proven. This proof can be done, for example, by the means of translation validation [24]:
  - Every transition of an AutoFOCUS automaton is annotated with a pre- and a postcondition.
  - The piece of code, generated from a particular transition, can be annotated with the same pre- and postconditions as the transition itself.
  - Finally, we verify that the pre- and postconditions hold for the generated code.
- For the deployment and scheduling the decision must be taken which tasks run on which ECUs. The schedules both for every ECU and for the communication bus must be also constructed. The schedules have to take into account casual dependencies of the tasks, the WCETs and the real-time requirements (e.g. specified response times).
  At the moment we do not construct schedules but only check existing schedules for correctness. Construction of schedules is a part of our future work.

In that way we obtain a continuous model-based development process, ranging from high-level system design to verified deployed code.

[4] The language C0 [21] is a Pascal-like restriction of C that is similar to MISRA C [22]. In safety critical applications it is feasible to use C in some restricted way to have less error-prone programming style.

5.2 AutoFOCUS Task Model (AFTM)

An AutoFOCUS task model is obtained from AutoFOCUS model components through encapsulation – Fig. 4 shows an AFTM task originating from an AutoFOCUS component app_logic. A task may contain a single component or a network consisting of several components – it is then treated as a product automaton (see *Behavior View* in Sect. 4). The transformation from an AutoFOCUS model into an AutoFOCUS task model is performed manually at the moment, but it is planned to extend the tool AutoFOCUS in order to automate this transformation.

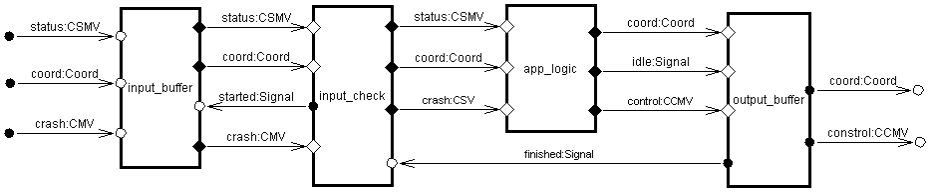

Fig. 4. AutoFOCUS task model of EmergencyCallTask

An AutoFOCUS task model consists of a set $\mathcal{T}$ of *tasks* T_i with $i \in \{1, \ldots, m\}$ for an arbitrary but constant m and a set of directed channels between them. We denote by $Pred, Succ : \mathcal{T} \rightarrow \mathcal{P}(\mathcal{T})$ the functions indicating data flow between tasks: $T_j \in Succ(T_i)$ means that there is a channel going from T_i to T_j and $T_j \in Pred(T_i)$ denotes a channel going from T_j to T_i.

Every component in an AutoFOCUS model runs continuously, whereas the execution of a reactive system is an infinite sequence of finite computations started by the OS scheduler and terminating with a special exit() system call. To match this computation model, we introduce in AFTM the notion of finite computations through *idle states*. An idle state is a state of the original logic component (or component network), where the computation can continue only after having received new input. In contrast to idle states, *non-idle* states are allowed to have outgoing transitions only without requiring any input. Thereby the set of control states is partitioned into two disjoint subsets:

$$S_{idle} = \{s \mid s \in S \wedge \forall (s \xrightarrow{pre:in:out:post} s') \in \delta(\mathcal{A}).\ in \neq \emptyset\} \tag{1}$$

$$S_{non\_idle} = \{s \mid s \in S \wedge \forall (s \xrightarrow{pre:in:out:post} s') \in \delta(\mathcal{A}).\ in = \emptyset\} \tag{2}$$

An AFTM task computation is a finite sequence of state transitions leading from an idle state to some other idle state:

$$\begin{aligned} c(val(P_c), s_0) = & s_0 \xrightarrow{pre_1:in:out_1:post_1} s_1 \xrightarrow{pre_2:\emptyset:out_2:post_2} \ldots \\ & \xrightarrow{pre_{n-1}:\emptyset:out_{n-1}:post_{n-1}} s_{n-1} \xrightarrow{pre_n:\emptyset:out_n:post_n} s_n \end{aligned} \tag{3}$$

where $s_0, s_n \in S_{idle}$, $s_1, \ldots, s_{n-1} \in S_{non\_idle}$, $(s_{i-1} \xrightarrow{pre_i:in_i:out_i:post_i} s_i) \in \delta(\mathcal{A})$ for all $i \in \{1, \ldots, n\}$, and $val(P_c)$ denotes the valuation of the component's ports. This

linkage between target platform task runs and AFTM task computations is also utilized for timing analysis (cf. Sect. 6.2).

In the *Mobile Phone* logic component STD (Fig. 3, page 5) the states no_connection, connection_ok and data_sent are idle. An example of the finite computations that can be performed by this automaton is connection_ok → sending_data → data_sent.

Upon reaching an idle state the encapsulated component of a task always sends a signal through a dedicated idle port:

$$\forall(s_1 \xrightarrow{pre:in:out:post} s_2) \in \delta(\mathcal{A}) : \ (s_2 \in S_{idle} \Leftrightarrow (\mathsf{idle!Present}) \in out) \qquad (4)$$

A distributed time-triggered system usually does not guarantee the simultaneous presence of all required input signals because of delays. As an AFTM task may start only when all required input data are available we introduce an *input check* for every task – it forwards the inputs and thus allows the task to start only after all required inputs have arrived and the task is not running, i.e. it is in an idle state.

We introduce two kinds of input checks: OR and AND. An OR-task T can start when at least one input from any task $T_i \in Pred(T)$ has arrived. For instance the EmergencyCallTask task on Fig. 2 is activated either to store new coordinates from the GPSTask or to perform an emergency call after having received a crash signal. The idea behind the AND-check is that the task can start only when all the inputs are available. For instance, the GPSTask (Fig. 2) may first start when both coordinate inputs x and y have arrived.

The input checks get their inputs solely from the *input buffers* (see e.g. input_buffer component in Fig. 4). These buffers store the arriving data as long as the tasks cannot process them. After the data gathered by the buffer has passed the input check, the input check sends the started signal to the buffer. Thereupon the input buffer is flushed and starts gathering a new data set. This simulates the behavior of the FTCom, which is used for task communication on the deployment platform (cf. Sect. 3).

The *output buffer* is necessary to assure well-defined points in time for communication and thus to make communication behavior predictable. The output buffer stores the outputs of the application logic and forwards them to the environment on receiving the idle signal. After that the output buffer is flushed and forwards the idle signal to the input check, indicating the completion of the task computation.

The introduced concepts of input checks and input/output buffers, as well as idle states, allow correct deployment on a distributed platform running with OSEKtime/FlexRay (cf. Sect. 3). The AFTM properties facilitating deployment are in particular the following: it models the behavior of the FTCom communication layer, it supports the notion of finite computations as suitable for time-triggered systems, it reads the input data at the beginning and communicates the results at the end of the computation, thus facilitating scheduling and modular verification.

5.3 Code generation

To run a task on a real system the representation of the model as code in some executable language is needed (e.g., for the automotive domain C code is usually used). Out of the

AFTM the corresponding C code (more precisely: C0 code) can be generated in a strict schematical way.

In the presented approach properties of a task are proven for the corresponding AFTM since this is more effective than verifying the generated C0 code. The equivalence of AFTM and C0 guarantees that the verified properties for the AFTM also hold for the C0 code. This equivalence can easily be proven using translation validation [24].

6 Verification

In the previous sections we have described how an AutoFOCUS component model can be packed into an AFTM. This section shows how the AFTM and the deployed system, based on AFTM, can be verified. The verification is accomplished in several steps: First, the AutoFOCUS model, packed into AFTM, is verified. This can be done, as the SMV model checker [25] is integrated with AutoFOCUS [26]. Then, it is necessary to show that the properties verified for the AutoFOCUS model remain valid in the deployed model. Formally, let $Model_{AFTM}$ denote the AutoFOCUS Task Model, $Model_{depl}$ the deployed model, and P the specification (a set of LTL properties). Then, the accomplishment of this procedure results in the following property.

$$(Model_{AFTM} \vDash P) \Rightarrow (Model_{depl} \vDash P) \tag{5}$$

The prerequisite for the fulfillment of this formula is that $Model_{AFTM}$ is a valid instance of the AFTM, as described in Sect. 5.2. This is discussed in Sect. 6.1. Sect. 6.2 shows that AFTM constraints, together with certain scheduling constraints put on the deployed system, imply behavioral equivalence between $Model_{AFTM}$ and $Model_{depl}$. Behavioral equivalence, in turn, provides the desired correctness of the deployed system, given that $Model_{AFTM}$ was verified.

6.1 Task and System Properties

To ensure the validity of a particular AutoFOCUS task model, certain properties have to be proven. First, it must be shown for every component network, deployed to one task, that all states in its product automaton that are marked as *idle* satisfy Formula 1. All the remaining states must satisfy Formula 2 (see page 8). These are purely syntactic checks on the AutoFOCUS task product automaton.

In the eCall example the automaton of the mobile phone component as shown in Fig. 3 (page 5) has three idle states (no_connection, connection_ok and data_send). In these states new inputs from the Emergency Call task and the radio link status are expected respectively. The remaining states are non-idle, thus, the transitions from these states are not allowed to access the input ports.

The second property to be verified is that every task sends a special idle signal before entering an idle state. This signal is needed in AFTM to affect input_check and output_buffer components (see Sect. 5.2). Formally it must be verified that Formula 4 holds for every transition.

Finally, it is necessary to verify that every sequence of transitions starting in an idle state always reaches some idle state within a finite number of steps. The first two

properties are simply syntactic checks on the model, while the last one can be verified via model checking. To verify the last property, it is possible to use the SMV back end of the tool AutoFOCUS.

6.2 Timing Properties

Timing properties ensure the equivalence of the data flow dependencies imposed by the AFTM channels and the task dependencies in the deployed system. The prerequisite for every timing analysis technique are the estimates for BCET/WCET for every task. In our setting these are minimal/maximal execution times between any pair of idle states. The estimation can be done e.g. using the static analysis techniques by AbsInt [23]. The task running time expressed in the logical time base of the deployment architecture (number of slots) is then obtained by dividing the BCET/WCET estimates by the slot length:

$$b^{slot} = \left\lceil \frac{bcet}{|slot|} \right\rceil, w^{slot} = \left\lceil \frac{wcet}{|slot|} \right\rceil \tag{6}$$

The assumption is that no two tasks (on the same ECU) can run in the same slot. By this, the logical running times of a task T which can be used in the scheduleability analysis lie within $[b^{slot}(T), \ldots, w^{slot}(T)]$. Thus, in the remainder of this section the notion of logical time is used in the scheduling constraints.

We describe the deployment of the system by the following definitions for arbitrary tasks $T_i, T_j \in \mathcal{T}$. The set of relative start times[5] is denoted by $start(T_i)$. W.l.o.g. we assume that $start(T_k) \neq \emptyset$ for all $T_k \in \mathcal{T}$. Let the predicate $ecu(T_i, T_j)$ denote the deployment of tasks on the same ECU. The messages produced by a task are sent through FlexRay in slots from the set $send(T_i)$[6]. Finally, the number of slots in the OS round is denoted by $|round|$.

The scheduleability analysis for the given technical infrastructure lies beyond the scope of this paper, however, the obtained scheduling tables have to be checked for their correctness for the given AFTM. We say a scheduling table to be correct, if the following properties hold.

Communication Jitter. In order to make the bus communication deterministic, the FlexRay slots reserved for the messages produced by a task T, must not lie within the following interval (see Fig. 5):

$$\forall s \in start(T) : \forall ss \in send(T) : ss \notin (s + b^{slot}(T), s + w^{slot}(T)]$$

In the case of a local communication, the consuming task T_2 must not be started before the WCET of its producing counterpart T_2 passes:

$$ecu(T_1, T_2) \wedge T_1 \in Pred(T_2) \Rightarrow$$
$$\forall s_1 \in start(T_1), s_2 \in start(T_2) : s_2 \notin (s_1 + b^{slot}(T_1), s_1 + w^{slot}(T_1)]$$

[5] A task can be started several times per OS round.

[6] Note that the set can be empty if the task sends its results only locally.

These properties allow us to define the *transport time* function tr. For a given pair of tasks T_1 and T_2 with $T_1 \in Pred(T_2)$ and a start time $s_1 \in start(T_1)$, $s_1 + tr(T_1, s_1, T_2)$ is the minimal time after which T_2 is allowed to access the messages produced by T_1 when T_1 is started at time s_1. An earlier access would violate the above properties. For the *local communication*, i.e., $ecu(T_1, T_2)$ holds, this time is $w^{slot}(T_1)$. Otherwise, $tr(T_1, s_1, T_2)$ is the longest distance to the next sending slot from $send(T_1)$, which transports data that T_2 is interested in, plus 1 slot for the transportation itself. The last case is illustrated in Fig. 6. Due to the above constraints, for any fixed triple of parameters the value of tr is constant.

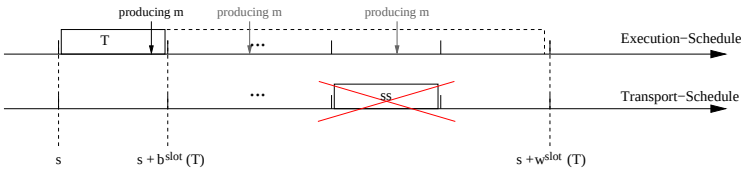

Fig. 5. Constrained Transport Times

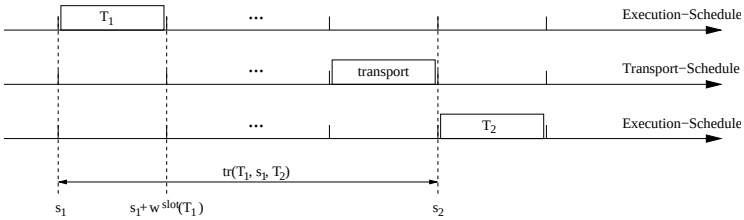

Fig. 6. Allowed Start Time

Control & Data Flow. The start times of tasks have to respect the data and control flow relations between them. We constraint starts/terminations allowed to occur between any pair of subsequent executions of a task T. For that purpose we define the following sets and predicates on them: For a task T let $start^{\perp}(T) = start(T) \cup \{-1\}$ and $start^{\top}(T) = start(T) \cup \{|round| + \min start(T)\}$. The value -1 denotes a fictitious start time smaller than any actual start time of T. The additional value of $start^{\top}(T)$ defines the first T's start in the next round. Obviously, it will be greater than any number in $start(T)$. Then for a given start time $s \in start^{\perp}(T)$ we denote T's next start time by $next(T, s) = s'$, with a *minimal* s', such that $s' \in start^{\top}(T)$ and $s' > s$. Further on, for a given start time $s \in start^{\top}(T)$ we define T's previous start time by $prev(T, s) = s'$, with a *maximal* s', such that $s' \in start^{\perp}(T)$ and $s' < s$.

Using the definitions from above, we can now formulate scheduling constraints for the both kinds of input check semantics. The AND-task T which needs outputs from

the tasks $T_1, \ldots, T_n$ demands, that these tasks must deliver their outputs between any pair of subsequent starts of T (see also Fig. 7).

$$\forall s \in start(T) : \forall T' \in Pred(T) : \exists s' \in start(T') : \\ prev(T, s) \leq s' + tr(T', s', T) < s$$

In the case of the OR-semantics for T, at least one of the tasks $T_1, \ldots, T_n$ has to deliver its outputs in this interval.

$$\forall s \in start(T) : \exists T' \in Pred(T) : \exists s' \in start(T') : \\ prev(T, s) \leq s' + tr(T', s', T) < s$$

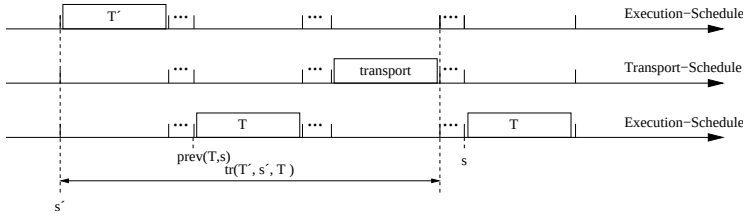

Fig. 7. Correct Data Flow

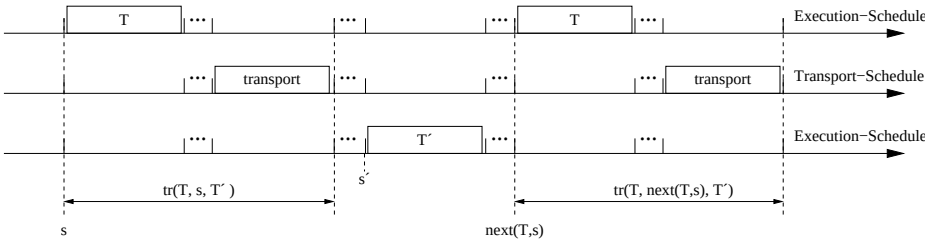

Fig. 8. No Data Loss

No Data Loss. In the presented work we consider systems where no data loss is allowed. By this a message has to be consumed by corresponding tasks, before it will be overwritten. This implies the following relationship between two subsequent transports from any producing task T and the start times of the corresponding consumers (see also Fig. 8).

$$\forall s \in start(T) : \forall T' \in Succ(T) : \exists s' \in start^{\top}(T) : \\ s + tr(T, s, T') < s' \leq next(T, s) + tr(T, next(T, s), T')$$

Additionally to the above timing constraints, the response times, which are an important part of the specification of real-time systems, have to be checked. Since the task running and transport times are calculable as described above, the response times can be easily estimated.

The positive accomplishment of the above proof obligations guarantees the correctness of deployment. Thus, the resulting system will work correctly within the assumed environment.

7 Related Work

In this paper we presented a concept for separate verification of application logic and infrastructure. The necessity of the separation of functionality and infrastructure verification is also argued for by Sifakis et al. [27]. They introduce a formal modeling framework and a methodology, addressing the analysis of correct deployment and timing properties. The extention in our task concept is the explicit modeling of task dependencies and explicit statements about task activation conditions.

There also exist other approaches for the verification of distributed real-time software. J. Rushby in [28] has presented a framework for a systematic formal verification of time–triggered communication. His framework allows to prove a simulation relationship between an untimed synchronous system, consisting of a number of communicating components (“processors”) and its implementation based on a time-triggered communication system. His approach considers only a one-to-one relationship between components and physical devices they run on, i.e. no OS, and no sequentialization of component execution is taken into account. This approach is insufficient because it neglects the current praxis of automotive software development: OS, bus and application logic are developed by different suppliers and therefore should be treated separately.

There are also constructive approaches, trying to keep the properties of the models during deployment. Examples of such approaches are Giotto [29] and AutoMoDe [30]. While the AutoMoDe approach suggests a bottom-up procedure, which is based on the inter-arrival times of periodic signals, it is more appropriate for digital systems for measurement and control, where this information is present at the design stage of development.

The system behavior realized for AutoFOCUS components using AFTM is inspired by Henzinger’s Giotto approach. The Giotto tasks, realized in C, are also activated only if all the needed inputs are available. Their outputs are issued after the time of their worst case execution is elapsed. In order to provide such behavior, Giotto installs a low-level system driver, called *E-machine*, which takes over the role of input and output check during the run-time. For this setting the construction of schedules was proven to be polynomial in [31]. However, in contrast to the presented approach, the data and control flow, which serves as an input for schedule synthesis, are extracted in a rather *ad hoc* manner, e.g. it cannot be proven, that they correspond to the actual behavior of the C-code tasks. Furthermore, no additional middleware like the E-machine is needed in the presented work.

As noted in [32], in the CASE tools typically used for model-based software development in the automotive domain, like MATLAB/Simulink [1], Rose RT [2], Auto-

FOCUS [3], there is no explicit deployment concept. In other tools, like ASCET-SD [33] or Cierto VCC [34], there is an ability to build a deployment model for one ECU only. However these tools allow the modeling of the systems only on a very low level of abstraction. The application of such tools in the early design phases would lead to unnecessary over-specification.

8 Conclusion

The task-based application model is simple enough to be verified using automated verification techniques such as model checking and robust enough to be deployed without violation of the verified properties. The special task construction using message buffering and input checks assures that the task behavior remains the same even after deployment. Thus, the properties verified for the pure task-based system remain valid for the deployed system.

It is important that in the presented approach the operating system and the communication bus are verified separately from the application logic. The verification tasks interact in the assumption/guarantee way: for the application verification we assume a certain behavior of the infrastructure and for the infrastructure we verify that it guarantees the assumed properties. This reduces the complexity of verification and results in a completely verified system. The separation of functional and timing properties brings additional reduction of the verification effort.

The only piece missing to provide a pervasively verified time-triggered system is the verified infrastructure (OS/bus) providing the properties that the application layer (deployed AFTM) relies on. Such a verified infrastructure is being developed in the Verisoft project [19]. In the context of this project the methodology presented in this paper will also be applied to the emergency call application [6] to achieve a proof-of-concept pervasive verification of an automotive system.

References

1. The MathWorks. http://www.mathworks.com (18.05.2006)
2. IBM Rational Rose Technical Developer. http://www-306.ibm.com/software/awdtools/developer/technical/ (18.05.2006)
3. AutoFocus. http://autofocus.in.tum.de/ (18.05.2006)
4. FlexRay Consortium: FlexRay Communication System - Protocol Specification - Version 2.0. (2004)
5. OSEK/VDX: Time-Triggered Operating System - Specification 1.0. (18.05.2006)
6. European Commission (DG Enterprise and DG Information Society): eSafety forum: Summary report 2003. Technical report, eSafety (2003)
7. Kopetz, H., Grünsteidl, G.: TTP — a protocol for fault-tolerant real-time systems. Computer **27**(1) (1994) 14–23
8. VxWorks: A Realtime Operating System (RTOS). http://www.windriver.com (18.05.2006)
9. QNX: A Realtime Operating System (RTOS). http://www.qnx.com (18.05.2006)
10. OSEK/VDX. http://www.osek-vdx.org (18.05.2006)
11. TTP/C. http://www.vmars.tuwien.ac.at/projects/ttp/ttpc.html (18.05.2006)
12. TTCan: Time Triggered Communication on CAN. http://www.ttcan.com (18.05.2006)

13. FlexRay Consortium. http://www.flexray.com (18.05.2006)
14. Botaschanjan, J., Kof, L., Kühnel, C., Spichkova, M.: Towards Verified Automotive Software. In: ICSE, SEAS Workshop, St. Louis, Missouri, USA (2005)
15. OSEK/VDX: Fault-Tolerant Communication - Specification 1.0. http://portal.osek-vdx.org/files/pdf/specs/ftcom10.pdf (18.05.2006)
16. FlexRay Consortium: FlexRay Communication System - Bus Guardian Specification - Version 2.0. (2004)
17. FlexRay Consortium: FlexRay Communication System - Electrical Physical Layer Specification - Version 2.0. (2004)
18. Huber, F., Schätz, B., Einert, G.: Consistent Graphical Specification of Distributed Systems. In: Industrial Applications and Strengthened Foundations of Formal Methods (FME'97). Volume 1313 of LNCS., Springer Verlag (1997) 122–141
19. Verisoft Project. http://www.verisoft.de (18.05.2006)
20. Verisoft–Automotive Project. http://www4.in.tum.de/v̈erisoft/automotive (18.05.2006)
21. Leinenbach, D., Paul, W., Petrova, E.: Towards the Formal Verification of a C0 Compiler. In: 3rd International Conference on SEFM, Koblenz, Germany (2005) 2–12
22. The Motor Industry Software Reliability Association (MISRA): Guidelines for the Use of the C Language in Critical Systems. Motor Industry Research Association (MIRA), Ltd., UK. (18.05.2006)
23. AbsInt Angewandte Informatik GmbH. Worst-Case Execution Time Analyzers. http://www.absint.com/profile.htm (18.05.2006)
24. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: Tools and Algorithms for the Construction and Analysis of Systems. Volume 1384 of LNCS. (1998) 151–166
25. Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An open source tool for symbolic model checking. In: Proceedings of CAV 2002, Copenhagen, Denmark (2002) 359–364
26. Wimmel, G., Lötzbeyer, H., Pretschner, A., Slotosch, O.: Specification based test sequence generation with propositional logic. Journal of STVR: Special Issue on Specification Based Testing (2000) 229–248
27. Sifakis, J., Tripakis, S., Yovine, S.: Building models of real-time systems from application software. Proceedings of the IEEE **91**(1) (2003) 100–111
28. Rushby, J.: Systematic formal verification for fault-tolerant time-triggered algorithms. In: Dependable Computing for Critical Applications—6. Volume 11., IEEE Computer Society (1997) 203–222
29. Henzinger, T.A., Horowitz, B., Kirsch, C.M.: Giotto: A time-triggered language for embedded programming. Proceedings of the IEEE **91** (2003) 84–99
30. Bauer, A., Romberg, J.: Model-Based Deployment in Automotive Embedded Software: From a High-Level View to Low-Level Implementations. In: Proceedings of MOMPES, satelite of ACSD'04, Hamilton, Canada (2004) 93–106
31. Henzinger, T.A., Kirsch, C.M., Majumdar, R., Matic, S.: Time-safety checking for embedded programs. In: EMSOFT. (2002) 76–92
32. Braun, P., Broy, M., Cengarle, M.V., Philipps, J., Prenninger, W., Pretschner, A., Rappl, M., Sandner, R. In: The automotive CASE. Wiley (2003) 211 – 228
33. ASCET-SD. http://de.etasgroup.com/products/ascet (18.05.2006)
34. Cadence Cierto VCC. http://www.cadence.com (18.05.2006)