DOI 10.1007/s00165-008-0097-0
BCS © 2008
Formal Aspects of Computing (2008) 20: 637–662

On the correctness of upper layers of automotive systems[1]

Jewgenij Botaschanjan[1], Manfred Broy[1], Alexander Gruler[1], Alexander Harhurin[1]
Steffen Knapp[2], Leonid Kof[1], Wolfgang Paul[2], Maria Spichkova[1]

[1] Institut für Informatik, Technische Universität München, Boltzmannstr. 3, 85748, Garching bei München, Germany.
E-mail: botascha@in.tum.de; broy@in.tum.de; gruler@in.tum.de; harhurin@in.tum.de; kof@in.tum.de; spichkov@in.tum.de
[2] Department of Computer Science, Saarland University, 66123 Saarbrücken, Germany.
E-mail: sknapp@wjpserver.cs.uni-sb.de; wjp@wjpserver.cs.uni-sb.de

Abstract. Formal verification of software systems is a challenge that is particularly important in the area of safety-critical automotive systems. Here, approaches like direct code verification are far too complicated, unless the verification is restricted to small textbook examples. Furthermore, the verification of application logic is of limited use in industrial context, unless the underlying operating system and the hardware are verified, too. This paper introduces a generic model stack, allowing the verification of all system layers as well as the concrete application models being used in the upper layers. The presented models and proofs close the gap between the correctness proof for the lower layers of car electronics developed at the Saarland University and the verification procedure for distributed applications developed at the Technische Universität München.

Keywords: Formal verification; Automotive software; Model-based development; Time-triggered systems

1. Introduction and overview

In order to provide a pervasively verified system, one must verify the applications in the top system layer as well as proving that every layer is simulated by the underlying one. Given a verified model stack, for every particular application it is then sufficient to verify only the upper system layer (the CASE tool model) in order to obtain a completely verified system. This kind of layered verification is the objective of the German Verisoft project [Ver06]. More concretely, the mission of the project is (i) to develop tools and methods permitting the pervasive formal correctness proofs of entire computer systems including hardware, system software, communication systems, and applications, and (ii) to demonstrate these methods and tools with examples of industrial complexity.

The goal of the subproject Verisoft-Automotive is to verify an automatic emergency call system, *eCall* [Eur03]. Starting in 2009, such an *eCall* functionality will become mandatory in Europe. The verification is based on the model stack, shown in Fig. 1, which incorporates the idea of a layered framework for the verification of automotive systems presented in [BKKS05]. The development starts with an AutoFOCUS task model (AFTM), a design

Correspondence and offprint requests to: Leonid Kof, E-mail: kof@in.tum.de
[1] The work of J. Botaschanjan, A. Gruler, A. Harhurin, S. Knapp, L. Kof, and M. Spichkova was partially funded by the German Federal Ministry of Education and Technology (BMBF) in Verisoft project under grant 01 IS C38.

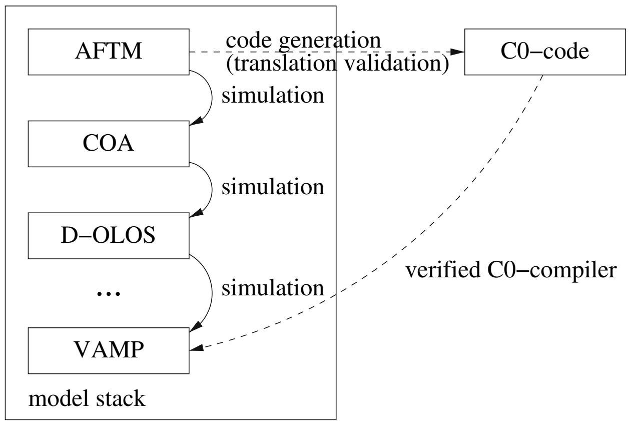


Fig. 1. Model stack

model in the CASE tool AutoFOCUS. Out of the AFTM we generate C0 code. The language C0 [LPP05] is a Pascal-like subset of C that is similar to MISRA C [Mot06]. In safety critical applications it is common to use a restricted version of C to ensure a less error-prone programming style. The generated C0 code is compiled and run on the VAMP processor [DHP05]. The intermediate models are necessary in order to verify that the AFTM and the runnable VAMP code are behaviorally equivalent under the assumption that the C0 code is equivalent to the AutoFOCUS tasks. This equivalence proof is non-trivial due to the fact that the models of computation of the AutoFOCUS tool (or, in general, of CASE tools) differ from real hardware.

Outline. The model stack, which is depicted in Fig. 1, includes the following elements: The hardware consists of Electronic Control Units (ECUs) connected via a FlexRay-like[2] bus [Con06, KS06, BBG+05]. The ECUs comprise a verified architecture micro processor (VAMP) [DHP05] and a FlexRay-like interface. The given system software is a verified C0 compiler [LPP05] and OLOS, an OSEKtime OS-like operating system [Kna05]. The system infrastructure is introduced in Sects. 2 and 3.

In [BBG+05, BKKS05, IdRLP05, KP06] a pervasive correctness proof for the lower layers of such systems has already been outlined starting from the gate level and reaching a communication model which is based on shared variables. In Sect. 4, we present a refinement of this communication model, called the distributed OLOS model (D-OLOS). In Sect. 5 the model of communicating automata (COA) is introduced, which bridges the gap between the model-based development and the CASE tool used for the application layer.

The key element of the transition from a CASE tool model to the real system code and real hardware is the mapping of modeled tasks to ECUs (deployment). Our deployment approach of the verified model onto a time-triggered architecture has already been sketched in [BGH+06]. Sections 6 and 7 continue this work and show how the application layer can be modeled in a CASE tool and how a schedule can be constructed to deploy the model. The resulting deployed model is behaviorally equivalent to the model in the CASE tool (proof: see Sect. 8). Although the proof of the theorem is simple, its consequences are far-reaching: We have a paper-and-pencil outline of a correctness proof for car electronics from the gate level up to the CASE tool models.

eCall example. To illustrate the concepts and notions introduced throughout the paper we will use the eCall case study as a running example. We model the eCall as a system consisting of three sub-systems, namely: A GPS navigation system, a mobile phone, and the actual emergency call application (cf. Fig. 2). External information (e.g. the crash sensor, the GPS signals, etc.) is considered to be part of the environment.

According to [Eur03], these components interact as follows: The navigation system sends periodically the vehicle's coordinates to the emergency call application. The crash sensor periodically sends 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

[2] The term 'like' refers to the fact that our implementation does not fully implement the corresponding standard. Differences are described in Sect. 2.

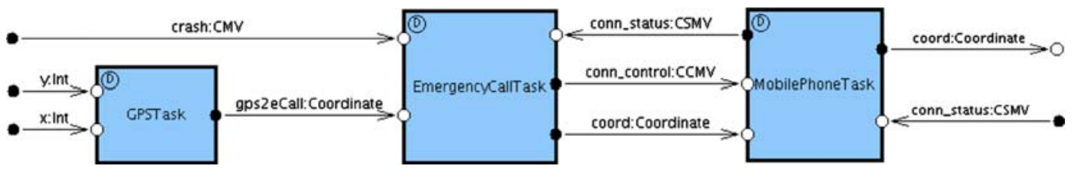


Fig. 2. The task architecture of the eCall case study (AutoFOCUS model, see also Sect. 6)

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 starting from the connection initiation step.

2. Deployment platform

To master the inherent complexity of automotive systems, industry came up with a number of standards based on the *time-triggered paradigm* [KG94]. These allow the realization of distributed systems with predictable time behavior, making them an appropriate deployment target for safety-critical real-time systems. In a time-triggered system, actions are executed at predefined points in time. Furthermore, time-triggered communication protocols use time synchronization as a global time base for the distributed communication partners. Thus by combining a time-triggered OS and a time-triggered network, deterministic system behavior with guaranteed response times can be achieved.

The target deployment platform of the presented work is a network of ECUs connected by a FlexRay-like bus with an OSEKtime OS-like operating system running on each node.

OSEKtime. OSEKtime OS [OSE01b] is an OSEK/VDX [OSE06] open operating system standard of the European automotive industry. The OSEKtime OS supports cyclic fixed-time scheduling and provides a fault tolerant communication mechanism.

An OSEKtime schedule defines when, within a so-called scheduling round, the dispatcher activates a user process. If another process is currently running at the scheduled activation time, it is preempted until the activated process has completed its computation. In addition OSEKtime monitors the deadlines of the processes, i.e. at predefined points within the scheduling round a process must have finished its computation, otherwise an error hook is executed. This imposes restrictions on the applications running under OSEKtime. The round-based scheduling procedure is repeated perpetually. All rounds have equal length and the scheduling tables for all rounds are the same.

FTCom [OSE01a] is the OSEKtime fault-tolerant communication layer that provides a number of primitives for interprocess communication. Messages kept in FTCom are uniquely identified by their IDs. For every message ID FTCom realizes a buffer of length one. Applications send or receive messages with certain IDs by invoking the communication primitives provided by FTCom.

For the scope of this paper we use the following simplifications compared to the OSEKtime standard:

- Every task is activated exactly once per scheduling round.
- Every task computation takes at most one scheduling slot, in particular we do not deal with preemption.
- The replication and replica determinate agreement (RDA [OSE01a]) mechanisms implemented in the FTCom are not used.

FlexRay. FlexRay [Con06] is a communication protocol for safety critical real-time automotive applications. It has been developed by the FlexRay Consortium [Fle06]. It is a static time division multiplexing network protocol that supports clock synchronization.

The static message transmission mode of FlexRay is based on FlexRay *rounds* consisting of a constant number of time slices of the same length, so called *slots*. A node can broadcast messages to other nodes within these slots. However, there can be at most one sender in a given slot.

A combination of the time-triggered OS and the time-triggered bus allows for synchronization of the computations and the communication. This is done by synchronizing the local ECU clock with the help of FlexRay and by setting the length of the OSEKtime scheduling round to be a multiple of the length of a FlexRay

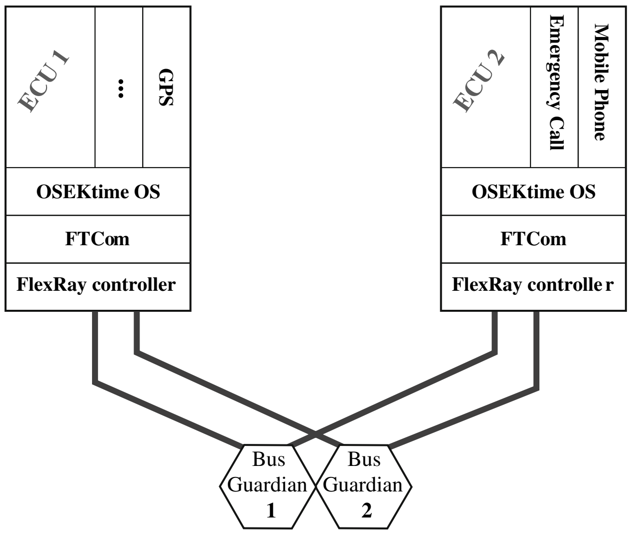


Fig. 3. An example distribution of the eCall tasks

round. A unit of computation is then also a FlexRay slot. The main aspects of FlexRay, OSEKtime OS and FTCom have been outlined in [KS06, KS07, dRK05], based on [BKKS05].

To reduce the complexity of the system, several aspects of FlexRay have been abstracted as well (see also [KS07]):

- We use only a simple clock synchronization- and FlexRay start-up algorithm.
- The model does not contain bus guardians [Con06] that protect channels on the physical layer from interference caused by communication that is not aligned with FlexRay schedules.
- Only the static segment of the communication cycle of FlexRay is used, as we are mainly interested in time-triggered systems.
- The system operates with one FlexRay channel only.
- Both the scheduling and the FlexRay slots have the same length.
- Both the scheduling and the FlexRay rounds have the same length.

These restrictions of our model preserve the idea of time-triggered systems and thus do not restrict the applicability of the presented ideas to real systems.

eCall example. Figure 3 shows a possible distribution of the eCall tasks on two ECUs connected by a double redundant FlexRay link.

3. Introduction to OLOS

The abbreviation OLOS is a shorthand for OSEKtime-like operating system. OLOS implements the core part of OSEKtime, namely cyclic time-triggered task scheduling and buffers for message exchange. In this section we will briefly summarize the semantics of OLOS [Kna05], which has been implemented as a dialect of the generic operating system kernel CVM [GHLP05]. Starting with the introduction to our notations in Sect. 3.1, we will sketch the scheduling mechanism and the communication behavior as well as the restrictions on the applications (Sect. 3.2).

3.1. Time-triggered system basics

Both FlexRay and OSEKtime use cyclic scheduling. This section introduces the mathematical notation necessary to formalize the cyclic schedules. For intervals of natural numbers we use the shorthand

$$[i : j] \equiv \{i, i + 1, \ldots, j\}$$

We consider p electronic control units $ECU(i)$, where $i \in [0 : p - 1]$. On each $ECU(i)$ there are n_i applications $A(i, j)$, where $j \in [0 : n_i - 1]$, running under the real-time operating system OLOS. These user programs are C0 programs.

Slots and rounds. Let ns be the number of slots in one round. Given a slot s we denote the slot following s by $s + 1$, which is a shorthand for $s + 1 \mod ns$. The slot prior to s is denoted by $s - 1$ which we define in an analogous way. If r is the round number, we can define the time unit as the pair (r, s). Then, the next time unit $(r, s) + 1$ is defined as follows:

$$(r, s) + 1 \equiv \begin{cases} (r, s + 1) & s < ns - 1 \\ (r + 1, 0) & \text{otherwise} \end{cases}$$

The previous slot $(r, s) - 1$ is defined in an analogous way.

Scheduling. The scheduling of all applications $A(i, j)$, as well as the inter ECU communication procedure via the FlexRay bus, is identical in each round r and only depends on the slot index s. The scheduling of all applications is defined by the global scheduling function $run(i, s)$. Note that according to the OSEKtime OS standard an 'idle' task is executed if no other application is specified in the scheduler. Since this idle task can be implemented as a normal application we do not treat it separately here. Function run returns the index $j \in [0 : n_i - 1]$ of the application being executed in the given slot s on $ECU(i)$:

$$run(i, s) \in [0 : n_i - 1]$$

Similarly, the global communication schedule defines which ECU is allowed to send a message in the given slot.

$$send(s) \in [0 : p - 1] \tag{1}$$

To complete the communication schedule, one must characterize which message is being sent. Such a function mapping slot numbers to message types is introduced below.

Communication. According to the FTCom specification [OSE01a], every ECU maintains a message buffer. The applications communicate solely via this buffer by writing messages to or reading messages from it. We formalize this buffer as an array storing the messages. Let nm be the number of message types to be sent in the distributed system. Then, the message buffer of the i-th ECU $MB(i)$ is an array with the index set $[0 : nm - 1]$. By $MB(i)(k)$ we denote the k-th element in the buffer $MB(i)$, where $k \in [0 : nm - 1]$.

The application scheduled on $ECU(i)$ can read and write $MB(i)(k)$ using two system-calls similar to those specified in [OSE01a]:

1. $ttSend(k, msg)$: The execution of this function on $ECU(i)$ results in copying the value of the C0 sub-variable with identifier msg into $MB(i)(k)$.
2. $ttRecv(k, msg)$: On invocation of this function on $ECU(i)$ the C0 sub-variable identified by msg is updated with the value of $MB(i)(k)$.

The semantics of both system-calls will be formalized in Sect. 4.4. A message that was sent by an application running on $ECU(i)$ is directly accessible for all applications running on the same ECU, whereas applications running on different ECUs can receive this message only after it has been broadcast via FlexRay.

Based on the MB definition, we can define the function mapping the communication slot onto the message to be sent. For a given slot s the message-type is defined by the global function $mtype$:

$$mtype(s) \in [0 : nm - 1]$$

Together with the definition of the function $send$ (Eq. 1) this results in a complete specification of the bus schedule. Intuitively, if $ECU(i)$ is the sender in a given slot s, i.e. if $send(s) = i$ holds, then $ECU(i)$ broadcasts $MB(i)(mtype(s))$. The message buffers of all the ECUs will be modified incrementally in each slot according to these two global schedules. Note that both the $mtype$ and the $send$ function are only partially defined: They are undefined for bus slots not used for communication.

3.2. Application structure and restrictions

Beside the two system-calls mentioned in Sect. 3.1 OLOS offers a third call named $ttExFinished()$. An application invoking this system-call indicates that it has completed its computation for the current slot and wants to return the control back to the operating system.

Cyclic task activation and deactivation is achieved by the following code structure: the application code is wrapped by a *while*-loop and *ttExFinished*() is invoked only once as the last statement of the loop body:

```
while(true) {
  "Application Code" ;
  ttExFinished() ;
}
```

Thus we enforce the application code to be executed once each time the application is scheduled (see Sect. 4.1). Intuitively, from an application programmer's point of view, the *ttExFinished*() system-call does nothing except for waiting until the application is scheduled again. The eCall example, put into such code structure, could look like this:

```
while(true) {
  if(crash)
    do something
  else
    do nothing
  ttExFinished() ;
}
```

We assume that all our applications comply with the given slot boundaries and thus run from one execution of *ttExFinished* to the next. Although this seems to be a hard assumption, it actually meets reality. Calculating the worst case execution time (WCET) of applications [Abs06], optimizing the application code for short WCET, and then choosing the proper length for the scheduling intervals is a necessary practice in the domain of real-time system programming.

4. The D-OLOS model

OLOS is appropriate to model a single ECU. In order to model a network of communicating ECUs, we introduce *distributed* OLOS, or D-OLOS. D-OLOS consists of an ECU set and a communication bus. We introduce the computational model of D-OLOS in two steps:

1. In Sect. 4.1, we introduce the configuration of a single application running under OLOS. In Sect. 4.2 we define the transition function on this configuration.
2. In a similar way Sects. 4.3 and 4.4 introduce global (distributed) configurations and global computations, respectively.

In a nutshell, the global configuration combines the configuration of all applications running on the same ECU. Furthermore it combines the configuration of all ECUs themselves.

4.1. Abstract C0 machine

The applications running under OLOS are C0 programs. C0 [LPP05] is a subset of the language C in which many error prone features like pointer arithmetic have been removed. The abstract model for the execution of a C0 program is called a C0 machine. To describe the current configuration c of a C0 machine, we refer to the rest of the program by $c.pr$. It denotes the list of statements that have not yet been executed. Further on, for reading or updating C0 variables x, we will use a value function $va(c, x)$ which denotes the current value of a C0 variable having the given identifier x in the given C0 machine configuration c. The detailed definition of this function as well as the complete configuration of a C0 machine is given in [LPP05].

A transition in a C0 machine corresponds to the execution of a C0 statement. The transition function δ_C executes the first statement in the rest of the program $c.pr$ according to its semantics given in [LPP05].

While statement. As an example we define the execution of a *while* statement. We denote the first statement of a statement list sl by $head(sl)$ and the remaining statements by $tail(sl)$.

Assume the next thing to be executed (the current rest of the program of configuration c) is a *while* loop, i.e. has the form $head(c.pr) = while(cond)\{a\}$ where a is a statement list. If the condition $cond$ is satisfied in the configuration c, i.e. $va(c, cond) = true$, then the rest of the program of the new configuration c' is:

$$c'.pr = \begin{cases} a;\ \ c.pr & \text{if } va(c, cond) = true \\ tail(c.pr) & \text{otherwise} \end{cases}$$

Thus, applications being wrapped by a *while*(*true*) loop, as assumed for our applications (compare Sect. 3.2), are executed time and again.

4.2. Local configurations and transitions

To define the semantics of a local computation, it is necessary to define the semantics of system-calls in addition to the usual programming language semantics. Furthermore, to match the "run-to-completion" semantics of tasks modeled in a CASE tool (introduced in Sect. 6) it is necessary to define the "run-to-completion" for OLOS tasks as well.

The definition of execution semantics is based on the local configuration lc, consisting of a C0 machine configuration and a message buffer, i.e. $lc = (lc.c, lc.MB)$. Informally, this represents an application process with an inter-process shared memory. A local computation is defined by the local transition function δ_{LC} which returns the successor configuration:

$$lc' = \delta_{LC}(lc)$$

The tuple lc' is defined on top of δ_C, which is the transition function representing the C0 semantics, as follows (unmentioned components remain unchanged):

- If the C0 machine $lc.c$ does not invoke an operating system-call, then an ordinary C0 transition is performed:

  $$lc'.c = \delta_C(lc.c)$$

- If the first statement of the rest of the program is a $ttRecv$ system-call, i.e. $head(lc.c.pr) = ttRecv(k, msg)$, the value of the C0 sub-variable having the identifier *msg* is updated with the content of the k-th element of the message buffer. Furthermore the system-call is deleted from the rest of the program:

  $$\begin{aligned} va(lc'.c, msg) &= lc.MB(va(lc.c, k)) \\ lc'.c.pr &= tail(lc.c.pr) \end{aligned}$$

- If the first statement of the rest of the program is a $ttSend$ system-call, i.e. $head(lc.c.pr) = ttSend(k, msg)$, the system-call itself is deleted from the rest of the program and the k-th element of the message buffer is updated with the value of the variable bound to identifier *msg*:

  $$\begin{aligned} lc'.c.pr &= tail(lc.c.pr) \\ lc'.MB(va(lc.c, k)) &= va(lc.c, msg) \end{aligned}$$

- If the application is done, i.e. the *ttExFinished* system-call is to be executed, i.e. $head(lc.c.pr) = ttExFinished()$, the application stalls:

  $$lc' = lc$$

A finite local computation is a sequence of configurations $lc^0, lc^1, \dots$ obeying $lc^{t+1} = \delta_{LC}(lc^t)$. Let $K(lc) > 0$ be the smallest step number K such that $\delta_{LC}^K(lc)$ has finished execution, i.e.

$$K(lc) = \min\{K \mid head(\delta_{LC}^K(lc).c.pr) = ttExFinished()\}$$

We define the result of the local computation $res_{LC}(lc)$ to be the local configuration after the K-th local transition:

$$res_{LC}(lc) \equiv \delta_{LC}^{K(lc)}(lc)$$

This way we abstract from the local application steps in the computation. Note that during a local computation the slot index and the current bus-value do not change.

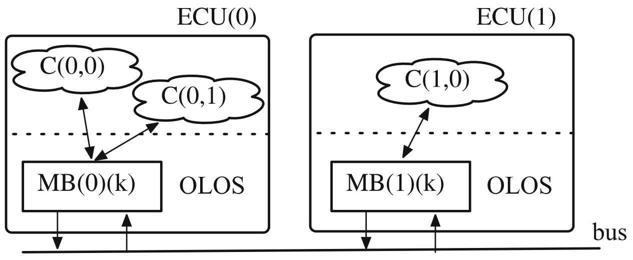


Fig. 4. Example D-OLOS configuration

4.3. D-OLOS configuration

D-OLOS consists of an ECU set and a communication bus. Analogously, a D-OLOS configuration consists of a set of local configurations and the current bus state (cf. Fig. 4). A D-OLOS configuration *dolos* is a tuple having the following components:

- $dolos.C(i, j)$ is the configuration (local state) of an abstract C0 machine representing application $A(i, j)$ for a process-number $i \in [0 : p - 1]$ and an ECU number $j \in [0 : n_i - 1]$.
- $dolos.MB(i)(k)$ is the k-th message in the message buffer of $ECU(i)$, where $k \in [0 : nm - 1]$.
- $dolos.s$ is the index of the current slot.
- $dolos.bus$ holds the message value of the message type currently being broadcasted.

The D-OLOS configuration *dolos* represents the global state of the system. An example scenario with three applications running on two ECUs and reading, or writing the kth entry on their corresponding FTCom buffers is depicted in Fig. 4. (This could be a possible deployment of the eCall example shown in Fig. 3.)

To define the global transition function, we need two predicates. The first predicate specifies which tasks are running and the second determines if all the tasks have finished their local computation. We define the predicate $running(dolos, i, j)$ to be true if application $A(i, j)$ is scheduled during slot $dolos.s$:

$$running(dolos, i, j) \equiv (j = run(i, dolos.s))$$

Let $done(dolos, i, j)$ be the predicate indicating that application $dolos.C(i, j)$ has reached the *ttExFinished*() statement. The predicate $newslot(dolos)$ indicates that all scheduled processes have finished their computation:

$$newslot(dolos) \equiv (\forall i, j : running(dolos, i, j) \Rightarrow done(dolos, i, j))$$

These predicates will be used in the following section to define the global transition function.

4.4. D-OLOS transition

The D-OLOS transition function δ_{DOLOS} takes a D-OLOS configuration *dolos* and returns its successor $dolos'$, i.e. $dolos' = \delta_{DOLOS}(dolos)$. In the definition of δ_{DOLOS} we split cases according to the *newslot* predicate. If $newslot(dolos)$ does not hold, the head of the rest of the program is executed in the scheduled application on each ECU if the application is not completed. If $newslot(dolos)$ holds, the inter-ECU communication procedure is performed.

In the following definitions we only mention configuration components that are updated; all other configuration components remain unchanged. We start with the definition of the first case, i.e. we assume $newslot(dolos)$ does not hold:

- Configurations of non-running applications, i.e. applications for which the $running(dolos, i, j)$ predicate does not hold, are not altered:

  $$\neg running(dolos, i, j) \Rightarrow dolos'.C(i, j) = dolos.C(i, j)$$

- In case of running applications, a local transition is performed:

  $$running(dolos, i, j) \Rightarrow (dolos'.C(i, j), dolos'.MB(i)) = \delta_{LC}(dolos.C(i, j), dolos.MB(i))$$

In the second case (assumed that *newslot*(*dolos*) holds) the slot index is updated, the message buffers are synchronized and the running applications are reset:

- The slot index is incremented:

  $$dolos'.s = dolos.s + 1$$

- The global communication is performed, i.e. the message buffer of every ECU is updated with the current content of *dolos.bus*. The value of *dolos.bus* itself is updated with the value of the message-type to be broadcast in the next slot:

  $$\forall i : dolos'.MB(i)(mtype(dolos.s)) = dolos.bus$$
  $$dolos'.bus = dolos.MB(send(dolos'.s))(mtype(dolos'.s))$$

- Finally the *ttExFinished* system-calls are removed from the rest of the program of the scheduled applications:

  $$dolos'.C(i, run(i, dolos.s)).pr = tail(dolos.C(i, run(i, dolos.s)).pr)$$

  This initializes the applications such that they can be scheduled again.

5. Model of communicating automata

The model of communicating OLOS automata (COA) bridges the gap between the D-OLOS model, which was defined using C0 transitions, and the automaton-based CASE tool models, which are introduced in Sect. 6. While the D-OLOS model uses a small-step synchronization approach based on the *newslot* predicate, the COA transition function combines the small steps into a single slot-step transition and thus abstracts from the local steps.

As in the previous section, we start with the definition of the system configuration and the transition function (Sects. 5.1 and 5.2, respectively). Then, in Sect. 5.3 we show how the COA model can be simulated by the D-OLOS model. This lays the basis for the model-based development in the following sections.

5.1. COA configuration

COA, as compared to D-OLOS, introduces a new transition function but no new configuration concepts. Thus, a COA configuration *coa* is a tuple in which all components are taken literally from the D-OLOS configuration *dolos*:

- $coa.C(i, j)$ is the configuration of the applications.
- $coa.MB(i)(k)$ stores the message values that are visible to $ECU(i)$.
- $coa.s$ is the current slot index.
- $coa.bus$ is the message value of the message type currently being broadcast.

The fundamental difference of COA from D-OLOS lies in its state transition function. This state transition function is introduced below.

5.2. COA transition

A COA transition corresponds to the sequence of the D-OLOS transitions that are executed during one slot. The COA transition function δ_{COA} is defined as follows: All runnable applications are executed until they invoke the *ttExFinished*() system-call, the others remain unchanged. Formally,

$$\forall i : (APP'(i), MB'(i)) = res_{LC}(coa.C(i, run(i, coa.s)), coa.MB(i))$$

The application automata are updated with the result of the local computation, so is the *coa.bus* component. Note that the *ttExFinished* system-call needs to be deleted from the rest of the program, so that the application is not stalled any longer.

The slot number is incremented and the message buffers are either updated with the message value which is currently being broadcast (if $k = mtype(coa.s)$) or with the result of the local computation:

$$\begin{aligned}
coa'.C(i, run(i, coa.s)).pr &= tail(APP'(i).pr) \\
coa'.C(i, run(i, coa.s)).x &= APP'(i).x \qquad \forall\, x \neq pr \\
coa'.s &= coa.s + 1 \\
coa'.bus &= MB'(send(coa'.s))(mtype(coa'.s)) \\
coa'.MB(i)(k) &= \begin{cases} coa.bus & k = mtype(coa.s) \\ MB'(i)(k) & \text{otherwise} \end{cases}
\end{aligned}$$

5.3. Simulation: D-OLOS versus COA

The transition functions of D-OLOS and COA are different. However, the execution of a sequence of D-OLOS transitions fitting in one slot and one COA transition will result in equivalent states. To properly verify behavioral equivalence between D-OLOS and COA, we need a simulation relation. The simulation relation between the D-OLOS and the COA model is straightforward. We define a simulation relation $dolos \cong coa$ between a D-OLOS configuration $dolos$ and a COA configuration coa by requiring for all i, j:

$$\begin{aligned}
dolos.C(i, j) &= coa.C(i, j) \\
dolos.MB(i) &= coa.MB(i) \\
dolos.s &= coa.s \\
dolos.bus &= coa.bus
\end{aligned}$$

The simulation theorem between the D-OLOS and the COA model reads as follows.

Theorem 1 (Simulation: D-OLOS vs. COA) Let u, k be D-OLOS and COA step-numbers such that the simulation relation holds, i.e. $dolos^u \cong coa^k$. Let $v > u$ be the smallest step-number such that the *newslot* predicate holds again, i.e. $v = min\{t \mid (t > u) \wedge newslot(\delta_{DOLOS}^{(t-u)}(dolos^u))\}$. Then, the simulation relation holds in each new slot after the global communication is performed:

$$dolos^u \cong coa^k \Rightarrow dolos^{v+1} \cong \delta_{COA}(coa^k)$$

Proof The proof is straightforward as the COA only abstracts from the local application steps. □

6. AutoFOCUS task model

The models introduced so far—D-OLOS and COA—both aim at formalizing the behavior of the lower system levels while having a focus on correctness proofs of system properties throughout these related models. Although laying the indispensable basis for the realization of a pervasive verification, neither D-OLOS nor COA are particularly well suited to capture the functionality of a distributed reactive system from a higher, more abstract, point of view, i.e. that in which a software designer is interested during a model-based development process of a system. In this section we describe the *AutoFOCUS Task Model* (AFTM) which seamlessly integrates on top of the model stack introduced so far (see also Fig. 1), while offering the possibility to abstract further from low level system details and to focus on modeling the pure functionality of a system. Here, with “integration into the model stack” we primarily mean that certain properties shown for the AutoFOCUS Task Model also provably hold in COA, and consequently in D-OLOS, as well.

As we will see in Sect. 6.1, the AFTM is based on the AutoFOCUS semantics, which itself is directly implemented by the AutoFOCUS CASE tool. In contrast to the AutoFOCUS semantics—and also to current CASE tools typically used for automotive software development, e.g. MATLAB/Simulink [Mat06] or Rose RT [IBM06]—the AFTM provides an explicit deployment concept [BGH+06]. This deployment concept guarantees the preservation of system properties. Such a preservation is essential since without deployment support it makes no sense to verify properties on the application model (AFTM), as such properties do not necessarily hold after deployment.

Thus, we now have a complete model stack which allows us to model a system at a high level (AFTM, supported by the respective AutoFOCUS CASE Tool) while being able to prove system properties throughout the related models and thus to (provably) guarantee the correctness of the entire system. Deployed on a verified hardware platform, this results in a pervasively verified system.

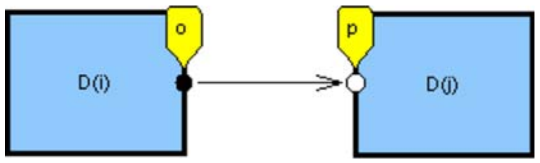


Fig. 5. Channel between the ports of connected components

6.1. The AutoFOCUS CASE tool

An AutoFOCUS task model (AFTM) is defined using AutoFOCUS [Aut06], which is a CASE tool for modeling and simulating reactive, distributed systems. It models the system as a finite network D of *components* $D(i)$, where $D = \{D(i) | i \in [1 : n]\}$ for a constant $n \in \mathbb{N}$. To simplify matters and to keep the notation short, we denote a component $D(i)$ in the following from time to time only by its index i, i.e. where actually an element of the set D is expected we simply use its index i of type $\mathbb{N}$.

The communication between components is accomplished by typed *channels* which connect components through *ports*. A channel can either connect a pair of components with each other or a single component to the environment. More precisely, for the set of components D there exists a relation *flow* of directed channels between them:

$$\mathit{flow} \subseteq D \times D$$

We restrict this relation by $\forall d \in D : (d, d) \notin \mathit{flow}$, i.e. no self-loops are allowed.

For a single component $D(i)$, we denote by $IP(i)$ ($OP(i)$) the set of all its input (output) ports. Further on, we denote the k-th input port of the component $D(i)$ by $IP(i, k)$, where $k \in [1 : P_i^{in}]$ for a constant $P_i^{in} = |IP(i)|$. A single output port $OP(i, k)$, where $k \in [1 : P_i^{out}]$ is defined similarly. In particular, ports (and the respective channels) are the only way for a component to communicate with its environment.

In order to model a concrete channel between the ports of connected components, we define the function *src* which maps an input port of a component to the corresponding output port of a connected component:

$$src(j, p) = (i, o) \tag{2}$$

This term means that for the given input port p of the component $D(j)$, the corresponding output port of the component $D(i)$ is o (cf. Fig. 5).

A single component $D(i)$ can either be hierarchically refined to a further component network or directly implemented by a non-deterministic, finite I/O automaton $\mathcal{A}_i$, which is defined by a set of states S, an initial state $s_0 \in S$, a set of local variables X, and a transition relation δ_i. Hereby, a state s not only reflects the current control state of the automaton but also the data state in terms of values of the local variables. Given a state s and an input message in_mes, the result of a single transition δ_i of the automaton $\mathcal{A}_i$ is a pair (s', out_mes) consisting of a state and a set of output messages, where

$$(s', out\_mes) \in \delta_i(s, in\_mes)$$

More precisely, on activation of the component $D(i)$ the attached automaton $\mathcal{A}_i$ can only execute a transition step in state s if the input ports have received the necessary input messages and the transition precondition is satisfied. If so, the transition can fire, which means that it writes data to the respective output ports, updates the local variables X, and puts the automaton into the state s'. A respective (transition) postcondition directly reflects these assignments. If in a certain state no transition can fire at all, an implicit *idle-loop* is executed, which means that the automaton simply remains in the current state s and produces no output. For more details see Sect. 6.3.1.

Once all automata are defined, AutoFOCUS can simulate the modeled system. The execution model for the automata is time-synchronous and message-asynchronous with buffer length one, driven by a global clock which divides time by so-called *ticks* into equal slices. Every tick starts a new time-slice, which consists of two phases: During the first phase every component $D(i)$ reads the values on its input ports and computes new values for its local variables and output ports, i.e. it performs a single state transition of its automaton $\mathcal{A}_i$. Then, during the second phase, the computed values are transmitted to the respective input ports of the connected components where they can be accessed at the following tick. Then the procedure is repeated. Thus, every component is activated (for computation) exactly once in each AutoFOCUS simulation step. If for one activation several transitions can be fired, a single one is picked non-deterministically for every simulation step. Furthermore, if an activated component is refined to a component network (and not directly implemented by an automaton), a component activation means that each sub-component of the refining network is activated exactly once. The above sketch of the main principles and features of AutoFOCUS is rather short, but it is sufficient for our purposes. Huber et al. give a more in-depth introduction to this CASE tool in [HSE97].

In addition to modeling and simulation, it is also possible to prove temporal-logic properties for AutoFOCUS models. There exists an SMV back-end [WLPS00] for this purpose.

eCall example. The following properties (formulated in linear temporal logic) were proved for the eCall system using the SMV model checker:

- The system remains in its initial state until it receives the “crash” signal.
- If the system receives the crash signal and there exists a period of time during which the radio link to the emergency center does not break down, the emergency call is performed correctly.

In order to preserve the LTL properties proved for the modeled system it is necessary to show that the deployment platform simulates the AutoFOCUS execution semantics [CGP99]. This simulation proof is given in Sect. 8.

6.2. AutoFOCUS tasks

As an acceptable abstraction level of model-based development we suggest modeling the functionality of the system at the level of communicating tasks using the AFTM which is described in this section. In [BGH+06] it is shown how the intended behavior of such AutoFOCUS tasks can be implemented using AutoFOCUS components, i.e. how a Task Model can be simulated in the AutoFOCUS semantics as sketched in the previous section.

The AFTM allows one to model a system as a set T of independent, communicating tasks $T(i)$, where

$$T \equiv \{T(i) \mid i \in [1 : M+1]\}$$

for a constant $M \in \mathbb{N}$. The tasks $T(i),\ 1 \leq i \leq M$, are considered to be regular systems tasks, whereas the task $T(M+1)$ is special as it is used to represent the environment of the system.

An AutoFOCUS task $T(i),\ 1 \leq i \leq M+1$, is itself composed of a network of AutoFOCUS components. In particular, this means that a task *is* an AutoFOCUS component. Thus, all concepts (e.g. ports, channels, etc.) defined for components are likewise applicable for tasks.

Tasks are divided into three disjoint sets: *AND*-tasks (T_{AND}), *OR*-tasks (T_{OR}) and the environment task $T(M+1)$.

$$T \equiv T_{AND} \uplus T_{OR} \uplus \{T(M+1)\}$$

The environment task $T(M+1)$ is assumed to be always running. An *OR*-task $T_{OR}(j)$ can be executed when at least one input from any preceding task $T(i)$ has arrived, where $(i, j) \in \mathit{flow}$. An *AND*-task can be executed only when all the inputs from every preceding task are available. The corresponding formal definition is captured by the predicate *runnable*:

$$\begin{aligned}\forall i \in [1 : M+1] :\ & \mathit{runnable}(T(i), \mathit{IP}(i)) \equiv \\ & i = (M+1)\ \vee \\ & (T(i) \in T_{OR} \wedge \exists j \in [0 : P_i^{in}] : \mathit{IP}(i,j) \neq \epsilon) \vee \\ & (T(i) \in T_{AND} \wedge \forall j \in [0 : P_i^{in}] : \mathit{IP}(i,j) \neq \epsilon)\end{aligned} \tag{3}$$

The absence of a value on a particular port is denoted by the special value ϵ. According to the AFTM semantics the task $T(i)$ can start the computation of the inputs $\mathit{IP}(i)$ only if the predicate $\mathit{runnable}(T(i), \mathit{IP}(i))$ is true (see below).

eCall example. Functionality of the eCall application was modeled in AutoFOCUS and bundled in three tasks: GPS, mobile phone, and the emergency call task (cf. Fig. 2, see also [BGH+06] for details). In the case study the GPSTask is an AND-task, which gets activated when both coordinates arrive. The remaining two tasks are OR-tasks: they must respond independently to any of their respective inputs.

Configuration. The configuration of an AFTM *aftm* consists of the following components:

- $aftm.T(i)$: The state (configuration) of a single task, where $aftm.T(i).st$ denotes the value of the control state of this task.
- $aftm.IP(i, j)$: The current value of the input port $IP(i, j)$.
- $aftm.OP(i, j)$: The current value of the output port $OP(i, j)$.
- $aftm.R$: The set of tasks (indicated by their indices) which are currently running.

For the sake of simplicity the notation above using (round) brackets is overloaded, since at one stage it is used to denote the *name* of an object (task) while in the other case the notation refers to the *value* of an object (message). However, since the respective meaning always becomes clear from the context, we will accept this inaccuracy.

Initial configuration. In the initial configuration $aftm^0$ all port variables are empty, i.e. they keep no messages. All tasks are in their initial states and $aftm^0.R = \{M + 1\}$, i.e. initially, the only running task is the environment.

6.3. AFTM execution semantics

AFTM bridges the gap between the deployment platform (COA) and the CASE tool AutoFOCUS. For this reason there are two views on the AFTM execution:

- macro-step, corresponding to the COA step.
- micro-step, corresponding to one simulation step of the AutoFOCUS tool.

To define the semantics of the whole AFTM system, it is necessary to consider the composition of single tasks as well as the communication between them. This results in the necessity of considering two views of the system: local (semantics of a single task) and global (semantics of the task communication). Together with micro- and macro-steps this yields four possible views of the AFTM semantics. These four views are introduced below.

6.3.1. Auto FOCUS task automaton: micro-steps (Ticks)

The micro-step of the task automaton is *the* basis for all the other definitions, because this micro-step directly reflects the execution semantics of the AutoFOCUS CASE tool. As the AFTM is implemented in the AutoFOCUS CASE tool the AutoFOCUS execution semantics is the basis of the AFTM execution semantics.

A single micro-step of an AFTM task is defined on the basis of a state transition in one AutoFOCUS tick. The transition function of a task $T(i)$ is realized over an AutoFOCUS I/O automaton with transition function δ_{AFA}. Here, δ_{AFA} is a function which simulates a single transition step, i.e. which returns exactly one successor state according to δ_i. If several successors exist (non-determinism), δ_{AFA} chooses one of them in a non-deterministic fashion. Since we consider the preservation of LTL properties (which must be true for all runs), such a way of dealing with non-determinism does not impose any restrictions. Formally,

$$\delta_{AFA}(T(i), IP(i)) = (T'(i), OP'(i)) \Rightarrow (T'(i).st, OP'(i)) \in \delta_i(T(i).st, IP(i))$$

For every task $T(i)$ the control states $S(i)$ of its automaton are disjointly divided by the predicate

$$idle : S \to \mathbb{B}$$

into *idle* and *non-idle* states. The concept of an idle state is necessary to synchronize AFTM tasks at predefined points. Such interrupt points are essential to deploy the tasks onto a platform like OLOS, where tasks are periodically activated and must terminate after a certain time. Depending on the type of the current control state $T(i).st \in S(i)$ and on the presence of inputs $IP(i)$, different behaviors are exhibited by the transition function δ_T, which is defined on top of δ_{AFA}.

To define δ_T it is necessary to introduce two abbreviations: With $E_{in}(i)$ ($E_{out}(i)$), we denote the tuple of empty inputs (outputs) for the task $T(i)$, i.e.

$$E_{in}(i) \equiv \underbrace{(\epsilon, \ldots, \epsilon)}_{P_i^{in} \text{ elements}} \qquad E_{out}(i) \equiv \underbrace{(\epsilon, \ldots, \epsilon)}_{P_i^{out} \text{ elements}}$$

We also need a conditional update operation $\oplus_\epsilon$, similar to the "?"–operation in the programming language C:

$$var' = var \oplus_\epsilon default \equiv var' = \begin{cases} var & var \neq \epsilon \\ default & \text{otherwise} \end{cases}$$

This operation can also easily be defined for tuples of variables:

$$\begin{aligned}(var'_1, \ldots, var'_n) &= (var_1, \ldots, var_n) \oplus_\epsilon (default_1, \ldots, default_n) \\ &\equiv \forall i \in [1:n] : var'_i = var_i \oplus_\epsilon default_i \end{aligned} \tag{4}$$

The function δ_T with the signature[3]

$$\delta_T(T(i), IP(i), OP(i)) \equiv (T'(i), OP'(i))$$

is then defined as follows:

1. In an idle state a transition step can only be performed if the task is *runnable*:

$$runnable(T(i), IP(i)) \wedge idle(T(i).st) \Rightarrow \begin{cases} T'(i) = \delta_{AFA}(T(i), IP(i)).T'(i) \\ OP'(i) = \delta_{AFA}(T(i), IP(i)).OP'(i) \oplus_\epsilon OP(i) \end{cases}$$

2. In a non-idle state all inputs are ignored:

$$\neg\, idle(T(i).st) \Rightarrow \begin{cases} T'(i) = \delta_{AFA}(T(i), E_{in}(i)).T'(i) \\ OP'(i) = \delta_{AFA}(T(i), E_{in}(i)).OP'(i) \oplus_\epsilon OP(i) \end{cases}$$

3. If neither of the cases apply, i.e. the task is idle but has not received required inputs, nothing happens:

$$\neg\, runnable(T(i), IP(i)) \wedge idle(T(i).st) \Rightarrow (T'(i), OP'(i)) = (T(i), OP(i))$$

For the environment task we assume that all its states are idle, i.e. it can react to the outputs of the system immediately.

6.3.2. *AutoFOCUS task automaton: macro-steps (Slots)*

A macro-step of an AFTM task automaton is a transition from one idle state to the next and conforms to a finite D-OLOS computation, i.e. to one step of a COA-Automaton. Each macro-step consists of a finite number of micro-steps. For the proper definition of the macro-step, it is necessary to introduce an additional abbreviation. We denote the result of $\delta_T(T(i), E_{in}(i), OP(i))$ by $\delta_{E_i}(T(i), OP(i))$, i.e.

$$\delta_{E_i}(T(i), OP(i)) \equiv \delta_T(T(i), E_{in}(i), OP(i))$$

Let $idle(T(i).st)$ hold and let $IP(i)$ be the set of its inputs. For each computation of the task automaton starting in an idle state, we require that it reaches another idle state after a finite number of transitions:

$$\forall i \in [1:M+1] : idle(T(i).st) \Rightarrow \exists n : idle(\delta_{E_i}^n(\delta_T(T(i), IP(i), E_{out}(i))).T'(i).st) \tag{5}$$

Let n_{macro} be the minimum n satisfying the above property for a given task configuration and set of inputs plus one. Thus, n_{macro} is the number of transition steps made from one idle state to the next with only non-idle states in between. Please note that if the task is not runnable, n_{macro} is equal to one. A macro-step, Δ_T, is defined as the result of n_{macro} micro-steps:

$$\Delta_T(T(i), IP(i)) \equiv \delta_{E_i}^{n_{macro}-1}(\delta_T(T(i), IP(i), E_{out}(i))) \tag{6}$$

This definition will be used in Sect. 6.3.4 to define the global AFTM execution step.

[3] Please note, that δ_T, and also δ_{AFA}, return a record. Thus, we will access its elements by their names, i.e. by $\delta_T.T'(i)$ and $\delta_T.OP'(i)$.

6.3.3. AutoFOCUS task model: micro step (*Ticks*)

The AFTM transition function δ_{AFTM} computes the successor of a given AFTM configuration, i.e. $\delta_{AFTM}(aftm) = aftm''$. A step of the system consists of a *micro-computation* and a *micro-communication* phase:

$$\delta_{AFTM} \equiv \delta_{comm} \circ \delta_{comp}$$

The original configuration is processed by the computation transition function δ_{comp} first: $\delta_{comp}(aftm) = aftm'$. Then the modified configuration serves as an input for the communication transition function δ_{comm}, i.e. $\delta_{comm}(aftm') = aftm''$. Both functions are described below.

We will use the following abbreviation of the *runnable* predicate introduced in Eq. 3:

$$runnable(aftm)(i) \equiv runnable(aftm.T(i), aftm.IP(i))$$

To keep the following definitions as short as possible, we list only modified parts of the configuration. The parts that are not mentioned explicitly remain unchanged.

1. *Micro-computation phase*. All AFTM tasks run in a lock-step mode, which means that all runnable tasks are started at the same micro step and run to completion. The lock-step execution is started if solely the environment is running. In this case we compute the set $newR$ depending on the *runnable*-predicate.

$$newR = \begin{cases} \{i \in [1 : M+1] \mid runnable(aftm)(i)\} & aftm.R = \{M+1\} \\ aftm.R & \text{otherwise} \end{cases}$$

   No other task is allowed to start during this phase, runnable tasks can run to completion.
   All tasks contained in the set $newR$ perform exactly one step with the values of the corresponding ports as arguments:

$$\forall i \in [1 : M+1] : i \in newR \\ \Rightarrow \begin{cases} aftm'.T(i) = \delta_T(aftm.T(i), aftm.IP(i), aftm.OP(i)).T'(i) \\ aftm'.OP(i) = \delta_T(aftm.T(i), aftm.IP(i), aftm.OP(i)).OP'(i) \end{cases}$$

   Then, for all tasks which are started in this step the input ports are flushed (the input messages are consumed):

$$\forall i \in newR \backslash aftm.R : aftm'.IP(i) = E_{in}(i)$$

   Finally, the set of running tasks is adjusted by removing the terminated ones:

$$aftm'.R = newR \backslash \{i \mid i \neq M+1 \wedge idle(aftm'.T(i).st)\}$$

   Note: (1) According to the AutoFOCUS task micro-step semantics, as defined in Sect. 6.3.1, the input port values are processed by δ_T only at the beginning of a macro step and ignored otherwise. This corresponds to the macro-step semantics of AutoFOCUS tasks defined in Sect. 6.3.2.

   (2) In the presented work we treat the concrete behavior of a task (represented by δ_{AFA}) in a black-box manner. E.g. if the task cannot consume all inputs at once at the beginning of its execution, they could be stored in local variables. Such architectural solutions were discussed in [BGH+06] and are beyond the scope of the present paper.

2. *Micro-communication phase*. All finished tasks transmit their produced values and flush their output ports.

$$\forall i, j \in [1 : M+1],\ o \in [1 : P_i^{out}],\ p \in [1 : P_j^{in}] : src(j, p) = (i, o) \wedge idle(aftm'.T(i).st) \\ \Rightarrow \begin{cases} aftm''.IP(j, p) = aftm'.OP(i, o) \oplus_\epsilon aftm'.IP(j, p) \\ aftm''.OP(i) = E_{out}(i) \end{cases}$$

6.3.4. AutoFOCUS task model: macro step (*Slots*)

The macro-step of AFTM is defined as the simultaneous execution of a macro-step of every runnable task. For a concrete configuration *aftm* of AFTM, let $n_{macro}^{(i)}(aftm)$ be the number of micro-steps in the macro-step to be performed by the task $T(i)$ (see also Sect. 6.3.2) and let $N_{macro}(aftm)$ be the maximal length of the macro-steps:

$$N_{macro}(aftm) \equiv \max_{i \in [1:M+1]} (n_{macro}^{(i)}(aftm))$$

During $N_{macro}(aftm)$ steps of the system only the runnable tasks will change their states. Thus, the macro step of an AFTM system can be defined as follows:

$$\Delta_{AFTM}(aftm) \equiv \delta_{AFTM}^{N_{macro}}(aftm)$$

Lemma 1 (Global idle state) For any configuration *aftm* and any task i

$$idle(\Delta_{AFTM}(aftm).T(i).st)$$

Proof This follows directly from the definitions above. □

To prove the correctness of the above definitions we need to show that every macro-lock-step (Δ_{AFTM}) can be simulated by parallel execution of the macro-steps of all tasks $T(i)$ with *runnable*(*aftm*)(i). This is shown in the following theorem.

Theorem 2 (Simulation: task automaton macro-step vs. macro-lock-step) The following holds for every configuration of AFTM *aftm*.

$$\begin{aligned}
&\bigwedge_{i\in[1:M+1]} idle(aftm.T(i).st) \Rightarrow \\
&\forall i \in [1:M+1]: \\
&\Delta_{AFTM}(aftm).T(i) = \Delta_T(aftm.T(i), aftm.IP(i)).T'(i) \quad \wedge \\
&\quad \forall o, p, j : src(j,p) = (i,o) \Rightarrow \\
&\qquad \Delta_{AFTM}(aftm).OP(i,o) = \epsilon \quad \wedge \\
&\qquad \Delta_{AFTM}(aftm).IP(j,p) \;=\; \begin{cases} \Delta_T(aftm.T(i), aftm.IP(i)).OP'(i,o) \oplus_\epsilon aftm.IP(j,p), \\ \qquad \text{if } \neg runnable(aftm)(j) \\ \Delta_T(aftm\,T(i), aftm.IP(i)).OP'(i,o) \quad \text{, otherwise} \end{cases}
\end{aligned}$$

Proof The proof follows directly from the above definition of Δ_{AFTM} and from the definitions in Sects. 6.3.2 and 6.3.3. □

7. Deployment

A system specified in a CASE tool with a synchronous lock step semantics has to be deployed onto a target architecture without loss of certain properties which have been already proven for it. The problem of property preservation during deployment arises due to the differences in the time-synchronous semantics of AFTM and the time-triggered semantics of the D-OLOS platform. The main difference is that AutoFOCUS components perform their computation steps simultaneously, driven by a global clock, whereas in a time-triggered system applications are scheduled sequentially. In the case of communication cycles in the system these differences would lead to deviations in the order of input processing of particular tasks. One possible solution would be the simulation of the AFTM semantics by the deployment platform, e.g. by establishing the global computation and communication phases in the schedules. However, this would lead to unnecessary communication delays and inefficient resource usage.

In this section, we present another approach to the deployment of time-synchronous models on time-triggered platforms. Section 7.1 defines the mapping from an AFTM system onto COA application automata. After that, a scheduler synthesis procedure is introduced (in Sect. 7.2) which, together with WCET estimations, guarantees the preservation of temporal properties proved to hold in a given AFTM system.

7.1. Mapping AFTM to COA

To specify the mapping from an AFTM onto a COA model, the following functions and relations are introduced:

- The mapping from AFTM tasks onto COA application automata.
- The mapping from AutoFOCUS output ports onto FlexRay slots and message types.

Distribution. Each AFTM task $T(i)$ is mapped onto the COA application $C(k, j)$ which simulates the task. Given $M + 1$ tasks and p ECUs the task deployment is declared by a injective mapping

$$depl \subseteq [1 : M + 1] \to [0 : p - 1] \times [0 : N - 1]$$

where N is the number of applications maximally allowed on an ECU:

$$N \equiv \max_{0 \leq k < p} n_k$$

The *depl*-function is defined for the whole input domain ($[1 : M + 1]$); however, there is no need to cover the whole range ($[0 : p - 1] \times [0 : N - 1]$). By this we allow further applications to be deployed on the same deployment platform. For reasons of simplicity we assume that the presence of alien functionality (non-AFTM tasks deployed on the same ECU) does not influence a system's behavior. On the one side this is guaranteed by the resource management mechanisms of OSEKtime and FlexRay; on the other side we demand that FTCom entries which belong to the system are never modified by alien tasks.

For a task $T(i)$ the first component of $depl(i)$ indicates the ECU on which $T(i)$ is deployed. The second component denotes the local application number of the ECU. Using the projections *fst*/*snd* we access the first/second component of the result tuple, respectively.

Scheduling. Apart from the definition of the mapping onto FlexRay slots, it is necessary to define the task activation time $start(i)$ within the round: The constraint required by AFTM is that every task is scheduled exactly once per round.

$$start : [1 : M + 1] \to [0 : ns - 1]$$

This function is dual to the function $run(k, s) \subseteq [0 : n_k - 1]$ from Sect. 3.1:

$$start(i) = s \Leftrightarrow depl(i) = (p, k) \ \wedge \ run(p, s) = k$$

Communication. In a similar way we define the partial function $depl_{op}(i, o)$ which maps an output port o of a task i onto the FlexRay slot transporting the corresponding message:

$$\begin{gathered} depl_{op} : [1 : M + 1] \times [1 : P^{out}] \to [0 : ns - 1] \\ \exists s : depl_{op}(i, o) = s \Leftrightarrow \exists j, p : (i, o) = src(j, p) \wedge fst(depl(i)) \neq fst(depl(j)) \end{gathered}$$

where ns is the number of slots in the FlexRay round (see Sect. 3.1) and P^{out} is the maximum number of output ports allowed:

$$P^{out} \equiv \max_{1 \leq i \leq M+1} P_i^{out}$$

The function describes global communication via FlexRay only. Thus it is defined solely for those output ports which are connected to a task which is deployed on another ECU. We also need a function that defines the points of time of any communication in the deployed system. For the local communication we set them to the slot in which the producer is scheduled:

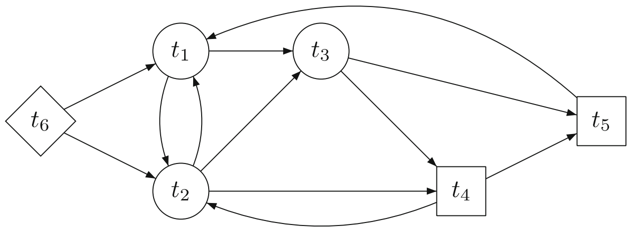
$$depl_{comm}(i, o) \equiv \begin{cases} s & \exists s : depl_{op}(i, o) = s \\ start(i) & \text{otherwise} \end{cases}$$

We define a function $mtype_{op}$ that returns for a given output port the corresponding message type:

$$mtype_{op} : [1 : M + 1] \times [1 : P^{out}] \to [0 : | OP | -1]$$

where OP is the set of all output ports in the system. The value produced by the task i on port o is written into $MB(fst(depl(i)))(mtype_{op}(i, o))$. The relation to the functions $send(s) \in [0 : p - 1]$ and $mtype(s) \in [0 : nm - 1]$ from Sect. 3.1 is:

$$depl_{op}(i, o) = s \Leftrightarrow send(s) = fst(depl(i)) \wedge mtype(s) = mtype_{op}(i, o)$$



(a)

Slot	ECU 1	ECU 2	Communication
0			(t_6, t_1)
1	t_1		(t_6, t_2)
2			(t_1, t_2)
3		t_2	
4			(t_2, t_3)
5	t_3		(t_2, t_1)
6			(t_3, t_4)
7		t_4	
8			(t_4, t_5)
9	t_5		

(b)

to \ from	t_6	t_1	t_2	t_3	t_4	t_5
t_1	1	-	1	-	-	0
t_2	1	0	-	-	0	-
t_3	-	1	1	-	-	-
t_4	-	-	1	1	-	-
t_5	-	-	-	1	1	-

(c)

Fig. 6. AutoFOCUS Task Graph ($M = 5$) with a sample schedule and delays

7.2. Scheduler synthesis procedure

For the deployment of an AFTM we assume that the system specification is given as a set of linear temporal logic (LTL) formulas (see also [CGP99]) and that the AFTM satisfies this specification. In the definition of the mapping from an AFTM onto a COA system as introduced so far, the temporal properties are not considered. In the following we describe the scheduler synthesis procedure, which aims to preserve the temporal properties already proven to hold in a given AFTM system. Thereby we will use a more elaborate example in spite of eCall (cf. Fig. 6). It is designed to exhibit the most interesting causal relations between tasks.

To simulate AFTM by a deployed system a scheduling constraint is necessary: if an application sends its outputs via FlexRay, the FlexRay communication must be scheduled *after* the producing task. Formally,

$$\forall i \in [1 : M + 1], o \in [1 : P_i^{out}] : \exists s : \mathit{depl}_{op}(i, o) = s \Rightarrow \mathit{start}(i) < \mathit{depl}_{op}(i, o) \quad (7)$$

The task graph in Fig.6 serves as an illustration of the presented concepts. There, the circles denote OR-tasks, the rectangles AND-tasks and the diamond denotes the environment. There is a directed edge from a task t_i to a task t_j if $(i, j) \in \mathit{flow}$.

Assuming the tasks from Fig. 6 are deployed on two ECUs (t_1, t_3, t_5 on ECU 1 and t_2, t_4 on ECU 2), then Fig. 6 shows a sample schedule for these tasks. The unallocated ECU computation slots can be used by other tasks which do not belong to this task graph. The environment task is not deployed explicitly. It is merely represented by the FlexRay slots for the input/output messages it produces/consumes. The assumption is that the behavior of the environment task in AFTM and the real environment (which can be realized by several ECUs/tasks) is equivalent.

(r, s)		$(0,1)$	$(1,1)$	$(2,1)$	$(3,1)$	$(4,1)$
1. *MB*		$i_6^{(1)}, \epsilon_2, \epsilon_5$	$i_6^{(2)}, i_2^{(1)}, \epsilon_5$	$i_6^{(3)}, i_2^{(2)}, \epsilon_5$	$i_6^{(4)}, i_2^{(3)}, \epsilon_5$	$i_6^{(5)}, i_2^{(4)}, i_5^{(1)}$
2.	(t_6, t_1)	$i_6^{(1)}$	$i_6^{(2)}$	$i_6^{(3)}$	$i_6^{(4)}$	$i_6^{(5)}$
	(t_2, t_1)	ϵ_2	$i_2^{(1)}$	$i_2^{(2)}$	$i_2^{(3)}$	$i_2^{(4)}$
	(t_5, t_1)	–	–	–	–	–
3. inputs		$\epsilon, \epsilon, \epsilon_5$	$i_6^{(1)}, \epsilon_2, \epsilon_5$	$i_6^{(2)}, i_2^{(1)}, \epsilon_5$	$i_6^{(3)}, i_2^{(2)}, \epsilon_5$	$i_6^{(4)}, i_2^{(3)}, i_5^{(1)}$

Fig. 7. Message/input buffer content & inputs of the task t_1

While an AFTM system is allowed to have arbitrary communication cycles (except for self-loops), according to the D-OLOS specification, there exists only one cycle with the fixed length of one round. For example, in AFTM the cycle between tasks t_1, t_3, and t_5 from Fig. 6 means that the environment task t_6 will send its output four times to t_1 before the first input from t_5 (together with the fifth one from t_6 and the third one from t_2) will arrive. In this context we will speak about the *age* of an input: the number of messages arrived through a specific channel so far. On the other side, in the corresponding D-OLOS system, having a schedule in which every task is scheduled exactly once (cf. Fig. 6), the first input from t_5 will be processed by t_1 together with the second input from t_6. By this a naïve deployment approach would lead to deviations from the communication semantics established by AFTM.

This problem is mastered by installing *delay buffers* of length one on communication links in the deployed system. They exactly simulate the communication semantics of AutoFOCUS. For a given channel between two tasks $T(i)$ and $T(j)$ the communication has to be either delayed by one if the data sent through this channel arrives *before* the task $T(j)$ is started, or the delay emerges naturally otherwise. By this, the set of all *input delays* $q(j, p)$ of a task $T(j)$ is defined for every connected input port by:

$$\forall p \in [1 : P_j^{in}] : q(j, p) = \begin{cases} 1 & depl_{comm}(src(j, p)) < start(j) \\ 0, & \text{otherwise} \end{cases} \tag{8}$$

As the result of the deployment every task $T(j)$ gets $|\{p \mid q(j, p) > 0\}|$ dedicated buffers for its corresponding inputs. Figure 6 shows the buffer lengths for each channel from our example. They are initialized with an ϵ. At the beginning of its activation the task reads its inputs from FTCom (*MB*) and puts them into the corresponding buffer. The values which fall out of the buffer are the actual inputs for the task logic.

The described behavior is illustrated by Fig. 7 for the task t_1 from Fig. 6a. There, the inputs for five invocations of the task according to the schedule from Fig. 6b are listed. The upper index denotes the age of an input, while the lower one is the index of its producer. The absence of a message from the task i is indicated by ϵ_i. The first line shows the content of the message buffer at the specified points in time. These values are inserted into the buffers of t_1, shown on the second line. The values, which fall out thereby, as well as the input from the task i_5 (for which no buffering is needed, cf. Fig 6c), are the actual inputs of t_1, listed in line three. This demonstrates that with the help of these buffers the tasks in AFTM and D-OLOS will work on consistent inputs (cf. the example given with the inputs of t_1 at (4, 1) in Fig. 7). This statement will be proved as a part of the simulation in Sect. 8.3.

Note: The delay buffers slow down the data flow in the system. Every message transfer lasts exactly one round. This solution is surely superfluous for applications with acyclic data flow dependencies. In these cases a sequentialization always exists, which allows one to establish slot-wise communication delays. On the other side, forbidding data-flow cycles would substantially constrain the set of deployable models and a buffer-free deployment procedure would reduce the flexibility of the schedule synthesis. Which solution is preferable depends on the concrete application domain.

Real-time scheduling. The procedure presented above is absolutely sufficient to construct slot-wise schedules. This means we can assign a FlexRay slot to a message and an OSEKtime slot to a task. However, we still need to discharge the assumption we made that the assigned task will terminate and the assigned message will be transmitted *before* the slot ends. Put another way, we need to estimate the sufficient slot length. Determining the necessary slot length is possible for concrete application code only: To determine the slot length, it is necessary to estimate the worst case execution time (WCET) for every application.

Task	WCET (ms)	
	δ_{AFA}	Δ_{T}
eCall	0.2	1.0
mobile phone	0.2	1.0
GPS	0.1	0.5

Fig. 8. Worst case execution times for eCall tasks

For the three tasks constituting the eCall (GPS navigation system, the mobile phone, and the actual emergency call application) the execution times were estimated with the tool aiT by AbsInt Angewandte Informatik GmbH [Abs06]. On the 300 MHz VAMP processor [DHP05] the tasks have WCETs, as shown in Fig. 8. The WCET data in Fig. 8 imply that we can set the slot length to 1 ms. If we compose one round of 20 slots, we get 20 ms per round, i.e. 50 rounds/s. This rate is representative for typical task activation rates used in the automotive industry.

8. Simulation proof for COA and AFTM

A system specified in a CASE tool with a synchronous lockstep semantics has to be deployed on a target architecture without loss of properties that have already been proven for it. In order to achieve this, three issues have to be addressed:

- The structure of the code generated from CASE tool models.
- Equivalence between particular AFTM tasks and the code generated out of them has to be ensured.
- Finally, for both the communication and the OS schedules, it has to be proven that the input/output relations of the AFTM model are preserved after deployment.

These issues are treated in Sects. 8.1–8.3, respectively.

8.1. Code generation

The structure of the generated code results from the OLOS requirements (see also Sect. 3.2), from the structure of the AutoFOCUS models, and from the operating system and communication schedules. The overall code structure is the following:

```
while(true) {
    "Code for copying the inputs" ;
    "Code generated from the AutoFOCUS model ($GC(depl(i))$)" ;
    "Code for copying the outputs" ;
    ttExFinished() ;
}
```

The code generated from AutoFOCUS is a schematic translation of finite automata to C0, it is denoted by $GC(depl(i))$ and is not treated further here. The code that copies the inputs for the task $T(i)$ needs a set of additional variables $\{buf_{i,p} \mid q(i,p) = 1\}$. It is built according to the following scheme:

```
// for ports with $q(i,p) = 1$
...
if ($va(GC(depl(i)), buf_{i,p}) \neq \epsilon$)
    $task\_var\_in_{i,p} = buf_{i,p}$ ;
```

```
$ttRecv(mtype_{op}(src(i, p)), buf_{i,p})$;
...
// for ports with $q(i, p) = 0$
...
if $(MB(fst(depl(i)))(mtype_{op}(src(i, p))) \neq \epsilon)$
  $ttRecv(mtype_{op}(src(i, p)), task\_var\_in_{i,p})$;
...
```

The code for copying one output is simpler: it consists of a $ttSend$-call followed by an assignment that flushes the corresponding local output variable:

```
// for all output ports of a task
...
$ttSend(mtype_{op}(i, o), task\_var\_out_{i,o})$;
$task\_var\_out_{i,o} = \epsilon$;
...
```

The wrapper code introduced above, together with the code generated from AutoFOCUS, behaves in exactly the same way as the model in the AutoFOCUS tool (see Proposition 1 below).

8.2. Translation validation

To show the property preservation, it is necessary to ensure the equivalence between particular AFTM tasks and the code generated out of them. This is accomplished using *translation validation* [PSS98]. The correctness of an AutoFOCUS task relies on the correctness of individual transition steps of the underlying I/O automaton. Their correctness can be guaranteed by generating assertions for every piece of code which implements an individual transition step. These assertions have to be proven in the real code. The Isabelle theorem prover was enriched by a verification environment for C0 code within the scope of the Verisoft project. In this environment assertions, expressed in Hoare-triple style, can be verified [Sch05].

The simulation relation between an AFTM task $T(i)$, its inputs and outputs $IP(i)$ and $OP(i)$, and an instantiated COA application $GC(depl(i))$ denoted as

$$GC(depl(i)) \cong (T(i), IP(i), OP(i))$$

is defined for all configurations of $T(i)$, where the control state $T(i).st$ is idle, and for all configurations $GC(depl(i))$ after a number of slot computations of the corresponding $C(depl(i))$ as follows:

$$\begin{array}{lrl} \forall x: & va(GC(depl(i)), x) & = T(i).x \\ \forall p: & va(GC(depl(i)), task\_var\_in_{i,p}) & = IP(i, p) \\ \forall o: & va(GC(depl(i)), task\_var\_out_{i,o}) & = OP(i, o) \end{array} \tag{9}$$

Here, x denotes a state component (control state/variable) of $T(i)/GC(depl(i))$. The behavioral equivalence is formulated in Proposition 1.

Proposition 1 An AFTM task $T(i)$ and a COA application $GC(depl(i))$ are in the *simulation relation* if for all inputs of the task $T(i)$ that are in the simulation relation with the set of internal input variables of the application $GC(depl(i))$, the results of the transition functions of the AFTM and COA models are also in the simulation relation. Hereby the content of the message buffer is not changed.

$$idle(T(i).st) \wedge GC(depl(i)) \cong (T(i), IP(i), OP(i)) \Rightarrow \begin{cases} GC'(depl(i)) \cong (T'(i), IP'(i), OP'(i)) \\ MB(fst(depl(i))) = MB'(fst(depl(i))) \end{cases}$$

where

$$\begin{aligned} (T'(i), OP'(i)) &= \Delta_T(T(i), IP(i)) \\ IP'(i) &= \begin{cases} E_{in}(i) & runnable(T(i), IP(i)) \\ IP(i) & \text{otherwise} \end{cases} \\ (GC'(depl(i)), MB'(fst(depl(i)))) &= res_{LC}(GC(depl(i)), MB(fst(depl(i)))) \end{aligned}$$

Please note that the second part of the above proposition means that code generated from AutoFOCUS automaton does not contain any $ttSend$ or $ttRecv$ statements. The simulation proof presented in the next section relies on this proposition: It is applicable only to the systems with behavioral equivalence between AutoFOCUS tasks and the corresponding generated C0 programs. Thus the proposition has to be proven for every individual task, e.g. by using the framework from [Sch05] and translation validation as discussed above.

8.3. Simulation proof

Proposition 1 shows that given the same inputs, an AFTM task and the code generated from this task behave in the same way. Additionally, it is necessary to show that the inputs processed by the task and the generated code are always the same.

Theorem 3 (Simulation: COA vs. AFTM) Let $TaskInput(j, p)$ be the input received by the task j on the port p.

$$TaskInput(j,p) = eti(j,p) \oplus_{\epsilon} va(C(depl(j)), task\_var\_in_{j,p}), \text{ where}$$
$$eti(j,p) = \begin{cases} MB(fst(depl(j)), mtype_{op}(src(j,p))) & q(j,p) = 0 \\ va(C(depl(j)), buf_{j,p}) & \text{otherwise} \end{cases}$$

Further on let the upper index $(.)^{r,s}$ denote the configuration of some component *at the beginning* of the corresponding slot. Then, the following simulation relation holds during the system run:

$$\begin{array}{ll} \forall i, r: & \\ \quad \forall x: & va(coa^{r,start(i)}.C(depl(i)), x) = aftm^{r}.T(i).x \\ \wedge \forall p: & TaskInput^{r,start(i)}(i,p) = aftm^{r}.IP(i,p) \\ \wedge \forall o: & va(coa^{r,start(i)}.C(depl(i)), task\_var\_out_{i,o}) = aftm^{r}.OP(i,o) = \epsilon \end{array}$$

Proof The theorem is obviously true in the initial system configuration: All AutoFOCUS tasks automata as well as the COA programs are in their initial states, the slot and round numbers are zero. The message-buffers of all ECUs and applications variables are empty:

$$\forall\, k \in [0 : p-1], l \in [0 : nm-1] : \begin{cases} coa.MB(k)(l) = \epsilon \\ va(coa.C(k,l), x) = \epsilon \end{cases}$$

and all port variables in AFTM are empty, i.e. they keep no messages:

$$\forall i \in [1 : M+1], \forall p \in [1 : P_i^{in}], \forall o \in [1 : P_i^{out}] : aftm.IP(i,p) = \epsilon \wedge aftm.OP(i,o) = \epsilon$$

Thus, at the beginning the AFTM model and the COA model are in the simulation relation. □

Induction step. Let us assume that simulation relation holds in some round r for any task $T(i)$. We show that these relations hold also in the round $r+1$ using the following lemmas. Every lemma has the induction assumption as a premise.

Lemma 2 (State invariance for inactive applications in COA). For all tasks i

$$coa^{(r,start(i))+1}.C(depl(i)) = coa^{(r+1,start(i))}.C(depl(i))$$

Proof The configuration of an application in COA is changed only once per round according to the scheduling constraint from Sect. 7.1. □

Now we prove the theorem by considering different execution phases of the AFTM and COA systems. We split the execution of a single task into three phases: (1) execution of the code generated from AutoFOCUS, (2) copying the produced values into the D-OLOS message buffer MB, (3) transmission of the values stored in MB to all the ECUs and copying the values from MB into task variables. Each of the following lemmas considers one execution phase, respectively.

Lemma 3 (State equivalence). Let GC' be defined as in Proposition 1, namely as the result of the application of the function res_{LC} to the code GC. Then, for all tasks i in $aftm^{r}$ and their state components x

$$va(coa^{(r+1,start(i))}.C(depl(i)), x) = \Delta_T(T(i), IP(i)).T'(i).x$$
$$\forall j : va(GC'(depl(i)), task\_var\_out_{i,j}) = \Delta_T(T(i), IP(i)).OP'(i,j)$$

Proof Note that the induction assumption guarantees assumptions of Proposition 1 to hold. The statement follows directly from Proposition 1. □

Lemma 4 (Output equivalence). For all tasks i in $aftm^r$, for all their output ports o, for all ECUs n, and for all slots (r', s') which lie in the interval

$$[(r, depl_{comm}(i, o)) + 1 : (r + 1, depl_{comm}(i, o))]$$

the following predicate holds:

$$coa^{(r', s')}.MB(n, mtype_{op}(i, o)) = \Delta_T(T(i), IP(i)).OP'(i, o)$$

Proof As a consequence of the second part of Lemma 3, the code structure, as introduced in Sect. 8.1, and the semantics of $ttSend$ from Sect. 4.2, we have

$$coa^{(r, start(i))+1}.MB(fst(depl(i)), mtype_{op}(i, o)) = \Delta_T(T(i), IP(i)).OP'(i, o)$$

Furthermore, the scheduling constraint (7) ensures that the value will be transported to all ECUs before the end of the round r and will not be changed until the next broadcast, according to $depl_{comm}(i, o)$. □

Lemma 5 (Broadcast correctness) For all tasks i, j in $aftm^r$ and all their ports o, p, such that $(i, o) = src(j, p)$, all the produced messages are correctly broadcast:

$$TaskInput^{r+1, start(j)}(j, p) = \Delta_T(T(i), IP(i)).OP'(i, o) \oplus_\epsilon va(C^{r+1, start(j)}(depl(j)), task\_var\_in_{j,p})$$

Proof The following is true for all tasks i:

1. In the case that the message is broadcast after the start of the corresponding task j ($q(j, p) = 0$), the following obviously holds:

$$(r + 1, start(j)) \in [(r, depl_{comm}(i, o)) + 1 : (r + 1, depl_{comm}(i, o))]$$

Thus, according to Lemma 4 and the definition of *TaskInput*,

$$\begin{aligned} eti^{r+1, start(j)}(j, p) &= coa^{(r+1, start(j))}.MB(fst(depl(j)), mtype_{op}(i, o)) \\ &= coa^{(r, depl_{comm}(i, o))+1}.MB(fst(depl(i)), mtype_{op}(i, o)) \\ &= \Delta_T(T(i), IP(i)).OP'(i, o) \end{aligned}$$

where eti is the external task input, as defined at the beginning of the theorem.

2. In the case that the message is broadcast before the start of the corresponding task j ($q(j, p) = 1$), from the induction assumption, Lemma 2, and the definition of *TaskInput* we have

$$\begin{aligned} eti^{r+1, start(j)}(j, p) &= va(coa^{r+1, start(j)}.C(depl(j)), buf_{j,p}) \\ &= va(coa^{(r, start(j))+1}.C(depl(j)), buf_{j,p}) \\ &= coa^{(r, start(j))}.MB(fst(depl(j)), mtype_{op}(i, o)) \\ &= \Delta_T(T(i), IP(i)).OP'(i, o) \end{aligned}$$

Since *TaskInput* is defined as

$$TaskInput^{r+1, start(j)}(j, p) = eti^{r+1, start(j)}(j, p) \oplus_\epsilon va(C^{r+1, start(j)}(depl(j)), task\_var\_in_{j,p}),$$

the statement is true in both cases. □

To prove the theorem, it is still necessary to move from Δ_T to Δ_{AFTM}. For this purpose we use Theorem 2. From Lemma 1 and Theorem 2 follows

$$\Delta_{AFTM}(aftm^r).T(i) = \Delta_T(aftm^r.T(i), aftm^r.IP(i)).T'(i)$$

and due to the first part of Lemma 3 follows the first line of the theorem statement. The second line of the theorem statement directly follows from Proposition 1:

$$\forall p : aftm^{r+1}.IP(i, p) = va(C^{r+1, start(i)}(depl(i)), task\_var\_in_{i,p}),$$

as well as Theorem 2, Lemma 5, and the fact that the code for copying the inputs behaves according to the definition of *TaskInput*. Finally, the last line of the theorem statement follows from Theorem 2 and the code structure. □

9. Related work

In this paper, we have presented a concept for a pervasive verification approach, separating verification of application logic and infrastructure. The necessity of this separation is also argued for by Sifakis et al. [STY03]. They introduce a formal modeling framework and a methodology, addressing the analysis of correct deployment and timing properties. The extension in our task concept is the explicit modeling of task dependencies and explicit statements about task activation conditions.

There are other approaches for the verification of distributed real-time software. Rushby has presented a framework for a systematic formal verification of time-triggered communication in [Rus97]. His framework allows one 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. However, 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. For current verification issues as encountered in the automotive industry, 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.

Put in a broader context the ideas of the presented work coincide with the CLI stack extension from [Moo03] in which a research proposal is sketched which aims at producing a stack of mechanically verified and formally integrated models of embedded systems. The models should range from gate-level up to a high-level programming language. The work presented here can be seen as a realization of the CLI stack vision for the upper system layers. In fact it makes a step beyond this vision by integrating a further layer (CASE tools) above the top-most CLI stack level (the C0 semantics in our case). The lower layers of the stack are also covered within the Verisoft Automotive subproject (cf. [KP06, Pau05] for details).

10. Conclusion

The results presented in this paper build a basis for our continuous verification approach, reaching from the application models in a CASE tool down to the gate level of the hardware. In particular, we presented a theorem stack, stating that every level of the system can be simulated by the underlying one. This theorem stack implies that the system modeled in a CASE tool is simulated by the system running on the real hardware. Combined with the verification of the model presented at the application layer in AutoFOCUS and translation validation for the generated code, this implies a completely verified system.

The presented proofs are more than just paper-and-pencil proofs. Major parts of the proofs are already formalized and proven in the automated theorem prover Isabelle [NPW02].

We see several significant contributions of our paper. First of all, it gives in a mathematical style a comprehensive description of the structure of an automotive hardware/software system, including relations to high level abstract application oriented models. Thus, it is a kind of refinement description of a system through several levels of abstraction including a proof of this refinement relation.

On the other hand it can be seen as a step towards giving a very precise mathematical description of what people like to call the architecture of a system. Such architectures include:

- The hardware, meaning the CPU and bus systems.
- The system-level software such as operating systems and device drivers (e.g. drivers for FlexRay in our case).
- The task level where software is represented by tasks that are scheduled; the way these tasks communicate over the bus infrastructure.
- The high-level application software description in terms of modeling techniques as used in AutoFOCUS.

We consider this contribution to be as important and interesting as the contribution to the overall structuring of hardware/software architectures to be found in automobiles. The basic idea and the approach is very much inspired by what people call levels of abstraction and architecture layers.

These terms ("levels of abstraction" and "architecture layers") are used in a way which is not always very precise. In our case, however, we give a complete mathematical description of such structures.

Acknowledgements

We want to thank Nicolas Fritz from AbsInt Angewandte Informatik GmbH for his work on worst case execution time estimation. Furthermore, we want to thank the anonymous referees who helped a lot to improve the paper.

References

[Abs06] AbsInt Angewandte Informatik. Worst-case execution time analyzers. http://www.absint.com/, 15.12.2006
[Aut06] AutoFocus Project. http://autofocus.in.tum.de, accessed 15.12.2006
[BBG$^+$05] Beyer S, Böhm P, Gerke M, Hillebrand M, In der Rieden T, Knapp S, Leinenbach D, Paul WJ (2005) Towards the formal verification of lower system layers in automotive systems. In: 23rd IEEE international conference on computer design: VLSI in computers and processors (ICCD'05). IEEE, New York
[BGH$^+$06] Botaschanjan J, Gruler A, Harhurin A, Kof L, Spichkova M, Trachtenherz D (2006) Towards modularized verification of distributed time-triggered systems. In: Formal methods 2006. LNCS, vol 4085. Springer, Heidelberg, August 23–25 2006
[BKKS05] Botaschanjan J, Kof L, Kühnel Ch, Spichkova M (2005) Towards verified automotive software. In: ICSE, SEAS Workshop, St. Louis, Missouri, USA, May 21 2005
[CGP99] Clarke EM, Grumberg O, Peled DA (1999) Model checking. The MIT Press, Cambridge
[Con06] FlexRay Consortium. FlexRay overview. http://www.flexray.com/products/protocol%20overview.pdf, accessed 15.12.2006
[DHP05] Dalinger I, Hillebrand M, Paul W (2005) On the verification of memory management mechanisms. In: Borrione D, Paul W (eds) CHARME 2005. LNCS. Springer, Heidelberg (to appear)
[dRK05] In der Rieden T, Knapp S (2005) An approach to the pervasive formal specification and verification of an automotive system (Status Report). In: Tenth international workshop on formal methods for industrial critical systems (FMICS 05)
[Eur03] European Commission (DG Enterprise and DG Information Society). eSafety forum: Summary report 2003. Technical report, eSafety, March 2003
[Fle06] FlexRay Consortium. http://www.flexray.com, accessed 15.12.2006
[GHLP05] Gargano M, Hillebrand M, Leinenbach D, Paul W (2005) On the correctness of operating system kernels. In: Hurd J, Melham T (eds) TPHOLs 2005. LNCS. Springer, Heidelberg
[HSE97] Huber F, Schätz B, Einert G (1997) Consistent graphical specification of distributed systems. In: Industrial applications and strengthened foundations of formal methods (FME'97). LNCS, vol 1313. Springer, Heidelberg, pp 122–141
[IBM06] IBM Rational Rose Technical Developer. http://www-306.ibm.com/software/awdtools/developer/technical/, accessed 18.05.2006
[IdRLP05] In der Rieden T, Leinenbach D, Paul WJ (2005) Towards the pervasive verification of automotive systems. In: Correct hardware design and verification methods. Lecture Notes in Computer Science, vol 3725. Springer, Heidelberg, pp 3–4
[KG94] Kopetz H, Grünsteidl G (1994) TTP—a protocol for fault-tolerant real-time systems. Computer 27(1):14–23
[Kna05] Knapp S (2005) Towards the verification of functional and timely behavior of an ecall implementation. Master's thesis, Universität des Saarlandes
[KP06] Knapp S, Paul W (2006) Realistic worst case execution time analysis in the context of pervasive system verification. In: Program analysis and compilation, theory and practice: essays dedicated to Reinhard Wilhelm, vol 4444, pp 53–81
[KS06] Kühnel Ch, Spichkova M (2006) Upcoming automotive standards for fault-tolerant communication: FlexRay and OSEKtime FTCom. In: International workshop on engineering of fault tolerant systems (EFTS 2006), Luxembourg, June 12–13
[KS07] Kühnel Ch, Spichkova M (2007) Fault-tolerant communication for distributed embedded systems. In: Software engineering and fault tolerance. Series on Software Engineering and Knowledge Engineering, vol 19. World Scientific Publishing, Singapore
[LPP05] Leinenbach D, Paul W, Petrova E (2005) Towards the formal verification of a C0 compiler. In: 3rd international conference on software engineering and formal method (SEFM 2005), Koblenz, Germany
[Mat06] The MathWorks. http://www.mathworks.com, accessed 18.05.2006
[Moo03] Strother Moore J (2003) A grand challenge proposal for formal methods: a verified stack. Lecture Notes in Computer Science, vol 2757/2003. Springer, Berlin
[Mot06] Motor Industry Software Reliability Association (MISRA). Guidelines for the use of the C language in critical systems, UK, 18.05.2006
[NPW02] Nipkow T, Paulson LC, Wenzel M (2002) Isabelle/HOL—a proof assistant for higher-order logic. LNCS, vol 2283. Springer, Heidelberg
[OSE01a] OSEK/VDX. Fault-Tolerant Communication—Specification 1.0, 2001. http://portal.osek-vdx.org/files/pdf/specs/ftcom10.pdf, accessed 15.12.2006
[OSE01b] OSEK/VDX. Time-Triggered Operating System—Specification 1.0, 2001. http://portal.osek-vdx.org/files/pdf/specs/ttos10.pdf, accessed 15.12.2006
[OSE06] OSEK/VDX. http://www.osek-vdx.org, accessed 15.12.2006
[Pau05] Paul W (2005) Lecture notes: computer architecture 2—automotive systems. http://www-wjp.cs.uni-sb.de/lehre/vorlesung/rechnerarchitektur2/ws0506/temp/060302_CA2_AUTO.pdf, December 2005
[PSS98] Pnueli A, Siegel M, Singerman E (1998) Translation validation. In: TACAS '98: proceedings of the 4th international conference on tools and algorithms for construction and analysis of systems, London, UK, 1998. Springer, Heidelberg
[Rus97] Rushby J (1997) Systematic formal verification for fault-tolerant time-triggered algorithms. In: Dependable computing for critical applications—6, vol 11. IEEE Computer Society, New York, pp 203–222
[Sch05] Schirmer N (2005) A verification environment for sequential imperative programs in Isabelle/HOL. In: Baader F, Voronkov A (eds) Logic for programming, artificial intelligence, and reasoning. LNAI, vol 3452. Springer, Heidelberg

[STY03] Sifakis J, Tripakis S, Yovine S (2003) Building models of real-time systems from application software. Proc IEEE 91(1):100–111

[Ver06] Verisoft Project. http://www.verisoft.de, accessed 15.12.2006

[WLPS00] Wimmel G, Lötzbeyer H, Pretschner A, Slotosch O (2000) Specification based test sequence generation with propositional logic. J STVR Special Issue on Specification Based Testing, 2000, 10:229–248

Received 23 November 2007

Accepted in revised form 1 August 2008 by C.B. Jones

Published online 25 November 2008