Formal Design Models for Distributed Embedded Control Systems

Christo Angelov, Krzysztof Sierszecki, Yu Guo

Mads Clausen Institute for Product Innovation
University of Southern Denmark
Alsion 2, 6400 Soenderborg, Denmark
{angelov, ksi, guo}@mci.sdu.dk

Abstract. The paper presents a formal specification of the software design models used in *COMDES-II* – a component-based framework for distributed control systems, featuring open architecture and predictable operation under hard real-time constraints. In this framework, an application is modelled as a network of distributed embedded actors that communicate transparently by exchanging labeled messages (signals), independent of their allocation on network nodes. Actors are configured from prefabricated executable components such as modal function blocks controlled by a master state machine, whereby actor structure is specified by a data flow model (function block network). Accordingly, actor behaviour is specified by composite functions representing signal transformations - from input to output signals, and system behaviour - by actor-level composite functions representing the overall sequence of computation – from system input to system output signals. Input and output signals are exchanged with the controlled plant at precisely specified time instants in accordance with the concept of Distributed Timed Multitasking, resulting in the elimination of transaction I/O jitter. System operation is ultimately described by a clocked synchronous model of computation featuring communicating actors, atomic (zero-time) execution of input and output actions and constant, non-zero execution time of system reactions.

Keywords: distributed control systems, component-based design of embedded software, domain-specific frameworks, correct-by-construction systems

1 Introduction

Nowadays, embedded software development is still dominated by conventional design methods and manual coding techniques. However, these are not able to cope with continuously growing demands for high quality of service, reduced development and operational costs, reduced time to market, as well as ever growing demands for software safety and dependability. In particular, software safety is severely affected by design errors that are typical for informal design methods, as well as implementation errors that are introduced during the process of manual coding.

This situation has stimulated the development of new software design methods based on formal design models (frameworks) specifying system structure and behaviour, which can be verified and validated before the generation of the program code [1, 2]. Furthermore, model-driven development can be combined with component-based design, whereby design models are implemented by means of reusable and reconfigurable components. Thus, embedded applications can be configured using repositories of prefabricated and validated components (rather than programmed), whereby the configuration specification is stored in data structures containing relevant information such as component parameters, input/output connections, execution sequences, etc. Hence, it is possible to reconfigure applications by updating data structures rather than reprogramming and reloading the entire application.

The main problem that has to be addressed with this method is to develop a *comprehensive*, yet *intuitive* and *open* framework for embedded systems. There are a considerable number of frameworks developed in the traditional Software Engineering domain that employ components with operational interfaces as well as various types of port-based objects, e.g. actor frameworks [4-8]. However, it can be argued that the architecture of the framework (i.e. models used to specify component functionality, interfacing and interaction) should be derived from areas such as Control Engineering and System Science, taking into account that modern embedded systems are predominantly control and monitoring systems. This approach has been used for some time with industrial control systems, whose software is built from component objects *(function blocks)* that implement standard application functions and interact by exchanging signals. Accordingly, function blocks are 'softwired' into function block networks that are mapped onto real-time control tasks, e.g. standards *IEC 61131-3* [10] and *IEC 61499* [11].

Unfortunately, this is a relatively low-level approach, which is inadequate for modern embedded applications. These vary from simple controllers to highly complex, time-critical and distributed systems featuring autonomous subsystems with concurrently running activities (tasks) that have to interact with one another within various types of distributed transactions. The above standards do not provide modeling techniques and component definitions at this level and do not define concurrency, whereby the mapping of function block networks on real-time tasks, as well as task scheduling and interaction are considered implementation details that are not a part of the standard.

In order to overcome the above problems, the Control Engineering models must be augmented with concepts and techniques developed in the Computer Science domain (concurrency, scheduling, communication, state machines, etc.), as advocated by leading experts in the area of Embedded Software Design, e.g. [2], [3]. The resulting framework must support compositionality and scalability through a well-defined hierarchy of reusable and reconfigurable components, including both actors and function blocks. On the other hand, it has to adequately specify system behaviour for a broad range of sequential, continuous and hybrid control applications.

These guidelines have been instrumental in developing the framework *COMDES-II* [13]. This is a domain-specific framework for time-critical distributed control applications, featuring a hierarchical component model as well as transparent signal-based communication at all levels of specification. In *COMDES-II*, an embedded application is composed from actors, which are configured from prefabricated function blocks. This is an intuitive and simple model that is easy to use and understand by application experts, i.e. control engineers.

An informal description of the above component models is given elsewhere [13]. This paper presents a formal specification of *COMDES-II* design models focusing on two interrelated aspects, i.e. system structure and behaviour. It is organized as follows: Section 2 presents a top-down specification of system structure in terms of data flow models describing actors and actor interactions, as well the internal structure of actors, which are composed of prefabricated function blocks. Section 3 presents a bottom-up specification of system behaviour starting with function block behaviour, followed by actor behaviour and finally - system behaviour. These are defined as composite functions specifying signal transformations - from input to output signals - of function blocks, actors and the system itself, respectively. Section 4 presents related research. The concluding section summarizes the main features of the framework and their implications for a software development process aimed at designing systems that are correct by construction.

2 Specification of System Structure

2.1 *COMDES-II* Design Models - an Introduction

In *COMDES-II*, an embedded system is conceived as a composition of active objects (actors) that communicate via labelled state messages (signals) encapsulating process variables, such as *speed*, *pressure*, *temperature*, etc. Communication is transparent, i.e. independent of the allocation of actors on network nodes. Accordingly, the system can be modelled by an actor network specifying constituent actors and the signals exchanged between them (see e.g. Fig. 1).

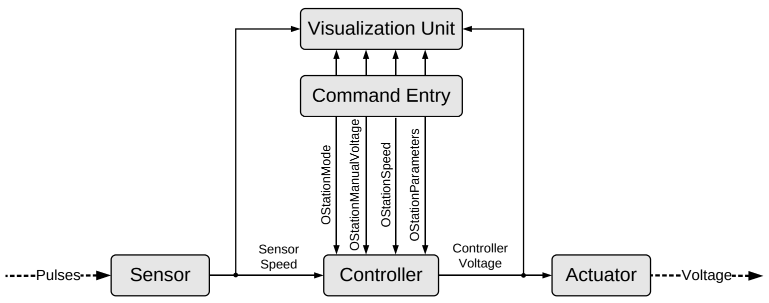

Fig. 1. COMDES-II actor network – an example: the DC Motor Control System

An actor is modelled as an integrated circuit consisting of a signal-processing block, which is mapped onto a non-blocking (basic) task, as well as input and output signal drivers that are used to exchange signals with other actors and the outside world (see Fig. 2). Actor tasks are configured from function blocks (FBs) and are modelled by function block networks. A function block is a *reusable executable component* that may have multiple instances within a given configuration. There are four kinds of function block: basic, composite, state machine and modal function blocks that can be used to implement a broad range of sequential, continuous and hybrid applications.

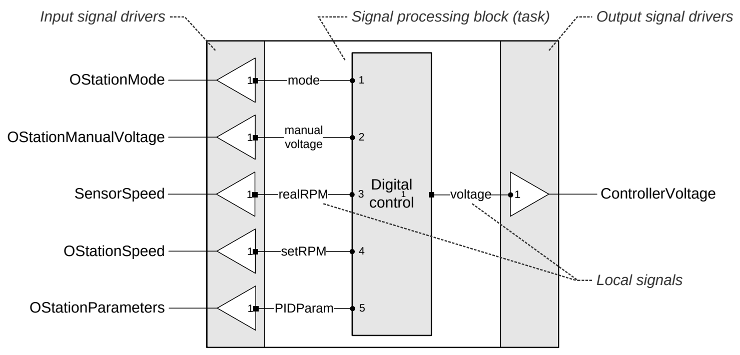

Fig. 2. COMDES-II *Controller* actor

Basic function blocks have simple stateless behaviour, which is specified by functions defining signal transformations - from input signals to output signals (e.g. a PID controller function block). Complex stateful behaviour is implemented with modal function blocks (MFBs). These may be viewed as a generalization of stateless function blocks: a MFB has a number of operational modes where each mode encapsulates one or more FB instances used to execute a control action associated with that mode. A modal function block receives indication of current mode from a supervisory state machine (SSM), whereby it executes the corresponding control action, in the context of a continuous or sequential control actor, e.g. *manual/automatic* control of DC motor rotation speed (see Fig. 3). A function block network may be encapsulated into a composite function block, which can be subsequently reused as an integral component.

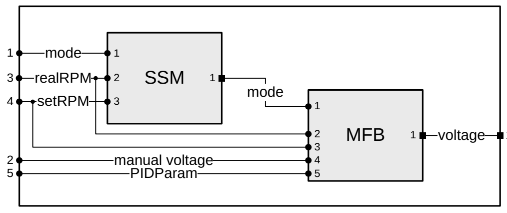

Fig. 3. The *Digital control* task composed of state machine and modal function blocks

Signal drivers are a special class of component - these are wrappers providing an interface to the system operational environment by executing kernel- or hardware-dependent functions. Specifically, signal drivers can invoke kernel primitives to transparently broadcast and receive signals, independent of the allocation of sender and receiver actors on network nodes [14].

A detailed informal description of the above component models is given elsewhere [13]. The following discussion presents a formal specification of *COMDES-II* components and component configurations. The latter takes into account the two levels of the framework, i.e. system and actor levels, which are treated in a top-down fashion. At the top level, the system is described as an *actor network* - a data flow model involving system actors and the global signals exchanged between them, as well as a definition of the signals in terms of identifiers and constituent signal variables. At the next level, each system actor is described by a *function block network*, i.e. a data flow model involving constituent function blocks and the local signals exchanged.

2.2 Distributed Control System Specification

A distributed embedded control system (ECS) is modelled as an actor network:

$$ECS = < \boldsymbol{A}, \boldsymbol{S}, \boldsymbol{C} > , \tag{1}$$

where $\boldsymbol{A}$ is the set of system actors, $\boldsymbol{S}$ is the set of system signals and $\boldsymbol{C}$ is the set of channels used to exchange signals between actors. The set of system actors $\boldsymbol{A}$ consists of environment actors $\boldsymbol{A}_{env}$ modelling the plant, and control actors $\boldsymbol{A}_{con}$ operating in a distributed system environment:

$$\boldsymbol{A} = \boldsymbol{A}_{env} \cup \boldsymbol{A}_{con} . \tag{2}$$

The set of system signals $\boldsymbol{S}$ can be represented as:

$$\boldsymbol{S} = \boldsymbol{S}_{in} \cup \boldsymbol{S}_{com} \cup \boldsymbol{S}_{out} , \tag{3}$$

where $\boldsymbol{S}_{in}$ is the subset of physical input signals, $\boldsymbol{S}_{com}$ is the subset of signals (messages) exchanged over the communication network, and $\boldsymbol{S}_{out}$ is the set of output physical signals. Furthermore, $\forall s_i \in \boldsymbol{S}$: $s_i = < Id_i, V_i >$, where Id_i is a signal identifier and V_i is a set of signal variables defined in terms of variable names and the corresponding data types:

$$V_i = \{ < s^i_1 : type^i_1 >, < s^i_2 : type^i_2 >, \; \ldots \; , < s^i_{ki} : type^i_{ki} > \} , \tag{4}$$

e.g. signal *OStationParameters* consisting of PID parameters, such as proportional, integral and derivative gain values (see Fig. 2).

The communication relationship between actors is specified in terms of channels that are defined by a *source* - *signal* - *destination* relation:

$$\boldsymbol{C} \subset \boldsymbol{A} \times \boldsymbol{S} \times 2^{A} , \tag{5}$$

e.g. one of the channels depicted in Fig. 1, which is specified by the tuple < *Sensor*, *Sensor_Speed*, {*Controller*, *Vizualization_Unit*} > .

In an actual implementation, control actors will be allocated to network nodes, and channels – to the network communication channel and physical I/O channels. The subsequent discussion assumes a real-time network with predictable message latency, such as CAN, which has been used for the experimental validation of *COMDES-II*.

A system control actor can be defined as:

$$a_{con} = < \boldsymbol{X} , L_{in}, N_{FB}, L_{out}, \boldsymbol{Y} > , \tag{6}$$

where: $\boldsymbol{X}$ is the set of input signals received by the actor, $\boldsymbol{X} \subset \boldsymbol{S}$, L_{in} is an input signal latch, N_{FB} is a signal-processing network of function blocks, L_{out} is an output signal latch and $\boldsymbol{Y}$ is a set of output signals generated by the actor, $\boldsymbol{Y} \subset \boldsymbol{S}$.

The input latch is used to receive input signals and decompose them into input signal variables constituting the set *V*, which may be viewed as local signals that are processed by the function block network. The latter computes output variables constituting the set *W*, which are used to compose the output signals generated by the output latch (see e.g. Figs. 2 and 3).

The I/O latches are composed of communication objects called *signal drivers*, denoted as D^{in} and D^{out}. In particular:

$$L_{in} = \{ D^{in}_i \};\ D^{in}_i :\ s^{in}_i \ \rightarrow\ V_i\ ,\ V_i \ \subset\ V\ ,$$
$$L_{out} = \{ D^{out}_i \};\ D^{out}_i :\ W_i \ \rightarrow\ s^{out}_i\ ,\ W_i \ \subset\ W\ , \qquad (\mathbf{7})$$

where V_i and W_i denote the constituent variables of the corresponding I/O signals s^{in}_i and s^{out}_i , respectively.

The I/O latches are activated at the release and deadline instants of the actor task. This is a basic (non-blocking) task, whose internal structure is specified as a function block network performing the transformation of input signal variables into output signal variables: $V \ \rightarrow\ W$.

The FB network is modelled by an *acyclic* data flow graph (see e.g. Fig. 3), which can be defined as follows:

$$N_{FB}\ =\ < B, Z, Con >\ , \qquad (\mathbf{8})$$

where *B* is a set of function blocks (FBs), *Z* is a set of FB network variables and *Con* is the set of FB network connections.

A function block performs the signal transformation $X \rightarrow Y$, where *X* is the set of FB input variables, $X \subset Z$, and *Y* is the set of FB output variables, $Y \subset Z$. Specifically, a function block can be defined as:

$$FB\ =\ < X, Y, P, F >\ , \qquad (\mathbf{9})$$

where *X*, *Y* and *P* denote input, output and persistent variables, respectively and *F* is a set of functions.

Input variables *X* are generated by input drivers or other function blocks, $X \subset Z$. These are used together with persistent variables to compute output variables *Y*, $Y \subset Z$. Persistent variables *P* represent the internal *state* of the function block, which is retained from one execution to the next, e.g. various types of controllers, filters, etc. [10]. Simple function blocks may not have internal state, e.g. arithmetic function blocks, comparators. Output variables are computed by functions $f \in F$ that are defined as $y = f(x, p)$, where $y \in Y$, $x \in X$ and $p \in P$.

The variables constituting the set *Z* may be viewed as *local signals* associated with the function block network:

$$Z = V\ \cup\ I\ \cup\ W\ , \qquad (\mathbf{10})$$

where the input signal variables *V* are generated by input drivers and processed by function blocks; internal variables *I* are generated and processed by function blocks; output signal variables *W* are generated by function blocks and used by output drivers to compose output signals (see e.g. Fig. 3).

FB network connections are used to wire function blocks with input and output signal drivers, and with each other. The corresponding set can be specified as a union of subsets denoting input, internal and output connections: $Con = Con_{in} \cup Con_{int} \cup Con_{out}$. These are defined as *source* - *local signal* - *destination* relations as follows:

$$\begin{aligned} Con_{in} &\subset L_{in} \times V \times B\,, \\ Con_{int} &\subset B \times I \times B\,, \\ Con_{out} &\subset B \times W \times L_{out}\,, \end{aligned} \tag{11}$$

e.g. the connection represented by the tuple < *SSM*, *mode*, *MFB* > shown in Fig. 3.

3 Specification of System Behaviour

3.1 *COMDES-II* Model of Computation – an Introduction

System operation is specified in terms of distributed transactions executed in accordance with a model of computation known as Distributed Timed Multitasking [12, 13], which is presently supported by the distributed real-time kernel *HARTEXμ* [14]. The distributed transaction involves a number of actors that execute transaction phases by invoking sequences of function blocks within the corresponding actor tasks. Actors interact with each other by exchanging labelled state messages (signals) using dedicated communication objects (signal drivers) that provide for transparent one-to-many communication between the actors involved.

Distributed Timed Multitasking (DTM) combines the concepts of Timed Multitasking [5] and transparent *signal-based* communication. With this model, it is assumed that signal drivers are short pieces of code that are executed atomically in logically *zero* time at precisely specified time instants, which is typical for control applications. Specifically, input signal drivers are executed when the actor task is released, and output drivers - when the task deadline arrives or when the task comes to an end, if it has no deadline (see Fig. 4). Consequently, task I/O jitter is effectively eliminated as long as the task comes to an end before its deadline.

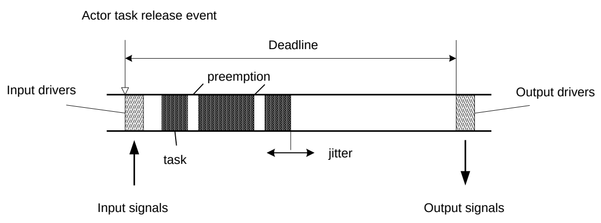

Fig. 4. Actor execution under Distributed Timed Multitasking

Jitter-free operation can be extended to distributed systems, e.g. a phased-aligned transaction involving the actors *Sensor* (*S*), *Controller* (*C*) and *Actuator* (*A*) from Fig. 1, which are triggered by a periodic timing event, such as a synchronization (*sync*) message denoting the initial instant of the transaction period (*T*), with deadline $D \le T$

(see Fig. 5). In this case, input and output signals are generated at transaction start and deadline instants, resulting in the elimination of transaction I/O jitter.

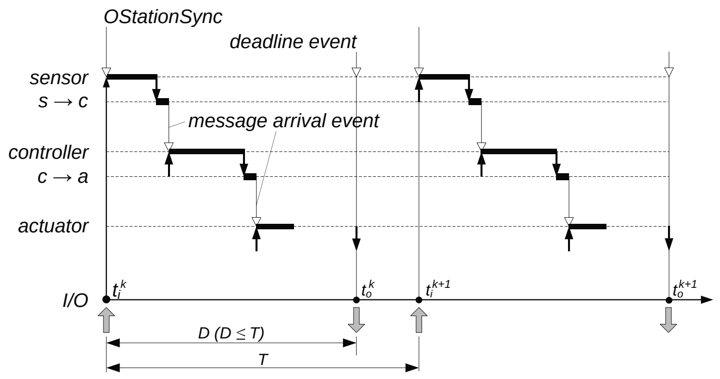

Fig. 5. Jitter-free execution of distributed transactions

The following discussion presents a formal specification of system operation, taking into account the adopted model of computation and the model of system structure developed in the preceding section.

3.2 Specification of Function Block Behaviour

Function block operation is specified with simple and/or composite functions from FB input variables $x(k)$ to FB output variables $y(k)$, $x \in X$, $y \in Y$, assuming periodic execution of system actors and constituent function blocks, which are invoked at time instants kT, $k = 1, 2, \ldots$, where T is the execution period of the host actor.

Basic function blocks implement standard signal-processing functions, such as:

$y(k) = f(x(k))$ - with simple FBs implementing various kinds of mathematical operations, comparators, etc. **(12)**

$y(k) = f(x(k), p(k-1), p(k-2), \ldots p(k-l))$ - with FBs having persistent state, **(13)**

where the state is defined in terms of one or more persistent variables $p(k-1), p(k-2), \ldots, p(k-l)$, retained from previous periods $1, 2 \ldots, l$ and updated during each period (as specified by the concrete FB algorithm, e.g. the discrete-time versions of filters, various control algorithms, etc. [10]).

A composite function block (CFB) encapsulates a FB network whose behaviour is described with one or more functions such as $y(k) = f(x(k))$, where f is a composite function specifying the transformation of signals from CFB inputs to CFB outputs, which is defined in terms of the functions executed by the constituent function blocks. Assuming that the CFB encapsulates a sequence of r function blocks, this function can be represented as:

$f = f_r \circ f_{r-1} \circ \ldots \circ f_1$, or using another notation: $y(k) = f_r (f_{r-1} (\ldots (f_1(x(k)))\ldots))$ **(14)**

In the general case, this function will have a different expression for each particular configuration of the FB network, which has to be always modelled by an *acyclic* data flow diagram. However, cycles are allowed at actor level but these are effectively broken by one-period delays due to the adopted clocked synchronous model of computation (see below).

The supervisory state machine (SSM) implements the reactive aspect of actor behaviour, in separation from the transformational (signal processing) aspect, which is delegated to the modal function block. The SSM generates two output signals - *m* and *u*, meaning *mode* and *mode-updated*, which are specified by the corresponding functions:

$m(k) = f(m(k\text{-}1), e(k), pr(e(k)))$ - a mode transition function, and

$u(k)$ - a Boolean function, which is defined as follows: (**15**)

$u(k) = true$ when $m(k) \neq m(k\text{-}1)$, i.e. when a mode transition has taken place,
$u(k) = false$ when $m(k) = m(k\text{-}1)$, and no transition has taken place.

In the above expression *e(k)* denotes a *transition trigger,* i.e. an event specified as a Boolean expression involving binary input signals that are *present* at time *kT*, *T* is the period of the host actor, and *pr(k)* is the priority of the event triggering the transition from *m(k-1)* to *m(k).*

The modal function block (MFB) implements the signal processing aspect of actor behaviour by executing constituent function blocks within the corresponding modes of operation. These compute control signals y_i, $i = 1, 2, ..., r,$ by invoking signal transformation functions $f_1, f_2, \ldots, f_r$ – from input to output signals. Subsets of these functions are selected for execution, depending on the *mode* and *mode-updated* input signals indicated by the state machine function block, such that:

$\forall y_i \in A_p, y_i(k) = f_i(x(k))$, and $\forall y_i \in A_q, q \neq p, y_i(k) = y_i(k\text{-}1)$ - when $m(k) = p$ and $u(k) = true;$ (**16**)

$\forall y_i, y_i(k) = y_i(k\text{-}1)$ - when $u(k) = false$,

where A_p denotes the control action, i.e. the subset of control signals generated in mode *p*, and f_i is the function executed by the corresponding function block(s) in order to generate the signal y_i, $y_i \in A_p$. For instance, the control signal *voltage* of Fig.3 will be generated by a PID function block if *mode* has been updated to *automatic.*

The composition of supervisory state machine and modal function block operates as a periodically executed event-driven state machine whose operational semantics and implementation are presented in [15]. This state machine is invoked within a periodically executing host actor but a state transition takes place only when the corresponding transition trigger is present, much in the same way as event-driven state machines triggered by external interrupts.

3.3 Specification of Actor Behaviour

Actors generate reactions to execution triggering events in the form:

$$\boldsymbol{e} \rightarrow \boldsymbol{Y_e}, \qquad (\mathbf{17})$$

where $\boldsymbol{Y_e} \subset \boldsymbol{Y}$, $\boldsymbol{Y_e}$ being the set of output signals generated by the actor in response to the execution trigger $\boldsymbol{e}$. The latter may be a local timing event $\uparrow(kT)$, a global timing event $\uparrow sync(kT)$ generated by a periodic synchronization message or an external event $\uparrow \boldsymbol{x}_{trigger}$, where $\boldsymbol{x}_{trigger}$ is one of the actor input signals (e.g. a message arrival event)[1].

Actor output signals $\boldsymbol{y} \in \boldsymbol{Y}$ are specified by functions of input signals $\boldsymbol{x} \in \boldsymbol{X}$ that are latched by input drivers at the time of input t_{in}. With periodic actors triggered by local or global timing events $t_{in} = kT, k = 0, 1, 2, \ldots$

Output signals are composed of output signal variables generated by the actor FB network, which has a zero *logical execution time (LET)*. Hence, the output signal variables are logically related to the input time instant *kT*:

$$w(k) = \varphi(v(k)) \, , \tag{18}$$

where φ is a composite function – from input signal variables $v \in V$ to output signal variables $w \in W$ that constitute actor input signals $\boldsymbol{x}$ and output signals $\boldsymbol{y}$, respectively.

With actors having purely transformational behaviour, φ can be defined like a CFB function, e.g.:

$$\varphi = f_r \circ f_{r-1} \circ \ldots.. \circ f_1 \, , \tag{19}$$

where f_i are basic and/or composite signal-transformation functions executed by constituent function blocks, $i = 1, 2, \ldots, r$.

With complex actors built from supervisory state machines coupled to modal function blocks, each mode generates certain control signals specified by the corresponding functions, for example:

$$\begin{aligned} w_1(k) &= \varphi^1 (v(k)) \quad \text{- generated in mode } 1 \\ w_2(k) &= \varphi^2 (v(k)) \quad \text{- generated in mode } 2 \\ &\ldots\ldots\ldots\ldots\ldots\ldots \\ w_s(k) &= \varphi^s (v(k)) \quad \text{- generated in mode } s \end{aligned} \tag{20}$$

In this case, for each φ^i, $\varphi^i = f_i \circ m$, where *m* is the mode transition function of the SSM function block and $f_i(v(k))$ is the signal transformation function executed by the modal function block when the supervisory state machine has indicated that $m(k) = i$.

In the general case:

$$\varphi^i = f_i \circ m \circ g \, , \tag{21}$$

where *g* denotes a pre-processing function. The latter is executed by a pre-processing (basic or composite) function block, generating a transition-trigger signal for the supervisory state machine (e.g. various types of arithmetic, comparators, counters, etc.)

The output variables generated by the actor task are used to compose output signals, which are latched into the output drivers at the time of output:

$$\boldsymbol{y}(t_{out}) = \varphi(\boldsymbol{x}(t_{in})) \, , t_{out} = t_{in} + D = kT + D, k = 0, 1, 2, \ldots \, ; 0 \le D \le T \, , \tag{22}$$

Hence:

$$\boldsymbol{y}(kT + D) = \varphi(\boldsymbol{x}(kT)) \, , \tag{23}$$

[1] Bold symbols denote actor-level events and input/output signals.

and the actor as a whole has a *clocked synchronous semantics* [19], chracterized by a *non-zero* logical execution time (LET).

In the special case of actor without deadline, it is assumed that $D = 0$, and $t_{in} = t_{out} = kT$. Hence: $\boldsymbol{y}(k) = \varphi(\boldsymbol{x}(k))$, and the actor has a perfect *synchronous* semantics (zero LET). This is the case with intermediate actors of phase-aligned transactions, where the deadline is usually associated with the last actor, which has to generate the control signal at the transaction deadline instant (see next section).

3.4 Specification of System Behaviour

System operation is specified in terms of distributed transactions, such as the transaction shown in Fig. 5, assuming: 1) Periodic phase-aligned transactions involving non-blocking *basic* tasks, such as the one shown in Fig. 5, which are typical for distributed control applications [18]; 2) Non-blocking signal-based communication; 3) Distributed Timed Multitasking, which is an extension of Timed Multitasking for distributed transactions.

Under these assumptions, a periodic phase-aligned transaction with a period T_{trans} can be represented as a sequence of transaction phases, involving a number of actors, which are executed in response to a global timing event $\uparrow sync(kT_{trans})$ represented by the arrival of a synchronisation (*sync*) message generated by a sync master node:

$$\begin{gathered} \uparrow sync(kT_{trans}) \rightarrow \boldsymbol{y}_1 ; \quad \boldsymbol{y}_1 = \varphi_1(\boldsymbol{x}_1), \\ \uparrow \boldsymbol{x}_2 \rightarrow \boldsymbol{y}_2 ; \quad \boldsymbol{y}_2 = \varphi_2(\boldsymbol{x}_2), \\ \ldots\ldots\ldots\ldots\ldots \\ \uparrow \boldsymbol{x}_n \rightarrow \boldsymbol{y}_n ; \quad \boldsymbol{y}_n = \varphi_n(\boldsymbol{x}_n), \end{gathered} \tag{24}$$

where: $\boldsymbol{x}_1 = \boldsymbol{x}_{in}, \boldsymbol{x}_2 = \boldsymbol{y}_1, \boldsymbol{x}_3 = \boldsymbol{y}_2, \ldots, \boldsymbol{x}_n = \boldsymbol{y}_{n-1}, \boldsymbol{y}_n = \boldsymbol{y}_{out}$.

Hence, transaction execution can be modelled with a composite function:

$$\Phi = \varphi_n \circ \varphi_{n-1} \circ \ldots\ldots \circ \varphi_1 , \tag{25}$$

where φ_i is the function implemented by the i-th actor, $i = 1, 2, \ldots., n$.

Taking into account Distributed Timed Multitasking, transaction execution can be represented as a transformation from input signals $\boldsymbol{x}_{in}(t_{in})$ to output signals $\boldsymbol{y}_{out}(t_{out})$, where t_{in} and t_{out} are determined by the transaction period T_{trans} and deadline D_{trans}:

$$\begin{gathered} \uparrow sync(kT_{trans}) \rightarrow \boldsymbol{y}_{out} , \\ \boldsymbol{y}_{out}(kT_{trans} + D_{trans}) = \Phi(\boldsymbol{x}_{in}(kT_{trans})); D_{trans} \leq T_{trans} . \end{gathered} \tag{26}$$

For the particular example illustrated by Figures 1 and 5, the behaviour of the control system can be represented in the form:

$$Voltage(kT_{trans} + D_{trans}) = \Phi(pulses(kT_{trans})),\ \Phi = \varphi_{actuator} \circ \varphi_{controller} \circ \varphi_{sensor} .$$

In the general case, the distributed system may consist of multiple subsystems executing distributed transactions with different rates of activation (multi-rate system), e.g. a multi-loop distributed control system. Accordingly, subsystem actors are allocated onto network nodes, and subsystem channels – onto the physical communication channel(s). This raises the issue of concurrent execution of transaction tasks/communications within the corresponding operational domains.

Following the adopted model of computation (Fig. 4), actor tasks are executed in a dynamic priority-driven scheduling environment provided by node-resident kernels, which are instances of the *HARTEXμ* timed multitasking kernel [14]. Communication takes place in a real-time network supporting predictable interactions, such as CAN. Transparent signal-based communication is supported by a dedicated protocol provided by the *HARTEXμ* kernel. With this protocol, signal drivers are executed *atomically* at precisely specified time instants that are fixed on the time axis. This makes it possible to eliminate the undesirable effects of task preemption and network communication, i.e. transaction I/O jitter, as long as transaction (end-to-end) response times are less than the corresponding end-to-end deadlines. This requirement can be checked using response time analysis developed for distributed real-time systems, e.g. the analysis method and tool presented in [18].

4 Related research

COMDES-II is a follow-on version of *COMDES-I* [12]. It employs an actor-based system model, whereby actors are conceived as units of concurrency as well as functionality (e.g., sensor, controller, actuator, etc.), whereas in the previous version a system is composed from function units encapsulating multiple threads of control. It also incorporates a different, i.e. composite state machine model emphasizing the separation of reactive and transformational (signal-processing) behaviour.

In *COMDES-II*, system operation is described by the Distributed Timed Multitasking (DTM) model of computation, which has been inspired by the original *Timed Multitasking* model [5] and is similar to the LET model adopted in the *xGiotto* language [6]. However, both of these models use port-based communication between actors, whereas DTM employs broadcast communication with labeled state messages (signals). This solution rules out artifacts such as ports, message queues, mailboxes, operational interfaces, etc., and provides for transparent interactions that are independent of the allocation of the actors on network nodes. Furthermore, the above frameworks use flat actor models with actors programmed in a conventional fashion, whereas *COMDES-II* actors are configured from prefabricated *executable* components – function blocks.

The adopted communication mechanism is characterized by complete separation of computation and communication, as recommended in [9], since signal drivers are executed in separation from actor tasks and from each other. That is not the case with port-based objects, where ports are usually defined as communication objects whose methods are invoked within task I/O drivers in a conventional call-return manner, see e.g. [5]. Consequently, the communication pattern is ‘hardwired’ in the code of I/O drivers and cannot be reconfigured without reprogramming.

The presented model of computation bears certain similarities with the models used in synchronous languages [20], and in particular: atomic execution of input and output actions; clocked operation similar to the execution pattern used in *LUSTRE* and *SIGNAL*; compositional data flow models inspired by the Control Engineering domain.

At the same time, there are substantial differences that have to be highlighted in order clearly differentiate the two models:

- Synthetic, component-based approach using prefabricated executable components vs. a conventional language-based approach used in synchronous languages
- True actor-level concurrency vs. conceptual concurrency, which is 'compiled away' during program compilation
- Constant *non-zero* reaction time vs. instantaneous (zero-time) reaction assumed by perfectly synchronous systems.

The last feature facilitates the engineering of distributed systems and eliminates problems related to fixpoints, instantaneous loops, etc., which have been major issues with synchronous systems. Furthermore, the synchronous model does not address the problem of task and transaction jitter because of the very nature of the synchrony hypothesis, whereas it is practically eliminated with the *COMDES* model of distributed computation.

5 Conclusion

The paper presents the formal specification of *COMDES-II* - a domain-specific framework for distributed embedded control systems, which combines open architecture and predictable behaviour under hard real-time constraints. The framework employs a hierarchical system model combining the concepts of both *actor* and *function block*: an embedded system is composed from autonomous system agents (actors), which are configured from prefabricated executable components – function blocks. Actors interact by exchanging signals, i.e. labeled messages with state message semantics, rather than using I/O ports or operational interfaces. This feature facilitates system reconfiguration and provides for transparent communication between actors, resulting in flexible and truly open distributed systems. Signal-based communication is also used for internal interactions involving constituent function blocks. That is why system configuration is specified by *data flow models* at all levels of specification. Consequently, actor behaviour is represented as a composition of component functions, and system behaviour – as a composition of actor functions. A synchronous model of computation is applied at the component level. A clocked synchronous model of execution is applied at the actor and system levels, i.e. Distributed Timed Multitasking.

The presented software architecture has important implications for software safety and predictability, as well as the entire software development process. In this case, applications are configured from prefabricated and validated (*trusted*) components, following strict composition rules that are derived from the syntax and static semantics of the framework. The behaviour of software components and applications is rigorously specified via a hierarchy of formal models that constitute the behavioural semantics of the framework. On the other hand, the use of timed multitasking makes it possible to engineer highly predictable systems operating in a flexible, dynamic scheduling environment.

This has been demonstrated in a number of experiments used to validate the framework, e.g. distributed computer control systems involving physical and computer models of plants, such as electric DC motor, production cell, steam-boiler, turntable machine, etc. It has also been applied in an industrial case study - a medical ventilator control system [17]. In all cases, the use of the framework helped reduce development time and increase software quality. This was quite obvious with some of the systems mentioned above, e.g. the production cell control system, which was developed in a relatively short time and became operational without extensive testing and debugging.

However, in order to guarantee that an application is correct by construction, it has to be proven correct with respect to the required functional and timing behaviour. That is only possible if a precise and unambiguous system model is developed, whose particular features would desirably facilitate the process of analysis. In *COMDES-II* that is accomplished through formal design models emphasizing the principle of *separation of concerns*, i.e. separate treatment of computation and communication, functional and timing behaviour, reactive and transformational behaviour, etc. Thus, different aspects of system behaviour can be verified in separation using appropriate techniques and tools. Functional behaviour can be analyzed using tools such as *Simulink* (with continuous systems) and *Uppaal* (with discontinuous systems), following semantics-preserving transformation of system design models into the corresponding analysis models, whereas timing behaviour can be verified through numerical response-time analysis.

In particular, *Simulink* can be used to analyse system behaviour via simulation. That is facilitated by the similarity between *COMDES-II* design models and *Simulink* analysis models representing the controller part of the system, both of which are discrete-time data flow models. Consequently, it is possible to export a *COMDES-II* design model to the *Simulink* environment, by wrapping *COMDES-II* components into S-functions and wiring them together, following the interconnection pattern of the original design model. This analysis method has been successfully experimented with the medical ventilator case study, whereby the *COMDES-II* design of the control system has been exported to *Simulink* and subsequently validated via numerical simulation.

The envisioned development process will make it possible to engineer embedded applications that are *correct by construction*. This will hopefully eliminate design errors, which are difficult and costly to repair. On the other hand, implementation errors will be eliminated through an automated configuration process supported by an integrated toolchain [16], which is based on meta-models that have been derived from the formal design models presented in this paper. Ultimately, the elimination of both design and implementation errors will considerably enhance software safety, which is of paramount importance for the overall safety of embedded applications.

6 References

1. B. Bouyssounouse and J. Sifakis (Eds.), "Embedded Systems Design. The ARTIST Roadmap for Research and Development", *LNCS 3436* (2005)
2. T.A. Henzinger and J. Sifakis, "The Embedded Systems Design Challenge", Proc. of the 14th International Symposium on Formal Methods FM 2006, *LNCS 4085* (2006), pp. 1-15
3. P. Caspi, "Some Issues In Model-Based Development for Embedded Control Systems", Invited Lecture, DIPES'2006, Braga, Portugal, Oct. 2006

4. D.B. Stewart, R.A. Volpe and P.K. Khosla, "Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects", *IEEE Transactions on Software Engineering*, vol. 23, No 12, 1997, pp. 759-776
5. J. Liu and E.A. Lee, "Timed Multitasking for Real-Time Embedded Software", *IEEE Control Systems Magazine: Advances in Software Enabled Control*, Feb. 2003, pp. 65-75
6. A. Ghosal, T.A. Henzinger, C.M. Kirsch and M.A. Sanvido, "Event-Driven Programming with Logical Execution Times", Proc. of HSCC 2004, *LNCS 2993* (2004), pp. 357-371
7. D. Isovic and C. Norström, "Components in Real-Time Systems", Proc. of the 8th International Conference on Real-Time Computing Systems and Applications RTCSA'2002, Tokyo, Japan, March 2002
8. H. Hansson, M. Åkerholm, I. Crnkovic and M. Törngren, "SaveCCM – A Component Model for Safety-Critical Real-Time Systems", Proc. of the 30th EUROMICRO Conference on Software Engineering and Advanced Applications SEAA 2004, pp. 627-635
9. A.L. Sangiovanni-Vincentelli and G. Martin, "Platform-Based Design and Software Design Methodology for Embedded Systems", *IEEE Design and Test of Computers*, vol. 18 (2001), pp. 23-33
10. K.H. John and M. Tiegelkamp, IEC 61131-3: *Programming Industrial Automation Systems*, Springer, 2001
11. R. Lewis, *Modeling Control Systems Using IEC 61499*, Institution of Electrical Engineers (2001)
12. C. Angelov, K. Sierszecki, N. Marian and J. Ma, "A Formal Component Framework for Distributed Embedded Systems", in I. Gorton et al. (Eds.): Proc. of CBSE 2006, *LNCS 4063* (2006), pp. 206-221
13. C. Angelov, X. Ke and K. Sierszecki, "A Component-Based Framework for Distributed Control Systems", Proc. of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications SEAA 2006, Cavtat, Dubrovnik, Croatia, Aug.-Sept. 2006, pp. 20-27
14. K. Sierszecki, C. Angelov and X. Ke, "A Run-Time Environment Supporting Real-Time Execution of Embedded Control Applications", Proc. of the 14th International IEEE Conference on Embedded and Real-Time Computing Systems and Applications RTCSA 2008, Kaohsiung, Taiwan, Aug. 2008
15. C. Angelov, X. Ke, Y. Guo and K. Sierszecki, "Reconfigurable State Machine Components for Embedded Applications", Proc. of the 34th EUROMICRO Conference on Software Engineering and Advanced Applications SEAA 2008, Parma, Italy, Sept. 2008, pp. 51-58
16. Y. Guo, K. Sierszecki and C. Angelov, "COMDES Development Toolset", Proc. of the 5th International Workshop on Formal Aspects of Component Software FACS 2008, Malaga, Spain, Sept. 2008, pp. 233-238
17. F. Zhou, W. Guan, K. Sierszecki and C. Angelov, "Component-Based Design of Software for Embedded Control Systems: the Medical Ventilator Case Study", Proc. of the International Conference on Embedded Software and Systems ICESS 2009, Hanchzhou, China, June 2009
18. W. Henderson, D. Kendall and A. Robson, "Improving the Accuracy of Scheduling Analysis Applied to Distributed Systems", *Real-Time Systems*, vol. 20, No 1 (2001), pp. 5-25
19. A. Jantsch, *Modeling Embedded Systems and SoCs - Concurrency and Time in Models of Computation*, Morgan Kaufmann, 2003
20. A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic and R. de Simone, "The Synchronous Languages 12 Years Later", *Proc. of the IEEE*, vol. 91, No 1, Jan. 2003, pp. 64-83