

A Formal Model of IEC 61499-Based Industrial Automation Architecture Supporting Time-Aware Computations

DMITRII DROZDOV ¹, VICTOR DUBININ², SANDEEP PATIL ¹ (Member, IEEE),
AND VALERIY VYATKIN ^{1,3} (Senior Member, IEEE)

¹ Luleå University of Technology, 97187 Luleå, Sweden

² Penza State University, Penza 440026, Russia

³ Aalto University, 02150 Helsinki, Finland

CORRESPONDING AUTHOR: DMITRII DROZDOV (e-mail: dmitrii.drozdoz@ltu.se)

This work was supported in part by Swedish Project DIACPA (Vetenskapsrådet dnr 2015-04675) and in part by the H2020 Project 1-SWARM, funded by the European Commission under Grant Agreement 871743.

ABSTRACT This paper proposes a formal model for industrial cyber-physical systems (CPS) with distributed control based on IEC 61499 standard and supporting time-aware computations for better adaptation to the ever changing environment conditions. Main features of the model include usage of timestamps, flattening, unified and independent behaviour of function block interfaces. This allows to make correct implementation of time-aware systems and significantly simplify the construction of models for verification and simulation, as well as ensure fairness of the model and determinism of the function block system execution at a resource level. The model formalism is based on a well-known abstract state machines (ASM) notion, which can be used as an intermediate formal representation to generate a variety of models for different purposes, e.g. formal verification, and executable code. This paper exemplifies this approach by the generation of models in the SMV language. The paper discusses the time-aware computation concept and its application in a few related case studies.

INDEX TERMS Abstract state machines, CPS, formal semantics, formal verification, IEC 61499, time-aware computations.

I. INTRODUCTION

Distributed automation systems is a vast class of cyber-physical systems (CPS) [1], providing many challenges for the CPS design and analysis. Formal models of computations [2] provide the underlying support for their implementation, testing, simulation and formal verification.

The automation CPS are subject to the environment uncertainties, the strongest of which is related to reliability of wireless communication. Algorithms, dealing with the uncertainties, require comprehensive validation. Model-based formal verification of software is a well-known mechanism for this purpose. To be applied efficiently, it requires creation of tool-chains seamlessly connecting the design models with the formal models of computations.

The use of block diagram languages for implementation of CPS is becoming a mainstream trend. The well-known

examples of that include Matlab/Simulink and LabView. The modelling environment Ptolemy II [3], developed at Berkeley, provides new modelling capabilities for heterogeneous CPS, combining complex physical system dynamics and cyber parts, based on different models of computations.

In industrial automation, the function block architecture of the IEC 61499 standard [4] is increasingly used for modelling complex distributed automation systems in such challenging applications as SmartGrid [5], process automation, and material handling systems [6]. IEC 61499 is based on the same concepts of event-driven block diagrams as Ptolemy II and, as such, allows for modelling of CPS composed of physical processes (a.k.a. plant) combined with control and communication. The model can be used for validation of system-level properties before deployment. The most common validation method is simulation, but it has well-known limitations in

the discovery of possible faults. Model-checking [7] presents a complementary method of exhaustive testing, however its applicability is also limited by possible complexity explosion when the original system uses rich data-types and computations. A recent survey [2] presents a summary of formal methods application in industrial automation. In the past 15 years of research certain progress [8], [9] was achieved in formal modelling of the IEC 61499 architecture, but it was limited by support of very simple data types, non-timed semantics and small scale systems. In the meantime the power of model-checking tools and sophistication of supported model-checking techniques has substantially improved and new opportunities emerge.

The notion of *Abstract State Machines* (ASM), proposed by Yu. Gurevich [10], has been proven during the past three decades as an efficient approach for the formal specification and analysis of computer hardware and software. It has been used in numerous modeling and verification projects, and later was introduced into development environments for practical usage in high-level system design and analysis [11].

The ASM can be used as an abstract modelling language for high-level system design and analysis [12], model-checking, and especially effective when different analysis and validation techniques may be applied to the same model. Various model-checking techniques have been applied to ASM models, starting with symbolic model-checking using the SMV tool, first proposed in [13] and later developed in [14], [15], and explicit state model-checking with SPIN [16], [17].

The Ptolemy II/PTIDES model of computations ensures execution determinism under an assumption that there is a known and rather small upper bound on network transmission delay. However, in distributed control systems the absolute upper bound is very large, while the average delay time stays reasonably small. This can happen when an occasional environment disturbance interferes with transfer of rather small number of messages. In this case, the solution of the PTIDES model to wait for the maximum time to process an event may lead to significant loss in control system's performance.

In this paper we propose a different idea, called *time-aware computations* (TAC). Instead of aiming at determinism, that is very expensive in distributed systems, it aims at adaptability and robustness. It is based on the same event-timestamping mechanism as PTIDES, but is intended to let the developer to handle each delay case individually thus minimizing its impact on functional properties of the automation system. It allows the controller to take into account actual point-point delay of the measured sensor readings, and adjust the control reaction accordingly, instead of trying to put the upper bound on it and wait for the maximum possible delay time.

We propose a comprehensive ASM-based formal model of the IEC 61499 function block architecture and apply it to the architecture extended with the event-timestamping mechanism of PTIDES. The developed model is an intermediate formal model representation, which can be used for the purposes of formal verification and as an underlying model of

run-time implementation. In particular, it is the back-end of the developed open-source software tool fb2smv.

The rest of the paper is organized as follows: Section II discusses the concept of time-aware computations and related works. Sections III and IV present the core original contribution: Section III gives the basic description of the model and Section IV defines core formal semantics for FB systems with timestamps. In Section V, SMV models and model-checking is discussed. Section VI combines several use-cases illustrating the applicability range of the proposed concepts and models. Finally, the Section VII concludes the paper.

II. TIME-AWARE COMPUTATIONS: THE CONCEPT AND RELATED WORK

We define *time-aware computations* as a set of methods, approaches and design patterns to create distributed control system software that can take advantage of precise information of the timing of computations and communications. The idea of TAC could be partially attributed to the precision time architecture (PRET) [18], [19], which was introduced to achieve timing repeatability at one computational node. On the contrary, in TAC we target distributed systems. In the following, the main mechanism for achieving TAC is timestamping of events, while in PRET it is achieved by taking into account timing of every micro-operation and calculating the worst-case execution time.

The main purpose of TAC in distributed automation is to achieve robustness of the distributed system behaviour in presence of various disturbances. One such common disturbance in the modern automation CPS is related to the jitter of wireless communication, which is becoming increasingly important in the automation context.

The works, enabling TAC, can be divided to several groups as follows.

A. DISTRIBUTED NETWORKED CONTROL SYSTEMS

The progress in distributed networked control systems provides control methods adjusted to the distributed systems setting. A cornerstone in the evolution towards distributed control systems was the introduction of event-based control [20], [21]. Prior to that, an earlier study [22] discussed issues of decentralized control systems with yet central control, but sensors and actuators connected over communication network and addressed questions of modelling random time delays that occur in communication networks.

B. IEC 61499 ARCHITECTURE

IEC 61499 was introduced as a system-level architecture for distributed automation systems, extending the software model of popular programmable logic controllers (PLC), known as IEC 61131-3 standard, with the means of describing complex distributed systems composed thereof.

The central structural unit of the IEC 61499 architecture is function block. As shown in Figure 1, function blocks have clearly defined interfaces of event and data inputs and outputs. Event inputs are used to activate the block. There are basic,

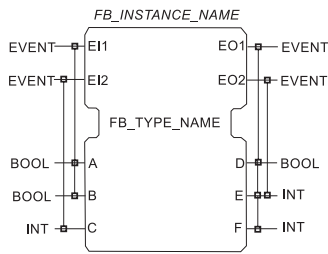


FIGURE 1. Interface of a function block.

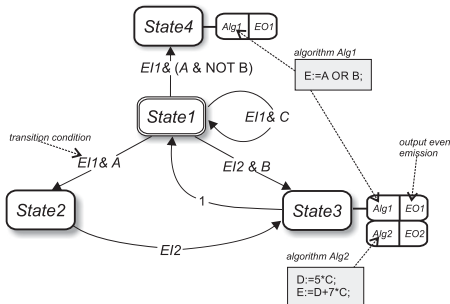


FIGURE 2. ECC: a state-machine, defining the behaviour of a basic function block.

composite and service-interface kinds of function blocks. A basic function block may have internal variables which are fully protected, i.e. not directly accessible from outside. As a result of internal computations the block may change output data variables and emit output events, which, if connected to event inputs of other blocks, will activate them.

The behaviour of a basic function block is determined by a state machine, called execution control chart (ECC), illustrated in Figure 2. Semantically, ECC is equivalent to a Moore type finite automaton. States of ECC can have associated actions, each consisting of invocation of an algorithm and emission of an output event. Algorithms can be programmed in different programming languages even within a single basic FB. Thus, basic FBs can be regarded as a portable abstract model of a single controller.

Function block instances can be connected one with another by event and data connection links, thus forming function block networks. The connections define control and data flow between FB instances thus determining the network’s execution semantics. FB networks are seen as a universal model of control systems, both distributed and centralized. Functionality of composite function blocks is defined by a network of function block instances, some of which can also be composite. This can lead to a hierarchical structure of applications, as illustrated in Figure 3 for the elevator control application, which will be discussed in more detail in Section VI-A.

In distributed systems FB instances included in a network can be regarded as independent processes. Communication between them is abstractly modelled by event and data passing.

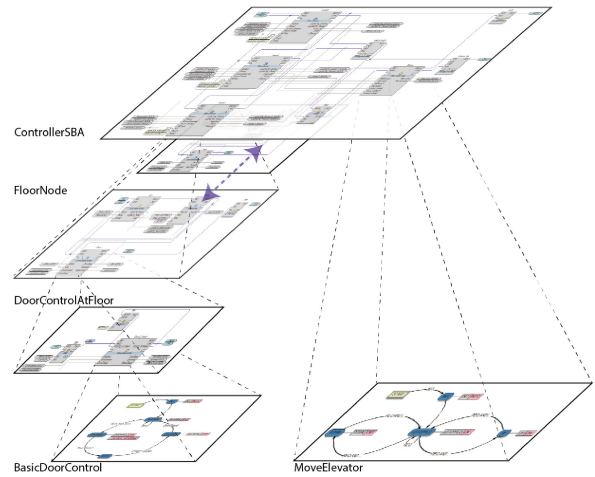


FIGURE 3. Hierarchical structure of an application in IEC 61499.

The IEC 61499 standard includes a mechanism to add more details to the abstract FB network model of a system. Application’s FBs can be allocated to distributed devices, and communication FBs inserted whenever event or data connections cross borders of devices.

The event-based communication of FBs in IEC 61499 can be seen as a mechanism making the behaviour of applications partially agnostic to the underlying distributed hardware architecture. This mechanism, which can be seen as an abstraction of message passing, preserves causality of events regardless of internal execution specifics of devices, where parts of the application are executed.

For more details on IEC 61499, we refer the reader to the proper introductory material, such as the book [23].

C. IEC 61499 FUNCTION BLOCKS FORMAL MODELLING AND OPERATIONAL SEMANTICS

The IEC 61499 design architecture for distributed automation appeared to improve determinism and reduce integration and reconfiguration effort thanks to the event-driven execution mechanism. The need to prove some properties of distributed automation systems rigorously has led to the development of the corresponding formal execution semantics. The international standard IEC 61499 for distributed automation defines an event-driven control architecture and a block-based programming language aimed to fulfill new demand for fully distributed and flexible control systems for Industry 4.0. The first version of the standard was released in 2005 with updated second release in 2012 [4]. While the standard itself well defines language syntax, tool requirements, compliance profiles, etc., there is still a need for strict mathematically-defined notation, which can be used as a basis to generate various models and implementations.

The first attempt to model IEC 61499 formally was presented in [24] by means of a Petri-net derivative called net condition-event systems. A good summary of early works on IEC 61499 modelling and verification can be found in [8].

An attempt to present a formal operational semantics of IEC 61499 is the work [25] by Yoong *et al.* This approach is, however, limited by the fact that the semantics was based on the synchronous paradigm, similar to Esterel. A semantic characterization of PLC programs based on extended λ -calculus appears in [26], leading towards theorem proving and model checking.

In [27] timed-models of plants and closed-loop automation systems are built as constrained timed discrete event systems and timed controller models respectively. These implementation-independent models are subsequently transformed into timed automata for verifying timing requirements such as urgency. Several works transform system artifacts into format accepted by formal tools. In [28], IEC 61499 systems are modelled as finite state machines, contracts or modules in synchronous languages, subsequently allowing the use of model checkers. Many algorithms use the UPPAAL model checker to verify timeliness. In [29], IEC 61499 systems are transformed into timed automata using set translation rules for verifying timing requirements. However, hardware configurations that affect a system's timing characteristics are not considered. Works [30]–[32] look at formally specifying buffered sequential execution model (BSENI) of the Fuber IEC61499 runtime. They also presents extended finite automata models that are suitable for formal verification of the proposed execution semantics.

The works [33]–[35] give the first description of FB formal model with ASM. This model can be (and is) used as a basis to build applied models for formal verification and simulations. For example, various works by the authors of this paper on SMV modelling and formal verification of IEC 61499 function blocks [36]–[38] and the fb2smv converter tool [39] are using underlying elements of the ASM-based formal semantics.

D. CPS MODELING WITH TIMESTAMPS

The concept of event timestamping was originally developed in research on determinism of cyber-physical systems. Cyber-physical system concept can be described as integration of tightly connected computer system and physical process where physical process can affect computations and vice versa [40].

A large research was done by Lee *et al.* on cyber-physical systems concept in general and distributed software [41], leading to creation of PTIDES [42] model of computation and Ptolemy II software framework for distributed CPS modelling and simulation. As mentioned in Section III-A, semantic basis and its implementation were suggested by Dai *et al.* [43], however, the authors did not consider the time-aware computations concept.

E. CYBER-PHYSICAL AGNOSTICISM

Possible enhancement of PTIDES model in application to IEC 61499-based control systems was first proposed in [44]. Instead of full determinism in exchange to control systems

performance derived from PTIDES model, a different approach was proposed: to process messages (events) immediately when they received, but take actual communication delay into account in the control algorithm. Instead of determinism in PTIDES model, this property was called cyber-physical agnosticism or CPA in short.

From the perspective of control theory, distributed control systems were studied in the late 1990-th by J. Nilsson *et al.* [22], [45], addressing the problems of optimal controller generation and control system modelling and analysis for distributed configuration with random delays. However, that work was focused on the configuration with centralized controller and remote sensors and actuators (in low-latency networks). It does not consider the case where the controller itself is distributed across several nodes.

III. MODEL BASICS

The semantic description of IEC 61499 extended with timestamps, presented in this section, is meant to be used as a formal basis for different models in implementation, validation and formal verification of IEC 61499-based control systems. The model is designed in a way to keep a certain level of simplicity in order to address the state space explosion problem widely observed when using formal verification techniques, preserving, at the same time, the crucial semantic features.

A. FLATTENED FB MODEL

As illustrated in Figure 3, a composite function block, or an IEC 61499 application can have hierarchical organisation due to the fact that composite function block definitions can include instances of other composite function blocks.

As for composite function blocks, their execution model has not been fully clarified in the standard. According to [46], there are two possibilities: to execute a composite FB as an entity or as a transparent container, i.e. a flattened FB network. In this work, we follow the transparent container model and further specify its implementation: our flattening approach, based on [47], [48], uses the concept of data valves, preserving the semantics of interfaces (the event-data associations and data latches). Events and data in our model are transferred through interface border in and out immediately when they appear.

A flat FB system consists of the following elements:

- Basic function blocks (BFB)
- Service interface function blocks (SIFB)
- Composite FB input interfaces
- Composite FB output interfaces

Fig. 4 shows an example of the flattened model structure of a composite FB “quad” on the top level, that was composed of the instances “tw1” and “tw2”. Each of the latter is also a composite function block, composed of two instances of the basic function blocks INC and DEC respectively.

Basic function blocks in our model are executed as atomic units, that is while a BFB is being executed, no other actions are possible (except buffering of input events from SIFBs),

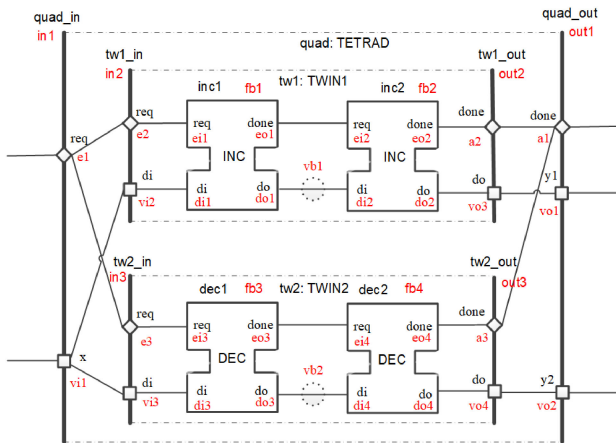


FIGURE 4. Flattened FB system.

therefore a basic FB execution time should be as short as possible.

The key difference of the CFB model from another IEC 61499 structure, called *subapplication*, is that events and data are still bound together by means of the WITH-associations, while in a subapplication events and data are always transferred independently. The WITH association between event and data interface elements of a function block determines that the data element is sampled only when the associated event occurs.

The presented modelling approach sets certain restrictions on FB systems that can be modelled, as follows:

- 1) A data input of one FB can be connected to only one data output of another FB. (Event connections are free of such restrictions and implicit usage of the E_SPLIT and E_MERGE standard function blocks is assumed for splitting and merging events respectively.)
- 2) Each event or data *input* of a component FB can be connected to only one input interface.
- 3) Each event or data *output* of a component FB can be connected to only one output interface.

Most of these restrictions do not limit the generality as they follow the rules of IEC 61499.

While the proposed model for composite function blocks does not support the IEC 61499 principle of “run to completion,” it can be further enhanced with this feature, but on account of some complications. One should note that different implementations of IEC 61499 treat the “run to completion” principle differently and the standard itself does not elaborate on this side of the composite FB execution model.

B. MODELLING TIMESTAMPS

The IEC 61499 model, introduced in this paper, uses event timestamps as proposed in [49] and further elaborated in [43], [50]. This mechanism is not yet a part of the standard, but is already supported by some of the implementation tools, such as IDE Neptune [51] and is considered to be a useful extension

for its ability to check timing guarantees and implement robust time-aware behaviour.

All input and output events of function blocks are loaded with *timestamps*. A timestamp is determined by two variables: the event *birth* time (we will further refer to it as TB-timestamp) and the event *last update* time (further referred as TL-timestamp). Intuitively, an event *birth* time is the system time, when the event was created by the event-producing FB and the *last update* time is when the event was last processed by a function block (or passed through an interface) firing a consequent event. The event birth time is created either by an event-producing SIFB, or by a user-defined event creating BFB. The closest to the above semantics and implementation was proposed by Dai *et al.* [43], however, the paper does not give a complete semantic description of FB systems and does not consider a possibility to use timestamp information in control logic.

To ensure a fully deterministic execution of a FB system (in a single resource), besides the timestamps, we also use the principle of unique priorities of execution elements. For example, in the whole system there should not be two input events (of any FBs) with equal priority. This is achieved by using two priority levels: a) unique FB priorities in the system and b) unique local priorities of event inputs/outputs within a single FB or interface. Priorities are taken into account iff timestamps of two or more active events are equal. The timestamps are meant for received events at the time of evaluation.

C. MODELLING EVENT SCHEDULER

The formal model definition consists of two parts: model schema and model dynamics. The model schema describes the static part of the model (including basic syntactic constructs, variables and common functions). The model dynamics is described by the rules of state variables (and functions) change.

In addition to the earlier mentioned structural restrictions for the FB system, the model-level abstraction is used to improve model-checking performance. The abstractions are based on the following assumptions:

- The model does not take into account clock synchronization error on different devices. It is assumed that the error is handled on the application level and not by the execution semantics.
- The model considers only FB application execution within a single resource.
- FB execution in a resource is always sequential. Resources (or devices) run separately and independent from each other.
- The model uses a discrete time notion, where time domain $\mathbb{T} = \{undef, 0, 1, 2, \dots\}$ is a set of discrete time values in the system. *undef* means that time is not defined or non-important.

However, too much independence in execution of composite FBs can lead to a behavior unwanted in some execution models, for instance:

- A composite FB could be triggered when not all its components finished executions (CFB reenterability).

- A component FB inside a CFB could be executed multiple times.
- Execution of component FBs from different CFB instances could interleave.

It is assumed that a resource includes a *scheduler* that controls execution of FBs. Execution of the FB system components in a single resource is always sequential (since a resource is usually implemented as non-preemptive to ensure deterministic behaviour) [52].

The scheduler is built upon a chronologically ordered event queue (events in a queue are ordered by TL-timestamps). The *scheduling elements* in the queue are events and *execution elements* are FBs and interfaces.

For the sake of model's simplicity, we will use an *implicit* event queue in the model instead of explicit queue in runtime implementations. An implicit queue is formed as a set of events buffered on event inputs of BFBs, SIFBs and CFB interfaces. There is a chronological order relation defined on this set. In case of implicit event queue, the elements are not bound into a list. When describing the work of the scheduler, it is convenient to use a queue of execution elements, i.e. FBs and interfaces, instead of a queue of events. It is possible to uniquely map the events to their associated execution elements.

For the reference, the listing below shows an abstract algorithm of a possible scheduler implementation. During the event transfer, we need to consider not only the FB that has just finished execution, but also the event-producing SIFBs. We assume that the need precautions are taken to insert the newly produced events into the event queue safely, asynchronously and concurrently to the scheduler algorithm.

```

scheduler = while execQueue not empty do
  choose  $x \in \text{execQueue}$  :
     $x$  has a minimum timestamp and
    the highest priority among other elements
    with the same minimum timestamp (if any)
     $p_m(x)$ 
  (1)

```

where $p_m(x)$ is the execution rule for the corresponding schedulable element x (e.g. a BFB). However, keeping in mind the performance of model-checking, we do not use explicit scheduler in the model, but instead the active event selection is done by special predicates directly in the execution rules.

Concluding the above, we can summarize main differences between formal model and actual implementations:

- 1) Model uses implicit event queue, while implementations use explicit event queue;
- 2) Model does not have an explicit scheduler, instead, it uses predicates for selection of an active event among timed prioritized events;
- 3) Model is based on sets, functions and ASM rules, while a real implementation uses FB artefacts and computations based on them.

Model operation is based on transitions from one model state to another according to certain ASM rules. This can be called a *rule-driven execution*, when the model operation is defined by execution of enabled ASM rules.

D. BASIC NOTATION

A model of a flat FB system S is formally defined as

$$M_S = (\text{Synt}_S, \text{Sem}_S) \quad (2)$$

The syntactic part is what is defined by the IEC 61499 abstract syntax and can be described as a tuple:

$$\text{Synt}_S = (FB_S, IN_S, OUT_S, PR_S, EVCONN_S, DCONN_S) \quad (3)$$

where FB_S is a set of component FB instances in the system (only BFB and SIFB, composite FBs are unfolded into interfaces and connections by flattening, as discussed in Section III), IN_S is a set of input CFB interface instances, OUT_S - set of output CFB interface instances, PR_S is a set of priorities, $EVCONN_S$ and $DCONN_S$ are sets of event and data connections in the system.

We call component FBs, input and output interface instances - *execution elements* and define a set $EL_S = FB_S \cup IN_S \cup OUT_S$ for this purpose. Every execution element has its unique priority from the PR_S set. A priority of an element x in the system is referred as $prior_S(x)$, where $prior_S : EL_S \rightarrow PR_S$.

The semantic part of the system description will be:

$$\text{Sem}_S = (EI_{FB}, EO_{FB}, EI_{IN}, EO_{OUT}, VI_{FB}, VO_{FB}, VI_{IN}, VO_{OUT}, \tau) \quad (4)$$

First eight elements of the tuple are event and data inputs and outputs of BFBs/SIFBs and CFB interface instances. (Different lower indexes FB , IN and OUT indicate subsets of component function blocks, input interfaces and output interfaces correspondingly) τ is a monitored 0-ary function, returning system (resource) clock value (in ASM model, the term "monitored" means the function's change is not defined by ASM rules, but read (monitored) from external environment, e.g. user input or external simulator). For all data variables we define a universal value function $Val : VAR_S \rightarrow VAL$, where $VAR_S = VI_{FB} \cup VO_{FB} \cup VI_{IN} \cup VO_{OUT}$ is a union of all sets of data variables in the model and $VAL = \mathbb{B} \cup \mathbb{I} \cup \mathbb{F}$ is a domain including all possible values (in this model we limit it to Boolean (\mathbb{B}), integer(\mathbb{I}) and float(\mathbb{F}) data types). Changes to variables are determined by the ASM rules, which are listed below in the section IV.

Generally, most of the actions performed in the model are delegated to the *execution elements*. On the top level it makes sense to define just the rules for active event selection and some basic commons.

First, we define three functions over an event $e \in EV_S$ ($EV_S = (EI_{FB} \cup EO_{FB} \cup EI_{IN} \cup EO_{OUT})$):

$Fired : EV_S \rightarrow \mathbb{B}$ defines whether event e is fired or not;

$Tb : EV_S \rightarrow \mathbb{T}$ is a function assigning TB-timestamp of the event e ;

$Tl : EV_S \rightarrow \mathbb{T}$ is a function assigning TL-timestamp of the event e .

The main predicate to be used for active event selection is as follows:

$$IsActive(e) = (Fired(e) \wedge Tl(e) = MinTl) \wedge prior(get_fb(e)) = MaxPrior(MinTl) \quad (5)$$

where $get_fb(e)$ is an execution element (in this case it is a FB) which correspond to event e ;

$MinTl$ is an 0-ary function showing current minimal TL timestamp value in the model;

$MaxPrior(MinTl)$ is the maximum priority of execution elements that have a minimum timestamp (in the current state);

Model dynamics on the top level is defined by the model's main rule, which is parallel execution of all its components' main rules:

$$main = \text{for all } x \in EL_S \text{ do } p_m(x) \quad (6)$$

When defining the ASM model of a FB system, static, derived, and dynamic functions are used. The derived functions are those coming with a specification or computation mechanism given in terms of other functions.

IV. FORMAL SEMANTICS WITH TIMESTAMPS

A. BASIC FB

1) MODEL SCHEMA

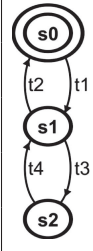
A basic function block model can be formally presented as $M_{FB} = (Synt_{FB}, Sem_{FB})$. As in [34], we define the syntactic part as follows:

$$Synt_{FB} = (Interface, ALG, IV, ECC, Val_0, t) \quad (7)$$

It consists of the interface, a set of algorithms (ALG), execution control chart (ECC), a set of internal variables(IV), a set of initial values (Val_0). We modify the basic semantic definition, given by the IEC 61499 standard, adding extra element t , which represents the last active event's TB-timestamp.

A basic function block interface is a tuple $Interface = (EI, EO, VI, VO, IW, OW)$, where $EI \subset EI_{FB}$ is a set of event inputs of the basic function block, $EO \subset EO_{FB}$ is a set of event outputs, $VI \subset VI_{FB}$ is a set of data inputs, $VO \subset VO_{FB}$ is a set of data outputs, IW and OW are sets of WITH-connections for the basic FB input and output interface correspondingly. For every WITH-connection $iw \in IW$ we define two functions: $Event : IW \rightarrow EI$ and $Data : IW \rightarrow 2^{VI}$. Result of $Event(iw)$ is a single event $e \in EI$ associated with iw ; $Data(w)$ returns a subset $[VI] \subseteq VI$ of associated data inputs. In a similar way we define $Event : OW \rightarrow EO$ and $Data : OW \rightarrow 2^{VO}$ for $ow \in OW$. We used the polymorphism principle defining the functions $Event(iw)/Event(ow)$

TABLE 1. ECC Operation State Machine



State		Operations
s0		(wait for input event)
s1		Evaluate transitions
s2		Perform actions
transition	Condition	Operations
t1	Input event occurs	Sample inputs
t2	No transitions clear	
t3	A transition clears	
t4	Actions completed	

and $Data(iw)/Data(ow)$, however, in certain implementations they can be just named differently, for example, using suffix notation.

As discussed in [34], the functioning of a Basic Function Block is defined by means of ECC. IEC 6499 ECC Operation State Machine (OSM) defines an internal state machine for ECC transition behavior. Table I below shows briefly the OSM behavior.

One must note that the set-theoretical ASM notation is quite complicated. To make the model presentation better readable and usable, as compared to [35], we follow a higher level of abstraction (without losing rigorosity) and switch (where it makes sense) from the notation of runtime variables and value update functions to abstract ASM functions, so we can produce more elegant and compact description.

The semantic part of the basic FB model is defined as a tuple of functions:

$$Sem_{FB} = (Ecc, Osm, Na, Ni, Buffer) \quad (8)$$

where $Ecc : FB_S \rightarrow ECSTATES$ is a function representing the ECC state for the function block instance fb , where $ECSTATES$ is a set of ECC states; $Osm : FB_S \rightarrow OSMSTATES$; $OSMSTATES = \{s_0, s_1, s_2\}$ - function representing OSM state for the function block instance fb .

$Na : FB_S \rightarrow \mathbb{N}$ is the ECC action pointer, showing the number of executing ECC action, where $\mathbb{N} = \{0, 1, 2, \dots\}$ is a set of non-negative integers. If $Na(fb) = 0$, then it is considered that no ECC actions are executing at the moment.

$Ni : FB_S \rightarrow \mathbb{N}$ is the algorithm step pointer if the basic FB $fb \in FB_S$. If $Ni(fb) = 0$, we consider that the algorithm execution is finished.

Also, since the input data has to be buffered in the BFB when there is an active input event, we define a function $Buffer : VI \rightarrow VAL$ which corresponds to the internal buffer value, associated with data input $d \in VI$, and we will use $Val : VI \rightarrow VAL$ to define the input (before buffering) value of $d \in VI$.

2) MODEL DYNAMICS

The Basic FB dynamics is defined by the following rule:

$$\begin{aligned}
 p_m^{FB}(fb) = & (p_{start}^{FB}(fb), p_{EvReset}^{FB}(fb), p_{OSM}^{FB}(fb), \\
 & p_{ECC}^{FB}(fb), p_{Ni}^{FB}(fb), p_{EvOut}^{FB}(fb), \\
 & p_{Finish}^{FB}(fb)) \quad (9)
 \end{aligned}$$

The rule is parameterised with fb , which means that for every $fb \in FB_S$ there should be executed an individual instance of the rule. p_m^{FB} on its own consists of several sub-rules responsible for changing specific state variables and functions for the function block instance fb .

The first sub-rule: $p_{Start}^{FB}(fb)$ is responsible for BFB instance execution start routine, selecting active input event and buffering data from input interface.

$$\begin{aligned}
 p_{Start}^{FB}(fb) = & \\
 & \text{with } e \in EI : \\
 & \quad isActive(e) \wedge Osm(fb) = s_0 \\
 & \text{do} \\
 & \quad t(fb) := Tb(e) \\
 & \text{forall } iw \in IW : Event(iw) = e \text{ do} \\
 & \quad \text{forall } d \text{ in } Data(iw) \text{ do} \\
 & \quad \quad Buffer(d) := Val(d) \quad (10)
 \end{aligned}$$

This rule is activated when the following conditions are met:

- 1) one of BFB event inputs is active;
- 2) BFB is “free” i.e. its OSM is in the state s_0 ;

When the rule is activated, the following actions are performed:

- 1) initialize “TB” timestamp variable with the value from the active event input;
- 2) sample the input data, associated with the active input event.

The next sub-rule: $p_{EvReset}^{FB}(fb)$ is responsible for resetting event buffers. When the OSM is in the state S_1 (which means that BFB has started the active execution phase) it resets the active event and the related timestamp buffers.

$$\begin{aligned}
 p_{EvReset}^{FB}(fb) = & \\
 & \text{forall } e \in EI : isActive(e) \wedge Osm(fb) = s_1 \text{ do} \\
 & \quad Fired(e) := false \\
 & \quad Tb(e) := undef \\
 & \quad Tl(e) := undef \quad (11)
 \end{aligned}$$

The operation state machine (OSM) is responsible for controlling basic function block’s operation state, i.e. executing data sampling, ECC transitions, algorithms, etc., in the correct order. It is defined by its state $Osm(fb)$ and transition rule p_{OSM}^{FB} . But before defining it, we need to define some auxiliary functions. First, we define a set of ECC transitions $ECTRAN$,

for each element $tr \in ECTRAN$ we need to individually define a guard condition $Guard : ECTRAN \rightarrow \mathbb{B}$ - is a derived function (predicate) to be used in ECC transition. Function $Ei : ECTRAN \rightarrow EI$ returns event $Ei(tr)$, associated with specific ECC transition tr (note that only one event can be associated with each transition). $Src : ECTRAN \rightarrow ECSTATE$ and $Dst : ECTRAN \rightarrow ECSTATE$ define source and destination ECC states correspondingly. Function $Enabled : ECTRAN \rightarrow \mathbb{B}$ will be used as a predicate to check if a specific ECC transition is enabled. It can be defined as follows: $Enabled(tr) = IsActive(Ei(tr)) \wedge Guard(tr)$.

We also define a function $ExistsEnabledEctran : FB_S \rightarrow \mathbb{B}$, such that $ExistsEnabledEctran(fb) = true$ if there is at least one enabled ECC transition in fb and $false$ otherwise. The function is parameterised with FB instance $fb \in FB_S$ which means it returns individual value for every FB instance. We do not use this parameter in the definition, but we assume that the transition set $ECTRAN$ belongs to the instance fb .

$$\begin{aligned}
 ExistsEnabledEctran(fb) = & \text{exist } tr \in ECTRAN : \\
 Enabled(tr) \wedge (Ecc(fb) = Src(tr)) \wedge (Ecc(fb) = Src(tr)) \quad (12)
 \end{aligned}$$

The OSM functioning is defined by the following rule:

$$\begin{aligned}
 p_{OSM}^{FB}(fb) = & \\
 & \text{if } (Osm(fb) = s_0 \wedge (\text{exist } e \in EI : \\
 & \quad isActive(e))) \text{ then} \\
 & \quad Osm(fb) := s_1 \\
 & \text{else if } (Osm(fb) = s_1 \wedge ExistsEnabledEctran(fb)) \text{ then} \\
 & \quad Osm(fb) := s_2 \\
 & \text{else if } (Osm(fb) = s_2 \wedge Na(fb) = 0) \text{ then} \\
 & \quad Osm(fb) := s_1 \\
 & \text{else if } (Osm(fb) = s_1 \wedge \neg ExistsEnabledEctran(fb)) \text{ then} \\
 & \quad Osm(fb) := s_0 \quad (13)
 \end{aligned}$$

OSM starts with transition from state s_0 to s_1 when there is an active input event. The OSM is transitioned to s_2 when its state is s_1 and there exists at least one enabled ECC transition, and returns to the state s_1 when all ECC actions are executed in the current ECC state. Finally, the OSM will return to s_0 from s_1 if there are no enabled ECC transitions.

The ECC functioning is defined by the following rule:

$$\begin{aligned}
 p_{ECC}^{FB}(fb) = & \\
 & \text{choose } tr \in ECTRAN : (Ecc(fb) = Src(tr) \\
 & \quad \wedge Enabled(tr) \wedge Osm(fb) = s_1) \\
 & \text{do } Ecc(fb) := Dst(tr) \quad (14)
 \end{aligned}$$

Every ECC state may have one or more ECC actions associated with the state. When OSM is in the state s_2 , these actions are executed in the order they’ve been defined. To

schedule execution of ECC actions, in the BFB model schema we defined a counter $Na(fb)$. Update of this counter value is done by the p_{Na}^{FB} sub-rule (15).

$$\begin{aligned}
 p_{Na}^{FB}(fb) = & \\
 & \text{if } Osm(fb) = s_1 \text{ then} \\
 & \quad Na(fb) := 1 \\
 & \text{elseif } Osm(fb) = s_2 \wedge Ni(fb) = 0 \wedge \\
 & \quad Na(fb) < MaxNa(Ecc(fb)) \text{ then} \\
 & \quad \quad Na(fb) := Na(fb) + 1 \\
 & \text{elseif } Osm(fb) = s_2 \wedge Ni(fb) = 0 \wedge \\
 & \quad Na(fb) = MaxNa(Ecc(fb)) \text{ then} \\
 & \quad \quad Na(fb) := 0
 \end{aligned} \tag{15}$$

When we say that an ECC action is executed, we actually mean execution of algorithm $alg \in ALG$ associated with this action and subsequent firing of associated output event. In BFB model schema we defined an algorithm step counter $Ni(fb)$. As before, we first need to define some auxiliary functions. A static function $MaxNa : ECSTATE \rightarrow \mathbb{N}$ defines maximum value for actions counter (total number of actions) for each specific ECC state. We assume that algorithm step counter $Ni(fb)$ is updated by the sub-rule p_{Ni}^{FB} during algorithm execution and therefore, the rule for its update is specific to each algorithm. As mentioned in the model schema definition, $Ni(fb) = 0$ always means that an algorithm execution has finished.

Finally, a basic FB has to output events and data and finish its execution. We assume that if there is an output event $e \in EO$ associated with ECC action, after algorithm execution finished the function $Out(e)$ will be set to *true*. Therefore, we can define the output sampling sub-rule p_{EvOut}^{FB} as follows:

$$\begin{aligned}
 p_{EvOut}^{FB}(fb) = & \\
 & \text{forall } e \in EO : Out(e) \text{ do} \\
 & \quad \text{forall } c \in EVCONN_S : Src(c) = e \text{ do} \\
 & \quad \quad Fired(Dst(c)) := true \\
 & \quad \quad Tb(Dst(c)) := t(fb) \\
 & \quad \quad Tl(Dst(c)) := \tau \\
 & \quad \text{forall } ow \in OW : Event(ow) = e \text{ do} \\
 & \quad \quad \text{forall } d \in Data(ow) \text{ do} \\
 & \quad \quad \quad \text{forall } dc \in DCONN_S : Src(dc) = d \text{ do} \\
 & \quad \quad \quad \quad Val(Dst(dc)) := Val(d)
 \end{aligned} \tag{16}$$

B. COMPOSITE FB

As noted in sec. III-A, we decompose a composite function block into two independent interface modules: input and output. The data valve concept, described in works [46]–[48],

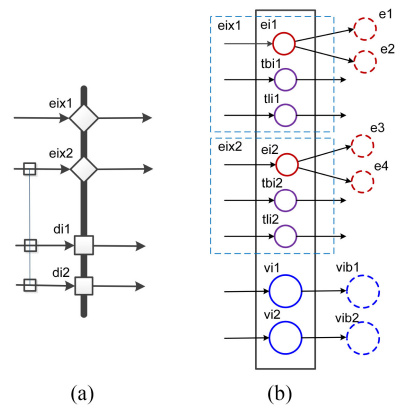


FIGURE 5. CFB input interface. Syntactic (a) and semantic (b) representations.

despite all its advantages, may result in more complicated models when a single data input belongs to more than one data valve. Therefore, keeping in mind model simplicity desired for formal verification, we represent CFB interface as a single entity instead of (possibly) several data valves. Figure 5 (a) and (b) show syntactic and semantic representations of the CFB input interface.

1) INPUT INTERFACE MODEL SCHEMA

As for basic function block, we simply define the model of input interface as $M_{IN} = (Synt_{IN}, Sem_{IN})$, omitting individual indices (i.e. $Synt_{IN}^i, Sem_{IN}^i$), but assume that all elements belong to an individual instance of an interface.

The syntactic part, defined as $Synt_{IN} = (EI, VI, IW)$, consists of input events EI , input data VI and WITH-connections IW .

In the semantic part, we do not define any more entities, but reuse some component definitions given in the overall model and the BFB descriptions.

2) INPUT INTERFACE MODEL DYNAMICS

Input interface operation consists of four steps:

- 1) Setting of the input event timestamps;
- 2) Transferring created output event to receivers;
- 3) Transfer of associated data to receivers;
- 4) Active (processed) input event reset.

It should be noted that the role of an input event in the input interface is changed. When the interface is “triggered,” the active input event becomes an output event for this interface and is transferred to the corresponding event receivers.

We assume that scheduler has chosen an active event input that belongs to the current interface.

Here we try to follow a unified approach, when a received event/data is held in an input buffer, associated with CFB input interface until it is processed (same as for BFB), and output events are transferred to destinations of the corresponding event and data connections by the FB/interface that triggers the event or creates the data. An input interface module

dynamics is defined as follows:

$$\begin{aligned}
p_m^{IN}(ii) = & \\
& \text{forall } e \in EI : IsActive(e) \text{ do} \\
& \quad \text{forall } c \in EVCONN_S : Src(c) = e \text{ do} \\
& \quad \quad Fired(Dst(c)) := true \\
& \quad \quad Tb(Dst(c)) := Tb(e) \\
& \quad \quad Tl(Dst(c)) := \tau \\
& \quad \text{forall } iw \in IW : Event(iw) = e \text{ do} \\
& \quad \quad \text{forall } d \in Data(iw) \text{ do} \\
& \quad \quad \quad \text{forall } dc \in DCONN_S : Src(dc) = d \text{ do} \\
& \quad \quad \quad \quad Val(Dst(dc)) := Val(d) \\
& \quad \quad Fired(e) := false \\
& \quad \quad Tb(e) := undef \\
& \quad \quad Tl(e) := undef
\end{aligned} \tag{17}$$

3) OUTPUT INTERFACE MODEL SCHEMA

Similar to the input interface, the output interface model is defined as a tuple $M_{OUT} = (Synt_{OUT}, Sem_{OUT})$ where $Synt_{OUT} = (EO, VO, OW)$ is a syntactic part of an interface, Sem_{OUT} - semantic part of an interface.

4) OUTPUT INTERFACE MODEL DYNAMICS

An output interface model dynamics is defined by the rule p_m^{OUT} .

$$\begin{aligned}
p_m^{OUT}(io) = & \\
& \text{forall } e \in EO : IsActive(e) \text{ do} \\
& \quad \text{forall } c \in EVCONN_S : Src(c) = e \text{ do} \\
& \quad \quad Fired(Dst(c)) := true \\
& \quad \quad Tb(Dst(c)) := Tb(e) \\
& \quad \quad Tl(Dst(c)) := \tau \\
& \quad \text{forall } ow \in OW : Event(ow) = e \text{ do} \\
& \quad \quad \text{forall } d \in Data(ow) \text{ do} \\
& \quad \quad \quad \text{forall } dc \in DCONN_S : Src(dc) = d \text{ do} \\
& \quad \quad \quad \quad Val(Dst(dc)) := Val(d) \\
& \quad \quad Fired(e) := false \\
& \quad \quad Tb(e) := undef \\
& \quad \quad Tl(e) := undef
\end{aligned} \tag{18}$$

Execution of the rule above will perform the following actions:

- 1) set timestamps for the output event;
- 2) send output event to corresponding event receivers;
- 3) output data, attached to the active output event by WITH-connections;
- 4) reset active output event buffer.

V. MODELLING IN SMV

As defined in the previous section, we use discrete time $Time = \{undef, 0, 1, 2, \dots\}$ in the model, where $0, 1, 2, \dots$ refers to abstract time steps, which could be of different length (e.g. millisecond, or ten milliseconds, or one second), depending on each particular case study. The model is built around the delayed-transition concept where certain transitions between model states require some time. However, some transitions may take just 1 or 2 time steps, and some other may take hundreds, or the opposite - delay is so small that we consider timing of this particular transition non important. In such a case, a uniform time flow would lead to creating a vast amount of intermediate states in the model state space (and counterexamples) where nothing important to a system behaviour happens, except incrementing a timer variable. To address this problem, in [38], [53] we proposed a shifting-time model, where the time of a nearest timed action is calculated by an especially introduced "time scheduler" module and the model is transitioned to that moment in time, avoiding "empty" time increment states.

IEC 61499 standard defines standard library function blocks E_DELAY and E_CYCLE as program timers to be used in control software. E_CYCLE function block can be easily represented by E_DELAY with a feedback event connection from EO to START, therefore we will describe only E_DELAY function block here.

To handle timestamps, we have to introduce global time in the system. The main problem is that the ever increasing global time variable would make the classic model-checking intractable. However, we can always perform model-checking for a limited trace length with bounded model-checking (BMC). To bound the model's state space, only the time interval from 0 to T_max is considered during verification. The global time variable is determined by the rule below, where γ means the end of all execution at the given moment of time (all component FBs' execution finished and there is no active input events in the system) and Do_i is the time scheduler's output variable for the i 'th timer block.

$$T_{Global} < T_{max} \vee \gamma \vee (\wedge (Do_i)) \Rightarrow T_{Global} = T_{Global} + D_{min} \tag{19}$$

Listing in Figure 6 shows the SMV code of time scheduler with global time for two delay blocks. Note that the SMV code for time scheduler is generated individually for a specific number of delay blocks it has to handle.

VI. CASE STUDIES OF TIME-AWARE COMPUTATIONS

We discuss in this paper three examples of the time-aware computations application:

- 1) Time-aware backtracking;
- 2) Continuous time-aware control;
- 3) Time-Complemented Event-Driven approach.

These examples are presented briefly in the following subsections. One should note that these case studies were presented in more detail in the respective separate publications. The purpose of presenting the cases here is to consolidate

```

MODULE TimeScheduler (D1o, D1i, D2o, D2i, beta, gamma)
VAR V1 : integer;
VAR V2 : integer;
VAR DMin : integer;
VAR TGlobal : integer; — global timestamp
ASSIGN
init(TGlobal):=0;
V1:=case
D1o >= 0 : D1o;
TRUE : Tmax;
esac;
V2:=case
D2o >= 0 : D2o;
TRUE : Tmax;
esac;
DMin:=case
V1 <= V2 : V1;
V1 > V2 : V2;
TRUE : 0;
esac;
D1i:=case
(gamma & D1o > 0) & TGlobal < Tmax : D1o - DMin;
TRUE: D1o;
esac;
D2i:=case
(gamma & D2o > 0) & TGlobal < Tmax : D2o - DMin;
TRUE: D2o;
esac;
next(TGlobal):=case
TGlobal<Tmax & (gamma & (D1o > 0 | D2o > 0)):
TGlobal + DMin;
TRUE: TGlobal;
esac;
DEFINE Tmax:= 3000;

```

FIGURE 6. Time scheduler for two delay blocks.

the experiences from the TAC use in different ways and in different application scenarios.

A. TIME-AWARE BACKTRACKING: ELEVATOR CASE STUDY

Time-aware backtracking is applicable to reversible processes with discrete control algorithms. It follows the simple idea that if late arrival of a message from another device in distributed control system indicated that controlled process has missed an important key point, it can be reversed back to that point and certain actions can be performed after that. In the paper [54] we showed a simple case study of where and how time-aware backtracking can be applied. It can be viewed as a sort of “toy” example, demonstrating, however, all necessary elements.

A building with three floors is equipped with an elevator as shown in Figure 7. Each floor has elevator doors, a call button and a cabin position sensor. When a user presses a call button, the elevator controller initiates the cabin to move towards the user’s floor (by sending up and down signals to the motor driver). When the elevator cabin reaches the desired floor, the position sensor sends the signal that the cabin is at the desired floor to the controller and controller stops the motor and opens doors. The decentralised function block control application, shown in Figure 7, is designed with distributed hardware architecture in mind, where nodes are connected with wired or wireless networks.

It is assumed that the cabin position sensors are connected to the controller via a wireless network and there can appear a random delay long enough to make the elevator overshoot

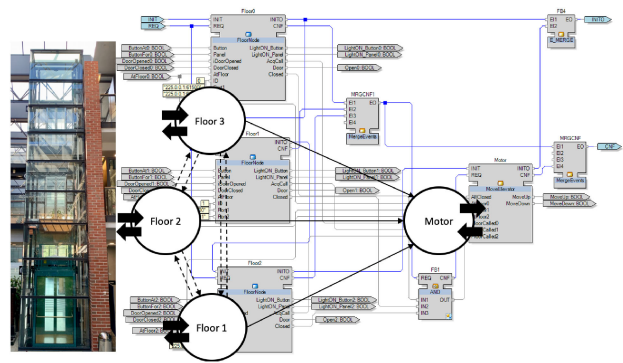


FIGURE 7. Elevator and its distributed function block application.

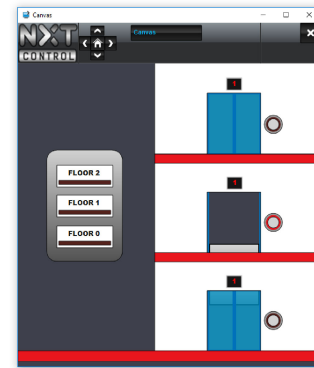


FIGURE 8. Elevator error.

by a sensible distance. Thereby it is possible to get the cabin stopped in a wrong position and have the doors opened there (see Figure 8). We assume that due to the random nature of the communication delay, it is impossible to predict when the elevator will miss the floor, but it is possible to detect it using event timestamps and perform actions to correct the position. When the sensor reading arrives, its timestamp is compared against the present time in the PLC and if the time difference $\Delta t = t_{present} - t_{timestamp}$ is greater than the maximum safety range, the sensor data is considered as outdated and a position correction must be performed.

To simplify the model, the elevator speed is considered constant and to correct the cabin position, the elevator motor is switched backward for the duration Δt . It is possible to calculate the actual elevator correction time for the motor considering acceleration/deceleration calculated with knowledge of weight, motor power, etc.

In the control logic, this approach was implemented by adding special “correction” states to the ECC diagram of the controller. Figure 9 shows the fragment of the ECC diagram with the state for the position correction (*CorrectUp*) when the elevator misses the floor 1 going up, a similar state is added to ECC for all six possible combinations of floor and direction.

The behaviour of the function block application was investigated by means of model-checking. The fb2smv model generator, that is based on the formal model, presented in

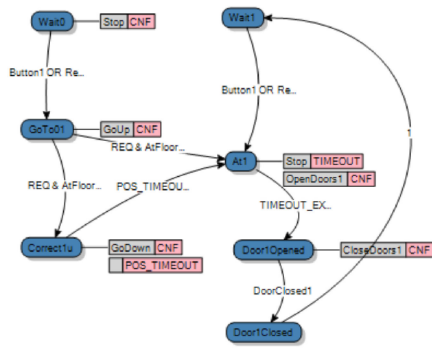


FIGURE 9. Fragment of ECC diagram with position correction state.

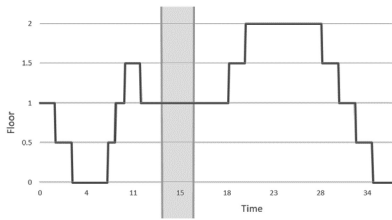


FIGURE 10. Plot from a trajectory in the model state space representing correction of the elevator's position.

this paper, was applied to generate the formal model in SMV language given the source code in IEC 61499. The model was checked against specifications stating that the elevator always stops in the required floor and in the position safe for door opening for a range of the communication jitter values.

Figure 10 shows a plot from nuSMV counterexample with corrected elevator position. The time scale is given in abstract time units, grey area shows when the elevator doors are open. To reduce the SMV model state space, elevator's continuous movement was modelled as a timed automaton with seven discrete positions where each transition between two subsequent positions takes 2 time units.

As pointed out in [50], model-checking has shown significant advantage as compared to testing by simulation in terms of the time, spent on the system's testing.

For more comprehensive details on this case study the reader is referred to [50], [54].

B. CONTINUOUS TIME-AWARE CONTROL

Continuous time-aware control is a wide category of patterns, where a knowledge of communication delay is used to adjust the control action accordingly.

Case study in [49], [53] shows an example of time-aware PID control. Figure 11(a) shows a linearly moving cylinder with centralized PID control implemented using function block diagram of IEC 61499. Figure 11(b) shows the same example, but with distributed control. The function block diagram is mapped onto two devices. Here the cylinder's position detected by a sensor is transferred via wireless communication network, where random message passing delays may occur. In a simulation model for this example we compared three

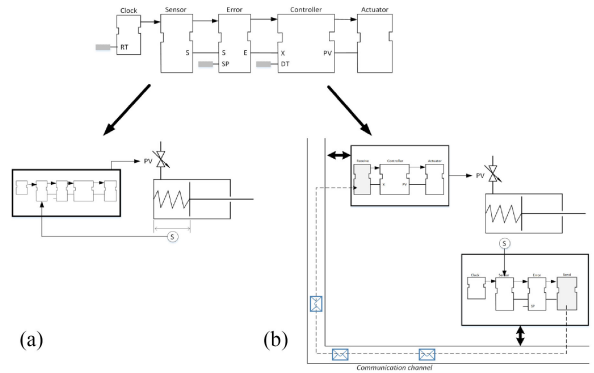


FIGURE 11. Cylinder with PID control: (a) centralized; (b) distributed.



FIGURE 12. Cylinder with PID control. Ideal (blue), distributed with random delay(red), distributed with TAC (green).

versions of control system running at a same time: first - ideal with centralized control and no communication delay (blue plot in Figure 12), second - with same control logic, but distributed control loop as mentioned before (red in Figure 12), and finally the third - with random delay and time-aware PID control (green in Figure 12).

The results show how time-aware PID control allows more smooth movement compared to just a classic control logic in case of distributed control system. Both are of course outperformed by the ideal control system case, where no communication delays occur.

This system was also verified using the SMV model-checker with the prior help of the fb2smv model generator [53].

C. TIME-COMPLEMENTED EVENT-DRIVEN (TCED) APPROACH

Time-complemented event-driven (TCED) approach in [55] proposes a control architecture, where a control decision is made in advance and events triggering control actions are scheduled to be executed at a certain time.

The idea is to provide the actuating events not with past, but with future time stamps, and use a smart actuator with built-in scheduler and synchronized clock to perform control actions at a certain time.

Implementation of this idea could be also done using the time-aware computations concept. For example, for the case of material handling systems from [55], the property to be verified would refer to the precision of the object diversion

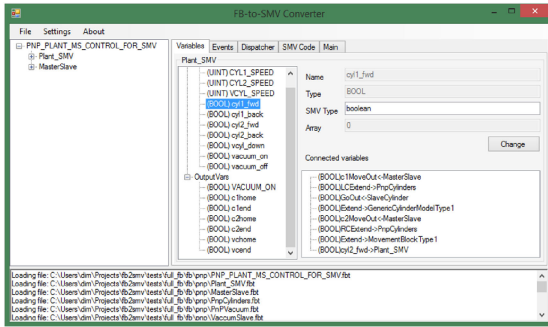


FIGURE 13. User interface of the fb2smv model generator.

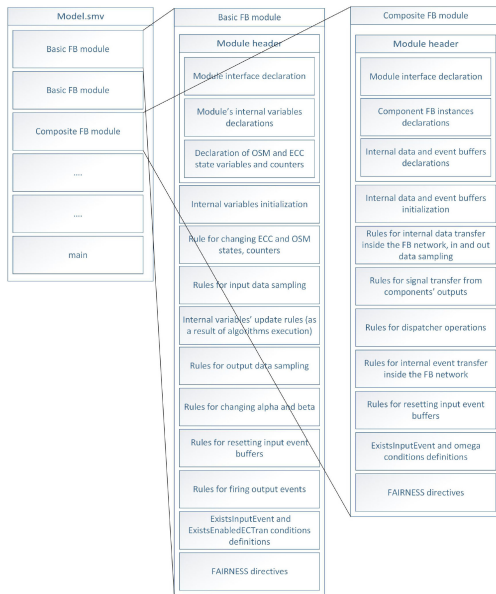


FIGURE 14. Structure of the generated SMV code.

in presence of disturbances, such as communication jitter. Implementation of the formal verification for TCED using fb2smv and NuSMV model-checker is planned in future work.

D. ROLE OF ASM MODELS IN FORMAL VERIFICATION TOOLCHAIN

The main motivation to use ASM is the need for an intermediate formal model, which can be used to generate different models (e.g. SMV) or executable code. For example, the ASM model of function blocks is used in the fb2smv converter [39], which converts IEC 61499 description files in XML to SMV code for formal verification. Its graphical user interface is presented in Figure 13. The window shows the loaded top-level FB PNP_PLANT_MS_CONTROL_FOR_SMV and its nested FBs. In addition, in the Connected variables field, one can observe all the variables associated with the selected cyl1_fwd variable.

The transformation rules to generate output model in fb2smv are based on the defined ASM semantics. The generated SMV model structure is presented in Figure 14. One

can see that the SMV code of both basic and composite FBs consists of the static declarations part (what was referred to as model's schema), and the rules part, which are the ASM rules, discussed in this paper and represented in the SMV language.

The fb2smv tool is a part of formal verification tool-chain, that includes the model checker NuSMV and the tool for counterexample analysis in terms of the original FB system.

VII. CONCLUSION AND FUTURE WORK

In this paper, we presented for the first time a novel formal model of IEC 61499 which can be used as a universal formal intermediate semantic layer. Unlike the existing formal models of IEC 61499, this one can be used for many purposes, providing a semantic reference to the entire function block system. For example, the model is used as a back-end of the fb2smv model generator: the generated SMV code is structured according to the proposed ASM structure and rules. Moreover, we modelled the semantic extension of IEC 61499 with event timestamping, paving the way to both formal analysis of the applications, following the extended syntax and semantics, and developing novel run-time execution engines, based on a formal model of execution.

Formalization of model semantics for IEC 61499 based control systems opens a path for building various models of such systems for simulation and formal verification. Synchronized device clocks and event timestamping plays a key role in dependable distributed control systems for Industry 4.0. In this work, we addressed the problem of formalization of such systems' semantics using ASM notation and proved its consistency on several examples where we applied formal verification methods (i.e. model-checking) to the models based on this semantics.

To claim for completeness, this model needs to be extended to address inter-resource (inter-device) communications and clock synchronization errors inevitably occurring in distributed systems. This is the matter for future work.

REFERENCES

- [1] E. A. Lee and S. A. Seshia, *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*. Cambridge, MA, USA: MIT Press, 2016.
- [2] R. Sinha, S. Patil, L. Gomes, and V. Vyatkin, "A survey of static formal methods for building dependable industrial automation systems," *IEEE Trans. Ind. Informat.*, vol. 15, no. 7, pp. 3772–3783, Jul. 2019.
- [3] C. Ptolemaeus, *System design, modeling, and simulation: Using ptolemy II*. Ptolemy. Org Berkeley, 2014, vol. 1, 2012.
- [4] "IEC 61499-1: Function Blocks Part 1: Architecture," 2nd ed., International Electrotechnical Commission, 2012.
- [5] G. Zhabelova and V. Vyatkin, "Multiagent smart grid automation architecture based on IEC 61850/61499 intelligent logical nodes," *IEEE Trans. Ind. Electron.*, vol. 59, no. 5, pp. 2351–2362, May 2012.
- [6] V. Vyatkin, "IEC 61499 as enabler of distributed and intelligent automation: State-of-the-art review," *IEEE Trans. Ind. Informat.*, vol. 7, no. 4, pp. 768–781, Nov. 2011.
- [7] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Trans. Program. Lang. Syst.*, vol. 8, no. 2, pp. 244–263, 1986.
- [8] H.-M. Hanisch, M. Hirsch, D. Missal, S. Preuße, and C. Gerber, "One decade of IEC 61499 modeling and verification—results and open issues," *IFAC Proc. Volumes*, vol. 42, no. 4, pp. 211–216, 2009.

- [9] J. O. Blech, P. Lindgren, D. Pereira, V. Vyatkin, and A. Zoitl, "A comparison of formal verification approaches for IEC 61499," in *Proc. IEEE 21st Int. Conf. Emerg. Technol. Factory Automat.*, 2016, pp. 1–4.
- [10] Y. Gurevich and E. Börger, "Evolving algebras 1993: Lipari guide," *Evolving Algebras*, p. 40, 1995.
- [11] E. Börger, "The origins and the development of the ASM method for high level system design and analysis," *J. Universal Comput. Sci.*, vol. 8, no. 1, pp. 2–74, 2002.
- [12] E. Börger, "The abstract state machines method for high-level system design and analysis," in *Proc. Formal Methods: State Art New Directions*. Berlin, Germany: Springer, 2010, pp. 79–116.
- [13] K. Winter, "Model checking for abstract state machines," *J. Universal Comput. Sci.*, vol. 3, no. 5, pp. 689–701, 1997.
- [14] P. Arcaini, A. Gargantini, and E. Riccobene, "AsmetaSMV: A way to link high-level ASM models to low-level NuSMV specifications," in *Proc. Int. Conf. Abstr. State Mach., Alloy, B. and Z.*, 2010, pp. 61–74.
- [15] G. D. Castillo and K. Winter, "Model checking support for the ASM high-level language," in *Proc. Int. Conf. Tools and Algo. Const. Anal. Sys.* Berlin, Germany: Springer, 2000, pp. 331–346.
- [16] R. Farahbod, U. Glässer, and G. Ma, "Model checking coreasm specifications," in *Proc. 14th Int. ASM Workshop.*, 2007.
- [17] G. Z. S. Ma, "Model checking support for CoreASM: Model checking distributed abstract state machines using Spin," Ph.D. dissertation, School Comput. Sci., Simon Fraser Univ., 2007.
- [18] I. Liu, J. Reineke, D. Broman, M. Zimmer, and E. A. Lee, "A PRET microarchitecture implementation with repeatable timing and competitive performance," in *Proc. IEEE 30th Int. Conf. Comput. Des.*, 2012, pp. 87–93.
- [19] S. Andalam, P. Roop, and A. Girault, "Predictable multithreading of embedded applications using PRET-C," in *Proc. 8th ACM/IEEE Int. Conf. Formal Methods Models Codeign.*, 2010, pp. 159–168.
- [20] K.-E. Årzén, "A simple event-based PID controller," *IFAC Proc. Volumes*, vol. 32, no. 2, pp. 8687–8692, 1999.
- [21] K. J. Aström, "Event Based Control," in *Proc. Anal. Des. Nonlinear Control Syst.* Berlin, Germany: Springer, 2008, pp. 127–147.
- [22] J. Nilsson, "Real-time control systems with delays," Ph.D. dissertation, Dept. Autom. Control, Lund Inst. Technol., Sweden, 1998.
- [23] A. Zoitl and R. Lewis, "Modelling control systems using IEC 61499: Applying function blocks to distributed systems," ROM, U.K., vol. 95, 2014.
- [24] V. Vyatkin and H.-M. Hanisch, "A modeling approach for verification of IEC1499 function blocks using net condition/event systems," in *Proc. 7th IEEE Int. Conf. Emerg. Technol. Factory Automat.*, 1999, pp. 261–270.
- [25] L. H. Yoong, P. S. Roop, V. Vyatkin, and Z. Salcic, "A synchronous approach for IEC 61499 function block implementation," *IEEE Trans. Comput.*, vol. 58, no. 12, pp. 1599–1614, Dec. 2009.
- [26] L. Xiao, R. Wang, M. Gu, and J. Sun, "Semantic characterization of programmable logic controller programs," *Math. Comput. Modelling*, vol. 55, no. 5, pp. 1819–1824, 2012.
- [27] M. Perin and J.-M. Faure, "Building meaningful timed models of closed-loop DES for verification purposes," *Control Eng. Pract.*, vol. 21, no. 11, pp. 1620–1639, 2013.
- [28] H. Prahofor and A. Zoitl, "Verification of hierarchical IEC 61499 component systems with behavioral event contracts," in *Proc. IEEE Int. Conf. Ind. Inform.*, 2013, pp. 578–585.
- [29] M. Stanica and H. Guéguen, "Using timed automata for the verification of IEC 61499 applications," in *IFAC Proc.*, vol. 37, no. 18, pp. 375–380, 2004.
- [30] G. Cengic and K. Åkesson, "Definition of the execution model used in the fuber IEC 61499 runtime environment," in *Proc. IEEE Int. Conf. Ind. Inform.*, 2008, pp. 301–306.
- [31] G. Cengic and K. Åkesson, "On formal analysis of IEC 61499 applications, Part A: Modeling," *IEEE Trans. Ind. Informat.*, vol. 6, no. 2, pp. 136–144, May 2010.
- [32] G. Cengic and K. Åkesson, "On formal analysis of IEC 61499 applications, part b: Execution semantics," *IEEE Trans. Ind. Informat.*, vol. 6, no. 2, pp. 145–154, May 2010.
- [33] S. Patil, V. Dubinin, C. Pang, and V. Vyatkin, "Neutralizing semantic ambiguities of function block architecture by modeling with ASM," in *Proc. Int. Andrei Ershov Memorial Conf. Perspectives Syst. Inform.* Berlin, Germany: Springer, 2014, pp. 76–91.
- [34] S. Patil, V. Dubinin, and V. Vyatkin, "Formal verification of IEC 61499 function blocks with abstract state machines and SMV-Modelling," in *Proc. IEEE Trustcom/BigDataSE/ISPA.*, vol. 3, 2015, pp. 313–320.
- [35] S. Patil, V. Dubinin, and V. Vyatkin, "Formal modelling and verification of IEC 61499 function blocks with abstract state machines and SMV-execution semantics," in *Proc. Int. Symp. Dependable Softw. Eng.: Theories, Tools, Appl.* Berlin, Germany: Springer, 2015, pp. 300–315.
- [36] S. Patil, V. Vyatkin, and M. Sorouri, "Formal verification of intelligent mechatronic systems with decentralized control logic," in *Proc. 17th Conf. Emerg. Technol. Factory Automat.*, 2012, pp. 1–7.
- [37] S. Patil, D. Drozdov, V. Dubinin, and V. Vyatkin, "Cloud-based framework for practical model-checking of industrial automation applications," in *Proc. Doctoral Conf. Comput., Elect. Ind. Syst.* Berlin, Germany: Springer, 2015, pp. 73–81.
- [38] D. Drozdov, S. Patil, V. Dubinin, and V. Vyatkin, "Formal verification of cyber-physical automation systems modelled with timed block diagrams," in *Proc. IEEE 25th Int. Symp. Ind. Electron.*, 2016, pp. 316–321.
- [39] D. Drozdov, "fb2smv: IEC 61499 Function Blocks XML Code to SMV Converter," <https://github.com/dmitydrozdov/fb2smv>
- [40] E. A. Lee, "Cyber physical systems: Design challenges," in *Proc. 11th IEEE Symp. Object Oriented Real-Time Distribution Comput.*, 2008, pp. 363–369.
- [41] E. Lee, "The past, present and future of cyber-physical systems: A focus on models," *Sensors*, vol. 15, no. 3, pp. 4837–4869, 2015.
- [42] P. Derler et al., "Ptides: A programming model for distributed real-time embedded systems," Dept. Elect. Eng Comput. Sci., California Univ. Berkeley, Tech. Rep., 2008.
- [43] W. Dai, C. Pang, V. Vyatkin, J. H. Christensen, and X. Guan, "Discrete-event-based deterministic execution semantics with timestamps for industrial cyber-physical systems," *IEEE Trans. Syst., Man, Cybern. Syst.*, vol. 50, no. 3, pp. 851–862, Mar. 2020.
- [44] C. Pang, W. Dai, and V. Vyatkin, "Towards IEC 61499 models of computation in Ptolemy II," in *Proc. IECON 2015-41st Annu. Conf. IEEE Ind. Electron. Soc.*, 2015, pp. 001988–001993.
- [45] J. Nilsson, B. Bernhardsson, and B. Wittenmark, "Stochastic analysis and control of real-time systems with random time delays," *Automatica*, vol. 34, no. 1, pp. 57–64, 1998.
- [46] C. Sunder, A. Zoitl, J. H. Christensen, M. Colla, and T. Strasser, "Execution models for the IEC 61499 elements composite function block and subapplication," in *Proc. 5th IEEE Int. Conf. Ind. Inform.*, vol. 2, 2007, pp. 1169–1175.
- [47] V. Dubinin and V. Vyatkin, "Towards a formal semantic model of IEC 61499 function blocks," in *Proc. 4th Int. Conf. Ind. Inform.*, 2006, pp. 6–11.
- [48] V. Dubinin and V. Vyatkin, "On definition of a formal model for IEC 61499 function blocks," *EURASIP J. Embedded Syst.*, vol. 2008, p. 7, 2008.
- [49] V. Vyatkin, C. Pang, and S. Tripakis, "Towards cyber-physical agnosticism by enhancing IEC 61499 with ptides model of computations," in *Proc. IECON 2015-41st Annu. Conf. IEEE Ind. Electron. Soc.*, 2015, pp. 001970–001975.
- [50] V. Shatrov and V. Vyatkin, "Formal verification of IEC 61499 enhanced with timed events," in *Proc. Doctoral Conf. Comput., Elect. Ind. Syst.* Berlin, Germany: Springer, 2020, pp. 168–178.
- [51] (2020) Function Block Builder. [Online]. Available: <https://www.yueyiautomation.com/Software?type=FB>
- [52] C. Pang, S. Patil, C.-W. Yang, V. Vyatkin, and A. Shalyto, "A portability study of IEC 61499: Semantics and tools," in *Proc., 12th IEEE Int. Conf.*, 2014, pp. 440–445.
- [53] D. Drozdov, S. Patil, and V. Vyatkin, "Formal modelling of distributed automation CPS with CP-agnostic software," in *International Workshop Service Orientation Holonic Multi-Agent Manufacturing* Berlin, Germany: Springer, 2017, pp. 35–46.
- [54] D. Drozdov, S. Patil, V. Dubinin, and V. Vyatkin, "Towards formal verification for cyber-physically agnostic software: A case study," in *Proc. IECON 2017-43rd Annu. Conf. IEEE Ind. Electron. Soc.*, 2017, pp. 5509–5514.
- [55] C. Pang, J. Yan, and V. Vyatkin, "Time-complemented event-driven architecture for distributed automation systems," *IEEE Trans. Syst., Man, Cybern. Syst.*, vol. 45, no. 8, pp. 1165–1177, Aug. 2015.



DMITRII DROZDOV received the B.Tech. degree in computer engineering and the M.Sc. degree in computer science from Penza State University, Penza, Russia, in 2013 and 2015, respectively.

He is currently working toward the Ph.D. degree with Luleå University of Technology, Luleå, Sweden, with major in industrial informatics. His research interests include distributed control systems in industrial automation, methods and means of their efficient design, testing, formal verification, and implementation.



VICTOR DUBININ received the Diploma degree in computer engineering, and the Ph.D. degree in computer engineering and Dr.Sc. degree in computer science from the University of Penza, Penza, Russia, in 1981, 1989, and 2014, respectively.

From 1981 to 1989, he was a Researcher, from 1989 to 1995, he was a Senior Lecturer, and from 1995 to 2015, he was an Associate Professor with the University of Penza. Since 2015, he has been a Professor with the Department of Computer Science, University of Penza. In 2011, he held a Visiting Researcher position with The University of Auckland, Auckland, New Zealand, and from 2013 to 2019, he was with the Luleå University of Technology, Luleå, Sweden. His research interests include formal methods for specification, verification, synthesis, and implementation of distributed and discrete event systems.

He was a recipient of DAAD-grants to work as a Guest Scientist with Martin-Luther-University Halle-Wittenberg, Halle, Germany, in 2003, 2006, and 2010, respectively.

He was a recipient of DAAD-grants to work as a Guest Scientist with Martin-Luther-University Halle-Wittenberg, Halle, Germany, in 2003, 2006, and 2010, respectively.



SANDEEP PATIL (Member, IEEE) received the bachelor's degree in computer science engineering from the CMR Institute of Technology, Bengaluru, India, in 2005, the Master of Computer Science degree in software engineering from the Illinois Institute of Technology, Chicago, IL, USA, in 2010, the Master of Engineering Studies degree in computer systems from The University of Auckland, Auckland, New Zealand, in 2011, and the Ph.D. degree in formal verification of cyber physical systems from the Luleå University of Technology, Luleå, Sweden, in 2018.

Sweden, in 2018.

He also works with formal verification techniques in the same application field. He is an Accomplished Software Engineering Professional with more than 14 years of research and development experience in systems and application software, including four years with Motorola India Pvt. Ltd., India, as a Senior Software Engineer. His research interests include software engineering principles and methodologies in distributed industrial automation, especially using the IEC 61499 paradigm.



VALERIY VYATKIN (Senior Member, IEEE) received the Doctoral degrees from Russia and Japan, in 1992 and 1999, respectively, and Habilitation degree from Germany, in 2002.

He is on joint appointment as a Chaired Professor with Luleå University of Technology, Sweden, and Full Professor with Aalto University, Helsinki, Finland. Previously, he was a Visiting Scholar with Cambridge University, Cambridge U.K., and had permanent academic appointments with New Zealand, Germany, Japan, and Russia. His research interests include dependable distributed automation and industrial informatics, software engineering for industrial automation systems, artificial intelligence, distributed architectures, and multi agent systems applied in various industry sectors, including smart grid, material handling, building management systems, data centres, and reconfigurable manufacturing.

He was awarded the Andrew P. Sage Award for the best IEEE Transactions paper in 2012. He has been the Chair of IEEE IES Technical Committee on Industrial Informatics since 2016.

He was awarded the Andrew P. Sage Award for the best IEEE Transactions paper in 2012. He has been the Chair of IEEE IES Technical Committee on Industrial Informatics since 2016.