Hierarchical and Concurrent ECCs for IEC 61499 Function Blocks

Roopak Sinha, *Member, IEEE*, Partha S. Roop, *Member, IEEE*, Gareth Shaw, Zoran Salcic, *Senior Member, IEEE*, and Matthew M. Y. Kuo

***Abstract*—IEC 61499 enables component-oriented descriptions of complex industrial processes facilitating model-driven engineering. One aspect that is lacking, however, is the ability to directly express Statecharts-like hierarchy and concurrency within basic function blocks (BFBs). Such features can significantly enhance function blocks and help create more succinct and readable specifications. We propose a new syntactic extension to the standard called hierarchical and concurrent execution control chart (HCECC). A major roadblock for any suggested changes to the standard is the need for compliance. Our approach extends the synchronous execution semantics of IEC 61499, where HCECCs are purely syntactic sugar. Using a revised synchronous semantics, our compiler generates standard compliant C code from HCECCs. Benchmarking and usability studies reveal the relative superiority of the proposed approach over existing approaches.**

***Index Terms*—Execution semantics, hierarchical state machines, IEEE 61499, parallel execution, statecharts.**

I. Introduction

THE INCREASING size and complexity of industrial control systems have motivated the development of new design paradigms to aid system designers. The IEC 61499 standard [1], [2] proposes component-oriented models called function blocks for developing distributed control systems. A function block describes both control and data flow within a single module. By providing a graphical specification framework, the standard simplifies system design with a top-down approach and encourages the reuse of function blocks.

The unit of execution in IEC 61499 is a basic function block (BFB), which executes a finite state machine (FSM) known as an execution control chart (ECC) that describes the control flow of the block. ECCs are flat Moore-machines that allow modeling of only sequential behaviors. Therefore, a basic block cannot model concurrent behaviors. In addition, ECCs do not allow users to separate high-level behavior such as initialization and error handling of a block. As a result, more states and transitions are required to describe the same behavior compared to other graphical formalisms.

The standard also provides CFBs that are networks of interconnected function block instances. Networks model concurrency and structural hierarchy that basic blocks cannot handle efficiently. However, networks can easily get very complex, making them difficult to design, understand, and maintain. Moreover, in networks containing many function block instances, a significant amount of execution time is required for the flow of events and data between instances. Not only do such complex networks require a long time to design manually, network behavior becomes nonobvious and there is a higher chance of software bugs.

This issue has been noted in Annex E.1 of the IEC 61499 draft standard [3] as well as recent scholarly work [2], [4]. These works highlight the urgent need to introduce statecharts-like [5] hierarchy and parallelism within basic blocks. Statecharts are extensions of FSMs that allow a single FSM to contain concurrent or parallel as well hierarchical or refined behaviors. Statecharts have found popular use in the design of distributed control software [6]. Unfortunately, statecharts semantics [4] are incompatible with the IEC 61499 standard (e.g., it does not handle data flow modeling, unlike IEC 61499), and currently no extension of BFBs to model Statecharts exists.

This paper proposes the first Statecharts-like extensions of IEC 61499 called hierarchical and concurrent ECCs (HCECCs), which allow explicit modeling of hierarchy and concurrency within a BFB using two operators: 1) parallel; and 2) refinement. Any nesting of these operators can be resolved to a standard CFB, making HCECCs completely compliant to the standard. A synchronous execution semantics [7], [8] for HCECCs is proposed, which allows a complete deterministic execution of HCECCs as well as standard function blocks. The main contributions of this paper are formalizing IEC 61499 function blocks syntax and presenting a simplified synchronous semantics for the execution of function blocks, formulating the HCECC framework by introducing the parallel and refinement operators, and showing how HCECCs execute using the proposed simplified synchronous semantics, and presenting a sound translation of HCECCs into standard-compliant CFBs.

This paper is organized as follows. Section II presents a review of related works. Section III introduces a formalism for IEC 61499 and a description of the semantics for basic and CFBs. HCECCs are introduced in Section IV, and Section V describes their transformation to standard-compliant composite

Manuscript received March 21, 2014; revised August 15, 2015; accepted October 14, 2015. Paper no. TII-15-0754.R1.

R. Sinha is with the School of Computer and Mathematical Sciences, Auckland University of Technology, Auckland, New Zealand (e-mail: roopak.sinha@aut.ac.nz).

P. S. Roop, Z. Salcic, and M. M. Y. Kuo are with the Department of Electrical and Computer Engineering, University of Auckland, Auckland, New Zealand (e-mail: p.roop@auckland.ac.nz; z.salcic@auckland.ac.nz; mkuo005@aucklanduni.ac.nz).

G. Shaw is with Fiserv New Zealand, Auckland, New Zealand (e-mail: gareth@garethnz.com).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TII.2015.2496262

blocks. Section IV reports experimental results and Section VII provides concluding remarks and future directions.

II. Literature Review

Model-driven engineering as expounded by IEC 61499 is founded on models of computation. The standard allows modeling control flow using events and state machines within function blocks, and data flow using variables and algorithms. More recently, researchers have employed more expressive models such as statecharts [9] and synchronous data flow [10], and communicating sequential processes [11] for modeling control flow in industrial strength systems. Similarly, extending data flow modeling in industrial control systems has also been studied extensively [12]. Some formalisms like *charts (pronounced starcharts) [13] have been used to combine control and data flow models. However, for IEC 61499 systems, it is not possible to modify the standard.

The existing pool of statecharts-based frameworks serves as the logical starting point to build IEC 61499 compatible models to capture concurrency and hierarchy. Statecharts extend FSMs and allow parallelism using state diagrams with broadcast-based communication and hierarchy using a nested structure of states. statecharts are used in a wide range of application domains for modeling complex systems, and have been extended to generate popular modeling frameworks such as UML statecharts [14]. Unfortunately, statecharts and current variants cannot be used in IEC 61499 because model only control flow and do not capture data flow.

The need to introduce statecharts-like extensions within basic blocks is well documented [2]–[4]. Comparative studies between IEC 61499 and other frameworks like UML-statecharts [15], [16] help highlight this gap. Unfortunately, existing solutions to introduce statecharts-like extensions in IEC 61499 have limited scope. In [17], component interaction diagrams are proposed to allow modeling UML-statecharts like hierarchy within function block applications. However, these diagrams do not allow designers to embed concurrency and hierarchy within a single function block. In [4], ECCs within BFBs are extended based on the Monaco domain-specific language, which models a subset of statecharts. However, the authors discard this extension because of an inability to retrieve hierarchy from flattened ECCs, and the associated re-engineering and refactoring costs. On the other hand, the proposed framework is not domain specific, and hierarchical and concurrent behaviors can be automatically translated to networks of function blocks. In [18] and [19], informal statecharts-based extensions to basic blocks are proposed. In all cases, the issue of complying to the standard is overlooked.

Many different execution semantics of IEC 61499 have been proposed [1], [2], [20]. As compared to other semantics, the synchronous semantics developed in [7] excels in many respects. Synchronous systems execute in logical time instances called ticks. During each tick, a system reads inputs, processes them, and then emits outputs. A tick or macro-step is atomic and instantaneous. Internally though, macro-steps are usually sequenced into micro-steps. A micro-step pertains to a reaction of an individual component or the propagation of information within the system. Synchronous systems may suffer from *causality* cycles caused by inter-dependent micro-steps [8]. In [7], causality cycles are avoided by introducing a one-tick delay in communications between the blocks in a network, which slows down event propagation in large networks. The semantics proposed in this paper avoids causality cycles and propagation by using a static schedule for executing blocks in a network, following the semantics of PRET-C [21].

HCECCs allow the inclusion of concurrent and hierarchical behaviors within basic blocks, and are a subset of Statecharts without inter-level transitions. The simplified synchronous semantics for IEC 61499 ensure that programs can never have causality cycles. The proposed HCECC syntax and semantics allow translating HCECCs into standard IEC 61499 function block networks, making them fully standard compliant.

III. Formalization of IEC 61499

This section presents the syntax and simplified synchronous semantics of IEC 61499. A baggage handling system (BHS) containing many baggage entry and exit points connected via a network of conveyor belts is used as an illustrative example.

A. Syntax

The following key concepts and notations are used in this paper. A set $A = \{a_1, \ldots, a_n\}$ is a totally ordered set *iff* for any two distinct elements $a_i, a_j \in A$, $a_i <_A a_j$ if $i < j$ or otherwise $a_j <_A a_i$. $<_A$ is the antisymmetric and transitive *ordering relation* for set A. The subscript A of $<_A$ is omitted when the context is clear. The linear sum $C = A \oplus B$ of two ordered and disjoint sets A and B is the union of A and B, with A's elements ordered as in A and B's elements ordered as in B, and $a <_C b$ for every $a \in A$ and $b \in B$.

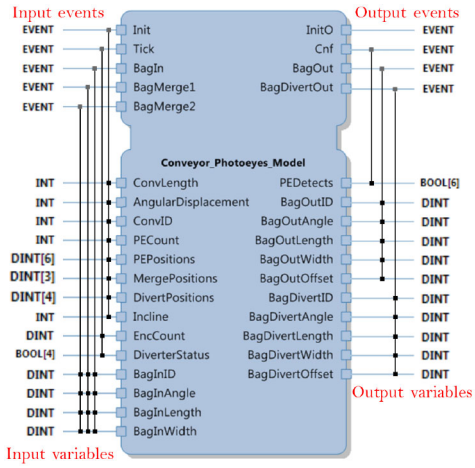
1) Interface: All function blocks have interfaces.

Definition 1 (Function block interface): A function block interface is a tuple $\mathcal{I} = \langle E_I, V_I, \alpha_I, E_O, V_O, \alpha_O \rangle$ where E_I, V_I, E_O, and V_O are finite sets of input events, input variables, output events and outputs variables, respectively. $\alpha_I \subseteq E_I \times V_I$ and $\alpha_O \subseteq E_O \times V_O$ are the sets of input and output associations.

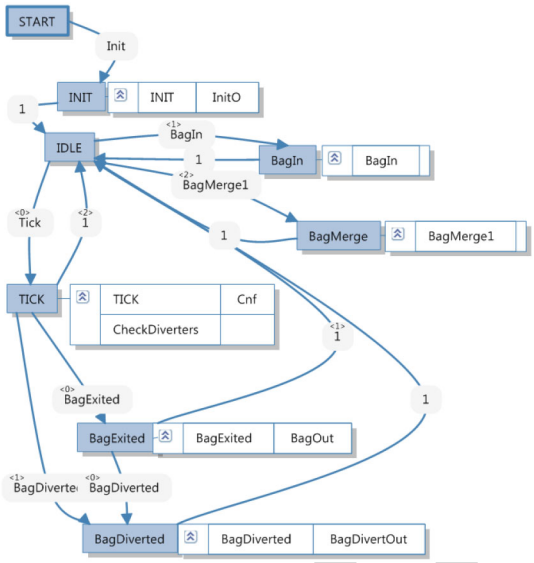
Fig. 1 shows the interface for `Conveyor_Photoeyes_Model`, which is used in the BHS to control a conveyor belt. The input events and variables appear on the left side, and outputs appear on the right side. Events are transient and are present in time instants when they are fired and absent otherwise. This is based on the well-known synchronous approach [8]. Variables have persistent values. Variables are sampled or updated every time an associated event is present. For example, in Fig. 1, event Init is associated with the variable *ConvLength*. Hence, *ConvLength* is sampled whenever `Init` is present.

2) Basic Function Block:

Definition 2 (BFB): `BFB` is a tuple $\langle \mathcal{I}, L, \texttt{ECC} \rangle$ with interface $\mathcal{I}$. The local declarations $L = \langle V_L, A_L \rangle$ contain an internal variables set V_L and a set of algorithms A_L. $\texttt{ECC} = \langle S, s_0, \lambda, T \rangle$ is an ECC with a finite set of states S, initial state $s_0 \in S$, an action function $\lambda : S \to (A_L \cup E_O)^*$ mapping states to a finite



F1:1 Fig. 1. Function block interface.



F2:1 Fig. 2. ECC.

sequence of algorithm executions and/or output event emissions, and the transition function $T : S \rightarrow 2^{(E_I \cup \{1\}) \times \mathcal{B}(\hat{V}) \times S}$. Here, $\hat{V}$ refers to the set of all combinations of valid values of internal, input and output variables, and $\mathcal{B}(\hat{V})$ refers to all Boolean expressions over $\hat{V}$. The transition function is restricted such that for any $s \in S$, $T(s)$ is totally ordered.

BFB has an interface, local declarations (internal variables and algorithms), and an ECC. The `Conveyor_Photoeyes_Model` block has the interface shown in Fig. 1, three internal variables BagModel, BagExited, and BagDiverted, and the set of algorithms $A_L = \{INIT, TICK, CheckDiverters, BagExited, BagDiverted, BagIn, BagMerge1\}$. Algorithms are written using any PLC or standard languages such as C or Java.

Fig. 2 shows the ECC of `Conveyor_Photoeyes_Model`. The ECC in Fig. 2 contains eight states with `START` as the initial state (highlighted with a bold outline). Each state has associated *actions* or a sequence of algorithm executions and output event emissions. For state `INIT`, the actions $\lambda(\texttt{INIT}) = INIT.InitO$ correspond to executing the algorithm $INIT$ and emitting the event $InitO$. For any state s, an outgoing transition $t \in T(s)$ is described as (e, b, s') (or $s \xrightarrow{e,b} s'$ in short-hand), where e is either a triggering input event or 1 (no triggering event), b is a Boolean expression evaluated on the values of internal, input and output variables, and s' is the destination state. For example, consider the transition `START` $\xrightarrow{\text{Init,true}}$ `INIT`. The triggering event is *Init*, the Boolean condition is $true$, and the destination state is `INIT`. For any state s the set of transitions $T(s)$ is totally ordered. Transition order is shown in Fig. 2 using the labels $\langle 0 \rangle$, $\langle 1 \rangle, \ldots$ For example, the first transition from state `IDLE` is to state `Tick` (labeled with $\langle 0 \rangle$). The order is omitted when a state only has one outgoing transition.

3) Composite Function Block: CFB contains a network of interconnected function block instances. Fig. 3 combines two basic blocks into a CFB.

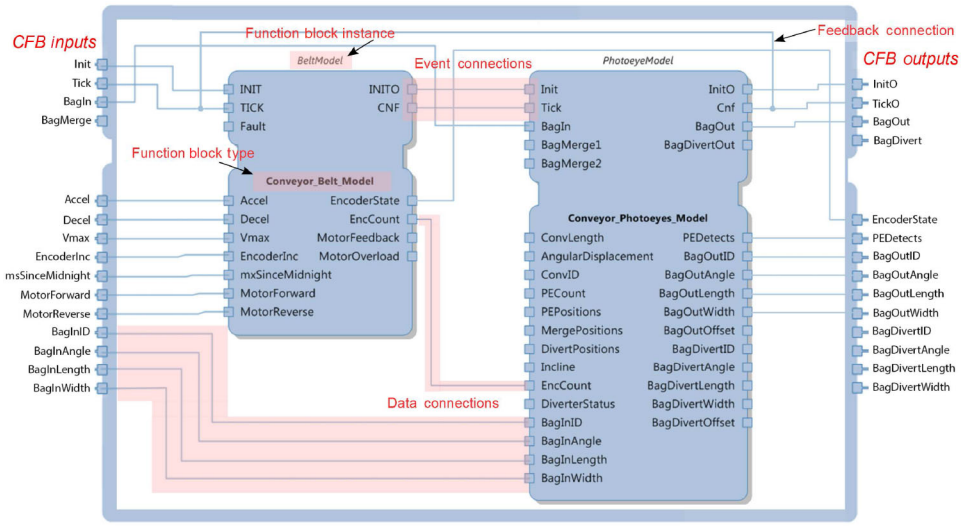
Function block instances: An instance is to a function block what an object is to a class in Java or C++: the former is a named *instance* of the latter *type*. An instance `fb` $= \langle name, FBT \rangle$ is called $name$ and has the type FBT (which can be a basic or composite block). The composite block in Fig. 3 contains two function block instances: $BeltModel$ of type `Conveyor_Belt_Model`, and $PhotoeyeModel$ of type `Conveyor_Photoeyes_Model`. Instances allow easy reuse of function blocks. The interface of an instance is used to connect it into a network. The notation $\langle$`name`$\rangle$.$\langle$`event/variable name`$\rangle$ refers to the inputs or outputs (events/variables) of an instance `fb`. For example, the output variable *PEDetects* of instance `PhotoeyeModel` is written as *PhotoeyeModel.PEDetects*. The instances of a composite block network are contained in a totally ordered set `FBs`.

Event connections: A CFB network contains the following types of event connections (the short-hand $s \mapsto d$ represents an event connection from s to d).

1) Input to input connections C_{EI2I}: Inputs of the CFB's interface may connect to inputs of instances in the network. For example, *Tick* $\mapsto$ `BeltModel`.*TICK*, as per Fig. 3.
2) Output to input connections C_{EO2I}: Output events of one instance may be read as inputs by another. For example, `BeltModel`.$CNF \mapsto$ `PhotoEyeModel`.*Tick* A *feedback* connection connects the output of an instance appearing later in `FBs` (an ordered set) to the input of an earlier instance in `FBs`. For example, `PhotoEyeModel`.$Cnf \mapsto$ `BeltModel`.*Tick*
3) Output to output connections C_{EO2O}: Output events of instances in the network may connect to output events of the CFB's interface. For example, `PhotoEyeModel`.*Cnf* $\mapsto TickO$.

Variable connections: Composite blocks have three types of variable connections, as illustrated in Fig. 3.

1) Input to input connections C_{VI2I} connect input variables of the interface to the input variables of the FB instances in the network, e.g., *Accel* $\mapsto$ `BeltModel`.*Accel*.
2) Output to input connections C_{VO2I} connect the output variable of one FB instance to the input of another, e.g., `BeltModel`.*EncCount* $\mapsto$ `PhotoEyeModel`.*EncCount*.



F3:1 Fig. 3. Network of blocks within the conveyor model CFB.

3) Output to output connections C_{VO2O} connect an output variable of an instance to an output of the interface, such as `PhotoEyeModel`.*PEDetects* $\mapsto$ `PEDetects`.

IEC 61499 restricts variable connections such that only a single source can be connected to a destination variable in a CFB. Moreover, only variables of the same type can be connected together. This paper omits this aspect for the sake of readability but without sacrificing expressiveness.

Definition 3 (CFB): A `CFB` is a tuple $\langle \mathcal{I}, \texttt{FBNetwork} \rangle$ with interface $\mathcal{I} = \langle E_I, V_I, \alpha_I, E_O, V_O, \alpha_O \rangle$, and a network `FBNetwork` $= \langle \texttt{FBs}, C_{\text{events}}, C_{\text{var}} \rangle$, where `FBs` $= \{\texttt{fb}_1, \texttt{fb}_2, \ldots \texttt{fb}_n\}$ is a finite and totally ordered set of function block instances of size n. Each $\texttt{fb}_i$ $(i \in [1, n])$ is a FB instance $\langle name_i, FBT_i \rangle$, with name $name_i$ and a function block (type) FBT_i with the interface $\mathcal{I}_i = \langle E_{Ii}, V_{Ii}, \alpha_{Ii}, E_{Oi}, V_{Oi}, \alpha_{Oi} \rangle$. The event connections set $C_{\text{events}} = C_{EI2I} \cup C_{EO2I} \cup C_{EO2O}$ (as per Section III-A3). The set of variable connections $C_{\text{var}} = C_{VI2I} \cup C_{VO2I} \cup C_{VO2O}$ (as per Section III-A3). For any two variable connections $(src_1, dest_1), (src_2, dest_2) \in C_{\text{var}}, dest_1 \neq dest_2$.

B. Synchronous Semantics for IEC 61499

The proposed synchronous semantics for IEC 61499 is similar to the semantics of PRET-C [21], and guarantees the absence of causality cycles due to a static ordering of execution. Execution of blocks is divided into logical time instants called *ticks* or macro-steps. During each tick, a predefined and ordered sequence of micro-steps is followed. This semantics differs from the synchronous semantics proposed in [7], where the order of execution of blocks may change between different ticks, and blocks can only read inputs generated in the previous tick. The proposed semantics does not require this delayed propagation of events used in [7].

1) Execution Semantics of BFBs: Each BFB initializes in its start state s_0. Each tick, its execution follows four steps. In step 1, the block samples input events (E_I) from the environment, and updates input variables (in V_I) when associated input events are present. In step 2, the outgoing transitions of the current state s are evaluated in order and the first enabled transition is taken. A transition is enabled when the related event is present and the Boolean condition evaluates to $true$. For example, if the ECC shown in Fig. 2 is in state `START`, and input event *Init* is present, the lone transition `START` $\xrightarrow{\text{Init,true}}$ `INIT` fires. If no outgoing transition is enabled, the current tick terminates. In step 3, after a transition to a destination state s' has fired, the actions $\lambda(s')$ are executed. For example, if state `INIT` is entered, the block executes algorithm $INIT$ and emits the event $InitO$. Finally, in step 4, output variable values are updated if associated output events were emitted in step 3. Thanks to the explicit ordering of ECC transitions, basic blocks execute deterministically, as in step 2, only the highest priority transition that is enabled fires.

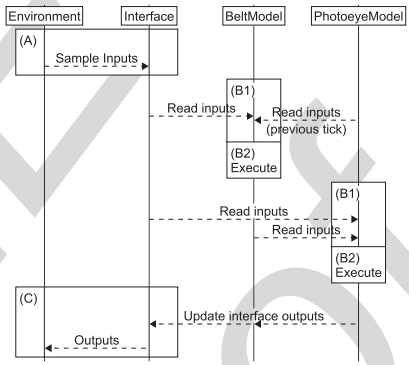
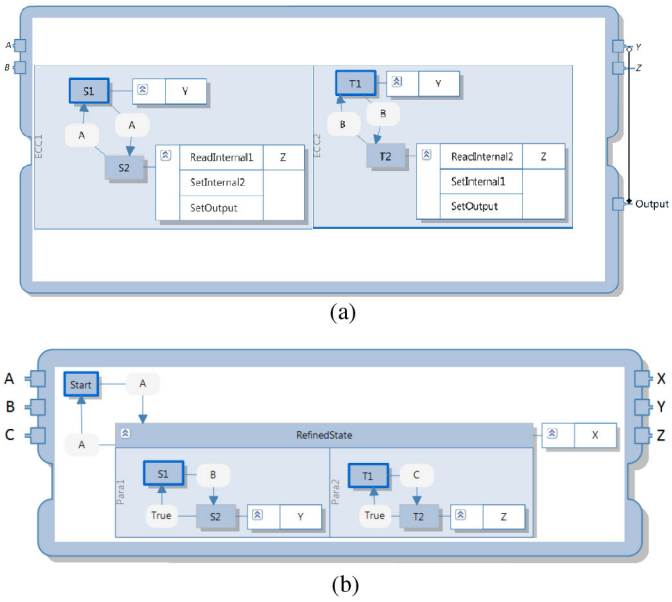


Fig. 4. Execution semantics for composite blocks. F4:1

2) Execution Semantics of Composite Blocks: A CFB is initialized with all instances in its network in their initial states. Each tick, the following sequence is executed. Fig. 4 illustrates this sequence for the CFB shown in Fig. 3. In step A, interface input events are sampled, and associated input variable values are updated. In step B, all FB instances in the network execute as per their order in the set `FBs`. For example, `BeltModel` executes before `PhotoeyeModel`. The execution of each instance involves two substeps. In step B1, input events are sampled and



F5:1 Fig. 5. HCECC examples. (a) Parallel HCECC. (b) Refined HCECC.

associated input variables are updated. For example, the input event `BeltModel`.*TICK* is present if any of the source events Tick of the block's interface or `PhotoeyeModel`.*Cnf* is present. For feedback connections, such as `PhotoeyeModel`.*Cnf* $\mapsto$ `BeltModel`.*TICK*, the presence or absence of the source event in the previous tick is considered. Then, in step B2, each FB instance is executed, based on its semantics (basic or composite). Fig. 4 shows how both steps B1 and B2 are repeated for the two instances in the CFB of Fig. 3. Finally, in step C, outputs of the CFB interface are updated. An output event is set to present if a source event connected to it is present during the current tick. All output variables associated with emitted output events are also updated.

We can recursively substitute composite block instances with their networks to translate a CFB network into a network of basic block instances. Under the proposed semantics, CFBs can be mapped to PRET-C programs with well-formed semantics [21]. It can then be shown that CFB execution is deterministic and reactive under the proposed semantics.

IV. HCECCs

HCECCs introduce parallel and refined block types to allow statecharts-like concurrency and hierarchy in basic blocks.

A. Syntax

Parallel HCECCs allow concurrency within a single block. Fig. 5(a) shows a parallel HCECC, which has an interface and local declarations. However, unlike a basic block, the HCECC contains two ECCs that share the interface and local declarations.

Definition 4 (parallel HCECC): A parallel HCECC $\texttt{HCECC}_{\|}$ is a tuple $\langle \mathcal{I}, L, \texttt{ECCs} \rangle$, with interface $\mathcal{I}$, local declarations L, and a finite ordered set $\texttt{ECCs} = \{\texttt{ECC}_1, \ldots, \texttt{ECC}_n\}$ of ECCs.

Refined HCECCs allow nesting of parallel ECCs within ECC states. Fig. 5(b) shows a refined HCECC containing an ECC with two states: `START` and `RefinedState`. The state `RefinedState` is refined and contains two concurrent ECCs: `Para1` and `Para2`. The top-level ECC is the refined ECC, `RefinedState` is the refined state, and `Para1` and `Para2` are the refining behaviors. All refined and refining ECCs share the interface and local declarations of the block.

Definition 5 (refined HCECC): A refined HCECC $\texttt{HCECC}_{\triangleright}$ is a tuple $\langle \mathcal{I}, L, \texttt{ECC}, \texttt{ECCs}, \triangleright \rangle$ containing an interface $\mathcal{I}$, local declarations L, a top-level refined ECC `ECC`, and a set `ECCs` of refining ECCs. The state-refinement function $\triangleright : S \rightarrow 2^{\texttt{ECCs}}$ (S is the set of states in `ECC`) maps states in `ECC` to an ordered subset of refining ECCs in `ECCs`.

For the refined HCECC shown in Fig. 5(b), $\texttt{ECCs} = \{\texttt{Para1}, \texttt{Para2}\}$, and $\triangleright(\texttt{START}) = \emptyset$ (nonrefined state) and $\triangleright(\texttt{RefinedState}) = \{\texttt{Para1}, \texttt{Para2}\}$. Def. 5 defines refined HCECCs with a single level of refinement, but HCECCs can have nested refinement, as discussed in the next section.

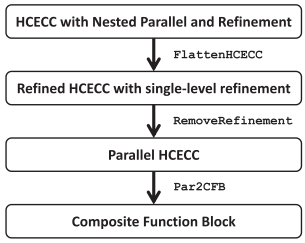
B. Synchronous Semantics for HCECCs

1) Execution Semantics of Parallel HCECCs: A parallel HCECC executes using the following sequence every tick. In step A, the block samples input events and updates input variables. In step B, ECCs contained in `ECCs` execute in order. For example, the HCECC in Fig. 5(a) executes ECC1 and then ECC2. Finally, in step C, an output event is emitted once if it is emitted by any ECC. Associated output variables are updated to the latest values assigned to them in the current tick following the order of execution of ECCs in B.

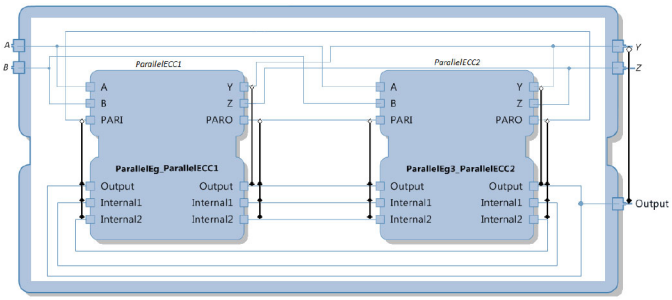
Variable updates propagate from the first ECC to the last ECC every tick. For example, for the HCECC in Fig. 5(a), if ECC1 updates any internal variables, ECC2 uses the updated values while executing in the same tick. ECC1 can read updates by ECC2 only in the next tick. The fixed order of execution ensures deterministic execution without causality issues. This semantics where a block is invoked only once relative to an event occurrence is fully compliant to ver. 2 of the IEC 61499 standard [1]. In ver. 1, a block can be invoked multiple times relative to an event occurrence, leading to ambiguities in execution semantics. Stability research can address such issues to ensure deterministic execution [22], [23]. However, stability research is not required for the proposed semantics.

2) Execution Semantics of Refined HCECCs: Every tick, a refined HCECC executes as follows. If the current state is not a refined state (case 1), the HCECC executes like a BFB (see Section III-B1). For example, the HCECC shown in Fig. 5(b) executes like a BFB when it is in state `Start`. If, during the current tick, a transition to a refined state is taken, the actions of the refined state, and the initial states of all refining behaviors are executed in order. For example, if a transition from state `Start` to the refined state `RefinedState` is taken, the block emits the output X (action of `RefinedState`), followed by executing the actions of the initial states S1 and T1 of the refining ECCs.

If the HCECC is in a refined state (case 2), it first samples the interface inputs and updates variables (step A). Then, in step B, the outgoing transitions of the refined state are evaluated. For example, if the HCECC from Fig. 5(b) is in state



F6:1 Fig. 6. Overview of HCECC translation.



F7:1 Fig. 7. Parallel ECCs flattened to a function block network.

`RefinedState` and the outgoing transition to state `Start` is enabled, execution is identical to case 1 above. In step C, if no transition fires in step B, the HCECC executes like a parallel HCECC (see Section IV-B1)—ECCs refining the current state execute in order. Finally, in step D, interface output events are emitted (if emitted by any of the ECCs), and output variable values are updated.

Refined HCECCs execute deterministically because in every tick, they follow the deterministic execution semantics of basic blocks (case 1) or parallel HCECCs (case 2).

V. Translating HCECCs to IEC 61499

As refined HCECCs execute like parallel HCECCs, and the semantics of parallel HCECCs are based on composite blocks, it is possible to create a sound translation from HCECCs to equivalent IEC 61499 function blocks as shown in Fig. 6. Detailed proofs can be found at http://tinyurl.com/hcecc2015.

A. Parallel HCECC to CFB Conversion

Algorithm 1 converts a parallel HCECC into a CFB by first creating multiple FB instances, and then connecting them. These steps are illustrated by transforming the parallel HCECC in Fig. 5(a) to the CFB network in Fig. 7.

Creating FB instances: Algorithm 1 transforms each ECC in the parallel HCECC into an instance of a newly created BFB (lines 1–22). The newly created instances are ordered in the same manner as the ECCs in the parallel HCECC (lines 8–21) and are contained in the set `FBs`.

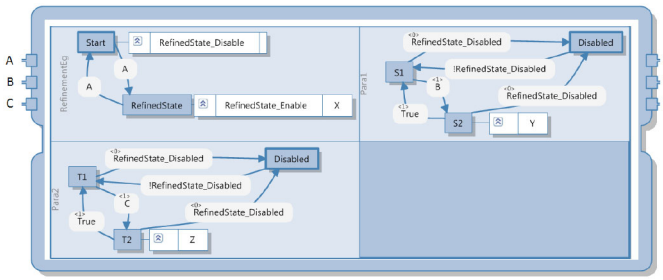
The newly created instances have the same interface $\mathcal{I}'$ (line 6) which has the same input and output events as the HCECC interface, and additional input and output events *PARI* and *PARO* (line 3). These additional events propagate updated variable values within the network (described in the next paragraph). $\mathcal{I}'$ has input variables corresponding to all input, output, and local variables of the parallel HCECC, and its output variables correspond to all local and output variables of the parallel HCECC (line 4). $\mathcal{I}'$ retains the input and output associations of the original HCECC. In addition, *PARI* and *PARO* are associated with each input and output variable of $\mathcal{I}'$, respectively. The local declarations L' for each instance contains no internal variables and has local versions of all algorithms of the parallel HCECC (line 7). A new algorithm $UpdateOutputs$ is used to ensure that each basic block instance propagates output data forward in every tick.

Events *PARI* and *PARO* replicate the communication between the ECCs of the parallel HCECC (using internal

Algorithm 1. Algorithm `Par2CFB`

Input: Parallel HCECC $\langle \mathcal{I}, L, \texttt{ECCs} \rangle$
Output: CFB `CFB`

1 //`Create FB instances`;
2 Ordered set `FBs` $= \emptyset$;
3 $E_I{}' = E_I \cup \{PARI\}$; $E_O{}' = E_O \cup \{PARO\}$;
4 $V_I{}' = V_I \cup V_L \cup V_O$; $V_O{}' = V_L \cup V_O$;
5 $\alpha_I{}' = \alpha_I \cup \{(PARI, v_I) | v_I \in V_L \cup V_O\}$;
$\alpha_O{}' = \alpha_O \cup \{(PARO, v_O) | v_O \in V_O\}$;
6 $\mathcal{I}' = \langle E_I{}', V_I{}', \alpha_I{}', E_O{}', V_O{}', \alpha_O{}' \rangle$;
7 $L' = \langle \emptyset, A_L \cup \{UpdateOutputs\} \rangle$;
8 **for** *each* $ECC = \langle S, s_0, \lambda, T \rangle \in ECCs$ **do**
9 $S' = S$; $s_0' = s_0$; Initialize λ', T';
10 **for** *each* $s \in S$ **do**
11 $\lambda'(s) = \lambda(s) \oplus \{UpdateOutputs, PARO\}$;
12 $T'(s) = T(s)$;
13 **if** *there is no transition* $s \xrightarrow{1} s_1$ **then**
14 Loop state s_l; $S' = S' \cup \{s_l\}$;
15 $\lambda'(s_l) = \{UpdateOutputs, PARO\}$;
16 $T'(s) = T'(s) \oplus \{s \xrightarrow{1} s_l\}$;
17 $T'(s_l) = \{s_l \xrightarrow{e,b} s_1 | s \xrightarrow{e,b} s_1\}$;
18 **end**
19 **end**
20 $\texttt{ECC}' = \langle S', s_0', \lambda', T' \rangle$; $\texttt{BFB}' = \langle \mathcal{I}', L', \texttt{ECC}' \rangle$;
21 $\texttt{fb} = \langle name_i, \texttt{BFB}' \rangle$; $\texttt{FBs} = \texttt{FBs} \oplus \{\texttt{fb}\}$;
22 **end**
23 //`Create connections`;
24 $n = |\texttt{FBs}|$; $C_{events} = \emptyset$; $C_{var} = \emptyset$;
25 **for** *each* $i \in [1, n]$ **do**
26 Select the i-th FB instance $\texttt{fb}_i \in \texttt{FBs}$;
27 $C_{events} = C_{events} \cup \{(e_I \mapsto \texttt{fb}_i.e_I) | e_I \in E_I\}$;
28 $C_{events} = C_{events} \cup \{(\texttt{fb}_i.e_O \mapsto e_O) | e_O \in E_O\}$;
29 **if** $i = 1$: $C_{var} = C_{var} \cup \{(v_I \mapsto \texttt{fb}_i.v_I) | v_I \in V_I\}$;
30 **if** $i = n$: $C_{var} = C_{var} \cup \{(\texttt{fb}_i.v_O \mapsto v_O) | v_O \in V_O\}$;
31 **if** $i < n$ **then** $\texttt{fb}_j = \texttt{fb}_{i+1}$ **else** $\texttt{fb}_j = \texttt{fb}_1$;
32 $C_{events} = C_{events} \cup \{(\texttt{fb}_i.PARO \mapsto \texttt{fb}_j.PARI)\}$;
33 $C_{var} = C_{var} \cup \{(\texttt{fb}_i.v_L \mapsto \texttt{fb}_j.v_L | v_L \in V_L\}$;
34 $C_{var} = C_{var} \cup \{(\texttt{fb}_i.v_O \mapsto \texttt{fb}_j.v_O | v_O \in V_O\}$;
35 **end**
36 //`Create CFB`;
37 $\texttt{FBNetwork} = \langle \texttt{FBs}, C_{events}, C_{var} \rangle$;
38 $\texttt{CFB} = \langle \mathcal{I}, \texttt{FBNetwork} \rangle$;
39 **return** CFB;



F8:1 Fig. 8. Parallel ECCs equivalent to refinement in Fig. 5(b)

variables) in the (to be created) CFB network by explicit propagation of data between the corresponding instances. Each instance emits *PARO* every tick, forcing an update of all output variables using algorithm $UpdateOutputs$ (line 11). The *PARO* output of a FB instance is connected to the *PARI* input of the *next* FB instance in the network (lines 31–35), allowing a forward propagation of data from each instance.

Each ECC in the HCECC is converted into an ECC′ of a corresponding BFB instance BFB′ (lines 8–22). ECC′ initially contains the same states and initial state as ECC (line 9) and each state s from ECC retains all original actions and transitions in ECC′, but also has two new actions—the execution of the algorithm $UpdateOutputs$ and the emission of the event *PARO* for forward data propagation.

Lines 13–18 create additional *loop* states in ECC. Loop states are needed to ensure that in any tick, if none of the original transitions inherited by the current state in ECC′ are taken, a transition to a loop state will ensure that the instance will still correctly propagate variable values through the network by doing actions $UpdateOutputs$ and *PARO*. Therefore, for every state s that does not have an always-true transition (line 13), a corresponding loop state s_l in ECC′ is introduced (line 14). Also, each state has an additional always-true but lowest priority transition to its corresponding loop state (line 16). The loop state itself has the same transitions as its corresponding state s (line 17).

Creation of connections: The event and variable connections for each FB instance are created as follows (the ith FB instance in FBs is noted as $\texttt{fb}_i$).

1) All input/ output events of the CFB interface $\mathcal{I}$ are connected to corresponding input and output events of $\texttt{fb}_i$ (lines 27, 28).
2) If $\texttt{fb}_i$ is the *first* or *last* FB instance in the CFB network, all input or output variables (respectively) of the HCECC interface $\mathcal{I}$ are connected to the corresponding input or output variables (respectively) of $\texttt{fb}_i$ (lines 29, 30).
3) The *PARO* output event of $\texttt{fb}_i$ is connected to the *PARI* input event of the next instance $\texttt{fb}_j$ (line 32). $\texttt{fb}_j$ is the next instance $\texttt{fb}_{i+1}$ after $\texttt{fb}_i$ in FBs, or if $\texttt{fb}_i$ is the last element of FBs, then $\texttt{fb}_j$ is the first element $\texttt{fb}_1$ of FBs (line 31). For example, in Fig 7, *PARO* of ParallelECC1 connects to *PARI* of ParallelECC2 and *PARO* of ParallelECC2 connects to *PARI* of instance ParallelECC1. The output variables of $\texttt{fb}_i$ corresponding to the internal and output variables of the parallel HCECC are connected to corresponding input variables of $\texttt{fb}_j$ (line 33).

Algorithm 2. Algorithm RemoveRefinement

Input: $\mathcal{I}, L = \langle V_L, A_L \rangle, \texttt{ECC} = \langle S, s_0, \lambda, T \rangle, s \in S, \texttt{ECCs}_s$
Output: Pair $(\texttt{ECCs}_\triangleright, L_\triangleright)$

1 //Modify locals;
2 Variable set $V_L\prime = V_L \cup \{s_\triangleright\_Disabled\}$ ($s_\triangleright\_Disabled$ is of type Boolean);
3 Algorithm set $A_L{}' = A_L \cup \{s_\triangleright\_Enable, s_\triangleright\_Disable\}$;
4 //Create new local declarations;
5 $L_\triangleright = \langle V_L\prime, A_L{}' \rangle$;
6 //Create $\texttt{ECCs}_\triangleright$;
7 Initialize ordered set of ECCs $\texttt{ECCs}_\triangleright = \emptyset$;
8 //Modify ECC;
9 Initialize empty action map λ_0;
10 //Modify states;
11 **for** *each state* $s \in S$ **do**
12 **if** $s = s_\triangleright$ **then**
13 $\lambda_0(s) = \lambda(s) \oplus \{s_\triangleright\_Enable\}$;
14 **end**
15 **else if** $s_\triangleright$ *has a transition* $(in\ T)$ *to* s **then**
16 $\lambda_0(s) = \lambda(s) \oplus \{s_\triangleright\_Disable\}$
17 **end**
18 **else**
19 $\lambda_0(s) = \lambda(s)$;
20 **end**
21 **end**
22 $\texttt{ECC}_0 = \langle S, s_0, \lambda_0, T \rangle$;
23 $\texttt{ECCs}_\triangleright = \texttt{ECCs}_\triangleright \oplus \{\texttt{ECC}_0\}$;
24 //Modify ECCs in $\texttt{ECCs}_s$;
25 **for** *each* $ECC_i = \langle S_i, s_{0_i}, \lambda_i, T_i \rangle \in \texttt{ECCs}_s$ **do**
26 Create set of states $S_{\triangleright i} = S_i \cup \{Disabled\}$;
27 Set state $s_{0_{\triangleright i}} = Disabled$;
28 **if** $s_\triangleright = s_0$ **then**
29 $s_{0_{\triangleright i}} = s_{0_i}$;
30 **end**
31 Initialize action map $\lambda_{\triangleright i} = \lambda_i$;
32 Add $\lambda_{\triangleright i}(Disabled) = \emptyset$;
33 Initialize transition function $T_{\triangleright i} = T_i$;
34 **for** *each* $s \in S_{\triangleright i}$ **do**
35 **if** $s = Disabled$ **then**
36 $T_{\triangleright i}(s) = \{s \xrightarrow{1,!s_\triangleright\_Disabled} s_{0_i}\}$;
37 **else**
38 $T_{\triangleright i}(s) = \{s \xrightarrow{1,s_\triangleright\_Disabled} Disabled\} \oplus T_i(s)$;
39 **end**
40 **end**
41 Create ECC $\texttt{ECC}_{\triangleright i} = \langle S_{\triangleright i}, s_{0_{\triangleright i}}, \lambda_{\triangleright i}, T_{\triangleright i} \rangle$;
42 $\texttt{ECCs}_\triangleright = \texttt{ECCs}_\triangleright \oplus \{\texttt{ECC}_{\triangleright i}\}$;
43 **end**
44 **return** $Pair\ (ECCs_\triangleright, L_\triangleright)$;

A proof by induction shows that the CFB generated from Algorithm 1 has the same behavior the source parallel HCECC.

B. Refinement to Parallel Conversion

AlgorithmRemoveRefinement (Algorithm 2) translates the refinement of a single state in a refined HCECC into parallel ECCs. For the HCECC in Fig. 5(b), the algorithm

T1:1 TABLE I
T1:2 HCECC AND ECC SIZE COMPARISON

		ECCs				HCECCs			
No.	**Benchmark**	**# Basic blocks**	**Connections**	**States**	**Transitions**	**Unique blocks**	**Connections**	**States**	**Transitions**
B1	Watch	3	21	17	24	1	0	14	20
B2	Distribution station	2	18	10	12	1	0	10	12
B3	Drill station	2	10	9	10	1	0	9	9
B4	Water monitor	4	31	16	38	3	28	16	38
B5	Cruise controller	5	32	15	42	1	0	11	38
B6	MP3player	4	22	14	24	1	0	10	15

produces the parallel ECCs shown in Fig. 8. The algorithm executes as follows. Lines 1–5 of Algorithm 2 modify the local declarations, adding a new internal variable ($s_{\triangleright}\_Disabled$) and two new algorithms (lines 2–3). The new algorithms ($s_{\triangleright}\_Enable, s_{\triangleright}\_Disable$) clear and set the $s_{\triangleright}\_Disabled$ variable respectively.

The first step in constructing the set $\texttt{ECCs}_{\triangleright}$ of parallel ECCs involves modifying the main ECC (lines 8–22). An algorithm $s_{\triangleright}\_Enable$ is added to the actions of $s_{\triangleright}$ (line 13), so that whenever state $s_{\triangleright}$ is reached, the variable $s_{\triangleright}\_Disabled$ is cleared (allowing refining ECCs to execute). Similarly, in every state s to which $s_{\triangleright}$ has an outgoing transition, an algorithm $s_{\triangleright}\_Disable$ is added to the actions to set the variable $s_{\triangleright}\_Disabled$ and to stop the refining ECCs from executing. For example, for the parallel ECCs in Fig. 8, the action map of the first ECC is changed as follows. $\lambda_0(\texttt{Start})$ takes the value $\{RefinedState\_Disable\}$, and $\lambda_0(\texttt{RefinedState})$ takes the value $\{RefinedState\_Enable, X\}$, where X was part of the original refined HCECC. The modified main ECC is inserted as the first element into the set $\texttt{ECCs}_{\triangleright}$. It, therefore, executes first, and may enable or disable other ECCs.

Lines 24–43 modify each refining ECC, adding a new state `Disabled` (line 26) to model the disabled behavior when the refined ECC is not in state $s_{\triangleright}$. Fig. 8 shows the `Disabled` state for each of the modified refining ECCs (`Para1` and `Para2`). `Disabled` is set as the initial state of each refining ECC (line 27). However, if the refined state $s_{\triangleright}$ is the initial state of the refined ECC, refining ECCs' original initial states are retained (line 29). The modified refining ECC states retain their actions from the original refining ECC (line 31) and state Disabled has no actions (line 32).

The transition function for the modified refining ECC retains all original transitions (line 33). New transitions are added to model the enabled/disabled behavior, based on the value of the variable $s_{\triangleright}\_Disabled$. All states have a highest priority transition to `Disabled` when $s_{\triangleright}\_Disabled$ is true (line 38). Disabled has a single outgoing transition to the original initial state of the refining ECC (line 36). For example, each state in ECC `Para1` in Fig. 8 has a transition to `Disabled` when $RefinedState\_Disabled$ is true, and `Disabled` has a transition to the original initial state `S1` when $RefinedState\_Disabled$ is false. Finally, line 41 constructs the modified refining ECC and adds it to $\texttt{ECCs}_{\triangleright}$ on line 42. After all refining ECCs have been processed in this way, line 44 returns $\texttt{ECCs}_{\triangleright}$ and modified local declarations $L_{\triangleright}$.

A refined HCECC containing multiple (m) refined states is transformed into a parallel HCECC by applying Algorithm 2 m times (once for each refined state). HCECCs with nested refinement are processed in a depth-first fashion starting from the inner-most HCECC, until the block is reduced to a refined HCECC with only single-level refinement. A proof of soundness of the proposed transformations can be done by induction, showing equivalence between the execution of a refined HCECC, the corresponding parallel HCECC, and the corresponding IEC 61499 program, under the proposed synchronous semantics.

T2:1 TABLE II
T2:2 DEVELOPMENT TIME (TIME IN MINUTES)

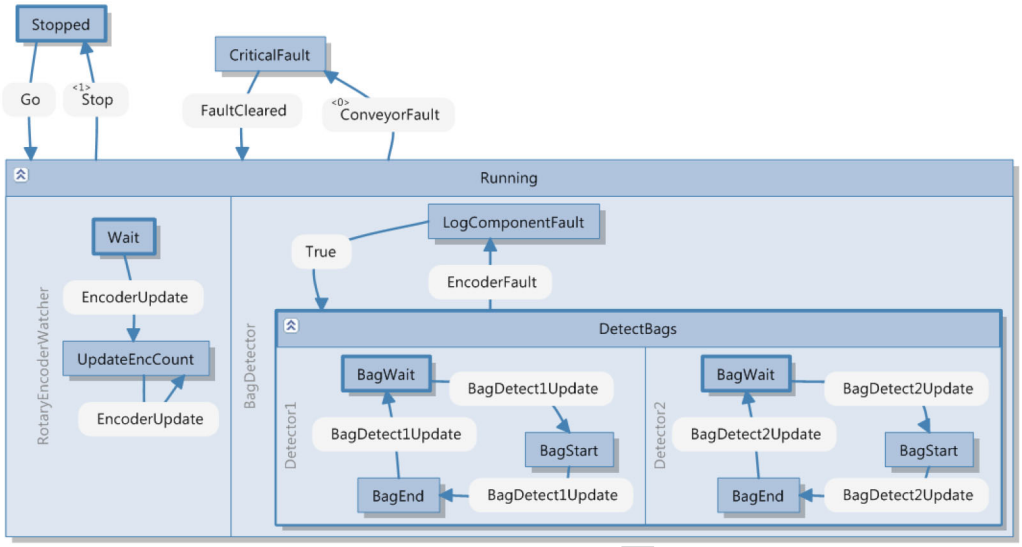
	Developer A		Developer B		Developer C	
No.	**ECCs**	**HCECCs**	**ECCs**	**HCECCs**	**ECCs**	**HCECCs**
B1	8	4	11	7	53	40
B2	5	3	14	10	50	34
B3	4	2	9	4	30	18
B4	11	10	23	20	70	45
B5	12	6	19	10	43	25
B6	8	3	20	9	35	16

VI. RESULTS

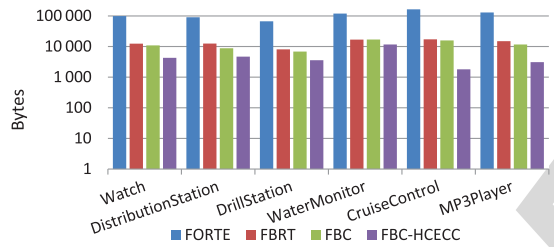
A qualitative and quantitative evaluation of HCECCs was done using the Auckland Function Block Benchmark [24], which contains designs varying in size and complexity of algorithms. Each benchmark was reimplemented using HCECCs, and subsequently compared to the original model. All benchmarks are available at http://tinyurl.com/hcecc2015.

Table I compares the sizes of the standard and HCECC versions of each benchmark. Columns 2–5, respectively, show the numbers of basic blocks, connections, states and transitions in the standard version. Columns 5–8, respectively, show the numbers of blocks, connections, states, and transitions in the HCECC version. For each benchmark, the size of the HCECC version was either smaller than or the same as the standard version. Most compression was achieved when refinement was used because enforcing refinement-like hierarchy in standard function blocks requires many more transitions and states.

Table II shows the times taken by three developers to develop the benchmarks shown in Table I. Developer A was an expert in HCECC design, developer B had some experience with statecharts and a working knowledge of IEC 61499, and developer C was completely new to both HCECCs and IEC 61499. Overall, all developers took significantly lesser amount of time to develop HCECCs. While these results are promising, a wider usability study is beyond the scope of this paper.



F9:1 Fig. 9. Multilevel refinement example.



F10:1 Fig. 10. Benchmark code size comparison.

T3:1 TABLE III

T3:2 PERFORMANCE COMPARISON (TIME IN MILLI-SECONDS)

	FORTE	FBRT	FBC ECC	FBC HCECC
B1	1962.6	77.0	75.0	24.0
B2	1532.0	120.4	68.2	30.0
B3	387.0	56.3	44.0	19.0
B4	1928.4	214.5	83.0	33.0
B5	877.0	130.0	64.0	27.8
B6	1251.2	144.7	123.0	96.0

HCECCs can model complex behaviors that cannot be easily created in standard function blocks. Fig. 9 shows the HCECC for a conveyor controller (algorithms omitted for readability). This model may react to up to three different events in a single tick. Modeling such behaviors using standard (flat) ECCs results in exponentially more states and transitions, requires extra effort in creating multiple blocks and connecting them, and takes longer to design and maintain.

We extended the function block compiler (FBC) [7] to create FBC-HCECC, a compiler to compile HCECCs into C code using the proposed synchronous semantics. Fig. 10 compares code sizes produced by FORTE v1.7.1, FBRT v20081003, FBC, and FBC-HCECC. For FBRT code, the Java virtual machine (JVM) size was removed for a fair comparison. Code generated by FBC-HCECC was on average 1.18 times smaller than FBC, which in turn produced much smaller code than FORTE and FBRT. In some cases, like benchmark B4, FBC-HCECC generated code was 1.70 times smaller than FBC. The reduction in code size comes from HCECCs allowing the same functionality within smaller models, as per Table I.

Code for each benchmark from different compilers was run on a windows PC with an Intel Core i7 920 processor and 18GB RAM. Java v7 (JDK and JRE) was used to compile and run FBRT-generated code. 4DIAC and FBDK front-ends were used to design and run FORTE and FBRT programs, respectively. C code from FBC and FBC-HCECC was compiled using Visual Studio's C compiler with the -O2 optimization switch. Each program was then executed with a common test-file containing a sequence of one million input vectors. Each vector contained a random event and random values for variables. A comparison of the execution times is shown in Table III. FBC-HCECC programs executed 1.3 to 3.1 times faster than FBC programs, with an average speedup of 2.3 possibly due to smaller model sizes. FBC-HCECC programs ran on average 3.8 and 42.7 times faster than FBRT and FORTE programs, which required additional runtimes.

Overall, HCECCs enable more compact designs that are faster to develop, smaller in code size, and execute faster than standard IEC 61499 designs. A side-effect of using the proposed semantics is that a fixed ordering of blocks within composite blocks (and also parallel and refined HCECCs) must be defined. The FBC-HCECC compiler uses the ordering as per XML descriptions, which is standard practice. Designers can change this ordering by editing the XML descriptions. HCECCs can increase cohesion and reduce coupling between blocks in some cases, such as exceptional handling scenarios. However, it is possible to misuse this framework and integrate loosely coupled blocks, reducing maintainability.

VII. CONCLUSION

This paper introduces HCECCs, consisting of hierarchical and concurrent operators for IEC 61499 ECCs, inspired by statecharts. Refined HCECCs efficiently model exception handling and other hierarchical behaviors, whereas parallel

HCECCs simplify the modeling of concurrent processes and enable the monitoring of simultaneous events in a single block. A synchronous semantics for IEC 61499 and HCECCs is proposed under which HCECCs can be translated to CFBs. Benchmarking results show that HCECCs provide reduced model sizes, which helps reduce development time, compiled code size, and execution time. Future directions include using HCECCs under other IEC 61499 semantics, optimizing HCECC to network translation, studying how HCECCs affect cohesion and coupling as well as code maintainability or reusability, and providing tool-support for HCECCs.

REFERENCES

[1] J. H. Christensen, T. Strasser, A. Valentini, V. Vyatkin, and A. Zoitl, "The IEC 61499 function block standard: Overview of the second edition," *ISA Autom. Week*, vol. 6, 2012.

[2] 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.

[3] IEC, *Committee Draft for Vote: IEC 61499-1: Function Block-Part 1 Architecture*. Geneva, Switzerland: IEC, 2004.

[4] A. Zoitl and H. Prahofer, "Guidelines and patterns for building hierarchical automation solutions in the IEC 61499 modeling language," *IEEE Trans. Ind. Informat.*, vol. 9, no. 4, pp. 2387–2396, Nov. 2013.

[5] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Programm.*, vol. 8, no. 3, pp. 231–274, Jun. 1987.

[6] J. Kim, I. Kang, J.-Y. Choi, and I. Lee, "Timed and resource-oriented statecharts for embedded software," *IEEE Trans. Ind. Informat.*, vol. 6, no. 4, pp. 568–578, Nov. 2010.

[7] L. H. Yoong, P. 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.

[8] M. Di Natale, Q. Zhu, A. Sangiovanni-Vincentelli, and S. Tripakis, "Optimized implementation of synchronous models on industrial LTTA systems," *J. Syst. Archit.*, vol. 60, no. 4, pp. 315–328, 2014.

[9] M. Bonfè, C. Fantuzzi, and C. Secchi, "Design patterns for model-based automation software design and implementation," *Control Eng. Pract.*, vol. 21, no. 11, pp. 1608–1619, 2013.

[10] J.-P. Talpin *et al.*, "Formal verification of synchronous data-flow program transformations toward certified compilers," *Front. Comput. Sci.*, vol. 7, no. 5, pp. 598–616, 2013.

[11] R. Nakamura, F. Arakawa, and M. Edahiro, "Simple one-to-one architecture for parallel execution of embedded control systems," in *Proc. IEEE Int. Conf. Cyber-Phys. Syst. Netw. Appl. (CPSNA)*, 2014, pp. 25–30.

[12] P. Gaj, J. Jasperneite, and M. Felser, "Computer communication within industrial distributed environment—A survey," *IEEE Trans. Ind. Informat.*, vol. 9, no. 1, pp. 182–189, Feb. 2013.

[13] A. Girault, B. Lee, and E. A. Lee, "Hierarchical finite state machines with multiple concurrency models," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 18, no. 6, pp. 742–760, Jun. 1999.

[14] D. Latella, I. Majzik, and M. Massink, "Towards a formal operational semantics of UML statechart diagrams," in *Formal Methods for Open Object-Based Distributed Systems*. New York, NY, USA: Springer, 1999, pp. 331–347.

[15] A. Barji, N. Hagge, and B. Wagner, "Comparative study of using CNet, IEC 61499, and statecharts for behavioral models of real-time control applications," in *Proc. IEEE Conf. Emerg. Technol. Fact. Autom. (ETFA'06)*, 2006, pp. 750–757.

[16] D. Tikhonov, D. Schutz, S. Ulewicz, and B. Vogel-Heuser, "Towards industrial application of model-driven platform-independent PLC programming using UML," in *Proc. 40th Annu. IEEE Conf. Ind. Electron. Soc. (IECON'14)*, 2014, pp. 2638–2644.

[17] K. Thramboulidis, "Model-integrated mechatronics-toward a new paradigm in the development of manufacturing systems," *IEEE Trans. Ind. Informat.*, vol. 1, no. 1, pp. 54–61, Jan. 2005.

[18] G. D. Shaw, P. S. Roop, and Z. Salcic, "A hierarchical and concurrent approach for IEC 61499 function blocks," in *Proc. IEEE Conf. Emerg. Technol. Fact. Autom. (ETFA'09)*, 2009, pp. 1–8.

[19] V. Dubinin, V. Vyatkin, and T. Pfeiffer, "Engineering of validatable automation systems based on an extension of UML combined with function blocks of IEC 61499," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA'05)*, 2005, pp. 3996–4001.

[20] V. N. Dubinin and V. Vyatkin, "Semantics-robust design patterns for IEC 61499," *IEEE Trans. Ind. Informat.*, vol. 8, no. 2, pp. 279–290, May 2012.

[21] S. Andalam, P. S. Roop, A. Girault, and C. Traulsen, "A predictable framework for safety-critical embedded systems," *IEEE Trans. Comput.*, vol. 63, no. 7, pp. 1600–1612, Jul. 2014.

[22] F. Schumacher and A. Fay, "Formal representation of GRAFCET to automatically generate control code," *Control Eng. Pract.*, vol. 33, pp. 84–93, 2014.

[23] M. Sogbohossou and A. Vianou, "Formal modeling of grafcets with time petri nets," *IEEE Trans. Control Syst. Technol.*, vol. 23, no. 5, pp. 1978–1985, Sep. 2015.

[24] L. H. Yoong, P. S. Roop, and Z. Salcic, "Implementing constrained cyber-physical systems with IEC 61499," *ACM Trans. Embedded Comput. Syst. (TECS)*, vol. 11, no. 4, pp. 78:1–78:22, 2012.

Roopak Sinha (S'03–M'13) received the B.E. (Hons), M.C.E., and Ph.D. degrees.

He has worked with INRIA, Grenoble, France, and the University of Auckland, Auckland, New Zealand. Currently, he is a Senior Lecturer with the School of Computer and Mathematical Sciences, Auckland University of Technology, Auckland. His research interests include next-generation formal frameworks for designing large-scale embedded software with application in industrial automation systems, Internet-of-Things, and intelligent transportation systems.

Partha S. Roop (M'94) received the Ph.D. degree in computer science (software engineering) from the University of New South Wales, Sydney, NSW, Australia, in 2001.

He is currently an Associate Professor and is the Director of the Computer Systems Engineering Program, Department of Electrical and Computer Engineering, University of Auckland, Auckland, New Zealand. His research interests include the design and verification of embedded systems and executable biology. In particular, he is developing techniques for the design of embedded applications in automotive, robotics, intelligent transportation, and medical devices that meet functional safety standards.

Gareth Shaw received the B.E. and the Ph.D. degrees in electrical and electronic engineering from the University of Auckland, Auckland, New Zealand, in 2007 and 2013, respectively.

He is a Mobile Development Team Lead with Fiserv New Zealand, Auckland. His research interests include design languages and their compilation, distributed systems, code generation, and complex digital system design.

Zoran Salcic (S'75–M'76–SM'98) received the B.E., M.E., and Ph.D. degrees in electrical and computer engineering from Sarajevo University, Sarajevo, Bosnia and Herzegovina, in 1972, 1974, and 1976, respectively.

He is a Professor of Computer Systems Engineering with the University of Auckland, Auckland, New Zealand. He has authored over 300 peer-reviewed journal and conference papers, and several books. His research interests include complex digital systems, custom-computing machines, embedded systems and their implementation, design automation tools, hardware-software co-design, models of computation and languages for concurrent and distributed systems, and cyber-physical systems.

Prof. Salcic is a Fellow of the Royal Society New Zealand. He was the recipient of the Alexander von Humboldt Research Award in 2010.

Matthew M. Y. Kuo received the B.E. (Hons) degree in electrical and computer systems engineering from the University of Auckland, Auckland, New Zealand. He is currently pursuing the Ph.D. degree.

He is currently involved in an industrial project focused on the Internet of Things with UniServices, Auckland. His research interests include synchronous programming, static timing analysis, and precision timed industrial automation systems.

QUERIES

Q1: Please provide postal code for affiliations.
Q2: Please provide page range for Ref. [1].
Q3: Please provide the field of study, institutional details, location, and year of all the degrees of the author Roopak Sinha.
Q4: Please spell out the term “INRIA”.
Q5: Please provide the institutional details of the Ph.D. degree of the author Matthew M. Y. Kuo.

Hierarchical and Concurrent ECCs for IEC 61499 Function Blocks

Roopak Sinha, *Member, IEEE*, Partha S. Roop, *Member, IEEE*, Gareth Shaw, Zoran Salcic, *Senior Member, IEEE*, and Matthew M. Y. Kuo

***Abstract*—IEC 61499 enables component-oriented descriptions of complex industrial processes facilitating model-driven engineering. One aspect that is lacking, however, is the ability to directly express Statecharts-like hierarchy and concurrency within basic function blocks (BFBs). Such features can significantly enhance function blocks and help create more succinct and readable specifications. We propose a new syntactic extension to the standard called hierarchical and concurrent execution control chart (HCECC). A major roadblock for any suggested changes to the standard is the need for compliance. Our approach extends the synchronous execution semantics of IEC 61499, where HCECCs are purely syntactic sugar. Using a revised synchronous semantics, our compiler generates standard compliant C code from HCECCs. Benchmarking and usability studies reveal the relative superiority of the proposed approach over existing approaches.**

***Index Terms*—Execution semantics, hierarchical state machines, IEEE 61499, parallel execution, statecharts.**

I. Introduction

THE INCREASING size and complexity of industrial control systems have motivated the development of new design paradigms to aid system designers. The IEC 61499 standard [1], [2] proposes component-oriented models called function blocks for developing distributed control systems. A function block describes both control and data flow within a single module. By providing a graphical specification framework, the standard simplifies system design with a top-down approach and encourages the reuse of function blocks.

The unit of execution in IEC 61499 is a basic function block (BFB), which executes a finite state machine (FSM) known as an execution control chart (ECC) that describes the control flow of the block. ECCs are flat Moore-machines that allow modeling of only sequential behaviors. Therefore, a basic block cannot model concurrent behaviors. In addition, ECCs do not allow users to separate high-level behavior such as initialization and error handling of a block. As a result, more states and transitions are required to describe the same behavior compared to other graphical formalisms.

The standard also provides CFBs that are networks of interconnected function block instances. Networks model concurrency and structural hierarchy that basic blocks cannot handle efficiently. However, networks can easily get very complex, making them difficult to design, understand, and maintain. Moreover, in networks containing many function block instances, a significant amount of execution time is required for the flow of events and data between instances. Not only do such complex networks require a long time to design manually, network behavior becomes nonobvious and there is a higher chance of software bugs.

This issue has been noted in Annex E.1 of the IEC 61499 draft standard [3] as well as recent scholarly work [2], [4]. These works highlight the urgent need to introduce statecharts-like [5] hierarchy and parallelism within basic blocks. Statecharts are extensions of FSMs that allow a single FSM to contain concurrent or parallel as well hierarchical or refined behaviors. Statecharts have found popular use in the design of distributed control software [6]. Unfortunately, statecharts semantics [4] are incompatible with the IEC 61499 standard (e.g., it does not handle data flow modeling, unlike IEC 61499), and currently no extension of BFBs to model Statecharts exists.

This paper proposes the first Statecharts-like extensions of IEC 61499 called hierarchical and concurrent ECCs (HCECCs), which allow explicit modeling of hierarchy and concurrency within a BFB using two operators: 1) parallel; and 2) refinement. Any nesting of these operators can be resolved to a standard CFB, making HCECCs completely compliant to the standard. A synchronous execution semantics [7], [8] for HCECCs is proposed, which allows a complete deterministic execution of HCECCs as well as standard function blocks. The main contributions of this paper are formalizing IEC 61499 function blocks syntax and presenting a simplified synchronous semantics for the execution of function blocks, formulating the HCECC framework by introducing the parallel and refinement operators, and showing how HCECCs execute using the proposed simplified synchronous semantics, and presenting a sound translation of HCECCs into standard-compliant CFBs.

This paper is organized as follows. Section II presents a review of related works. Section III introduces a formalism for IEC 61499 and a description of the semantics for basic and CFBs. HCECCs are introduced in Section IV, and Section V describes their transformation to standard-compliant composite

Manuscript received March 21, 2014; revised August 15, 2015; accepted October 14, 2015. Paper no. TII-15-0754.R1.

R. Sinha is with the School of Computer and Mathematical Sciences, Auckland University of Technology, Auckland, New Zealand (e-mail: roopak.sinha@aut.ac.nz).

P. S. Roop, Z. Salcic, and M. M. Y. Kuo are with the Department of Electrical and Computer Engineering, University of Auckland, Auckland, New Zealand (e-mail: p.roop@auckland.ac.nz; z.salcic@auckland.ac.nz; mkuo005@aucklanduni.ac.nz).

G. Shaw is with Fiserv New Zealand, Auckland, New Zealand (e-mail: gareth@garethnz.com).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TII.2015.2496262

blocks. Section IV reports experimental results and Section VII provides concluding remarks and future directions.

II. Literature Review

Model-driven engineering as expounded by IEC 61499 is founded on models of computation. The standard allows modeling control flow using events and state machines within function blocks, and data flow using variables and algorithms. More recently, researchers have employed more expressive models such as statecharts [9] and synchronous data flow [10], and communicating sequential processes [11] for modeling control flow in industrial strength systems. Similarly, extending data flow modeling in industrial control systems has also been studied extensively [12]. Some formalisms like *charts (pronounced starcharts) [13] have been used to combine control and data flow models. However, for IEC 61499 systems, it is not possible to modify the standard.

The existing pool of statecharts-based frameworks serves as the logical starting point to build IEC 61499 compatible models to capture concurrency and hierarchy. Statecharts extend FSMs and allow parallelism using state diagrams with broadcast-based communication and hierarchy using a nested structure of states. statecharts are used in a wide range of application domains for modeling complex systems, and have been extended to generate popular modeling frameworks such as UML statecharts [14]. Unfortunately, statecharts and current variants cannot be used in IEC 61499 because model only control flow and do not capture data flow.

The need to introduce statecharts-like extensions within basic blocks is well documented [2]–[4]. Comparative studies between IEC 61499 and other frameworks like UML-statecharts [15], [16] help highlight this gap. Unfortunately, existing solutions to introduce statecharts-like extensions in IEC 61499 have limited scope. In [17], component interaction diagrams are proposed to allow modeling UML-statecharts like hierarchy within function block applications. However, these diagrams do not allow designers to embed concurrency and hierarchy within a single function block. In [4], ECCs within BFBs are extended based on the Monaco domain-specific language, which models a subset of statecharts. However, the authors discard this extension because of an inability to retrieve hierarchy from flattened ECCs, and the associated re-engineering and refactoring costs. On the other hand, the proposed framework is not domain specific, and hierarchical and concurrent behaviors can be automatically translated to networks of function blocks. In [18] and [19], informal statecharts-based extensions to basic blocks are proposed. In all cases, the issue of complying to the standard is overlooked.

Many different execution semantics of IEC 61499 have been proposed [1], [2], [20]. As compared to other semantics, the synchronous semantics developed in [7] excels in many respects. Synchronous systems execute in logical time instances called ticks. During each tick, a system reads inputs, processes them, and then emits outputs. A tick or macro-step is atomic and instantaneous. Internally though, macro-steps are usually sequenced into micro-steps. A micro-step pertains to a reaction of an individual component or the propagation of information within the system. Synchronous systems may suffer from *causality* cycles caused by inter-dependent micro-steps [8]. In [7], causality cycles are avoided by introducing a one-tick delay in communications between the blocks in a network, which slows down event propagation in large networks. The semantics proposed in this paper avoids causality cycles and propagation by using a static schedule for executing blocks in a network, following the semantics of PRET-C [21].

HCECCs allow the inclusion of concurrent and hierarchical behaviors within basic blocks, and are a subset of Statecharts without inter-level transitions. The simplified synchronous semantics for IEC 61499 ensure that programs can never have causality cycles. The proposed HCECC syntax and semantics allow translating HCECCs into standard IEC 61499 function block networks, making them fully standard compliant.

III. Formalization of IEC 61499

This section presents the syntax and simplified synchronous semantics of IEC 61499. A baggage handling system (BHS) containing many baggage entry and exit points connected via a network of conveyor belts is used as an illustrative example.

A. Syntax

The following key concepts and notations are used in this paper. A set $A = \{a_1, \ldots, a_n\}$ is a totally ordered set *iff* for any two distinct elements $a_i, a_j \in A$, $a_i <_A a_j$ if $i < j$ or otherwise $a_j <_A a_i$. $<_A$ is the antisymmetric and transitive *ordering relation* for set A. The subscript A of $<_A$ is omitted when the context is clear. The linear sum $C = A \oplus B$ of two ordered and disjoint sets A and B is the union of A and B, with A's elements ordered as in A and B's elements ordered as in B, and $a <_C b$ for every $a \in A$ and $b \in B$.

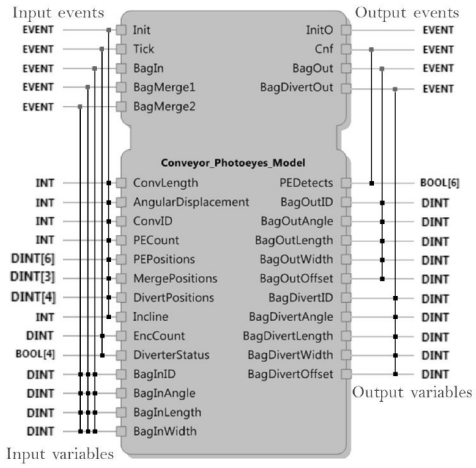
1) Interface: All function blocks have interfaces.

Definition 1 (Function block interface): A function block interface is a tuple $\mathcal{I} = \langle E_I, V_I, \alpha_I, E_O, V_O, \alpha_O \rangle$ where E_I, V_I, E_O, and V_O are finite sets of input events, input variables, output events and outputs variables, respectively. $\alpha_I \subseteq E_I \times V_I$ and $\alpha_O \subseteq E_O \times V_O$ are the sets of input and output associations.

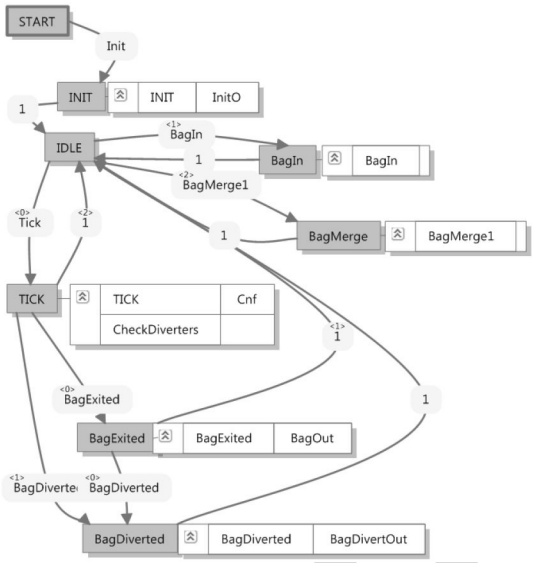
Fig. 1 shows the interface for `Conveyor_Photoeyes_Model`, which is used in the BHS to control a conveyor belt. The input events and variables appear on the left side, and outputs appear on the right side. Events are transient and are present in time instants when they are fired and absent otherwise. This is based on the well-known synchronous approach [8]. Variables have persistent values. Variables are sampled or updated every time an associated event is present. For example, in Fig. 1, event Init is associated with the variable *ConvLength*. Hence, *ConvLength* is sampled whenever `Init` is present.

2) Basic Function Block:

Definition 2 (BFB): `BFB` is a tuple $\langle \mathcal{I}, L, \texttt{ECC} \rangle$ with interface $\mathcal{I}$. The local declarations $L = \langle V_L, A_L \rangle$ contain an internal variables set V_L and a set of algorithms A_L. $\texttt{ECC} = \langle S, s_0, \lambda, T \rangle$ is an ECC with a finite set of states S, initial state $s_0 \in S$, an action function $\lambda : S \to (A_L \cup E_O)^*$ mapping states to a finite



F1:1 Fig. 1. Function block interface.



F2:1 Fig. 2. ECC.

sequence of algorithm executions and/or output event emissions, and the transition function $T : S \rightarrow 2^{(E_I \cup \{1\}) \times \mathcal{B}(\hat{V}) \times S}$. Here, $\hat{V}$ refers to the set of all combinations of valid values of internal, input and output variables, and $\mathcal{B}(\hat{V})$ refers to all Boolean expressions over $\hat{V}$. The transition function is restricted such that for any $s \in S$, $T(s)$ is totally ordered.

BFB has an interface, local declarations (internal variables and algorithms), and an ECC. The `Conveyor_Photoeyes_Model` block has the interface shown in Fig. 1, three internal variables BagModel, BagExited, and BagDiverted, and the set of algorithms $A_L = \{INIT, TICK, CheckDiverters, BagExited, BagDiverted, BagIn, BagMerge1\}$. Algorithms are written using any PLC or standard languages such as C or Java.

Fig. 2 shows the ECC of `Conveyor_Photoeyes_Model`. The ECC in Fig. 2 contains eight states with `START` as the initial state (highlighted with a bold outline). Each state has associated *actions* or a sequence of algorithm executions and output event emissions. For state `INIT`, the actions $\lambda(\texttt{INIT}) = INIT.InitO$ correspond to executing the algorithm $INIT$ and emitting the event $InitO$. For any state s, an outgoing transition $t \in T(s)$ is described as (e, b, s') (or $s \xrightarrow{e,b} s'$ in short-hand), where e is either a triggering input event or 1 (no triggering event), b is a Boolean expression evaluated on the values of internal, input and output variables, and s' is the destination state. For example, consider the transition $\texttt{START} \xrightarrow{\text{Init,true}} \texttt{INIT}$. The triggering event is *Init*, the Boolean condition is $true$, and the destination state is `INIT`. For any state s the set of transitions $T(s)$ is totally ordered. Transition order is shown in Fig. 2 using the labels $\langle 0 \rangle$, $\langle 1 \rangle, \ldots$ For example, the first transition from state `IDLE` is to state `Tick` (labeled with $\langle 0 \rangle$). The order is omitted when a state only has one outgoing transition.

3) Composite Function Block: CFB contains a network of interconnected function block instances. Fig. 3 combines two basic blocks into a CFB.

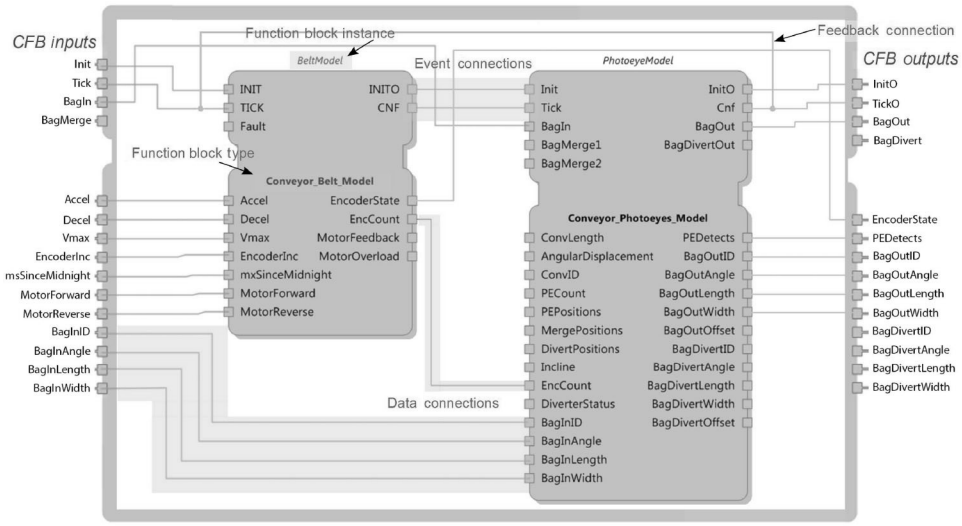
Function block instances: An instance is to a function block what an object is to a class in Java or C++: the former is a named *instance* of the latter *type*. An instance $\texttt{fb} = \langle name, FBT \rangle$ is called $name$ and has the type FBT (which can be a basic or composite block). The composite block in Fig. 3 contains two function block instances: $BeltModel$ of type `Conveyor_Belt_Model`, and $PhotoeyeModel$ of type `Conveyor_Photoeyes_Model`. Instances allow easy reuse of function blocks. The interface of an instance is used to connect it into a network. The notation $\langle \texttt{name} \rangle.\langle \texttt{event/variable name} \rangle$ refers to the inputs or outputs (events/variables) of an instance `fb`. For example, the output variable *PEDetects* of instance `PhotoeyeModel` is written as *PhotoeyeModel.PEDetects*. The instances of a composite block network are contained in a totally ordered set `FBs`.

Event connections: A CFB network contains the following types of event connections (the short-hand $s \mapsto d$ represents an event connection from s to d).

1) Input to input connections C_{EI2I}: Inputs of the CFB's interface may connect to inputs of instances in the network. For example, *Tick* $\mapsto$ `BeltModel`.*TICK*, as per Fig. 3.
2) Output to input connections C_{EO2I}: Output events of one instance may be read as inputs by another. For example, `BeltModel`.$CNF \mapsto$ `PhotoEyeModel`.*Tick* A *feedback* connection connects the output of an instance appearing later in `FBs` (an ordered set) to the input of an earlier instance in `FBs`. For example, `PhotoEyeModel`.$Cnf \mapsto$ `BeltModel`.*Tick*
3) Output to output connections C_{EO2O}: Output events of instances in the network may connect to output events of the CFB's interface. For example, `PhotoEyeModel`.*Cnf* $\mapsto TickO$.

Variable connections: Composite blocks have three types of variable connections, as illustrated in Fig. 3.

1) Input to input connections C_{VI2I} connect input variables of the interface to the input variables of the FB instances in the network, e.g., *Accel* $\mapsto$ `BeltModel`.*Accel*.
2) Output to input connections C_{VO2I} connect the output variable of one FB instance to the input of another, e.g., `BeltModel`.*EncCount* $\mapsto$ `PhotoEyeModel`.*EncCount*.



F3:1 Fig. 3. Network of blocks within the conveyor model CFB.

3) Output to output connections C_{VO2O} connect an output variable of an instance to an output of the interface, such as `PhotoEyeModel`.*PEDetects* $\mapsto$ `PEDetects`.

IEC 61499 restricts variable connections such that only a single source can be connected to a destination variable in a CFB. Moreover, only variables of the same type can be connected together. This paper omits this aspect for the sake of readability but without sacrificing expressiveness.

Definition 3 (CFB): A `CFB` is a tuple $\langle \mathcal{I}, \texttt{FBNetwork} \rangle$ with interface $\mathcal{I} = \langle E_I, V_I, \alpha_I, E_O, V_O, \alpha_O \rangle$, and a network $\texttt{FBNetwork} = \langle \texttt{FBs}, C_{\text{events}}, C_{\text{var}} \rangle$, where $\texttt{FBs} = \{\texttt{fb}_1, \texttt{fb}_2, \ldots \texttt{fb}_n\}$ is a finite and totally ordered set of function block instances of size n. Each $\texttt{fb}_i$ $(i \in [1, n])$ is a FB instance $\langle name_i, FBT_i \rangle$, with name $name_i$ and a function block (type) FBT_i with the interface $\mathcal{I}_i = \langle E_{Ii}, V_{Ii}, \alpha_{Ii}, E_{Oi}, V_{Oi}, \alpha_{Oi} \rangle$. The event connections set $C_{\text{events}} = C_{EI2I} \cup C_{EO2I} \cup C_{EO2O}$ (as per Section III-A3). The set of variable connections $C_{\text{var}} = C_{VI2I} \cup C_{VO2I} \cup C_{VO2O}$ (as per Section III-A3). For any two variable connections $(src_1, dest_1), (src_2, dest_2) \in C_{\text{var}}, dest_1 \neq dest_2$.

B. Synchronous Semantics for IEC 61499

The proposed synchronous semantics for IEC 61499 is similar to the semantics of PRET-C [21], and guarantees the absence of causality cycles due to a static ordering of execution. Execution of blocks is divided into logical time instants called *ticks* or macro-steps. During each tick, a predefined and ordered sequence of micro-steps is followed. This semantics differs from the synchronous semantics proposed in [7], where the order of execution of blocks may change between different ticks, and blocks can only read inputs generated in the previous tick. The proposed semantics does not require this delayed propagation of events used in [7].

1) Execution Semantics of BFBs: Each BFB initializes in its start state s_0. Each tick, its execution follows four steps. In step 1, the block samples input events (E_I) from the environment, and updates input variables (in V_I) when associated input events are present. In step 2, the outgoing transitions of the current state s are evaluated in order and the first enabled transition is taken. A transition is enabled when the related event is present and the Boolean condition evaluates to $true$. For example, if the ECC shown in Fig. 2 is in state `START`, and input event *Init* is present, the lone transition $\texttt{START} \xrightarrow{\text{Init,true}} \texttt{INIT}$ fires. If no outgoing transition is enabled, the current tick terminates. In step 3, after a transition to a destination state s' has fired, the actions $\lambda(s')$ are executed. For example, if state `INIT` is entered, the block executes algorithm $INIT$ and emits the event $InitO$. Finally, in step 4, output variable values are updated if associated output events were emitted in step 3. Thanks to the explicit ordering of ECC transitions, basic blocks execute deterministically, as in step 2, only the highest priority transition that is enabled fires.

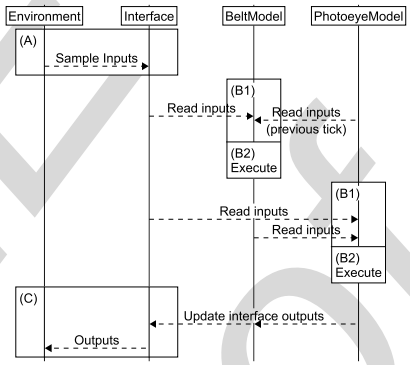
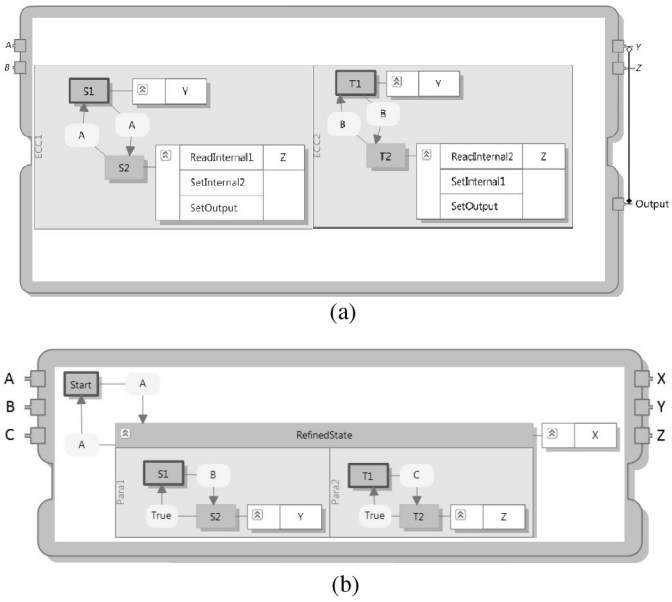


Fig. 4. Execution semantics for composite blocks. F4:1

2) Execution Semantics of Composite Blocks: A CFB is initialized with all instances in its network in their initial states. Each tick, the following sequence is executed. Fig. 4 illustrates this sequence for the CFB shown in Fig. 3. In step A, interface input events are sampled, and associated input variable values are updated. In step B, all FB instances in the network execute as per their order in the set `FBs`. For example, `BeltModel` executes before `PhotoeyeModel`. The execution of each instance involves two substeps. In step B1, input events are sampled and



F5:1 Fig. 5. HCECC examples. (a) Parallel HCECC. (b) Refined HCECC.

associated input variables are updated. For example, the input event `BeltModel`.*TICK* is present if any of the source events Tick of the block's interface or `PhotoeyeModel`.*Cnf* is present. For feedback connections, such as `PhotoeyeModel`.*Cnf* $\mapsto$ `BeltModel`.*TICK*, the presence or absence of the source event in the previous tick is considered. Then, in step B2, each FB instance is executed, based on its semantics (basic or composite). Fig. 4 shows how both steps B1 and B2 are repeated for the two instances in the CFB of Fig. 3. Finally, in step C, outputs of the CFB interface are updated. An output event is set to present if a source event connected to it is present during the current tick. All output variables associated with emitted output events are also updated.

We can recursively substitute composite block instances with their networks to translate a CFB network into a network of basic block instances. Under the proposed semantics, CFBs can be mapped to PRET-C programs with well-formed semantics [21]. It can then be shown that CFB execution is deterministic and reactive under the proposed semantics.

IV. HCECCs

HCECCs introduce parallel and refined block types to allow statecharts-like concurrency and hierarchy in basic blocks.

A. Syntax

Parallel HCECCs allow concurrency within a single block. Fig. 5(a) shows a parallel HCECC, which has an interface and local declarations. However, unlike a basic block, the HCECC contains two ECCs that share the interface and local declarations.

Definition 4 (parallel HCECC): A parallel HCECC $\texttt{HCECC}_{\|}$ is a tuple $\langle \mathcal{I}, L, \texttt{ECCs} \rangle$, with interface $\mathcal{I}$, local declarations L, and a finite ordered set $\texttt{ECCs} = \{\texttt{ECC}_1, \ldots, \texttt{ECC}_n\}$ of ECCs.

Refined HCECCs allow nesting of parallel ECCs within ECC states. Fig. 5(b) shows a refined HCECC containing an ECC with two states: `START` and `RefinedState`. The state `RefinedState` is refined and contains two concurrent ECCs: `Para1` and `Para2`. The top-level ECC is the refined ECC, `RefinedState` is the refined state, and `Para1` and `Para2` are the refining behaviors. All refined and refining ECCs share the interface and local declarations of the block.

Definition 5 (refined HCECC): A refined HCECC $\texttt{HCECC}_{\triangleright}$ is a tuple $\langle \mathcal{I}, L, \texttt{ECC}, \texttt{ECCs}, \triangleright \rangle$ containing an interface $\mathcal{I}$, local declarations L, a top-level refined ECC `ECC`, and a set `ECCs` of refining ECCs. The state-refinement function $\triangleright : S \rightarrow 2^{\texttt{ECCs}}$ (S is the set of states in `ECC`) maps states in `ECC` to an ordered subset of refining ECCs in `ECCs`.

For the refined HCECC shown in Fig. 5(b), $\texttt{ECCs} = \{\texttt{Para1}, \texttt{Para2}\}$, and $\triangleright(\texttt{START}) = \emptyset$ (nonrefined state) and $\triangleright(\texttt{RefinedState}) = \{\texttt{Para1}, \texttt{Para2}\}$. Def. 5 defines refined HCECCs with a single level of refinement, but HCECCs can have nested refinement, as discussed in the next section.

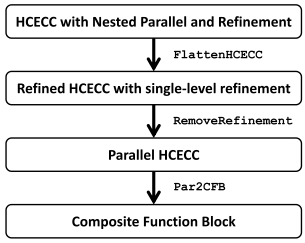
B. Synchronous Semantics for HCECCs

1) Execution Semantics of Parallel HCECCs: A parallel HCECC executes using the following sequence every tick. In step A, the block samples input events and updates input variables. In step B, ECCs contained in `ECCs` execute in order. For example, the HCECC in Fig. 5(a) executes ECC1 and then ECC2. Finally, in step C, an output event is emitted once if it is emitted by any ECC. Associated output variables are updated to the latest values assigned to them in the current tick following the order of execution of ECCs in B.

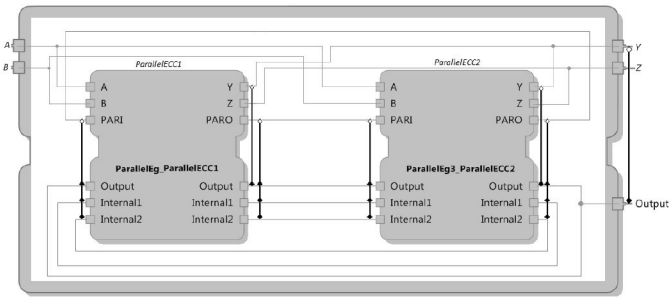
Variable updates propagate from the first ECC to the last ECC every tick. For example, for the HCECC in Fig. 5(a), if ECC1 updates any internal variables, ECC2 uses the updated values while executing in the same tick. ECC1 can read updates by ECC2 only in the next tick. The fixed order of execution ensures deterministic execution without causality issues. This semantics where a block is invoked only once relative to an event occurrence is fully compliant to ver. 2 of the IEC 61499 standard [1]. In ver. 1, a block can be invoked multiple times relative to an event occurrence, leading to ambiguities in execution semantics. Stability research can address such issues to ensure deterministic execution [22], [23]. However, stability research is not required for the proposed semantics.

2) Execution Semantics of Refined HCECCs: Every tick, a refined HCECC executes as follows. If the current state is not a refined state (case 1), the HCECC executes like a BFB (see Section III-B1). For example, the HCECC shown in Fig. 5(b) executes like a BFB when it is in state `Start`. If, during the current tick, a transition to a refined state is taken, the actions of the refined state, and the initial states of all refining behaviors are executed in order. For example, if a transition from state `Start` to the refined state `RefinedState` is taken, the block emits the output X (action of `RefinedState`), followed by executing the actions of the initial states S1 and T1 of the refining ECCs.

If the HCECC is in a refined state (case 2), it first samples the interface inputs and updates variables (step A). Then, in step B, the outgoing transitions of the refined state are evaluated. For example, if the HCECC from Fig. 5(b) is in state



F6:1 Fig. 6. Overview of HCECC translation.



F7:1 Fig. 7. Parallel ECCs flattened to a function block network.

`RefinedState` and the outgoing transition to state `Start` is enabled, execution is identical to case 1 above. In step C, if no transition fires in step B, the HCECC executes like a parallel HCECC (see Section IV-B1)—ECCs refining the current state execute in order. Finally, in step D, interface output events are emitted (if emitted by any of the ECCs), and output variable values are updated.

Refined HCECCs execute deterministically because in every tick, they follow the deterministic execution semantics of basic blocks (case 1) or parallel HCECCs (case 2).

V. Translating HCECCs to IEC 61499

As refined HCECCs execute like parallel HCECCs, and the semantics of parallel HCECCs are based on composite blocks, it is possible to create a sound translation from HCECCs to equivalent IEC 61499 function blocks as shown in Fig. 6. Detailed proofs can be found at http://tinyurl.com/hcecc2015.

A. Parallel HCECC to CFB Conversion

Algorithm 1 converts a parallel HCECC into a CFB by first creating multiple FB instances, and then connecting them. These steps are illustrated by transforming the parallel HCECC in Fig. 5(a) to the CFB network in Fig. 7.

Creating FB instances: Algorithm 1 transforms each ECC in the parallel HCECC into an instance of a newly created BFB (lines 1–22). The newly created instances are ordered in the same manner as the ECCs in the parallel HCECC (lines 8–21) and are contained in the set `FBs`.

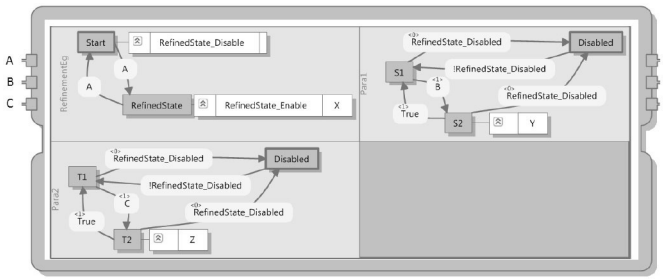
The newly created instances have the same interface $\mathcal{I}'$ (line 6) which has the same input and output events as the HCECC interface, and additional input and output events *PARI* and *PARO* (line 3). These additional events propagate updated variable values within the network (described in the next paragraph). $\mathcal{I}'$ has input variables corresponding to all input, output, and local variables of the parallel HCECC, and its output variables correspond to all local and output variables of the parallel HCECC (line 4). $\mathcal{I}'$ retains the input and output associations of the original HCECC. In addition, *PARI* and *PARO* are associated with each input and output variable of $\mathcal{I}'$, respectively. The local declarations L' for each instance contains no internal variables and has local versions of all algorithms of the parallel HCECC (line 7). A new algorithm $UpdateOutputs$ is used to ensure that each basic block instance propagates output data forward in every tick.

Events *PARI* and *PARO* replicate the communication between the ECCs of the parallel HCECC (using internal

Algorithm 1. Algorithm `Par2CFB`

Input: Parallel HCECC $\langle \mathcal{I}, L, \texttt{ECCs} \rangle$
Output: CFB `CFB`

1 //Create `FB instances`;
2 Ordered set `FBs` $= \emptyset$;
3 $E_I{}' = E_I \cup \{PARI\}$; $E_O{}' = E_O \cup \{PARO\}$;
4 $V_I{}' = V_I \cup V_L \cup V_O$; $V_O{}' = V_L \cup V_O$;
5 $\alpha_I{}' = \alpha_I \cup \{(PARI, v_I) | v_I \in V_L \cup V_O\}$;
$\alpha_O{}' = \alpha_O \cup \{(PARO, v_O) | v_O \in V_O\}$;
6 $\mathcal{I}' = \langle E_I{}', V_I{}', \alpha_I{}', E_O{}', V_O{}', \alpha_O{}' \rangle$;
7 $L' = \langle \emptyset, A_L \cup \{UpdateOutputs\} \rangle$;
8 **for** *each* $ECC = \langle S, s_0, \lambda, T \rangle \in ECCs$ **do**
9 $S' = S$; $s_0' = s_0$; Initialize λ', T';
10 **for** *each* $s \in S$ **do**
11 $\lambda'(s) = \lambda(s) \oplus \{UpdateOutputs, PARO\}$;
12 $T'(s) = T(s)$;
13 **if** *there is no transition* $s \xrightarrow{1} s_1$ **then**
14 Loop state s_l; $S' = S' \cup \{s_l\}$;
15 $\lambda'(s_l) = \{UpdateOutputs, PARO\}$;
16 $T'(s) = T'(s) \oplus \{s \xrightarrow{1} s_l\}$;
17 $T'(s_l) = \{s_l \xrightarrow{e,b} s_1 | s \xrightarrow{e,b} s_1\}$;
18 **end**
19 **end**
20 $\texttt{ECC}' = \langle S', s_0', \lambda', T' \rangle$; $\texttt{BFB}' = \langle \mathcal{I}', L', \texttt{ECC}' \rangle$;
21 $\texttt{fb} = \langle name_i, \texttt{BFB}' \rangle$; $\texttt{FBs} = \texttt{FBs} \oplus \{\texttt{fb}\}$;
22 **end**
23 //Create `connections`;
24 $n = |\texttt{FBs}|$; $C_{events} = \emptyset$; $C_{var} = \emptyset$;
25 **for** *each* $i \in [1, n]$ **do**
26 Select the i-th FB instance $\texttt{fb}_i \in \texttt{FBs}$;
27 $C_{events} = C_{events} \cup \{(e_I \mapsto \texttt{fb}_i.e_I) | e_I \in E_I\}$;
28 $C_{events} = C_{events} \cup \{(\texttt{fb}_i.e_O \mapsto e_O) | e_O \in E_O\}$;
29 **if** $i = 1$: $C_{var} = C_{var} \cup \{(v_I \mapsto \texttt{fb}_i.v_I) | v_I \in V_I\}$;
30 **if** $i = n$: $C_{var} = C_{var} \cup \{(\texttt{fb}_i.v_O \mapsto v_O) | v_O \in V_O\}$;
31 **if** $i < n$ **then** $\texttt{fb}_j = \texttt{fb}_{i+1}$ **else** $\texttt{fb}_j = \texttt{fb}_1$;
32 $C_{events} = C_{events} \cup \{(\texttt{fb}_i.PARO \mapsto \texttt{fb}_j.PARI)\}$;
33 $C_{var} = C_{var} \cup \{(\texttt{fb}_i.v_L \mapsto \texttt{fb}_j.v_L | v_L \in V_L\}$;
34 $C_{var} = C_{var} \cup \{(\texttt{fb}_i.v_O \mapsto \texttt{fb}_j.v_O | v_O \in V_O\}$;
35 **end**
36 //Create `CFB`;
37 $\texttt{FBNetwork} = \langle \texttt{FBs}, C_{events}, C_{var} \rangle$;
38 $\texttt{CFB} = \langle \mathcal{I}, \texttt{FBNetwork} \rangle$;
39 **return** CFB;



F8:1 Fig. 8. Parallel ECCs equivalent to refinement in Fig. 5(b)

variables) in the (to be created) CFB network by explicit propagation of data between the corresponding instances. Each instance emits *PARO* every tick, forcing an update of all output variables using algorithm $UpdateOutputs$ (line 11). The *PARO* output of a FB instance is connected to the *PARI* input of the *next* FB instance in the network (lines 31–35), allowing a forward propagation of data from each instance.

Each ECC in the HCECC is converted into an ECC′ of a corresponding BFB instance BFB′ (lines 8–22). ECC′ initially contains the same states and initial state as ECC (line 9) and each state s from ECC retains all original actions and transitions in ECC′, but also has two new actions—the execution of the algorithm $UpdateOutputs$ and the emission of the event *PARO* for forward data propagation.

Lines 13–18 create additional *loop* states in ECC. Loop states are needed to ensure that in any tick, if none of the original transitions inherited by the current state in ECC′ are taken, a transition to a loop state will ensure that the instance will still correctly propagate variable values through the network by doing actions $UpdateOutputs$ and *PARO*. Therefore, for every state s that does not have an always-true transition (line 13), a corresponding loop state s_l in ECC′ is introduced (line 14). Also, each state has an additional always-true but lowest priority transition to its corresponding loop state (line 16). The loop state itself has the same transitions as its corresponding state s (line 17).

Creation of connections: The event and variable connections for each FB instance are created as follows (the ith FB instance in FBs is noted as $\texttt{fb}_i$).

1) All input/ output events of the CFB interface $\mathcal{I}$ are connected to corresponding input and output events of $\texttt{fb}_i$ (lines 27, 28).
2) If $\texttt{fb}_i$ is the *first* or *last* FB instance in the CFB network, all input or output variables (respectively) of the HCECC interface $\mathcal{I}$ are connected to the corresponding input or output variables (respectively) of $\texttt{fb}_i$ (lines 29, 30).
3) The *PARO* output event of $\texttt{fb}_i$ is connected to the *PARI* input event of the next instance $\texttt{fb}_j$ (line 32). $\texttt{fb}_j$ is the next instance $\texttt{fb}_{i+1}$ after $\texttt{fb}_i$ in FBs, or if $\texttt{fb}_i$ is the last element of FBs, then $\texttt{fb}_j$ is the first element $\texttt{fb}_1$ of FBs (line 31). For example, in Fig 7, *PARO* of ParallelECC1 connects to *PARI* of ParallelECC2 and *PARO* of ParallelECC2 connects to *PARI* of instance ParallelECC1. The output variables of $\texttt{fb}_i$ corresponding to the internal and output variables of the parallel HCECC are connected to corresponding input variables of $\texttt{fb}_j$ (line 33).

Algorithm 2. Algorithm RemoveRefinement

Input: $\mathcal{I}, L = \langle V_L, A_L \rangle, \texttt{ECC} = \langle S, s_0, \lambda, T \rangle, s \in S, \texttt{ECCs}_s$
Output: Pair $(\texttt{ECCs}_\triangleright, L_\triangleright)$

1 //Modify locals;
2 Variable set $V_L\prime = V_L \cup \{s_\triangleright\_Disabled\}$ ($s_\triangleright\_Disabled$ is of type Boolean);
3 Algorithm set $A_L{}' = A_L \cup \{s_\triangleright\_Enable, s_\triangleright\_Disable\}$;
4 //Create new local declarations;
5 $L_\triangleright = \langle V_L\prime, A_L{}' \rangle$;
6 //Create $\texttt{ECCs}_\triangleright$;
7 Initialize ordered set of ECCs $\texttt{ECCs}_\triangleright = \emptyset$;
8 //Modify ECC;
9 Initialize empty action map λ_0;
10 //Modify states;
11 **for** *each state* $s \in S$ **do**
12 **if** $s = s_\triangleright$ **then**
13 $\lambda_0(s) = \lambda(s) \oplus \{s_\triangleright\_Enable\}$;
14 **end**
15 **else if** $s_\triangleright$ *has a transition* $(in\ T)$ *to* s **then**
16 $\lambda_0(s) = \lambda(s) \oplus \{s_\triangleright\_Disable\}$
17 **end**
18 **else**
19 $\lambda_0(s) = \lambda(s)$;
20 **end**
21 **end**
22 $\texttt{ECC}_0 = \langle S, s_0, \lambda_0, T \rangle$;
23 $\texttt{ECCs}_\triangleright = \texttt{ECCs}_\triangleright \oplus \{\texttt{ECC}_0\}$;
24 //Modify ECCs in $\texttt{ECCs}_s$;
25 **for** *each* $ECC_i = \langle S_i, s_{0_i}, \lambda_i, T_i \rangle \in \texttt{ECCs}_s$ **do**
26 Create set of states $S_{\triangleright i} = S_i \cup \{Disabled\}$;
27 Set state $s_{0_{\triangleright i}} = Disabled$;
28 **if** $s_\triangleright = s_0$ **then**
29 $s_{0_{\triangleright i}} = s_{0_i}$;
30 **end**
31 Initialize action map $\lambda_{\triangleright i} = \lambda_i$;
32 Add $\lambda_{\triangleright i}(Disabled) = \emptyset$;
33 Initialize transition function $T_{\triangleright i} = T_i$;
34 **for** *each* $s \in S_{\triangleright i}$ **do**
35 **if** $s = Disabled$ **then**
36 $T_{\triangleright i}(s) = \{s \xrightarrow{1, !s_\triangleright\_Disabled} s_{0_i}\}$;
37 **else**
38 $T_{\triangleright i}(s) = \{s \xrightarrow{1, s_\triangleright\_Disabled} Disabled\} \oplus T_i(s)$;
39 **end**
40 **end**
41 Create ECC $\texttt{ECC}_{\triangleright i} = \langle S_{\triangleright i}, s_{0_{\triangleright i}}, \lambda_{\triangleright i}, T_{\triangleright i} \rangle$;
42 $\texttt{ECCs}_\triangleright = \texttt{ECCs}_\triangleright \oplus \{\texttt{ECC}_{\triangleright i}\}$;
43 **end**
44 **return** $Pair\ (ECCs_\triangleright, L_\triangleright)$;

A proof by induction shows that the CFB generated from Algorithm 1 has the same behavior the source parallel HCECC.

B. Refinement to Parallel Conversion

AlgorithmRemoveRefinement (Algorithm 2) translates the refinement of a single state in a refined HCECC into parallel ECCs. For the HCECC in Fig. 5(b), the algorithm

T1:1 TABLE I
T1:2 HCECC AND ECC SIZE COMPARISON

		ECCs				HCECCs			
No.	**Benchmark**	**# Basic blocks**	**Connections**	**States**	**Transitions**	**Unique blocks**	**Connections**	**States**	**Transitions**
B1	Watch	3	21	17	24	1	0	14	20
B2	Distribution station	2	18	10	12	1	0	10	12
B3	Drill station	2	10	9	10	1	0	9	9
B4	Water monitor	4	31	16	38	3	28	16	38
B5	Cruise controller	5	32	15	42	1	0	11	38
B6	MP3player	4	22	14	24	1	0	10	15

produces the parallel ECCs shown in Fig. 8. The algorithm executes as follows. Lines 1–5 of Algorithm 2 modify the local declarations, adding a new internal variable ($s_{\triangleright}\_Disabled$) and two new algorithms (lines 2–3). The new algorithms ($s_{\triangleright}\_Enable, s_{\triangleright}\_Disable$) clear and set the $s_{\triangleright}\_Disabled$ variable respectively.

The first step in constructing the set $\texttt{ECCs}_{\triangleright}$ of parallel ECCs involves modifying the main ECC (lines 8–22). An algorithm $s_{\triangleright}\_Enable$ is added to the actions of $s_{\triangleright}$ (line 13), so that whenever state $s_{\triangleright}$ is reached, the variable $s_{\triangleright}\_Disabled$ is cleared (allowing refining ECCs to execute). Similarly, in every state s to which $s_{\triangleright}$ has an outgoing transition, an algorithm $s_{\triangleright}\_Disable$ is added to the actions to set the variable $s_{\triangleright}\_Disabled$ and to stop the refining ECCs from executing. For example, for the parallel ECCs in Fig. 8, the action map of the first ECC is changed as follows. $\lambda_0(\texttt{Start})$ takes the value $\{RefinedState\_Disable\}$, and $\lambda_0(\texttt{RefinedState})$ takes the value $\{RefinedState\_Enable, X\}$, where X was part of the original refined HCECC. The modified main ECC is inserted as the first element into the set $\texttt{ECCs}_{\triangleright}$. It, therefore, executes first, and may enable or disable other ECCs.

Lines 24–43 modify each refining ECC, adding a new state `Disabled` (line 26) to model the disabled behavior when the refined ECC is not in state $s_{\triangleright}$. Fig. 8 shows the `Disabled` state for each of the modified refining ECCs (`Para1` and `Para2`). `Disabled` is set as the initial state of each refining ECC (line 27). However, if the refined state $s_{\triangleright}$ is the initial state of the refined ECC, refining ECCs' original initial states are retained (line 29). The modified refining ECC states retain their actions from the original refining ECC (line 31) and state Disabled has no actions (line 32).

The transition function for the modified refining ECC retains all original transitions (line 33). New transitions are added to model the enabled/disabled behavior, based on the value of the variable $s_{\triangleright}\_Disabled$. All states have a highest priority transition to `Disabled` when $s_{\triangleright}\_Disabled$ is true (line 38). Disabled has a single outgoing transition to the original initial state of the refining ECC (line 36). For example, each state in ECC `Para1` in Fig. 8 has a transition to `Disabled` when $RefinedState\_Disabled$ is true, and `Disabled` has a transition to the original initial state `S1` when $RefinedState\_Disabled$ is false. Finally, line 41 constructs the modified refining ECC and adds it to $\texttt{ECCs}_{\triangleright}$ on line 42. After all refining ECCs have been processed in this way, line 44 returns $\texttt{ECCs}_{\triangleright}$ and modified local declarations $L_{\triangleright}$.

A refined HCECC containing multiple (m) refined states is transformed into a parallel HCECC by applying Algorithm 2 m times (once for each refined state). HCECCs with nested refinement are processed in a depth-first fashion starting from the inner-most HCECC, until the block is reduced to a refined HCECC with only single-level refinement. A proof of soundness of the proposed transformations can be done by induction, showing equivalence between the execution of a refined HCECC, the corresponding parallel HCECC, and the corresponding IEC 61499 program, under the proposed synchronous semantics.

T2:1 TABLE II
T2:2 DEVELOPMENT TIME (TIME IN MINUTES)

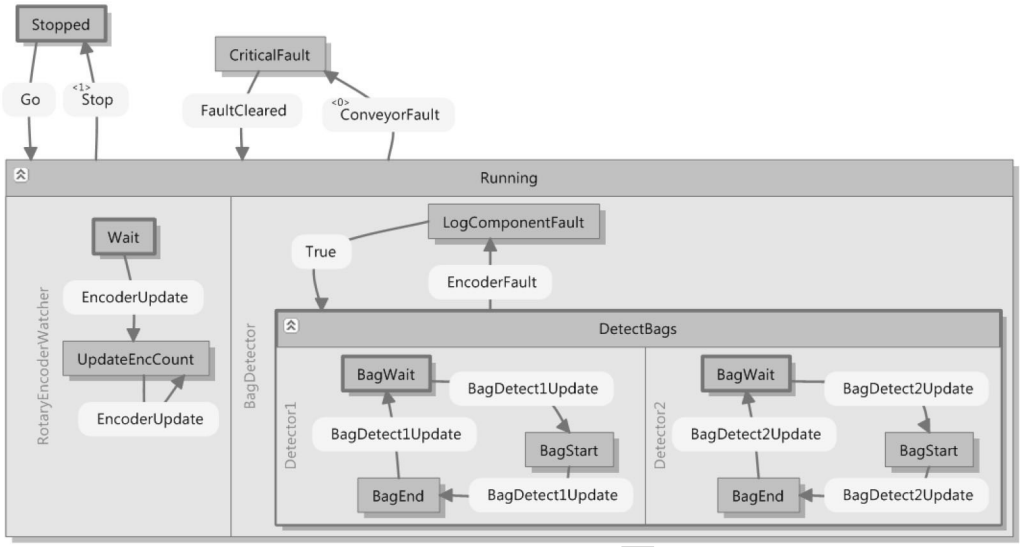
	Developer A		Developer B		Developer C	
No.	**ECCs**	**HCECCs**	**ECCs**	**HCECCs**	**ECCs**	**HCECCs**
B1	8	4	11	7	53	40
B2	5	3	14	10	50	34
B3	4	2	9	4	30	18
B4	11	10	23	20	70	45
B5	12	6	19	10	43	25
B6	8	3	20	9	35	16

VI. RESULTS

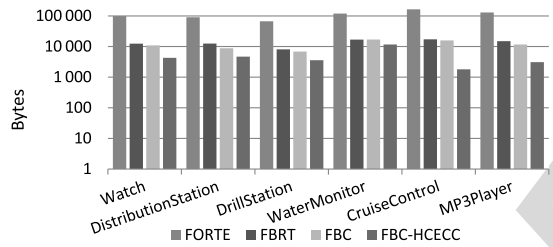
A qualitative and quantitative evaluation of HCECCs was done using the Auckland Function Block Benchmark [24], which contains designs varying in size and complexity of algorithms. Each benchmark was reimplemented using HCECCs, and subsequently compared to the original model. All benchmarks are available at http://tinyurl.com/hcecc2015.

Table I compares the sizes of the standard and HCECC versions of each benchmark. Columns 2–5, respectively, show the numbers of basic blocks, connections, states and transitions in the standard version. Columns 5–8, respectively, show the numbers of blocks, connections, states, and transitions in the HCECC version. For each benchmark, the size of the HCECC version was either smaller than or the same as the standard version. Most compression was achieved when refinement was used because enforcing refinement-like hierarchy in standard function blocks requires many more transitions and states.

Table II shows the times taken by three developers to develop the benchmarks shown in Table I. Developer A was an expert in HCECC design, developer B had some experience with statecharts and a working knowledge of IEC 61499, and developer C was completely new to both HCECCs and IEC 61499. Overall, all developers took significantly lesser amount of time to develop HCECCs. While these results are promising, a wider usability study is beyond the scope of this paper.



F9:1 Fig. 9. Multilevel refinement example.



F10:1 Fig. 10. Benchmark code size comparison.

T3:1 TABLE III
T3:2 PERFORMANCE COMPARISON (TIME IN MILLI-SECONDS)

	FORTE	FBRT	FBC ECC	FBC HCECC
B1	1962.6	77.0	75.0	24.0
B2	1532.0	120.4	68.2	30.0
B3	387.0	56.3	44.0	19.0
B4	1928.4	214.5	83.0	33.0
B5	877.0	130.0	64.0	27.8
B6	1251.2	144.7	123.0	96.0

HCECCs can model complex behaviors that cannot be easily created in standard function blocks. Fig. 9 shows the HCECC for a conveyor controller (algorithms omitted for readability). This model may react to up to three different events in a single tick. Modeling such behaviors using standard (flat) ECCs results in exponentially more states and transitions, requires extra effort in creating multiple blocks and connecting them, and takes longer to design and maintain.

We extended the function block compiler (FBC) [7] to create FBC-HCECC, a compiler to compile HCECCs into C code using the proposed synchronous semantics. Fig. 10 compares code sizes produced by FORTE v1.7.1, FBRT v20081003, FBC, and FBC-HCECC. For FBRT code, the Java virtual machine (JVM) size was removed for a fair comparison. Code generated by FBC-HCECC was on average 1.18 times smaller than FBC, which in turn produced much smaller code than FORTE and FBRT. In some cases, like benchmark B4, FBC-HCECC generated code was 1.70 times smaller than FBC. The reduction in code size comes from HCECCs allowing the same functionality within smaller models, as per Table I.

Code for each benchmark from different compilers was run on a windows PC with an Intel Core i7 920 processor and 18GB RAM. Java v7 (JDK and JRE) was used to compile and run FBRT-generated code. 4DIAC and FBDK front-ends were used to design and run FORTE and FBRT programs, respectively. C code from FBC and FBC-HCECC was compiled using Visual Studio's C compiler with the -O2 optimization switch. Each program was then executed with a common test-file containing a sequence of one million input vectors. Each vector contained a random event and random values for variables. A comparison of the execution times is shown in Table III. FBC-HCECC programs executed 1.3 to 3.1 times faster than FBC programs, with an average speedup of 2.3 possibly due to smaller model sizes. FBC-HCECC programs ran on average 3.8 and 42.7 times faster than FBRT and FORTE programs, which required additional runtimes.

Overall, HCECCs enable more compact designs that are faster to develop, smaller in code size, and execute faster than standard IEC 61499 designs. A side-effect of using the proposed semantics is that a fixed ordering of blocks within composite blocks (and also parallel and refined HCECCs) must be defined. The FBC-HCECC compiler uses the ordering as per XML descriptions, which is standard practice. Designers can change this ordering by editing the XML descriptions. HCECCs can increase cohesion and reduce coupling between blocks in some cases, such as exceptional handling scenarios. However, it is possible to misuse this framework and integrate loosely coupled blocks, reducing maintainability.

VII. CONCLUSION

This paper introduces HCECCs, consisting of hierarchical and concurrent operators for IEC 61499 ECCs, inspired by statecharts. Refined HCECCs efficiently model exception handling and other hierarchical behaviors, whereas parallel

HCECCs simplify the modeling of concurrent processes and enable the monitoring of simultaneous events in a single block. A synchronous semantics for IEC 61499 and HCECCs is proposed under which HCECCs can be translated to CFBs. Benchmarking results show that HCECCs provide reduced model sizes, which helps reduce development time, compiled code size, and execution time. Future directions include using HCECCs under other IEC 61499 semantics, optimizing HCECC to network translation, studying how HCECCs affect cohesion and coupling as well as code maintainability or reusability, and providing tool-support for HCECCs.

REFERENCES

[1] J. H. Christensen, T. Strasser, A. Valentini, V. Vyatkin, and A. Zoitl, "The IEC 61499 function block standard: Overview of the second edition," *ISA Autom. Week*, vol. 6, 2012.

[2] 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.

[3] IEC, *Committee Draft for Vote: IEC 61499-1: Function Block-Part 1 Architecture*. Geneva, Switzerland: IEC, 2004.

[4] A. Zoitl and H. Prahofer, "Guidelines and patterns for building hierarchical automation solutions in the IEC 61499 modeling language," *IEEE Trans. Ind. Informat.*, vol. 9, no. 4, pp. 2387–2396, Nov. 2013.

[5] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Programm.*, vol. 8, no. 3, pp. 231–274, Jun. 1987.

[6] J. Kim, I. Kang, J.-Y. Choi, and I. Lee, "Timed and resource-oriented statecharts for embedded software," *IEEE Trans. Ind. Informat.*, vol. 6, no. 4, pp. 568–578, Nov. 2010.

[7] L. H. Yoong, P. 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.

[8] M. Di Natale, Q. Zhu, A. Sangiovanni-Vincentelli, and S. Tripakis, "Optimized implementation of synchronous models on industrial LTTA systems," *J. Syst. Archit.*, vol. 60, no. 4, pp. 315–328, 2014.

[9] M. Bonfè, C. Fantuzzi, and C. Secchi, "Design patterns for model-based automation software design and implementation," *Control Eng. Pract.*, vol. 21, no. 11, pp. 1608–1619, 2013.

[10] J.-P. Talpin *et al.*, "Formal verification of synchronous data-flow program transformations toward certified compilers," *Front. Comput. Sci.*, vol. 7, no. 5, pp. 598–616, 2013.

[11] R. Nakamura, F. Arakawa, and M. Edahiro, "Simple one-to-one architecture for parallel execution of embedded control systems," in *Proc. IEEE Int. Conf. Cyber-Phys. Syst. Netw. Appl. (CPSNA)*, 2014, pp. 25–30.

[12] P. Gaj, J. Jasperneite, and M. Felser, "Computer communication within industrial distributed environment—A survey," *IEEE Trans. Ind. Informat.*, vol. 9, no. 1, pp. 182–189, Feb. 2013.

[13] A. Girault, B. Lee, and E. A. Lee, "Hierarchical finite state machines with multiple concurrency models," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 18, no. 6, pp. 742–760, Jun. 1999.

[14] D. Latella, I. Majzik, and M. Massink, "Towards a formal operational semantics of UML statechart diagrams," in *Formal Methods for Open Object-Based Distributed Systems*. New York, NY, USA: Springer, 1999, pp. 331–347.

[15] A. Barji, N. Hagge, and B. Wagner, "Comparative study of using CNet, IEC 61499, and statecharts for behavioral models of real-time control applications," in *Proc. IEEE Conf. Emerg. Technol. Fact. Autom. (ETFA'06)*, 2006, pp. 750–757.

[16] D. Tikhonov, D. Schutz, S. Ulewicz, and B. Vogel-Heuser, "Towards industrial application of model-driven platform-independent PLC programming using UML," in *Proc. 40th Annu. IEEE Conf. Ind. Electron. Soc. (IECON'14)*, 2014, pp. 2638–2644.

[17] K. Thramboulidis, "Model-integrated mechatronics-toward a new paradigm in the development of manufacturing systems," *IEEE Trans. Ind. Informat.*, vol. 1, no. 1, pp. 54–61, Jan. 2005.

[18] G. D. Shaw, P. S. Roop, and Z. Salcic, "A hierarchical and concurrent approach for IEC 61499 function blocks," in *Proc. IEEE Conf. Emerg. Technol. Fact. Autom. (ETFA'09)*, 2009, pp. 1–8.

[19] V. Dubinin, V. Vyatkin, and T. Pfeiffer, "Engineering of validatable automation systems based on an extension of UML combined with function blocks of IEC 61499," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA'05)*, 2005, pp. 3996–4001.

[20] V. N. Dubinin and V. Vyatkin, "Semantics-robust design patterns for IEC 61499," *IEEE Trans. Ind. Informat.*, vol. 8, no. 2, pp. 279–290, May 2012.

[21] S. Andalam, P. S. Roop, A. Girault, and C. Traulsen, "A predictable framework for safety-critical embedded systems," *IEEE Trans. Comput.*, vol. 63, no. 7, pp. 1600–1612, Jul. 2014.

[22] F. Schumacher and A. Fay, "Formal representation of GRAFCET to automatically generate control code," *Control Eng. Pract.*, vol. 33, pp. 84–93, 2014.

[23] M. Sogbohossou and A. Vianou, "Formal modeling of grafcets with time petri nets," *IEEE Trans. Control Syst. Technol.*, vol. 23, no. 5, pp. 1978–1985, Sep. 2015.

[24] L. H. Yoong, P. S. Roop, and Z. Salcic, "Implementing constrained cyber-physical systems with IEC 61499," *ACM Trans. Embedded Comput. Syst. (TECS)*, vol. 11, no. 4, pp. 78:1–78:22, 2012.

Roopak Sinha (S'03–M'13) received the B.E. (Hons), M.C.E., and Ph.D. degrees.

He has worked with INRIA, Grenoble, France, and the University of Auckland, Auckland, New Zealand. Currently, he is a Senior Lecturer with the School of Computer and Mathematical Sciences, Auckland University of Technology, Auckland. His research interests include next-generation formal frameworks for designing large-scale embedded software with application in industrial automation systems, Internet-of-Things, and intelligent transportation systems.

Partha S. Roop (M'94) received the Ph.D. degree in computer science (software engineering) from the University of New South Wales, Sydney, NSW, Australia, in 2001.

He is currently an Associate Professor and is the Director of the Computer Systems Engineering Program, Department of Electrical and Computer Engineering, University of Auckland, Auckland, New Zealand. His research interests include the design and verification of embedded systems and executable biology. In particular, he is developing techniques for the design of embedded applications in automotive, robotics, intelligent transportation, and medical devices that meet functional safety standards.

Gareth Shaw received the B.E. and the Ph.D. degrees in electrical and electronic engineering from the University of Auckland, Auckland, New Zealand, in 2007 and 2013, respectively.

He is a Mobile Development Team Lead with Fiserv New Zealand, Auckland. His research interests include design languages and their compilation, distributed systems, code generation, and complex digital system design.

Zoran Salcic (S'75–M'76–SM'98) received the B.E., M.E., and Ph.D. degrees in electrical and computer engineering from Sarajevo University, Sarajevo, Bosnia and Herzegovina, in 1972, 1974, and 1976, respectively.

He is a Professor of Computer Systems Engineering with the University of Auckland, Auckland, New Zealand. He has authored over 300 peer-reviewed journal and conference papers, and several books. His research interests include complex digital systems, custom-computing machines, embedded systems and their implementation, design automation tools, hardware-software co-design, models of computation and languages for concurrent and distributed systems, and cyber-physical systems.

Prof. Salcic is a Fellow of the Royal Society New Zealand. He was the recipient of the Alexander von Humboldt Research Award in 2010.

Matthew M. Y. Kuo received the B.E. (Hons) degree in electrical and computer systems engineering from the University of Auckland, Auckland, New Zealand. He is currently pursuing the Ph.D. degree.

He is currently involved in an industrial project focused on the Internet of Things with UniServices, Auckland. His research interests include synchronous programming, static timing analysis, and precision timed industrial automation systems.

QUERIES

Q1: Please provide postal code for affiliations.
Q2: Please provide page range for Ref. [1].
Q3: Please provide the field of study, institutional details, location, and year of all the degrees of the author Roopak Sinha.
Q4: Please spell out the term “INRIA”.
Q5: Please provide the institutional details of the Ph.D. degree of the author Matthew M. Y. Kuo.