A hierarchical and concurrent approach for IEC 61499 function blocks

Gareth D. Shaw, Dr. Partha S. Roop, Prof. Zoran Salcic
Department of Electrical and Computer Engineering
University of Auckland, Auckland

Abstract—The IEC 61499 function block standard proposes a new specification language for describing distributed industrial control systems. The standard specifies the use of an execution control chart (ECC) for state control, with algorithm calls for data handling. The design of complex industrial systems such as baggage handling systems can be difficult because of large state-spaces or complicated component interactions. Additionally, the flat state machines used in the standard do not provide a simple method for specifying error handling within the process’s execution. State machines from synchronous languages, however, have hierarchy and concurrent constructs to aid the developer. This paper presents a Hierarchical and Concurrent extension to ECCs, which we call HCECCs, which presents new design constructs adapted from synchronous languages in order to improve system specification with function blocks. The semantics of HCECCs, which are backward compatible with the standard, are described and design using HCECCs is compared with other specification approaches.

Index Terms—Modelling, IEC 61499, Function Blocks, Semantics, Synchronous extension

I. INTRODUCTION

The ever increasing complexity of control and automation systems has driven the need for new specification languages and design techniques. From this need, the IEC 61499 function block standard was developed. The standard provides an object oriented approach for complex distributed control. IEC 61499 also attempts a natural step from traditional industrial languages, allowing the reuse of algorithms written in the older IEC 61131 set of languages.

Modelling with IEC 61499 function blocks is appealing for a number of reasons. Such models are independent of implementation target. The high-level of abstraction adopted by function blocks simplifies the development of complex systems by making the design specification more intuitive and less error-prone. Fully executable code can then be automatically synthesized from these descriptions.

Adoption of the IEC 61499 standard, however, is currently hindered by semantic ambiguities, see ([1], [2]). Major ambiguities arise due to incoherent execution semantics of blocks and assumptions regarding the life-time of events. In addition, the authors of [3] explain four factors which contribute to the slow to non-existent adoption of IEC 61499 by the major control system equipment vendors. Of the four factors that the authors propose as being causes for this lack of acceptance, this paper’s syntactic extensions and semantic definitions will address: maintainability, predictability and extensibility. The fourth factor mentioned in the paper, scalability, is not addressed by this work as it has been addressed in [4].

The extension and semantics, called HCECCs, were developed to both address the semantics ambiguities of IEC 61499 as well as specification deficiencies. Utilising concepts from synchronous languages [5], HCECC semantics adopt the synchronous approach proposed in [6] and introduce ARGOs [7] like concurrency and hierarchy. The benefits of the synchronous approach, namely predictability of execution, have already been demonstrated by [6]. HCECCs extend the specification capabilities of function blocks, adding concurrency and hierarchy at the state-machine level, thus improving maintainability of existing code and extensibility by reducing the design complexity. Unlike other approaches to function block design [8], [9], HCECCs are directly compatible with the standard.

D. Drusinsky et al. discuss the benefits of a synchronous graphical language (Statecharts) over traditional finite state machines in [10]. They note the unsuitability of traditional state-based approach without concurrency and multi-level nesting for describing complex components, stating the deficiencies as almost universally accepted and inherent limitations. As with Statecharts [11], HCECCs add information-condensing hierarchy and orthogonality (modeling concreteness) to address these shortcomings. Unlike Statecharts, however, HCECCs do not allow internal broadcast communication within a block as internal events are not available in the IEC 61499 standard.

In what follows, Section II introduces the IEC 61499 standard using an example which illustrates the deficiencies of the standard. In Section III, this example is then reimplemented using HCECCs to demonstrate the advantages provided by hierarchy and refinement. The operators used within HCECCs are defined in Section IV and features of the semantics are explained. The tools developed and used for editing and executing HCECCs are presented in Section V, along with benchmarks and comparisons. In Section VI, comparisons are made with other languages and other extensions. Finally, Section VII concludes this work.

II. IEC 61499 FUNCTION BLOCKS

To demonstrate function blocks, this section presents the model for a conveyor controller. A single conveyor controller maintains an array of bag data for the bags on the conveyor. Control decisions are then made based on the quantities of
bags, the spacing between them, and each individual bag’s proximity to an exit point on the conveyor. The conveyor controller must therefore simultaneously monitor inputs to maintain an array of bag data, control the conveyor’s motor and monitor the bag array for exiting bags. This concurrent behaviour must also be preemptable by an error event.

A function block can be either basic or composite. Basic function blocks contain a state-machine for control flow and algorithms, called by the state machine, for data management. Composite function blocks instead contain a network of connected function blocks, encapsulating their functionality and possibly hiding internal events and data. Both types of blocks have an interface consisting of input and output events and data. Basic blocks also have internal data variables which can be used by algorithms.

Fig. 1 shows the interface for the composite conveyor controller block. Many of the data inputs are static and only required as constants for calculation. The non-static input EncCount on the other hand, contains the position of the conveyor belt at the current instance of time, updated once per execution of the function block. This specifies the measure of the distance the conveyor belt has moved since an arbitrary start time. The two events BagLeadingEdgeDetected and BagTrailingEdgeDetected, both signal a change in state in one or more of the infra-red object detectors placed along the length of the conveyor. The only output data is the control of the motor MotorOn. A value of true tells the motor to go and false tells it to stop. In IEC 61499 function blocks, input and output variables are tied with event(s). For example the EncCount data input is only read from the environment when either BagLeadingEdgeDetected or BagTrailingEdgeDetected occur. Similarly, the MotorOn data output is only set whenever the state-machine emits the MotorCommand event.

At the lowest specification level, basic function blocks (FBs) contain an Execution Control Chart (ECC) state machine. ECCs are flat finite state machines and thus not ideal for the specification of complex systems. Networks of FBs can be used to some extent to implement the desired synchronous functionalities. Concurrency can only be implemented using two separate basic function blocks in a network. This concurrency is not the same as concurrency within a state-machine, because it is performed at the function block level.

The semantics of the two interfaces interfere with what may be the desired functionality. Similarly refinement is limited to FB networks within another block, which merely hides the functionality of lower levels of blocks. For the conveyor controller example, the concurrent operations of conveyor control, the functionalities are split into the two blocks shown in the network in Fig. 2.

The main conveyor control is handled by the ConveyorController block, where the ECC is shown in Fig. 3. This ECC constantly monitors the number of bags on the conveyor and ensures the conveyor is going when there is a bag present and stopped when there are no bags. Management of the bag data is handled by the ConveyorController_BagData block, using the ECC shown in Fig. 4. Positions of each bag on the conveyor are updated and stored in an array in the Conv_HandleBagDetect algorithm. Using this data the Conv_DetectExitBag algorithm determines whether a bag needs to be handed over to the next downstream conveyor. It should be noticed that in both ECCs preemption is implemented with exiting transitions with the condition of the Error event on each state, which again complicates the design.

An ECC commences execution on the occurrence of an external event. As transitions may have a condition over an
event or a pure boolean condition, the ECC may then take a series of transitions before reaching a state which can only be exited upon the occurrence of an event. Exiting transitions are evaluated with priorities dependant on the textual order of the block’s specification. An always true condition in our semantics is often used as an else, meaning that if no other transition triggers, this low-priority transition is taken. States within the ECC have actions, each of which may consist of Algorithm and Output event. Algorithms are executed sequentially in the order the actions were declared and outputs are emitted similarly. Algorithms can use incoming data inputs or internal variables and assign values to outgoing data outputs. The values of these data outputs are only updated when a state emits an event that is associated with the data output.

Using Fig. 3 as an example, the state START with a double outline is the initial state. If the event Error occurs, the ECC transitions to state ERROR and executes the Conv_LogError algorithm. The ECC will then stay in this state, until the ErrorCleared event is detected. In the Error state therefore, the ECC is unreactive to all other events. After the ErrorCleared event, the ECC transitions back to the START state.

III. HCECC IMPLEMENTATION

The FB network, shown in Fig. 2, can alternatively be represented by a single FB containing an HCECC. The HCECC block has the same interface as the composite block in Fig. 1. The basic initialisation and error handling behaviour is represented by the ECC in Fig. 5. The NORMAL state within this machine is refined to contain the parallel ECCs shown in Fig. 6. From this it is much easier to understand its functionality. Beginning with START state, the block must be initialised, then the monitoring operations, bag-edge detections and bag exiting, and motor control are all performed simultaneously.

This example shows us the simplified representation of otherwise complex control. Table I shows the number of states and transitions for some example applications when using standard ECC and HCECC, respectively. HCECC specifications tend to be more concise and contain a smaller number of states and transitions. Also, it is easier to maintain a single block than a composite block that consists of multiple basic blocks as there are a smaller number of block-level connections to make.

Editing of an HCECC is performed in the same way as editing an ECC, making both approaches nearly identical in that respect.

A. HCECC OPERATORS

The operators used in HCECC are compositional, but not associative nor commutative. As such the operations are performed bottom up, with using operators with the format parent operator child. From the HCECC in Figs. 5 and 6, the three ECCs in parallel are first combined into one ECC, then the resulting ECC, which refines a state in the parent ECC, is flattened to form the final flat ECC. All of the HCECC operators ensure that a standards compatible ECC is generated [12].

The synchronous parallel operator $||$ merges two ECCs executing in parallel within a single FB into a single larger ECC.

Fig. 7 shows two of the ECCs which are in parallel in the HCECC. Using a graphical approach similar to ARGOS [7], concurrent ECCs are drawn next to each other with bounding boxes separating them. The operator creates the equivalent ECC in Fig. 8. Each state in the resulting ECC has only one always-true exiting transition. The action within the ECC with an always true condition on one state is merely replicated to the parallel state machine.

The refinement operator ↓ flattens the hierarchy of ECCs. Fig. 9 shows the parent or top most ECC, with the NORMAL state refined by the ECC formed using the parallel operator. Again similar to ARGOS, refined states are drawn as a box containing the refining machine or machines. The refinement operator removes the hierarchy to create the ECC in Fig. 10. The transitions into the refined state now point to the starting state of the refined ECC and all outgoing transitions are replicated on each of the nested states.

The Conveyor Controller example is flattened into an ECC containing 7 states and 26 transitions. Comparing this to the manually created ECC version which has 11 states and 24 transitions between 3 blocks, the flattened HCECC translation
TABLE I
STATE MACHINE COMPARISON

<table>
<thead>
<tr>
<th>Application</th>
<th>ECC</th>
<th>HCECC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Blocks</td>
<td>States</td>
</tr>
<tr>
<td>Watch</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>Cruise Controller</td>
<td>3</td>
<td>11</td>
</tr>
<tr>
<td>MP3Player</td>
<td>5</td>
<td>13</td>
</tr>
<tr>
<td>Conveyor Controller</td>
<td>3</td>
<td>11</td>
</tr>
</tbody>
</table>

is reasonably compact. The original HCECC specification is easier to maintain and extend, and an efficient ECC is automatically created from it. The next section explains the semantics in more detail.

IV. HCECC SEMANTICS AND FEATURES

This section presents the formal semantics for HCECCs. Equation 1 provides the formalisation of a function block. The synchronous parallel operator (\(\|\)) and the refinement operator (\(\downarrow\)), defined later, operate only on ECCs.

\[
FB = (I, V_I, f_I, O, V_O, f_O, V_{in}, ECC, A)
\]  

(1)

where \(I\) is the set of input events, \(V_I\) is the set of input variables, the function \(f_I : I \rightarrow 2^{V_I}\) maps input events to input variables, \(O\) is the set of output events, \(V_O\) is the set of output variables, the function \(f_O : O \rightarrow 2^{V_O}\) maps output events to output variables, \(V_{in}\) is the set of internal variables, \(ECC\) is the ECC within the block - defined in Eq. 2 below, and \(A\) is the set of algorithms.

\[
ECC = (S, s_0, \lambda, T)
\]  

(2)

where \(S\) is the set of states, \(s_0\) is the initial state, \(\lambda\), Eqn. 3, maps states to actions including algorithms and output events. \(T\) is the transition function, Eqn. 4, which maps the current state and events and boolean data from the previous tick to the next state. The function \(pre\) provides the value of the event or variable from the previous tick.

\[
\lambda : S \rightarrow (A \times O)
\]  

(3)

\[
T : S \times pre(I) \times B(pre(V_I) \cup pre(V_{in})) \rightarrow S
\]  

(4)

In these semantics, as in Yoong et al. [13], the previous value of an event or data is used. In this way causality issues [5] are mitigated and producer and consumer relationships become acyclic, thus reducing implementation complexities. These semantics also define the life cycle of an event to be over one transition, regardless of whether or not it is consumed in a taken transition.

Also as with Yoong Et al. [13], the notion of a function block run is used. A run is initiated when any incoming event is present and ends when no further transitions evaluate to true.

In both the synchronous parallel and refinement operators are denoted as \(A \| B\) or \(A \downarrow B\), where \(A\) and \(B\) are ECCs, the algorithms of \(A\) will be executed before the algorithms of \(B\) when the states are merged.

The synchronous semantics of an ECC in this approach can be summarised as follows. A transition is one tick, or instant of time and an event lasts for one tick regardless of whether it is used in a taken transition. Transitions use the previous values of signals and data, thus removing causal cycles, and the priorities of out-going transitions are determined by their order, which must be specified for each transition by system designer. The IEC 61499 standard does not guarantee determinism - only one valid transition for given inputs, or reactivity - in all states an input event results in a transition. As such the HCECCs operators are designed to maintain the reactivity and determinism present in the original specification.

HCECC operators

The proposed HCECC based semantics of IEC61499 is inspired by the synchronous Statecharts ARGOS [7]. However, unlike ARGOS, the intra-function block operators of HCECCs differ from ARGOS in several ways. These differences are intentional in order to stay compatible with the standard and also because ECCs are Moore FSMs as opposed to ARGOS’s
Mealy FSMs. For example, the synchronous parallel operator does not allow the creation of transitions whose conditions are dependent on more than one event. Note also that the implicit transitions which are conditional on the absence of an event remain implicit and are not merged with explicit transitions of the other machines. As the operators operate on ECC Moore machines as inputs, producing a Moore machine as the output ensures better control over the generated ECC, as opposed to translating via a Mealy machine.

As a result of Moore-type FSMs, the application of HCECC operators have the potential to result in an increased number of states. This is due to the fact that the actions are associated with states. Because of this, in situations where a resulting ECC needs to implicitly loop on a state which has an action, an additional auxiliary state must be created. These auxiliary states feature the same outgoing transitions as the original state, but they do not execute an action.

In both HCECC operators operating on ECCs, the transition priorities of each ECC are maintained in their relative orders, with always true conditions ensured to be the lowest priority of a group of exiting transitions.

**Synchronous Parallel operator**

The synchronous parallel operator merges two ECCs executing in parallel within a single FB into a single larger ECC.

**Definition 1:** Synchronous parallel operator $\parallel$

$$\parallel: E_1 \times E_2 \rightarrow E$$

$$(S_1, s_01, T_1, \lambda_1) \parallel (S_2, s_02, T_2, \lambda_2) = (S, (s_01, s_02), T, \lambda)$$

The new set of states $S$ is defined as:

$$S = \{S_1 \times S_2\} \cup \{\hat{S}_1 \times S_2\}$$

Notice that due to the transitions generated in $T$ some states may never be entered. For this reason a clean-up function is executed after the synchronous parallel operator is used. The clean-up function is described later in this section.

The function $\text{Aux}$, defined below $\text{Aux}$ creates a set of auxiliary states from states that have algorithms or outputs ($\lambda(s) \neq \phi$) and do not have transitions that have always true conditions i.e. there are no outgoing transitions where $(e = \perp) \land (c = 1)$ is true. Using this function, the set of auxiliary states of machine one is $S_1 = \text{Aux}(E_1)$ and the set of auxiliary states of machine two is $S_2 = \text{Aux}(E_2)$.

$$\text{Aux} : E \rightarrow S$$

$$\text{Aux}(E) = \{s|s \in S, \lambda(s) \neq \text{null} \land \forall (s, e, c, s') \in T : ((e = \perp) \land (c = 1))\}$$

The new action map $\lambda$ below, is the cross product of the original maps, with additional mappings for states combined with auxiliary states. The combined states of the form $(s_1, s_2)$, are mapped to the actions of both original states, noting that the action of machine $E_1$ will be executed before the actions of machine $E_2$. In the event that a part of the combined state is an auxiliary state, only the action of the non-auxiliary state is mapped.

$$\lambda = \{(s_1, s_2) \rightarrow (\lambda_1(s_1) \cup \lambda_2(s_2))\mid s_1 \in S_1, s_2 \in S_2\}$$

$$\cup$$

$$\{(s_1, s_2) \rightarrow \lambda_2(s_2)\mid s_1 \in S_1, s_2 \in S_2\}$$

$$\cup$$

$$\{(s_1, s_2) \rightarrow \lambda_1(s_1)\mid s_1 \in S_1, s_2 \in S_2\}$$

The new transition function $T$, Eqn. 5, is comprised of three sets of transitions. $T'$ is the set of transitions created when both machines take a synchronised transition, $T''$ is the set of transitions where either of the single machines take an interleaved transition while the other machine loops in its current state. $T'''$ copies all transitions in $T''$ so the auxiliary states have the same outgoing transitions are their original state counter-parts. These definitions may result in always false transitions or transitions to removed states (see definition of $S$). Because of this a clean up function is used after the parallel operator completes.

$$T = T' \cup T'' \cup T'''$$

As ECC transitions can only be conditional on one or no events, the synchronised transitions ($T'$) are only the transitions where both ECCs transition on the same event ($e_1 = e_2$) or where at least one of the transitions does not depend on an event ($e_1 = \perp$) or ($e_2 = \perp$). The conditions are then combined to form a larger boolean condition, which if always false will be removed later in the clean up function.

$$T' = $$

$$\{(s_1, e_1 \cup e_2, c_1 \land c_2, (s_1', s_2'))\mid (s_1, e_1, c_1, s_1') \in T_1 \land (s_2, e_2, c_2, s_2') \in T_2 \land ((e_1 = e_2) \lor (e_1 = \perp) \lor (e_2 = \perp))\}$$

$T''$ defined below contains interleaved transitions out of original states. For all transitions in the original ECC for all the states of the other ECC, a new transition is created for each instance where one machine takes a transition while the other machine loops in its current state. If the current state has an action then the transition must go to an auxiliary state.
transitions into a single flattened ECC where the state of one machine is
Refinement operator

and state, an input condition results in only one valid transition leaving any state are deterministic, meaning that from any
auxiliary state to the same destination state.

Definition 2: The refinement operator

\[ T'' = \]
\[ \{(s_1, s_2), e_1, c_1, (s_1', s_2)\} \]
\[ (s_1, c_1, s_1') \in T_1, \]
\[ s_2 \in S_2 \rightarrow s_2 = s_2', \]
\[ s_2 \notin S_2 \rightarrow s_2 = s_2' \}
\[ \cup \]
\[ \{(s_1, s_2), e_2, c_2, (s_1', s_2)\} \]
\[ (s_2, c_2, s_2') \in T_2, \]
\[ s_1 \in S_1 \rightarrow s_1 = s_1', \]
\[ s_1 \notin S_1 \rightarrow s_1 = s_1' \}

\[ T''' \] contains interleaved transitions out of auxiliary states. If there is an auxiliary version of either of the transition’s source
states (s1 or s2) then a transition is added that goes from this auxiliary state to the same destination state.

\[ T''' = \]
\[ \{(s_1, s_2), c, c, (s_1', s_2')\} \]
\[ ((s_1, s_2), c, c, (s_1', s_2')) \in T'', \]
\[ s_1 \in S_1 \}
\[ \cup \]
\[ \{(s_1, s_2), c, c, (s_1', s_2')\} \]
\[ ((s_1, s_2), c, c, (s_1', s_2')) \in T'', \]
\[ s_2 \in S_2 \}
\[ \cup \]
\[ \{(s_1, s_2), c, c, (s_1', s_2')\} \]
\[ ((s_1, s_2), c, c, (s_1', s_2')) \in T'', \]
\[ s_1 \in S_1, s_2 \in S_2 \}

The set of states S consists of S1 excluding state q combined with the states from the refining machine and an extra state if state q has an action (λ(q) ≠ φ). In this case, similar to the synchronous parallel operator, an additional auxiliary state is required. In the case of the refinement operator, the set S2 is equivalent to q, S2 so the auxiliary state becomes q, S02 where q’s action is performed.

\[ S = (S_1 \setminus q) \]
\[ \cup \]
\[ (S_2) \]
\[ \cup \]
\[ \{(q, S_02)\} \]
\[ \lambda_1(q) \neq \phi \}

The new action mapping function λ is as below. States in S1 except for q keep their original actions. The states of S2 only have the actions of s2 and if present the state (q, s02) has the actions of both q and s02. As with the || operator, q’s actions are performed first because of the order in the textual syntax: A ⊥q B.

\[ \lambda = \{s_1 \rightarrow (A, O)\] \]
\[ \forall s_1 \in S_1, \lambda_1(s_1) \rightarrow (A, O), s_1 \neq q} \]
\[ \cup \]
\[ \{s_2 \rightarrow (A, O)\] \]
\[ \forall s_2 \in S_2, \lambda_2(s_2) \rightarrow (A, O)\}
\[ \cup \]
\[ \{(q, s_02) \rightarrow (A_1 \cup A_2, O_1 \cup O_2)\] \]
\[ \lambda_1(q) \rightarrow (A_1, O_1), \]
\[ \lambda_2(s_02) \rightarrow (A_2, O_2)\}

The transition function T is defined below. It consists of several subsets: transitions in E1 that do not involve q; transitions in E1 that go to q now go to the composed initial state; transitions looping on state q also go to the composed initial state; transitions that exit q; unmodified transitions in E2; finally duplications of transitions exiting s02 are added to q, s02 if present.

\[ T = \{(s_1, c_1, s_1')\] \]
\[ \forall (s_1, c_1, s_1') \in T_1, \]
\[ s_1 \neq q \wedge s_1' \neq q} \]
\[ \cup \]
\[ \{(s_1, c_1, s_2)\] \]
\[ \forall (s_1, c_1, q) \in T_1, \]
\[ \lambda_1(q) \neq \phi \rightarrow s_2 = (q, s_02), \]
\[ \lambda_1(q) = \phi \rightarrow s_2 = s_02\}
\[ \cup \]
\[ \{(s_2, c_1, (q, s_02))\]
leaving only the parallel operator removes non-deterministic transitions, such a clean up function similar to the clean up function of the parallel operator defines semantics between function blocks. The interactions between function blocks are in synchronous parallel as with Esterel’s[14] synchronous parallel operator. Due to the use of previous event values, compositions of blocks are guaranteed to be acyclic. Simultaneous event emission between parallel blocks is possible, but due that an event maybe lost if the associated function block instead reacts on a different event.

Using this operator on the two blocks in Fig. 2, the ConveyorController block and the ConveyorController BagData are said to be in parallel. In each execution of the ConveyorController_WithECC composite block, which contains these two, each block is executed once in a determined sequence, and outputs are passed back to the interface of the ConveyorController_WithECC block.

Semantics of a network of function blocks

At the block level HCECCs utilise the semantic ideas developed in [13]. In order to encompass the interconnectivity of function blocks and to remedy some of the other semantic ambiguities the block parallel operator defines semantics between function blocks. The interactions between function blocks are in synchronous parallel as with Esterel’s[14] synchronous parallel operator. Due to the use of previous event values, compositions of blocks are guaranteed to be acyclic. Simultaneous event emission between parallel blocks is possible, but due that an event maybe lost if the associated function block instead reacts on a different event.

Using this operator on the two blocks in Fig. 2, the ConveyorController block and the ConveyorController BagData are said to be in parallel. In each execution of the ConveyorController_WithECC composite block, which contains these two, each block is executed once in a determined sequence, and outputs are passed back to the interface of the ConveyorController_WithECC block.

V. TOOLS AND RESULTS

Two tools were developed in order to use HCECCs. The first tool was developed as a plug-in to the University of Auckland’s FB workbench FBench [15]. The plug-in, shown in Fig. 11, allows the visualisation and editing of function blocks that use HCECCs. HCECCs are stored in the traditional XML format, with additions to the schema to add elements and attributes.

Using the plug-in embedded ECCs can be still be edited graphically, manipulating layout, states and transitions. The plug-in then also allows the creation of ECCs, which refine existing states or are concurrent with an existing ECC. The visualisation of refinement and parallelism uses an approach similar to Esterel [14] and StateCharts [11], providing a simple view for the designer.

The second tool, called HCECC2ECC was developed to flatten an HCECC into an IEC 61499 standard ECC. The tool takes the XML file of a basic function block which contains an HCECC definition, and generates an identical function block file replacing the HCECC with an equivalent ECC.

Table II compares the sizes of manually created models using ECCs with automatically generated ECCs from HCECC specifications. In all of the given examples, the generated ECC has a smaller number of states and the specification uses only one function block. The number of transitions depends heavily on the application and the use of refinement and concurrency. This shows that the automatically generated ECC is more efficient in terms of function block space, although as all the states are in one block it is likely not as easy to manage at this level.

HCECCs are executed by translating the flattened block into C using FBtoEsterel developed in [13]. FBtoEsterel is also used to generate C code for the manually created models. Table III compares the execution speeds of both implementations mentioned above. A set of 1 million pseudo-random input vectors were generated for each example. The measured times do not include the time to run the test bench, so that the test bench would not influence the results. To accomplish this, each test bench was run with calls to the example’s run function, and the execution time without the function call was subtracted. These times were measured on a standard PC with 2GB of RAM.

In all cases, the code produced from an HCECC specification performed at least 2.9x faster than the manually created counterpart. The speed up can be attributed, predominantly to the reduction in the number of blocks that must be executed, but also to the reduction in states which reduces the number of if statements the ECC must check through for each execution. These size and performance comparisons demonstrate that not only are HCECCs easier to design, but more efficient code can be generated. The MP3Player example which benefited from this level.

\[
\begin{align*}
\forall (q, e_1, e_2, q') & \in T_1, \\
\forall s_1 & \in (S_1 \cup \{(q, s_1) | \lambda q \neq \phi\}) \\
\{ (s_1, e_1, c_1, s_1') \} & \in T_1, \\
\forall s_2 & \in (S_2 \cup \{(q, s_2) | \lambda q \neq \phi\}) \\
\{ (s_2, e_2, s_2') \} & \in T_2 \\
\{ ((q, s_2), e_2, s_2') \} & \lambda_1 (q) \neq \phi, \\
\forall (s_2, e_2, s_2') & \in T_2
\end{align*}
\]
most from the reduction in the number of function blocks executed 4.9x faster, showing that by removing the necessity for separate blocks, much greater performance can be attained whilst not reducing maintainability.

VI. RELATED WORK

In [16] A. Barji et al. compare IEC 61499 function blocks to CNet [9] and UML Statecharts [11]. The authors discuss many of the areas where function blocks are considered less expressive compared to the alternatives, specifically: representation of concurrency, synchronization and mutual exclusion; and clarity of graphical representation. The use of concurrency and hierarchy within HCECCs directly supplements these shortcomings to create a more desirable language.

In [8] Statecharts were used as a specification language, providing a simpler interface for complex designs. A function block network was then created from this specification. Methods such as these improve specification, but it is then necessary to use other development tools prior to function block implementation. The issues with function block semantics still remain in this work as opposed to our proposed approach.

VII. CONCLUSION

The use of concurrency and hierarchy within the HCECC semantics presented in this paper directly supplements the short comings of the IEC 61499 standard. The specification of complex examples is greatly simplified by their use, reducing the number of states and transitions required to represent a component. This in turn makes maintaining and debugging of code easier. The adopted synchronous semantics also mitigates the ambiguities of the standard. These features of HCECCs are fairly unique amongst function block specification enhancements.

As HCECCs are backwards compatible with the standard, existing translation, simulation and verification tools can be used. Additionally, using FBToEsterel [13], the code generated from HCECC specifications is up to 4.9x faster than models created using traditional function blocks.

Future work may also provide a mechanism for allowing internal events within an ECC. This would allow the specification of more complex systems, providing internal broadcast communications. A possible implementation may create an interface event that loops from output to input.

REFERENCES