Soundness and Determinism of HCECC Operators

1 Introduction

In the main paper, we present HCECCs as some syntactic sugar to allow designers to use Statecharts-like parallel and refinement features within basic function blocks in IEC 61499. HCECCs can then be translated into standard-compliant composite function blocks, as shown in Fig. 1. We also present in the main paper a new synchronous semantics for IEC 61499.

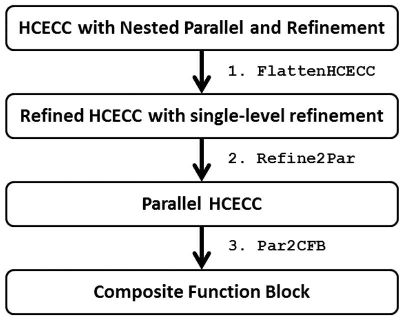

HCECC with Nested Parallel and Refinement
1. FlattenHCECC
Refined HCECC with single-level refinement
2. Refine2Par
Parallel HCECC
3. Par2CFB
Composite Function Block

Figure 1: Overview of HCECC translation

This addendum contains proofs to show the soundness of our approach. Section 2 shows the soundness of the translation of HCECCs to composite function blocks. Section 3 shows that the proposed synchronous semantics is deterministic and reactive by showing the equivalence of the developed semantics with that of the PRET-C language [1].

2 Soundness of HCECC to CFB transformation

2.1 Multi-level refined HCECC to single-level refined HCECC

Algorithm `FlattenHCECC` is used to flatten a refined HCECC with multiple levels of nesting to a refined HCECC with single levels of refinement. `FlattenHCECC` removes all refinement in a depth-first fashion by using `RemoveRefinement` for the bottom-most HCECC with no further refinement (line 8) in a recursive fashion, until the block is reduced to a refined HCECC with only a single level of refinement for any state.

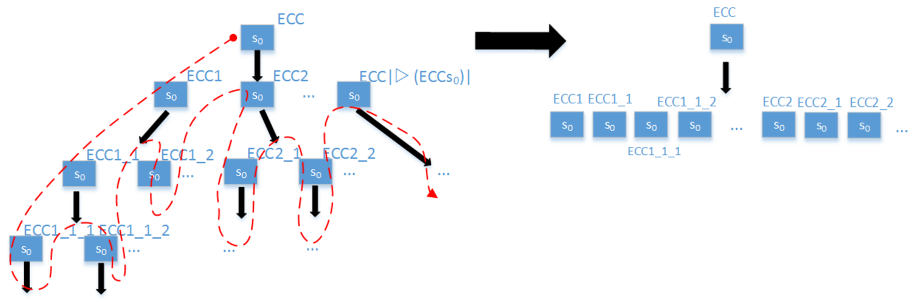


Figure 2: In-order depth first traversal of the refinement tree in Algorithm `FlattenHCECC`

Fig. 2 shows how `FlattenHCECC` executes on a multi-level refined HCECC A to produce a single-level refined HCECC B. Note that both HCECCs have the same interface and main ECC ECC (line 17). The algorithm goes through each refined state in the main ECC (lines 2–16) and every refined state is then recursively treated as a sub-tree, resulting in an in-order traversal of the tree.

The following theorem proves the soundness of Algorithm `FlattenHCECC`.

Theorem The execution of a multi-level refined HCECC A is identical to the execution of the single-level refined HCECC B obtained using Algorithm `FlattenHCECC`, assuming the same environment behaviour in every tick.

Proof. A proof by induction can be done as follows.

Base case: The interface of both the refined HCECCs is the same (line 17). Assuming the same environment behaviour, we can observe that at the start of the first tick, both the HCECCs have the same input events furnished by the environment, and the status of their input variables is also the same.

The base case looks at the first transition (first micro-step) that will be fired in both HCECCs at the start of the first tick. Both HCECCs will be in the same start state s_0 of ECC ECC. The following possibilities need to be considered:

1. Some transition of s_0 is enabled: Both the HCECCs initialize in this non-refined state by definition. The outgoing transitions set in the two HCECCs is the same at the start of the tick, transitions are evaluated in the same order, and the same highest priority transition fires.

2. No transition of s_0 is enabled: If no transition of s_0 is enabled, then there are the following possibilities:

   (a) s_0 is not a refined state: In this case, no transition fires in both HCECCs.

```
Input: Refined HCECC $\langle \mathcal{I}, L, \texttt{ECC} = \langle S, s_0, \lambda, T\rangle, \texttt{ECCs}, \triangleright\rangle$, $Refine()$ (a map
       from an ECC $\texttt{ECC}_i \in \texttt{ECCs}$ to a set of parallel ECCs ($\texttt{ECCs}_i$) and a
       refinement operator ($\triangleright_i$) such that $\langle \mathcal{I}, L\mathcal{I}, \texttt{ECC}_i, \texttt{ECCs}_i, \triangleright_i\rangle$ is a refined
       HCECC)
Output: Flattened refined HCECC $\langle \mathcal{I}, L', \texttt{ECC}, \texttt{ECCs}', \triangleright'\rangle$
1  $L' = L$, $\texttt{ECCs}' = \emptyset$, initialize $\triangleright'$;
2  foreach refined state $s \in S$ do
3      Ordered set of ECCs $\texttt{ECCs}_s := \emptyset$;
4      foreach $ECC_i \in \triangleright(s)$ do
5          if $Refine(ECC_i)$ is not null then
6              $Refine(\texttt{ECC}_i)$=$(\texttt{ECCs}_i, \triangleright_i)$;
7              Refined HCECC $\texttt{HCECC}_{\triangleright i} = \langle \mathcal{I}, L', \texttt{ECC}_i, \texttt{ECCs}_i, \triangleright_i\rangle$;
8              Parallel HCECC $\langle \mathcal{I}, L', \texttt{ECCs}'\rangle$ = RemoveRefinement(
               FlattenHCECC($\texttt{HCECC}_{\triangleright i}$, $Refine()$));
9              $\texttt{ECCs}_s = \texttt{ECCs}_s \oplus (\texttt{ECCs}'_{||})$;
10         end
11         else
12             $\texttt{ECCs}_s = \texttt{ECCs}_s \oplus \texttt{ECC}_i$;
13         end
14     end
15     $\triangleright'(s) = \texttt{ECCs}_s$, $\texttt{ECCs}' = \texttt{ECCs}' \cup \texttt{ECCs}'$;
16 end
17 return $\texttt{HCECC}_{ret} = \langle \mathcal{I}, L', \texttt{ECC}, \texttt{ECCs}', \triangleright'\rangle$;
```

Algorithm 1: Algorithm `FlattenHCECC`.

(b) s_0 is a refined state but initial states of the refining ECCs in $\triangleright(s_0)$ are not refined: The transitions of the initial state of the first ECC $ECC1 \in \triangleright(s_0)$ in A are evaluated. If no transitions are enabled, then the transitions of the initial state of the second ECC $ECC2 \in \triangleright(s_0)$ are evaluated, and so on. By definition, as there is no further nesting of refinement, s_0 has the same set of refining ECCs in B (line 12). Hence, either the same first enabled transition fires in both, or both HCECCs do not experience a transition (when none of the transitions of the initial states of the refining behaviours are enabled).

(c) s_0 is a refined state and some initial states of the refining ECCs in $\triangleright(s_0)$ are refined: Again, in A, transition sets are evaluated in the following order: transitions of the initial state of $ECC1$, transitions of the initial state of the first refining ECC $ECC1\_1$ refining the initial state of $ECC1$, ..., transitions of the initial state of $ECC2$, transitions of the initial state of the first refining ECC $ECC2\_1$ refining the initial state of $ECC2$, ..., and so on. Assuming that Algorithm `RemoveRefinement` is sound (to be proved later), we can see that the same transitions sets are also evaluated in the same order in B. In this case, the order of transitions will be transitions of the initial state of $ECC1$, transitions of the initial state of $ECC1\_1$ (the next ECC $\triangleright'(s_0)$ as per line 9 and 15), ..., transitions of the initial state of $ECC2$, transitions of the initial state of $ECC2\_1$ (the next ECC $\triangleright'(s_0)$ as per line 9 and 15), $\ldots$, and so on.

We can note that at the same first transition fires in the first tick. We can also extend this result to say that since variable modifications happen at transitions or in the actions of the resulting states, at the end of the first transition, variable values are the same in both HCECCs. Similarly, we can say that since the same transition is taken, the same actions are performed in the resulting states, and hence the same output events are emitted by both HCECCs.

Inductive step: We assuming that the same n-th micro-step (transitions in refining ECCs fires during the first tick in both A and B. Consequently, the state of the variables and output events are also the same after this n-th transition.

Given the above hypothesis, we prove that the same $n+1$-th transition fires in both HCECCs in the first tick.

Given that more than 1 transitions have fired in the multi-level refined HCECC, we can deduce the following:

- The initial state s_0 is a refined state. If it was not a refined state, the tick would have ended after a single or no outgoing transition had fired from s_0 (as per point 1 in the base case).
- No outgoing transition of s_0 was enabled: Again, as per the above observation, the tick would have ended after a single enabled transition had fired from s_0.

A can have levels 1 (single-level refinement) to m where $m \geq 1$ of nested refinement. Assuming that the n-th transition was at level l where $1 \leq l \leq m$. There are now the following possibilities:

1. $l = 1$: In this case, one of the refining ECCs $ECC\_i \in \triangleright(s_0)$ of state s_0 had an enabled transition in its initial state. If this ECC had any refined behaviours, they will be ignored, and the order of evaluating outgoing transitions will be from the initial state of the next refining ECC $ECC\_i+1 \in \triangleright(s_0)$ of state s_0, then the initial states in order of the refining ECCs of the initial state of ECC_i+1 (and so on until m levels), then the initial state of the next refining ECC $ECC\_i+2 \in \triangleright(s_0)$ of state s_0, and so on.

   The same order of evaluation is followed in the single-level HCECC. The order of states is the initial state of the next refining ECC $ECC\_i+1 \in \triangleright'(s_0)$ of state s_0, then the initial states in order of the refining ECCs of the initial state of ECC_i+1 (as all ECCs added to the set of refining ECCs of s_0 in lines 5-10 will be disabled assuming correctness of `RemoveRefinement`), then the initial state of $ECC\_i+1\_1 \in \triangleright'(s_0)$ corresponding to the first refining ECC of the initial state of ECC_i+1, $\ldots$, $ECC\_2\_1 \in \triangleright'(s_0)$, and so on.

2. $l > 1$: Like in the previous point, in the multi-level refined HCECC, the next transition to be triggered will be evaluated by following a *depth-first* order of initial states of first further refining ECCs (levels $l{+}1$ and beyond) and then to next higher level. This is equivalent to an in-order traversal of the refinement tree, as shown in Fig. 2.

   In the single-level refined HCECC, the order of initial states of ECCs in $\triangleright'(s_0)$ is sequential, but is the same as the order for the multi-level refined HCECC.

The proof that variable values remain the same, and that the output events being emitted are also the same at the end of the $n{+}1$-th evaluated transition follows from the same description.

Given that in the first tick, given the same inputs, both the HCECCs experience the same transitions (in the same order), have the same internal state (variables) and emit the same outputs, we can extend the proof, using the induction given above, to prove that given the same internal state and environment inputs, both HCECCs experience the same transitions, have the same variables and emit the same outputs in ***any*** tick.

□

2.2 Single-level refined HCECC to parallel HCECC

Algorithm `RemoveRefinement` (page 7 of the main paper and reproduced here) is used to convert a single-level refined HCECC to a parallel HCECC with single levels of refinement.

The following theorem proves the soundness of Algorithm `RemoveRefinement`.

Theorem The execution of a single-level refined HCECC A is identical to the execution of the corresponding parallel HCECC B obtained using Algorithm `RemoveRefinement`, assuming the same environment behaviour in every tick.

Proof. A proof by induction can be constructed as follows.

Base case: The interface of both the HCECCs is the same. Assuming the same environment behaviour, we can observe that at the start of the first tick, both the HCECCs have the same input events furnished by the environment, and the status of their input variables is also the same.

The base case looks at the first transition (first micro-step) that will be fired in both HCECCs at the start of the first tick. The refined HCECC is in state s_0 of the main ECC ECC. The parallel HCECC is also in the same state because ECC is the first parallel ECC (line 23). There are different possibilities of which transition will fire:

1. Some transition of s_0 is enabled: As both HCECCs are in the same initial state, the outgoing transitions set in the two HCECCs is also the same. The first enabled transition from s_0 will be taken (in order) in both cases. The outgoing transition from s_0 in the refined HCECC will disable all its

```
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 $(\texttt{ECCs}_{\triangleright}, L_{\triangleright})$
//Modify locals;
Variable set $V_L\prime = V_L \cup \{s_{\triangleright}\_Disabled\}$ ($s_{\triangleright}\_Disabled$ is of type Boolean);
Algorithm set $A'_L = A_L \cup \{s_{\triangleright}\_Enable, s_{\triangleright}\_Disable\}$;
//Create new local declarations;
$L_{\triangleright} = \langle V_L\prime, A'_L \rangle$;
//Create $\texttt{ECCs}_{\triangleright}$;
Initialize ordered set of ECCs $\texttt{ECCs}_{\triangleright} = \emptyset$;
//Modify ECC;
Initialize empty action map $\lambda_0$;
//Modify states;
for each state $s \in S$ do
    if $s = s_{\triangleright}$ then
        $\lambda_0(s) = \lambda(s) \oplus \{s_{\triangleright}\_Enable\}$;
    end
    else if $s_{\triangleright}$ has a transition (in $T$) to $s$ then
        $\lambda_0(s) = \lambda(s) \oplus \{s_{\triangleright}\_Disable\}$;
    end
    else
        $\lambda_0(s) = \lambda(s)$;
    end
end
$\texttt{ECC}_0 = \langle S, s_0, \lambda_0, T \rangle$;
$\texttt{ECCs}_{\triangleright} = \texttt{ECCs}_{\triangleright} \oplus \{\texttt{ECC}_0\}$;
//Modify ECCs in $\texttt{ECCs}_s$;
for each $\texttt{ECC}_i = \langle S_i, s_{0_i}, \lambda_i, T_i \rangle \in \texttt{ECCs}_s$ do
    Create set of states $S_{\triangleright i} = S_i \cup \{Disabled\}$;
    Set state $s_{0_{\triangleright i}} = Disabled$;
    if $s_{\triangleright} = s_0$ then
        $s_{0_{\triangleright i}} = s_{0_i}$;
    end
    Initialize action map $\lambda_{\triangleright i} = \lambda_i$;
    Add $\lambda_{\triangleright i}(Disabled) = \emptyset$;
    Initialize transition function $T_{\triangleright i} = T_i$;
    for each $s \in S_{\triangleright i}$ do
        if $s = Disabled$ then
            $T_{\triangleright i}(s) = \{s \xrightarrow{1,!s_{\triangleright}\_Disabled} s_{0_i}\}$;
        else
            $T_{\triangleright i}(s) = \{s \xrightarrow{1,s_{\triangleright}\_Disabled} Disabled\} \oplus T_i(s)$ ;
        end
    end
    Create ECC $\texttt{ECC}_{\triangleright i} = \langle S_{\triangleright i}, s_{0_{\triangleright i}}, \lambda_{\triangleright i}, T_{\triangleright i} \rangle$;
    $\texttt{ECCs}_{\triangleright} = \texttt{ECCs}_{\triangleright} \oplus \{\texttt{ECC}_{\triangleright i}\}$;
end
return Pair $(\texttt{ECCs}_{\triangleright}, L_{\triangleright})$;
```

Algorithm 2: Algorithm `RemoveRefinement`

refining behaviours, if any. In the parallel HCECC, the same transition will result in the algorithm $s_{\triangleright}\_Disable$ to be executed in the resulting state (line 15-16), resulting in all ECCs added in lines 24–43 taking a transition to their respective *Disabled* state (line 36).

2. No transition of s_0 is enabled: If no transition of s_0 is enabled, then there are the following possibilities:

   (a) s_0 is not a refined state: In this case, no transition fires in both HCECCs.

   (b) s_0 is a refined state: The transitions of the initial state of the first ECC $ECC1 \in \triangleright(s_0)$ are evaluated, followed by those of the initial state of the second ECC $ECC1 \in \triangleright(s_0)$, and so on, in the refined HCECC.

   In the case of the parallel HCECC, if s_0 is the refined state, the algorithm $s_{\triangleright}\_Enable$ will fire (line 12-13), due to which corresponding ECCs $ECC1\ldots$ will be enabled to evaluate the transitions from their original initial states as in the refined HCECC (line 28-29).

   The correspondence between the two HCECCs is shown in Fig. 3.

   As the same sets of transitions are evaluated in the same order, the first enabled transition is taken, resulting also in the same state of internal variables, as well as output events.

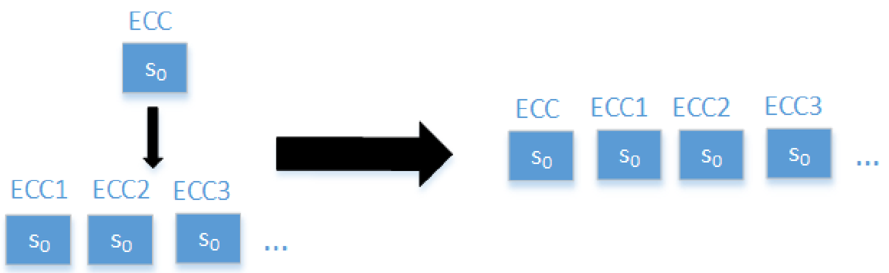


Figure 3: HCECC translation by Algorithm `RemoveRefinement`

Inductive step: We assume that the same n-th micro-step (transition in a refining behaviours in A and a corresponding transition in some parallel ECC in B) fires during the first tick. Consequently, the state of the variables and output events are also the same between A and B.

Given this hypothesis, we now prove that the same $n+1$-th transition fires in both HCECCs during the same tick.

Given that more than one transition ($n \geq 2$) has fired in the refined HCECC, we can deduce that (a) the initial state s_0 is a refined state, and that no outgoing transition from s_0 was enabled. If any of these were true, no more than 1 micro-steps could have happened in the refined HCECC. We can also say that in the parallel HCECC, no outgoing transition of s_0 was enabled because it has the same transition set as s_0 in the refined HCECC.

For the refined HCECC, given that n micro-steps (or transitions in the refining ECCs of s_0) have happened, we know that the transitions of $ECC1$, $ECC2, \ldots ECCm$ have been evaluated, where $m \geq n$. We look at the following possibilities:

1. $m = |\triangleright(s_0)|$: if $ECCm$ was the last refining ECC of s_0, no further transitions are to be enabled (the tick has ended). In the parallel HCECC also, all refining ECCs from the refined HCECC are appended into the ordered set of parallel ECCs after addition of enabling/disabling behaviour (line 42). Hence, the number of these appended ECCs is the same as the number of refining ECCs in the refined HCECC, or $|\triangleright(s_0)|$.

   In the parallel HCECC, all other parallel ECCs (if any) beyond the $|\triangleright(s_0)|$-th ECC are disabled because they relate to the refining behaviour of another state (not s_0) and hence they do not take any transition (line 36), resulting in the end of the tick without any further change.

2. $m < |\triangleright(s_0)|$: if there are more ECCs $ECCm+1, \ldots$ remaining after the n-th transition taken from the initial state of $ECCm$, then the order of evaluation of transitions will be from the initial states of $ECCm+1, \ldots$ until the last element of the set of refining ECCs. In the parallel HCECC also, all refining ECCs from the refined HCECC are appended into the ordered set of parallel ECCs after addition of enabling/disabling behaviour (line 42). Hence, the number of these appended ECCs is the same as the number of refining ECCs in the refined HCECC, or $|\triangleright(s_0)|$. Also, all of these ECCs are enabled (lines 28-29) because the initial state s_0 of the main ECC ECC is a refined state. The order of evaluation of transitions in the parallel HCECC therefore follows the same order as in the refined state. So if a transition is enabled in any of the further refining behaviours in the refined HCECC, the same corresponding transition will also be the next one to be triggered in the parallel HCECC, resulting in the same state, variables, and output events at the end of the $n+1$-th tick.

The proof that variable values remain the same, and that the output events being emitted are also the same at the end of the n+1-th evaluated transition follows from the same description.

Given that in the first tick, with the same inputs, both HCECCs have the same transitions (in the same order), have the same internal state (variables) and emit the same outputs, we can extend the proof, using the induction given above, to prove that given the same internal state and environment inputs, both HCECCs experience the same transitions, have the same variables and emit the same outputs in ***any*** tick.

$\square$

2.3 Parallel HCECC to Composite Function Block

Algorithm `Par2CFB` is used to convert a parallel HCECC to composite function block network.

The following theorem proves the soundness of Algorithm `RemoveRefinement`.

Theorem The execution of a parallel HCECC A is identical to the execution of the corresponding composite function block B obtained using Algorithm `RPar2CFB`, assuming the same environment behaviour in every tick.

Proof. A proof by induction can be constructed as follows.

Base case: The interface of both the HCECC and the CFB is the same (line 38). Assuming the same environment behaviour, we can observe that at the start of the first tick, both the models have the same input events furnished by the environment, and the status of their input variables is also the same (initialized).

The base case looks at the first transition (first micro-step) that will be fired in both models at the start of the first tick. The parallel HCECC has a set of ECCs $ECC1, \ldots$ and the composite block has the same number of instances $fb1, \ldots$, derived from the corresponding ECCs in the parallel HCECC. Also, both sets have the same corresponding orders (lines 8-21).

In the parallel HCECC, the transitions out of the initial state of the first ECC $ECC1$ will be evaluated in order. There are two possibilities:

1. If a transition in $ECC1$ is enabled: The first enabled transition is taken. In the composite block, the transitions of the initial state of the first instance $fb1$ will be evaluated. $fb1$ has access to the same input information as on the interface of the composite block (lines 25-35). Also, the initial state of $fb1$ has the same transitions (in the same order) as the initial state of $ECC1$ in the parallel HCECC (line 16). As the transitions are identical, the same corresponding transition will fire in $fb1$. Additional actions *UpdateOutputs* amd $PARO$ do not change further the status of variables, but merely transfer any updated values to the next instance (lines 11).

2. If no transition in $ECC1$ is enabled: The transitions in $ECC2$ are evaluated. In the composite block, in $fb1$, a transition is taken to the loop state (line 16) where actions *UpdateOutputs* amd $PARO$ do not change the status of variables or outputs, but merely transfer any updated values to the next instance (lines 11). The loop transition therefore does not affect the status of $fb1$. Next, the transitions of $fb2$ are evaluated. The same process continues until either a transition is taken in some ECC ECC' in HCECC, and in that case a corresponding transition is taken in instance fb' in the composite block. Leading up to fb', all previous instances $fb1, \ldots$ take loop transitions to merely forward input information forward to the next instance.

   If no transition is enabled in the parallel HCECC, all instances in the composite block take only loop transitions.

```
Input: Parallel HCECC $\langle \mathcal{I}, L, \mathtt{ECCs} \rangle$
Output: CFB $\mathtt{CFB}$
//Create FB instances;
Ordered set $\mathtt{FBs} = \emptyset$;
${E_I}' = E_I \cup \{PARI\}$; ${E_O}' = E_O \cup \{PARO\}$;
${V_I}' = V_I \cup V_L \cup V_O$; ${V_O}' = V_L \cup V_O$;
${\alpha_I}' = \alpha_I \cup \{(PARI, v_I) | v_I \in V_L \cup V_O\}$;
${\alpha_O}' = \alpha_O \cup \{(PARO, v_O) | v_O \in V_O\}$;
$\mathcal{I}' = \langle {E_I}', {V_I}', {\alpha_I}', {E_O}', {V_O}', {\alpha_O}' \rangle$;
$L' = \langle \emptyset, A_L \cup \{UpdateOutputs\} \rangle$;
for each $\mathtt{ECC} = \langle S, s_0, \lambda, T \rangle \in \mathtt{ECCs}$ do
    $S' = S$; $s'_0 = s_0$; Initialize $\lambda'$, $T'$;
    for each $s \in S$ do
        $\lambda'(s) = \lambda(s) \oplus \{UpdateOutputs, PARO\}$;
        $T'(s) = T(s)$;
        if there is no transition $s \xrightarrow{1} s_1$ then
            Loop state $s_l$; $S' = S' \cup \{s_l\}$;
            $\lambda'(s_l) = \{UpdateOutputs,\ PARO\}$;
            $T'(s) = T'(s) \oplus \{s \xrightarrow{1} s_l\}$;
            $T'(s_l) = \{s_l \xrightarrow{e,b} s_1 \mid s \xrightarrow{e,b} s_1\}$;
        end
    end
    $\mathtt{ECC}' = \langle S', s'_0, \lambda', T' \rangle$; $\mathtt{BFB}' = \langle \mathcal{I}', L', \mathtt{ECC}' \rangle$;
    $\mathtt{fb} = \langle name_i, \mathtt{BFB}' \rangle$; $\mathtt{FBs} = \mathtt{FBs} \oplus \{\mathtt{fb}\}$;
end
//Create connections;
$n = |\mathtt{FBs}|$; $C_{events} = \emptyset$; $C_{var} = \emptyset$;
for each $i \in [1, n]$ do
    Select the $i$-th FB instance $\mathtt{fb}_i \in \mathtt{FBs}$;
    $C_{events} = C_{events} \cup \{(e_I \mapsto \mathtt{fb}_i.e_I) \mid e_I \in E_I\}$;
    $C_{events} = C_{events} \cup \{(\mathtt{fb}_i.e_O \mapsto e_O) \mid e_O \in E_O\}$;
    if $i = 1$: $C_{var} = C_{var} \cup \{(v_I \mapsto \mathtt{fb}_i.v_I) \mid v_I \in V_I\}$;
    if $i = n$: $C_{var} = C_{var} \cup \{(\mathtt{fb}_i.v_O \mapsto v_O) \mid v_O \in V_O\}$;
    if $i < n$ then $\mathtt{fb}_j = \mathtt{fb}_{i+1}$ else $\mathtt{fb}_j = \mathtt{fb}_1$;
    $C_{events} = C_{events} \cup \{(\mathtt{fb}_i.PARO \mapsto \mathtt{fb}_j.PARI)\}$;
    $C_{var} = C_{var} \cup \{(\mathtt{fb}_i.v_L \mapsto \mathtt{fb}_j.v_L \mid v_L \in V_L\}$;
    $C_{var} = C_{var} \cup \{(\mathtt{fb}_i.v_O \mapsto \mathtt{fb}_j.v_O \mid v_O \in V_O\}$;
end
//Create CFB;
$\mathtt{FBNetwork} = \langle \mathtt{FBs}, C_{events}, C_{var} \rangle$;
$\mathtt{CFB} = \langle \mathcal{I}, \mathtt{FBNetwork} \rangle$;
return CFB;
```

Algorithm 3: Algorithm Par2CFB

At the end of the first micro-step, the status of variables, states of individual ECCs (instances) and outputs remain the same in both models.

Inductive step: Assuming that the same n-th transition fired during the first tick (and hence the state of the variables and output events are also the same), we prove that the same $n+1$-th transition fires in both models during the same tick.

Given that n micro-steps (or transitions in the parallel ECCs) have happened, we know that the transitions of $ECC1, ECC2, \ldots ECCm$ have been evaluated, where $m \geq n$. Similarly, transitions in the function block instances $fb1, \ldots, fbm$ have been evaluated in the composite block, with exactly n having taken a transition corresponding to the transitions in the parallel HCECC, and the rest $m-n$ must have taken loop transitions.

We look at the following possibilities:

1. $m = |ECCs|$: if $ECCm$ was the last parallel ECC, no further transitions are to be enabled. In the composite block also, since fbm was the last instance in the set of instances, no further transitions can be evaluated. The tick has ended.

2. $m < |\triangleright(s_0)|$: if there are more ECCs $ECCm+1, \ldots$ remaining after the n-th transition taken from the initial state of $ECCm$, then the order of evaluation of transitions will be from the initial states of $ECCm+1, \ldots$ until the last ECC in $ECCs$.

   In the composite block also, the transitions out of the initial states of the ECCs contained in the instances $fbm+1$ onwards will be evaluated in the same order as the parallel HCECC. If the next enabled transition in the parallel HCECC is in the initial state of some ECC $ECCm+i$ $(i \geq 1)$, the same transition in the initial state of $fbm+i$ will also be enabled as the transition sets are the same (lines 13-18) and connections between instances ensure all previous updates in the current tick are available (lines 25-35). In all ECCs after $ECCm$ and before $ECCm+i$, only loop transitions were enabled, resulting in no change in data but forwarding of data using *UpdateOutputs* amd $PARO$ (lines 15, 16).

The proof that variable values remain the same, and that the output events being emitted are also the same at the end of the n+1-th evaluated transition follows from the same description.

Given that in the first tick, with the same inputs, both models have the same transitions (in the same order), have the same internal state (variables) and emit the same outputs, we can extend the proof, using the induction given above, to prove that given the same internal state and environment inputs, both models experience the same transitions, have the same variables and emit the same outputs in ***any*** tick. The only additional observation required to prove this is that in the parallel HCECC, internal variables retain their values after the end of a tick for the next tick. In the composite block, a feedback connection between the last instance and the first instance ensures that this data is persistent (line 30).

□

3 Determinism and Reactivity of the Proposed Synchronous Semantics

The HCECC semantics is inspired by the PRET-C language. The previous section provided the soundness of HCECC to composite function blocks transformation. We now show the equivalence of a composite block to a PRET-C program. This allows us to show that the semantics of function blocks as presented in the main document is the same as that of PRET-C programs, which have proven deterministic and reactive semantics [1].

3.1 Introduction of PRET-C

PRET-C [1] is a synchronous extension to the C language for efficient concurrent programming. PRET-C is better suited for embedded real-time applications compared to other synchronous languages because C is the language of choice for embedded designers. Unlike the earlier synchronous C extensions such as ReactiveC [2], PRET-C is specially designed for predictable execution. To enforce behavioral and timing predictability, PRET-C only allows the usage of a subset of C. The use of dynamic pointers, dynamic memory allocation, unbounded loops and unbounded recursion are forbidden because their behavioral cannot be statically determined.

Table 1: PRET-C synchronous extensions

Statement	**Description**
ReactiveInput I	Declare an input I sampled from the environment.
ReactiveOutput O	Declare an output O emitted to the environment.
thread T() {p}	Declare a thread T with the body p.
PAR(T_1,...T_n)	Spawn n number of threads to execute concurrently in fixed order.
EOT	Mark the end of a tick.
[weak] abort p when exp	Preempt p when the expression is true.
await(exp)	Wait until a tick where the expression is true.

Table 1 presents the key synchronous extensions introduced by PRET-C. The ReactiveInput and ReactiveOutput statements declare input and output variables for the system respectively. The values for input variables are sampled from the environment at the start of a tick and output variables are emitted to the environment at the end of a tick or reaction. The THREAD statement declares a thread in PRET-C. Like a C function, the body of the thread is defined inside curly brackets. The PAR statement suspends the parent thread and spawns a set of concurrently executing child threads. The thread resumes execution once

```
 1 ReactiveInput int I;
 2 ReactiveOutput int O;
 3 int X = 0;
 4
 5 void main() {
 6     X = 0; O = 0;
 7     while (1) {
 8         abort {
 9             PAR(T1,T2);
10         } when (X >= 3);
11         X = 0; O = 0;
12         EOT;
13     }
14 }
```

```
14 thread T1() {
15     while(1) {
16         X += I;
17         EOT;
18     }
19 }
20 thread T2() {
21     while(1) {
22         if (X >= 3) {
23             O = 1;
24         }
25         EOT;
26     }
27 }
```

Tick	**I**	**X**	**O**
1	0	0	0
2	1	1	0
3	2	3	1
4	3	0	0

Figure 4: Example of a PRET-C program

all child threads terminate. The PAR statement is unlike the traditional parallel (||) of other synchronous languages such as Esterel [3] follows the ordered design pattern presented previously in section **??**. The order of execution is defined by the order the threads appear in the PAR statement. For example, in the statement $\text{PAR}(\text{T}_1, \text{T}_2, \text{T}_3)$, the execution within a tick will be T_1 followed by T_2 then T_3. The EOT statement declares the end of the tick of the thread, referred to as the thread's *local tick*. The EOT statement is a synchronization barrier between active threads. Only when all active threads reach their respective *local tick*, a *global tick* happens allowing the threads to begin executing their next tick. The **abort** statement preempts a program body when the conditional expression evaluates to be true. A strong abort (ABORT) evaluates the condition at the beginning of the tick and a weak abort evaluates the condition at the end of the tick. The **await** statement, pauses at each tick and terminates when the expression evaluates to true.

Fig. 4 shows a simple PRET-C program containing a main thread (lines 5 to 14) and two child threads t1 (lines 14 to 19) and t2 (lines 20 to 27). During each tick, the program accumulates the values from input I onto the global variable X. Once X has reached the threshold value of 3, the output O is set to 1. The table in Fig. 4 presents the values of the variables for a given trace. In the first tick, the main thread executes the PAR statement and spawns the threads T1 and T2 (line 9). The PAR statement executes T1 followed by T2. T1 increments X by the value of 0 (I) and reaches its *local tick* (line 17). T2 then begins execution and evaluates the conditional statement (line 22) to false and T2 reaches its *local tick* (line 25). This marks the end of the first *global tick* of the program. The values of both X and O remain 0. In the second tick, T1 resumes execution from the EOT (line 17) statement and increments X by the value of I (i.e. 1) then T1 completes its *local tick*. T2 then resumes execution from its EOT (line 25) statement and the conditional statement(line 22) is still evaluated to false. The value for X is now 1. In the third tick, T1 again resumes execution from the EOT statement and increments X to 3 and reaches its *local tick*. T2 then resumes execution. In this tick the conditional statement (line 22) is evaluated to true and output O is set to 1. The ABORT statement (line 8) is a strong abort, which evaluates the condition at the start of the *global tick* to preempt the abort body (code between the curly brackets). Therefore, during the fourth tick the preemption will occur and execution will resume from (line 11)and the variables X and O will be reset to 0.

3.2 Ensuring semantic equivalence between composite function blocks and PRET-C

Table 2: Mapping between IEC 61499 elements and PRET-C features

Function block element	**PRET-C feature**
Function blocks	PRET-C threads
Input/output events	Global boolean variables
Input/output data	Global variables
Event-data association	Conditional statements
Internal variables	Local variable
ECC states	Defined by a label and EOT
ECC Transition	Goto statement
Algorithms	Bounded C functions
Transition conditions	Condition statement
Function block network	Ordered composition (*PAR* statement)
Function block hierarchy (composite function blocks)	Thread nesting

We provide a mapping between IEC 61499 elements and the features of PRET-C (shown in Table 2) to give formal semantics to IEC 61499. Intuitively, each function block is translated into a PRET-C thread. Thus, the function block network naturally maps to ordered composition (PAR statements). The order of threads declared in the PAR statement is defined by the order in which function block instances occur within a composite block. The ordered composition $\otimes$ is equivalent to the PAR statement, that is, PAR(t1,t2) is the same as t1$\otimes$t2. Event and data connections within the network are mapped to global variables in PRET-C. Access to these variables are guaranteed to be thread-safe and deterministic because of the fixed ordered execution of threads in PRET-C. ECC states are mapped to labels where the state boundaries are defined by EOT statements. ECC algorithms are mapped to bounded C functions. ECC transitions are mapped to goto statements guarded by conditional statements. The reason behind the mapping of ECCs to labels and goto statements is to produce PRET-C and assembly code that is easily identifiable to ECC states. To illustrate this, an example is presented in the next section.

3.2.1 Basic function blocks

The mapping of basic function block is composed of two components: the function block structure and the function block body. The function block structure implements the function block interface (i.e., the input and output ports) and any internal variables of the function block. The function block body implements the behavior of the function block such as the ECC, ECC algorithms and event-data associations. Fig.5 presents an example of the PRET-C structure for the `ConveyorSensor` basic function block. The numbers on the figure show the mapping between the function block presented in Fig.5a and the code in Fig.5b.

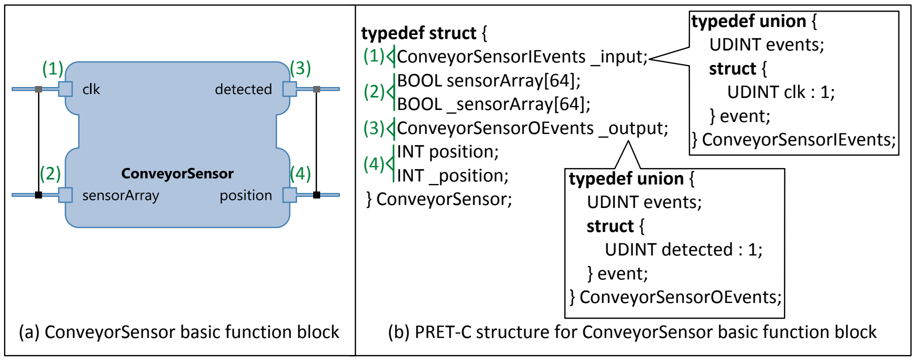
Event ports: For each basic function block, two C unions are used to implement the event ports. One implements the values of input event ports (i.e., `ConveyorSensorIEvents`) and another implements the values of output event ports (i.e, `ConveyorSensorOEvents`). The unions contain two elements: the `event` structure and `events` variable. The `event` structure maps the values of each event port to a single bit (i.e., `clk : 1`). The `events` variable provides a quick interface to reset the event during each reaction. The value of the `events` variable is the concatenation of each bit in the `event` structure since elements of a union share the same memory space.

Data ports: For the basic function block, the value of each data port is persistent and updated only when the associated event is present. For example, the value of `sensorArray` is only updated when the `clk` event is present. Therefore, each data port is implemented using two variables: a buffer variable and a data port variable. The buffer variable represents the latest value available to the data port. The data port variable represents the value currently available to the function block. The data port variable is updated to the value of the buffer variable only when the associated event is present. To distinguish these two variables, the buffer variable is prefixed with an underscore (i.e., `_sensorArray`) while the data port variable is not (i.e., `sensorArray`).

Fig.6a presents an example of the PRET-C code for the body of the `ConveyorSensor` basic function block.

Line 1: presents the declaration of the previously presently interface variable.

Line 2: presents the PRET-C thread which implements the behavior of the basic function block (i.e., the ECC). The # symbol in the figure is a prefix placeholder for the instance name when the function block is placed within a network. For example, the `PickUpSensor` and the `DropOffSensor` function blocks in the robotic arm example are instances of the `ConveyorSensor` basic function block. Therefore, the PRET-C code for the `PickUpSensor` function block will replace the # symbol with "PickUpSensor" (i.e., "thread PickUpSensor").


(1)
clk
detected
(3)
(2)
ConveyorSensor
(4)
sensorArray
position
(a) ConveyorSensor basic function block
(1)
(2)
(3)
(4)
(b) PRET-C structure for ConveyorSensor basic function block

```
typedef struct {
 ConveyorSensorIEvents _input;
 BOOL sensorArray[64];
 BOOL _sensorArray[64];
 ConveyorSensorOEvents _output;
 INT position;
 INT _position;
} ConveyorSensor;
```

```
typedef union {
   UDINT events;
   struct {
      UDINT clk : 1;
   } event;
} ConveyorSensorIEvents;
```

```
typedef union {
   UDINT events;
   struct {
      UDINT detected : 1;
   } event;
} ConveyorSensorOEvents;
```

Figure 5: Mapping of `ConveyorSensor` basic function block interface to PRET-C code

Lines 3-4: The initializations of the thread involve creating a pointer called `me` (line 3) and sampling the inputs (line 4). The `me`pointer allows the programmer to conveniently refer to the function block interface variable when designing ECC algorithms.

Lines 5-10: presents the implementation of the ECC and each ECC state. Each ECC state is implemented using a *generic state structure* presented below:

① Execute all algorithms associated with the ECC state.
② Set output events associated with the state.
③ Update output event-data associations.
④ Reset input events so they can be set to new values in the next reaction.
⑤ End of tick.
⑥ Reset output events so they can be set new values in this reaction.
⑦ Sample input from the environment/network and update input event-data associations.
⑧ Test state transitions and goto the respective state.
⑨ If no transitions are activated goto ④ to retry in the next tick.

Fig.6b presents a mapping of each ECC in the `ConveyorSensor` basic function block to the PRET-C code. The PRET-C code for lines 5 to 10 are elaborated in Fig.6b using the same color codings. The numbers in Fig.6b reflect the code to the steps of the *generic state structure*. For example, in the `Start` state all steps except ② and ⑨ are implemented because the state does not emit any output and the state has a default transition which always activates.

The update inputs function (i.e., `#_ConveyorSensor_UpdateInputs`) and update outputs function (i.e., `#_ConveyorSensor_UpdateOutputs`) is presented in Fig. 7 lines 1 to 15 and lines 16 to 21 respectively. The update function is split into two parts. The first part (lines 3 to 6) is to sample events and data from the network/environment. The code for this part is specific to each function block instance because each function block instance is connected to different inputs. The example shown in the figure (highlighted in green) is specific for the `PickUpSensor` function block. The `PickUpSensor` reads the event and data from the robotic arm interface.

The second part (lines 9 to 14) is to update the value of the input data port variable (i.e., `sensorArray`) with the value of the input buffer variable (i.e., `_sensorArray`) according to the event-data association. For the `ConveyorSensor` function block the `sensorArray` is associated with the `clk` event. Therefore, the value from the input buffer is only copied to the input data port variable when `clk` is evaluated to *true*.

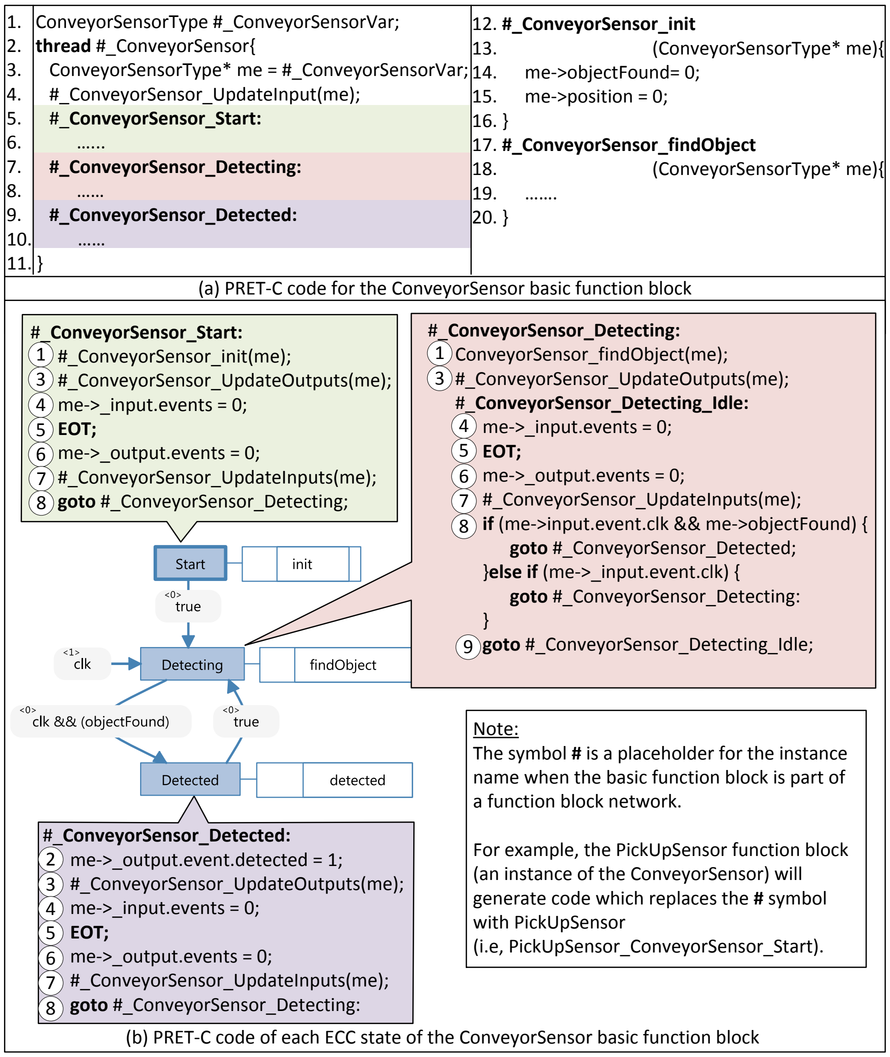
The update output function is responsible for updating the output data port according to the event-data association. Therefore, the code is similar to the second part of the update input function. For example, lines 16 to 21 only update the `position` variable when the `detected` event is present.

Lines 12-20: ECC algorithms are implemented directly using C functions.

3.2.2 Composite function blocks

The mapping of composite function blocks are also composed into two components: the function block structure and the function block body. The key difference compared to code for basic function blocks is twofold. One, the composite function block encapsulates other function blocks. Two, the behavior of the composite function block is to invoke the encapsulated blocks instead of executing an ECC. Fig.8 presents an example of the PRET-C code for the `PickUpConveyorSystem` composite function block. Fig.8a depicts the composite function block. It is composed of the `PickUpSensor` function block and the `PickUpConveyor` function block. The `PickUpSensor` is responsible for detecting objects on the conveyor belt. The `PickUpConveyor` is responsible for the control logic for the conveyor. Fig.8b depicts the function block structure. The numbers on the figure show the mapping between the function block elements in Fig.8a to the code. The code for the event and data ports (i.e., (1) to (4)) follows the same process as the previously presented basic function blocks. In addition to the event and data ports, the structure of the composite function block also contain variables for each encapsulated block. For example, the `PickUpConveyorSystem` function block encapsulates an instance of the `ConveyorSensor` function block called the `PickUpSensor` therefore, the structure in Fig. 8b contains the variable `PickUpSensor`.

Similar to basic function blocks, each composite function block is also implemented using PRET-C threads. Fig.8c depicts the function block body for the `PickUpConveyorSystem` composite function block. The behavior of the composite function block is implemented using the PRET-C *PAR* statement. Each thread of the encapsulated function blocks is spawned using the *PAR* statement.



```
ConveyorSensorType #_ConveyorSensorVar;
thread #_ConveyorSensor{
  ConveyorSensorType* me = #_ConveyorSensorVar;
  #_ConveyorSensor_UpdateInput(me);
  #_ConveyorSensor_Start:
      ......
  #_ConveyorSensor_Detecting:
      ......
  #_ConveyorSensor_Detected:
      ......
}
```

```
#_ConveyorSensor_init
                          (ConveyorSensorType* me){
    me->objectFound= 0;
    me->position = 0;
}
#_ConveyorSensor_findObject
                          (ConveyorSensorType* me){
    .......
}
```

(a) PRET-C code for the ConveyorSensor basic function block

```
#_ConveyorSensor_Start:
(1) #_ConveyorSensor_init(me);
(3) #_ConveyorSensor_UpdateOutputs(me);
(4) me->_input.events = 0;
(5) EOT;
(6) me->_output.events = 0;
(7) #_ConveyorSensor_UpdateInputs(me);
(8) goto #_ConveyorSensor_Detecting;
```

```
#_ConveyorSensor_Detecting:
(1) ConveyorSensor_findObject(me);
(3) #_ConveyorSensor_UpdateOutputs(me);
    #_ConveyorSensor_Detecting_Idle:
    (4) me->_input.events = 0;
    (5) EOT;
    (6) me->_output.events = 0;
    (7) #_ConveyorSensor_UpdateInputs(me);
    (8) if (me->input.event.clk && me->objectFound) {
            goto #_ConveyorSensor_Detected;
        }else if (me->_input.event.clk) {
            goto #_ConveyorSensor_Detecting:
        }
    (9) goto #_ConveyorSensor_Detecting_Idle;
```

```
#_ConveyorSensor_Detected:
(2) me->_output.event.detected = 1;
(3) #_ConveyorSensor_UpdateOutputs(me);
(4) me->_input.events = 0;
(5) EOT;
(6) me->_output.events = 0;
(7) #_ConveyorSensor_UpdateInputs(me);
(8) goto #_ConveyorSensor_Detecting:
```

Note:
The symbol **#** is a placeholder for the instance name when the basic function block is part of a function block network.

For example, the PickUpSensor function block (an instance of the ConveyorSensor) will generate code which replaces the **#** symbol with PickUpSensor (i.e, PickUpSensor_ConveyorSensor_Start).

(b) PRET-C code of each ECC state of the ConveyorSensor basic function block

Figure 6: Mapping of `ConveyorPickUp` basic function block ECC to PRET-C code

```
void #_ConveyorSensor_UpdateInputs(ConveyorSensorType* me) {
        //Sample data from enviroment or network interfaces
        RoboticArm->PickUpSensor._input.event.clk =
            RoboticArm->_input.event.clk;
        int i;
        for (i = 0; i < 64; i++)
                RoboticArm->PickUpSensor._sensorArray[i] =
                    RoboticArm->_pickUpArray[i];
        }
        //Update input data ports when the assoiciated input event is
            present
        if (me->_input.event.clk) {
                int i;
                for (i = 0; i < 64; i++) {
                        me->sensorArray[i] = me->_sensorArray[i]
                }
        }
}
void #_ConveyorSensor_UpdateOutputs(ConveyorSensorType* me) {
        //update output data port when the assoiciated output event is
            present
        if (me->_output.event.detected) {
                me->_position = me->position;
        }
}
```

Figure 7: Update input and output functions for the PickUpSensor basic function block

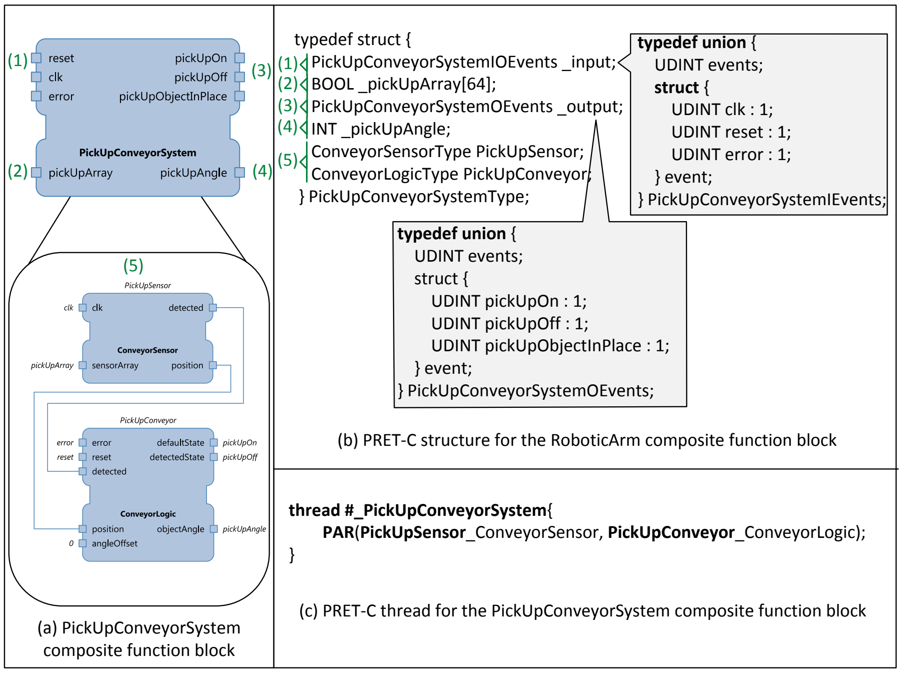
The order of thread declared in the *PAR* statement follows the ordered syncrhonous semantic of PRET-C. The ordered syncrhonous semantics of PRET-C has shown to be both reactive and deterministic by Andalam in [1].

References

[1] S. Andalam, P. S. Roop, A. Girault, and C. Traulsen, “Predictable framework for safety-critical embedded systems,” *IEEE Transactions on Computers*, vol. 99, no. 1, pp. 100 – 114, Feb. 2013.

[2] F. Boussinot, “Reactive c: An extension of c to program reactive systems,” *Softw. Pract. Exper.*, vol. 21, no. 4, pp. 401–428, Apr. 1991. [Online]. Available: http://dx.doi.org/10.1002/spe.4380210406

[3] G. Berry and G. Gonthier, “The Esterel Synchronous Programming Language: design, semantics and implementation,” *Science of Computer Programming*, vol. 19, no. 2, pp. 87–152, 1992.



(a) PickUpConveyorSystem composite function block

```
typedef struct {
(1) PickUpConveyorSystemIOEvents _input;
(2) BOOL _pickUpArray[64];
(3) PickUpConveyorSystemOEvents _output;
(4) INT _pickUpAngle;
(5) ConveyorSensorType PickUpSensor;
    ConveyorLogicType PickUpConveyor;
} PickUpConveyorSystemType;
```

```
typedef union {
  UDINT events;
  struct {
    UDINT clk : 1;
    UDINT reset : 1;
    UDINT error : 1;
  } event;
} PickUpConveyorSystemIEvents;
```

```
typedef union {
  UDINT events;
  struct {
    UDINT pickUpOn : 1;
    UDINT pickUpOff : 1;
    UDINT pickUpObjectInPlace : 1;
  } event;
} PickUpConveyorSystemOEvents;
```

(b) PRET-C structure for the RoboticArm composite function block

```
thread #_PickUpConveyorSystem{
      PAR(PickUpSensor_ConveyorSensor, PickUpConveyor_ConveyorLogic);
}
```

(c) PRET-C thread for the PickUpConveyorSystem composite function block

Figure 8: Mapping of the `PickUpConveyorSystem` composite function block to PRET-C code