A Formal Perspective on IEC 61499 Execution Control Chart Semantics

Per Lindgren¹ Marcus Lindner¹ David Pereira² Luís Miguel Pinho²

¹Luleå University of Technology
Email:{per.lindgren, marcus.lindner}@ltu.se

²CISTER / INESC TEC, ISEP
Email: {dmrpe, lmp}@isep.ipp.pt

Presentation at ETFA 4DIAC workshop September 9th 2015
1 Background
   - IEC 61499
   - Formal methods

2 Well-Formed Execution Control Charts
   - Notation
   - ECC Execution Semantics
   - ECC liveness conditions

3 Coq Formalization
   - Coq
   - Definitions
   - Extraction
   - Integration

4 Conclusion / Future Work
   - Conclusion
   - Future Work
What is IEC 61499?

- A model for loosely coupled distributed systems.
- Component Based (Function Blocks)
- Asynchronous Events with Event/Data association.
- Function Block *networks* mapped to *resources*.
- *Resources* mapped to *devices*.

Does your model implement the intended behavior?

Two sides of the problem:

1. **Model-level verification**
   - Well-formedness (soundness)
   - Intended behavior

2. **Tool-chain verification**
   - Analysis, e.g. well-formedness check
   - Compilation & Deployment
   - Run-time systems & Networking

*Verification needs a formal underpinning!"
Contributions

Our contributions in short:

- Semantics of IEC 61499 (sub-set) formalized in Coq
- Well-formedness criterion for scheduling progression
- Graph-based methods for static (compile-time) analysis
- Methods implemented in Coq (not yet proven)
- A prototype implementation based on *extracted* code
Related to the original problem:

1. **Model-level verification**
   - Well-formedness (w.r.t scheduling progression)
   - Intended behavior

2. **Tool-chain verification**
   - Analysis (w.r.t scheduling progression)
   - Compilation & Deployment
   - Run-time systems & Networking

*We provide a formal underpinning for verification*
Function Block Interface:

**Events** Input and output events,

**Variables** Input, output, and local variables, and

**With** Association between events and data.
Function Block Types:

**BFB** Basic Function Blocks
used to specify general behavior,

**SIFB** Service Interface Function Blocks
used to interface the environment of a FB network,

**CFB** Composite Function Blocks
composition of BFBs/SIFBs and (inner) CFBs
mapped as a single element for deployment, and

**SUB** Sub-application
composition of BFBs/SIFBs/CFBs and (inner) SUBs
each inner element mapped separately.
Execution Control Chart (ECC) for Basic Function Block

- Used to specify *stateful* behaviour,
- Each *state* may be associated to a sequence of *actions*. An *action* is defined by:
  - An (optional) algorithm
  - An (optional) output event
The standard gives an informal specification of the IEC 61499 semantics. In literature we find numerous approaches to formalization, including:


Different methods to verification:

- **Model checking**
  - Define (some) property of the model
  - (+) Automatic checking
  - (-) May lead to state explosion
  - (-) May need to re-check whole model, even on subtle change

- **Deductive reasoning**
  - Define (some) property of the model &
    - *prove* obligation(s)/goal(s)
  - (-) Manual or Semi-automatic
  - (+) Once proven, holds forever!
  - (+) Re-use of lemmas
  - (+) Tools may allow for extraction of *certified* code
Tools for Deductive reasoning

- **Coq** (INRIA) is a theorem-based proof assistant:
  - Definitions are given in a typed $\lambda$-calculus that features:
    - polymorphism,
    - dependent types and
    - very expressive (co-)inductive types
  - Proofs are done *semi-automatic* (through applying tactics)
  - Proofs are *automatically* checked

- **why3** (INRIA) is an extension to Hoare logic:
  - derives proof obligations from pre- and post-conditions
  - interfaces to (1st order logic) *automatic* provers, e.g. Alt-Ergo, CVC3/CVC4, Spass, Z3, etc.
  - can also export definitions and goals to Coq
    (in case automatic methods does not succeed)
# Table of Contents

1. Background
   - IEC 61499
   - Formal methods

2. Well-Formed Execution Control Charts
   - Notation
   - ECC Execution Semantics
   - ECC liveness conditions

3. Coq Formalization
   - Coq
   - Definitions
   - Extraction
   - Integration

4. Conclusion / Future Work
   - Conclusion
   - Future Work
The ECC specification is defined as a graph:

\[ ECC \triangleq \langle Q, T \rangle, \]

where \( Q \) is a finite set of ECC states \( q \in Q \), and \( T \) is the finite set of arcs or transitions \( t \in T \).

A transition \( t \in T \) is defined as the triple

\[ t = \langle q_s, c, q_d \rangle, \]

where \( q_s \) and \( q_d \), the source/destination state, and \( c \) a Boolean guard condition encoded via the functional signature,

\[ c : e_i \times D_i \times D_o \times D_l \rightarrow \text{Bool}, \]

where \( e_i \in E_i \).
BFB-states $s \in S$ are quadruples of the form $\langle d_i, d_o, d_l, q \rangle$. The initial state in more detail,

$$S^0 \triangleq \langle d^0_i, d^0_o, d^0_l, q^0 \rangle,$$

where $d^0_i \in D_i$, $d^0_o \in D_o$, and $d^0_l \in D_l$ are input, output, and local data variables, respectively, and $q^0$ defines the initial $ECC$ state.
The standard defines the "ECC operation state machine":

- **s0** Idle (initial) state,
- **s1** evaluate transitions,
- **s2** execute actions,
- **t1** on event *sample* data,
- **t3** on guard expression *true* cross transition,
- **t4** on all actions executed, and
- **t2** on all guard expressions *false*

**Figure:** ECC\(_{\text{ex}}\) state machine behavior
ECC Execution Semantics

Quoting the standard:

1. ...the resource shall ensure that no more than one input event occurs at any given instant in time . . . ;

2. ...Algorithm execution in a basic function block shall consist of the execution of a finite sequence of operations . . . ;

3. ...If state s1 was entered via t1, only transition conditions associated with the current input event, or transition conditions with no event associations, shall be evaluated. If state s1 was entered via t4, only transition conditions with no event associations shall be evaluated . . . .

Figure: ECC$_{ex}$ state machine behavior

Per Lindgren, Marcus Lindner, David Pereira, Luís Miguel Pinho

A Formal Perspective on IEC 61499 Execution Control Chart Semantics
Liveness is a common property to all well-formed models, and specifies that at some point *progression* is ensured.

In our case we define liveness by *scheduling progression*.

We discriminate between:

- *well-formed* models that ensures scheduling progression
- *ill-formed* models that do not ensure scheduling progression

Key observation:

Only on transition $t_1$ (from $s_0$) new events are received.
How to ensure progression?

\( ECC_{ex} \) must (eventually) reach state \( s_0 \) to accept a new event:

- On \( ECC_{ex} \) invocation \( s_0 \xrightarrow{t_1} s_1 \) is taken.
- The transition conditions \( s_1 \) of ECC state \( q_n \) lead either to:
  - transition \( s_1 \xrightarrow{t_2} s_0 \) and consequent liveness, or
  - transition \( s_1 \xrightarrow{t_3} s_2 \) and action execution
- statement 2 (finite sequence of operations), ensures termination of \( s_2 \), thus:
  - checking that \( s_1 \xrightarrow{t_2} s_0 \) is eventually taken is a sufficient and necessary liveness criterion, seen as a function:

\[
\forall q_n, e, ECC_{ex}(ECC, q_n, e) \xrightarrow{\star} s_0,
\]

where \( ECC \) is the ECC graph, \( q_n \) any state and \( e \) any event.
Theorem (Necessary and Sufficient Liveness Condition)

If each edge in the ECC is crossed a bound number of times, then \( s_1 \xrightarrow{t_2} s_0 \) will eventually be taken.

Ensuring this is generally hard! It involves proving termination condition \( t_2 \) under arbitrary algorithms (and their side effects to local variables \( d_l \) and output variables \( d_o \)).
Theorem (Sufficient Liveness Condition)

If each edge in the ECC is crossed at most one time, then \( s_1 \xrightarrow{t_2} s_0 \) will eventually be taken.

Limits expressivity, (we do not allow arbitrary loops in the ECC)

However:

The IEC 61499 standard stipulates, statement 3:

... If state \( s_1 \) was entered via \( t_1 \), only transition conditions associated with the current input event, or transition conditions with no event associations, shall be evaluated. If state \( s_1 \) was entered via \( t_4 \), only transition conditions with no event associations shall be evaluated (...).
Sufficient Liveness Condition

We can now formulate a sufficient (safe) condition:

- Let \( ev(t) : T \rightarrow \text{Bool} \) be a mapping from a transition \( t \) to \( true \) if the corresponding guard condition from the respective ECC holds an event dependency.

- Let the function \( SCC(ECC) \) result in the set of strongly connected components (sub-graphs) of the ECC.

The following generalization is possible:

\[
\forall scc \in SCC(ECC), \exists t \in scc, ev(t) = true,
\]

i.e., each cyclic path must have at least one edge for which the guard involves an event (i.e., \( ev(t) \) holds).
Example: Well-formed ECC (1/2)

Well-formed ECC ($ECC_{wf}$):
Green arrow indicate a transition $t$, where $ev(t) = true$. 

**Figure:** $ECC_{wf}$. 
Example: Well-formed ECC (2/2)

Well-formed ECC (\(ECC_{wf}\)):
Green arrow indicate a transition \(t\), where \(ev(t) = true\).

Figure: ECC\(_{wf}\).
Example: Ill-formed ECC

Ill-formed ECC ($ECC_{ill}$):
Red cycles indicate an ill-formed transition chain.

Figure: $ECC_{ill}$. 

Figure: Sub-graph paths of $ECC_{ill}$.
From graph theory, it is known that for any directed graph, the set of *maximal* SCC can be derived in linear time.

A *maximal* SCC may have inner SCCs, thus we need to enumerate and check $v_i \xrightarrow{*} v_j$ and $v_j \xrightarrow{*} v_i$, $(v_i, v_j \in scc)$.

However (related) the enumeration of *minimal* SCCs, is known to be NP complete.

We can turn the problem into a pre-processing alternate by applying $ev(t)$ to the ECC prior to deriving the corresponding SCCs. Let us define, as follows:

$$ECC^{pre} = ECC \setminus \{t \in ECC \mid ev(t) = true\}$$

Well-formedness can now be formulated as the following set emptiness check:

$$SCC(ECC^{pre}) = \{\emptyset\}$$
Example: Pre-processing of well-formed ECC

The example $SCC(ECC_{\text{pre}}^\text{wf}) = \{\emptyset\}$, i.e., $ECC_{\text{pre}}^\text{wf}$ has no strongly connected components (cycles).

(a) $ECC_{\text{wf}}$

(b) $ECC_{\text{pre}}^\text{wf}$

(c) $SCC(ECC_{\text{pre}}^\text{wf}) = \emptyset$
The example $SCC(ECC_{ill}^{pre}) \neq \{\emptyset\}$, i.e., $ECC_{ill}$ has a strongly connected component (cycle).
## Table of Contents

1. **Background**
   - IEC 61499
   - Formal methods

2. **Well-Formed Execution Control Charts**
   - Notation
   - ECC Execution Semantics
   - ECC liveness conditions

3. **Coq Formalization**
   - Coq
   - Definitions
   - Extraction
   - Integration

4. **Conclusion / Future Work**
   - Conclusion
   - Future Work
Computational definitions can be *proven* and *extracted* to *certified* functional code.

- Realistic sized programs: CompCert C
- However it is not easy (CompCert C > 10 years)
- Our work, just a proof of concept ....
Coq Definitions: guard

The Basic Function Block (BFB) notations can be captured by record types and plain definitions in Coq.

As an example, the definition of the transition guard expression.

```
1 Definition nodeId_t := nat.
2 Definition eventId_t := nat.
3
4 Record guard_t := mkGuard {
5   onEvent : option eventId_t;
6   onExp : bool
7 }.
```

Listing 1: Coq definitions (excerpt).

This is a simplification, considering boolean guard expression:

```
onExp : d_i → d_l → d_o → bool
```
The *computational* evaluation function **clear** takes an event **eid** and a guard expression **guard** and evaluates to \((\text{true}|\text{false})\).

```coq
1 Definition guard_target_t := prod guard_t nodeId_t.
2 Definition edge_t := prod nodeId_t guard_target_t.
3 Definition node_t := list action_t.
4 Definition nodes_t := list (prod nodeId_t node_t).
5 Definition edges_t := list edge_t.

(* Checks if guard expression is true *)
6 Definition clear (eid:eventId_t) (guard:guard_t) :=
   let cEvent :=
      match onEvent guard with
      | None ⇒ true
      | Some eid' ⇒ beq_nat eid eid' (* beq_nat is equality on nat *)
      end in
   cEvent && (onExp guard).
```

Listing 2: Coq definitions (excerpt).
And the complete well-formedness check...

1  Definition well (edges:edges_t) (n:nat) :=
2  (* remove edges with event conditions *)
3  let pre_edges := filter no_edge edges in
4
5  (* get the set of edge sources (nodes) *)
6  let (pre_ids,_) := split pre_edges in
7
8  (* compute cycles, None is no cycle *)
9  let pre_cycle :=
10     map (ecc_cyclic pre_edges n nil) pre_ids in
11
12  (* check so all sources are free of cycles *)
13  forallb (isNone (list nat)) pre_cycle.

Listing 3: Well formedness check
The process of extracting executable code from Coq definitions consists in discarding all the logical contents and translating the computational definitions into the language of OCaml.

In order to facilitate integration, the Coq types bool, list, prod are set to syntactically match the corresponding OCaml counterparts.

1. Extract Inductive bool ⇒ "bool" ["true" "false"].
2. Extract Inductive list ⇒ "list" ["[]" "(::)"].
3. Extract Inductive prod ⇒ "(*)" ["(,)"].
4. Extraction "Well.ml" well.
A (prototype) IEC 61499 tool was developed, re-using OCaml code from our earlier work on the RTFM-core compiler. Conversions between OCaml types and Coq generated types are easily defined as sketched below:

```ocaml
(* to nat (Coq representation) *)
let rec int_to_nat = function
| 0 -> Well.O
| n -> Well.S (int_to_nat (n - 1))

(* to int (OCaml representation) *)
let rec nat_to_int = function
| Well.O -> 0
| Well.S n -> 1 + (nat_to_int n)

(* to nat (Coq representation) *)
let ecc_to_nat ec = ...

(* to int (OCaml representation) *)
let ecc_to_int ecc = ...
```
Table of Contents

1 Background
   • IEC 61499
   • Formal methods

2 Well-Formed Execution Control Charts
   • Notation
   • ECC Execution Semantics
   • ECC liveness conditions

3 Coq Formalization
   • Coq
   • Definitions
   • Extraction
   • Integration

4 Conclusion / Future Work
   • Conclusion
   • Future Work
A formalization of IEC 61499 (subset) in Coq

*Liveness* defined in terms of ECC scheduling *progress*

A necessary and sufficient condition is defined
  - Complex (and may not be what you want)

A sufficient (stronger) condition is defined
  - Simple and useful

Graph theoretical solution (SCC)
  - Requires inner SCC *enumeration* (NP complete)

Addressed by pre-processing
  - *Linear* complexity (DFS)

Encoded in Coq and extracted to OCaml, integrated in the RTFM-4FUN, proof of concept tool
Future Work

- Proof of semantics, rendering fully certified code (for now only proof of algorithm termination)
- We are looking into why3 as a (simpler) alternative to Coq
- Extend well-formedness conditions to FB networks
- Formalize a real-time semantics for IEC 61499
- Ultimately certified
  - compilers and tools for IEC 61499
  - run-time systems for IEC 61499
  - ... your code here ...
Coq, Basics

- Grounded in *Calculus of Inductive Constructions* (CIC) a typed λ-calculus that features:
  - polymorphism,
  - dependent types and
  - very expressive (co-)inductive types.
- Curry-Horward’s isomorphism *programs-as-proofs* (CHi)
  In CHi, any typing relation $t : A$ can either be seen as a value $t$ of type $A$, or as $t$ being a proof of the proposition $A$.
- Any type in Coq is in the set of sorts $S = \{\text{Prop}\} \cup \{\text{Type}(i) \mid i \in \mathbb{N}\}$. The $\text{Type}(0)$ sort represents computational types, while the $\text{Prop}$ type represents logical propositions.
- Computational types can be *extracted* to functional programs → *certified programs*. 
An inductive type is introduced by a collection of constructors, each with its own arity.

A value of an inductive type is a composition of such constructors.

As an example, natural numbers are encoded as follows:

Example (nat: inductive definition of natural numbers)

```coq
Inductive nat : Type :=
  | O : nat
  | S : nat → nat.
```
Inductive Types in Coq 2(2)

- Coq automatically generates induction and recursion principles for each new inductive type.
- In Coq, functions must be provably terminating, e.g., recursive calls on structurally smaller arguments. As an example, consider the function `plus` that adds two natural numbers.

Example (plus: adds two natural numbers)

```coq
Fixpoint plus(n m:nat){struct n}:nat :=
match n with
| O  ⇒ m
| S p ⇒ S (plus p m)
end.
```

Per Lindgren, Marcus Lindner, David Pereira, Luís Miguel Pinho