A Formal Perspective on IEC 61499  
Execution Control Chart Semantics

Per Lindgren and Marcus Lindner  
Luleå University of Technology  
Email: \{per.lindgren, marcus.lindner\}@ltu.se

David Pereira and Luís Miguel Pinho  
CISTER / INESC TEC, ISEP  
Email: \{dmrpe, lmp\}@isep.ipp.pt

Abstract—The IEC 61499 standard proposes an event driven execution model for distributed control applications for which an informal execution semantics is provided. Consequently, run-time implementations are not rigorously described and therefore their behavior relies on the interpretation made by the tool provider. In this paper, as a step towards a formal semantics, we focus on the Execution Control Chart semantics, which is fundamental to the dynamic behavior of Basic Function Block elements. In particular, we develop a well-formedness criterion that ensures a finite number of Execution Control Chart transitions for each triggering event. We also describe the first step towards the mechanization of the well-formedness checking algorithm in the Coq proof-assistant so that, ultimately, we are able to show, once and for all, that this algorithm is effectively correct with respect to our proposed execution semantics. The algorithm is extractable from the mechanization in a correct-by-construction way, and can be directly incorporated in certified toolchain for analysis, compilation and execution of IEC 61499 models. As a proof of concept a prototype tool RTFM-4FUN has been developed. It performs well-formedness checks on Basic Function Blocks using the extracted algorithm’s code.

I. INTRODUCTION

The IEC 61499 standard offers an event driven execution model for distributed control applications. In the standard, the execution semantics is informally described. Thus, the behavior of run-time implementation emerges from the specific interpretations of the execution semantic underlying the tool chain at hand. As a consequence, correctness can only be argued from a deployment perspective, and not at the model level, with adverse implications to portability, interoperability and re-use of IEC 61499 models.

The standard was first established 2005 and later refined in 2012 [1]. The second edition had the aim of addressing ambiguities documented in, e.g., [2], [3], [4]. However, even in the second edition not all issues have yet been resolved, as is indicated by [5], [6], [7].

Already in [8], the need for formal methods to verify IEC 61499 models was identified. Classical criteria for correctness involves liveness (something good eventually happens, i.e., progression), safety (something bad must not happen), and celerity (time related properties) [9]. Two major approaches can be identified [8] for the purposes of verification: theorem-proving (TP) and model-checking (MC).

So far, most of the attention and research efforts have been put into MC, like in the cases of [8], [10], mainly to take advantage of the fact that MC is an automatic proof method. In the case of TP, essentially no work exists regarding the formalization of IEC 61499 related topics [11]. The main reason for that is the theoretical limits of the automation with TP in general and, although technologies like satisfiability and satisfiability modulo have advanced enormously in the past decades and currently automate many functional related properties, they are not able to provide the needed automation for extra functional concerns like timing, which is standard in industrial automation domain.

The power of proof assistants (also known as interactive theorem provers) like Coq [12] and Isabelle [13] cannot be neglected as they can bring significant advantages for verification efforts like the one that is the focus of this work. Within these systems, theorems stating fundamental properties of IEC 61449 can be formulated directly in a language specifically tailored for encoding mathematical statements, and these languages are clearly much more powerful than the usual families of temporal logics that are the base of MC tools. Examples of the power of considering proof assistant in rigorous software development can be seen in certified compilers [14] and certified micro-kernels [15]. Furthermore, in a clear contrast with MC based approaches, the use of proof assistants like Coq usually allows automatic extraction of code from a formal development. Ultimately, from proven definitions, this results in certified code that can be integrated into tools for compilation, analysis and execution, such as a certified toolchain for IEC 61499 models.

In this paper, we demonstrate how a subset of the IEC 61499 standard, namely the Execution Control Charts (ECCs), can be formalized in the Coq proof-assistant. As an example, we study a liveness condition for its execution, defined as the ability for ECC scheduling to progress. To this end, we develop a constructive definition for well-formedness that implies scheduling progression, i.e. liveness. Moreover, we discuss the proof obligations that have to be formulated and discharged in Coq in order to obtain a certified extraction. Additionally, as a proof of concept we describe the RTFM-4FUN development, and show that the extracted code can be straightforwardly integrated into the prototype tool. Finally, open problems and directions for future work are discussed.

II. BACKGROUND

A. IEC 61499

The IEC 61499 standard [16] provides a non-deterministic executable model for distributed control systems in terms of interacting function blocks. The execution semantics is informally defined, and thus subject to interpretation (as no
official reference implementation is present). For the purpose of the presented work, we briefly summarize key features of the standard. For a comprehensive overview see e.g., [17].

1) Design Elements: Function Block (FB) types:

- Basic Function Blocks (BFBs), used to specify general behavior,
- Service Interface Function Blocks (SIFBs), used to interface the environment of a FB network, and
- Composite Function Blocks, emerging from a composition of BFBs/SIFBs and (inner) composite FBs.

In common all FB types provide an interface defining input events and associated input data connections (ports in the following), and output events and associated output ports. An abstract view of the operation (input/output sequence) can optionally be defined as a Service Sequence (compliant to the ISO TR 8509 and ISO/IEC 10731:1994 standards).

The operation of a BFB is defined (in a finite state machine like manner) by its Execution Control Chart (ECC), algorithms, input/output events, and input/output/local variables.

The operation and implementation of SIFBs are left undefined in the standard. CFBs provide a hierarchical abstraction and carry no functionality in their own right [18].

2) Deployment: For the deployment, the application is partitioned onto a set of resources, which in turn are mapped onto a set of devices. Communication crossing resource boundaries (e.g., inter device communication) must pass through SIFBs.

The standard also associates other properties to the notions of resources and devices, allowing shielded execution and dynamic re-configuration. This is however outside the scope of this presentation, and hence not further mentioned.

3) Execution model: The execution model is event driven (asynchronous), where each event may be associated with a set data connections (as defined in the function block interface). A device may provide one (or more) resource(s) responsible for the scheduling of events. The order of event delivery and execution is undefined, hence the execution model is non-deterministic. However the standard stipulates that the resource (scheduler) presents a single event at each time for the execution of the receiving FB.

B. Coq

The Coq proof assistant is an implementation of the Calculus of Inductive Constructions (CIC) [19], a typed \( \lambda \)-calculus that features polymorphism, dependent types and very expressive (co-)inductive types. Coq provides users with the means to define data-structures and functions, as in standard functional languages, and also allows to define specifications and to build proofs in the same language, if we consider the underlying \( \lambda \)-calculus as a higher-order logic under the Curry-Howard’s isomorphism programs-as-proofs principle (CHI) [20]. In CHI, any typing relation \( t : A \) can either be seen as a value of type \( A \), or as \( t \) being a proof of the proposition \( A \). Any type in Coq is in the set of sorts \( S = \{ Prop \} \cup \{ Type(i) \ i \in \mathbb{N} \} \). The \( Type(0) \) sort represents computational types, while the \( Prop \) type represents logical propositions.

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:

\[
\text{Inductive nat : Type :=}
\begin{align*}
& | 0 : \text{nat} \\
& | S : \text{nat} \to \text{nat}.
\end{align*}
\]

Coq automatically generates induction and recursion principles for each new inductive type. More complex type families can be defined by combining inductive constructions and dependent types in Coq. We now introduce the subset types since they are used further ahead in this paper.

In Coq, functions must be provably terminating. Termination is ensured by a guard predicate that checks that recursive calls are always performed on structurally smaller arguments. As an example, consider the function plus that adds two natural numbers.

\[
\text{Fixpoint plus}(n m : \text{nat}) \text{struct n} \text{nat} :=
\begin{align*}
& \text{match n with} \\
& \text{match m with} \\
& | 0 \Rightarrow m \\
& | S p \Rightarrow S (\text{plus p m})
\end{align*}
\]

The basic way of the Coq proof construction process is to explicitly build CIC terms. However, proofs can be built more conveniently and interactively in a backward fashion (from the initial goal term to be proven). This step by step process is done by the use of proof tactics.

Another appealing feature of Coq is the possibility to extract the constructive parts of proof development into correct by construction functional programs. Since the underlying logic of Coq is constructive, any value, proof included, can be seen as a (functional) program. The extraction mechanism keeps the computational counterparts and translate them into standard functional programs. On the other hand, purely logical sub-terms are discarded since they are computationally non-informative.

III. Basic Function Block Semantics

In this section we formally describe BFBs and define their execution semantics. Furthermore, we non-ambiguously establish the notion of liveness based on progress as established by the principles of ECC scheduling. The notations and definitions that we present here are partially inspired by the works of [4] and [21].

A. Basic Function Block Notation

Let \( S \) be a non-empty, finite set of BFB-states. A BFB is a tuple \( b \triangleq (fbi, ecc, A, s^0) \) where \( fbi \) is a function block interface, \( ecc \) is a specification of an ECC, \( A \) is a set of algorithms, and \( s^0 \in S \) is the BFB’s initial state. The interface \( fbi \) is defined as

\[
fbi \triangleq (E_i, E_o, D_i, D_o, D_l, W_i, W_o),
\]

such that \( E_i \) and \( E_o \) are, respectively, sets of input and output events, \( D_i, D_o, \) and \( D_l \), respectively, the sets of input, output, and local variables, and \( W_i \) and \( W_o \) are associations from input and output events to input and output variables.
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 $t \in T$. Each ECC state $q \in Q$ is a structure $q \triangleq k_1, k_2, \ldots, k_n$, where the $k_i$ are the ECC’s state actions. Each such $k_i$ is defined as a pair $k \triangleq \langle a, e_0 \rangle$ such that $a \in A$ and $e_0 \in E_o$. Either the algorithm $a$ or the output event $e_0$ can be omitted, but never both simultaneously (as this would be a redundant action). A transition $t \in T$ is defined as the triple $t = \langle q_s, e, q_d \rangle$, where $q_s$ and $q_d$ is the source/destination state, and $e$ 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 $e_i \in E_i$. Any guard condition may be dependent on the occurrence of a single input event, solely on data, or may be simply the constant mapping to the value true \in \text{Bool}.

**Example**: Let us study a BFB (partially) defined by;

$$
\begin{align*}
\text{bfb} &= \langle f\text{bi}, ecc, \_, s^0 \rangle \\
\text{fbi} &= \langle \{a, b, d, e, f\}, \_, \_, \_, \_ \rangle \\
s^0 &= \langle \_, \_, \_, A \rangle
\end{align*}
$$

Figure 1. depicts the example ecc, with the states $A, \ldots, F$. For the example, actions and data variables are irrelevant (thus omitted), while Boolean guard expressions range either over single events $a, \ldots, f$ or are set true. The ecc has the initial state $q^0 = A$, with the edges (transitions) to ECC states $B$, and $E$ guarded by (input) events $b, e$ respectively.

### B. Sequential execution

The IEC 61499 standard defines the execution semantics according to Figure 2, where the underlying finite automata (ECC operation state machine) $ECC_{ex}$ is depicted, with initial state $s0$. The standard stipulates that:

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 (…).

![Fig. 2. ECC_{ex} state machine behavior.](image)

A consensus interpretation is the so-called sequential execution model, where iterative ECC traversals should reach a fixed-point by terminating in the state $s0$, before receiving the next event. This implies that actions are executed synchronously (run-to-completion) in $s2$ before $t4$ is taken to $s1$, where further transition conditions are evaluated. As a consequence, an input event is considered consumed when transitioning from s1, independently of whether the event was actually part of the transition condition (guard expression) evaluation or not.

### IV. ECC liveness conditions

The sequential execution model, discussed in the previous section, does not provide meaningful results for all possible ECC models. Hence it makes sense to distinguish between well-formed models – which have a meaningful and well defined behavior – and ill-formed models – which lacks meaning and/or have an undefined behavior. Liveness is a common property to all well-formed models, and specifies that at some point progression is ensured. Note that, an IEC 61499 system may still produce outputs without actually progressing. Thus, lack of progression may be hard or even impossible to detect by mere observation of the system’s output. To this end, concluding liveness of a model at design or compile time is of key importance. (We will return to this problem, in Section IV-D.)
A. Liveness and event scheduling progression

In order for event scheduling to progress, the corresponding \(ECC_{ex}\) must (eventually) reach state \(s_0\) to accept a new event\(^1\).

We can see \(ECC_{ex}\) as a function for executing an \(ECC\), parametrized by the triggering event \(e\) and the current \(ECC\) state \(q\), resulting in a new \(ECC\) state. The \(ECC_{ex}\) automaton depicted in Figure 2 defines such intended behavior. Conceptually, \(ECC_{ex}\) has the initial state \(s_0\) on invocation and terminates when reaching \(s_0\). From this perspective, liveness can be seen as the termination of \(ECC_{ex}\).

On invocation \(t_1\) is taken and the \(ECC_{ex}\) state \(s_1\) is reached. The transition conditions of \(ECC\) state \(q_n\), which lead either to following along \(t_2\) – transition to \(s_0\) and consequent termination – or taking \(t_3\) to reach a new \(ECC\) state \(q_{n+1}\) and also transition to \(ECC_{ex}\) state \(s_2\). Given that the computation made on the \(ECC\) must be terminating – the IEC 61499 standard states that the sequence of actions should be finite and each algorithm should amount to a finite sequence of operations, (Section III-B, 2) – then the evaluations performed when in state \(s_2\) do terminate and \(t_4\) will eventually be taken and \(ECC_{ex}\) state \(s_1\) reached once again. From \(s_1\), the transitions conditions of \(ECC\) state \(q_{n+1}\) are evaluated, and so forth in a transitive manner, until no transition condition is found true and the \(ECC_{ex}\) state \(s_0\) reached. From this we can define liveness as:

\[\forall e \in E_1, s \in S, ECC_{ex}(e,s) \rightarrow s_0. \quad (4)\]

1) Strongly Connected Components and Subgraphs:

In order to study termination, we turn to graph theory basics. Given a directed graph \(G = (V, E)\), where \(V\) is the set of vertices and \(E\) is a set of edges, a Strongly Connected Component (SCC) in \(G\) is a maximal set of vertices \(v_{sec} \subseteq V\) such that for any pair of vertices \(v_i\) and \(v_j\) \((v_i, v_j \in v_{sec})\) there exists a (directed) path between \(v_i\) and \(v_j\), and also from \(v_j\) to \(v_i\). In our model, however, we exclude the trivial SCC formed by a single state (node). A SCC may contain (inner) Strongly Connected Subgraphs (SCSs).

Example: We revisit our example ecc, Figure 1. Without considering the transition guards, we find the following set of strongly connected subgraphs, \{\{B,C,D\}, \{B,D\}, \{E,F\}\}, (Figure 3). Focusing in on the first subgraph \((a), scs_a = \{B,C,D\}\), we find the (cyclic) path \(B \rightarrow C \rightarrow D \rightarrow B\). Hence, (without considering the transition guards) an invocation of \(ECC_{ex}(\_,\ldots, q)\), with \(q \in scs_a\), would lead to non-termination (and consequentially an undefined behavior). However, taken the guard \(b\) into consideration for the edge \(D \rightarrow B\), we derive the following terminating traces:

1) \(ECC_{ex}(b,\ldots, D)\), starting from \(ECC\) state \(D\), and visiting \(B\) (by matching the event \(b\)), and the states \(C\) and finally terminating in state \(D\) (where we no longer match the event, Section III-B, 3).

2) \(ECC_{ex}(b',\ldots, D)\), starting from \(ECC\) state \(D\), triggered by the event \(b'\) such that \(b' \neq b\), thus directly terminating in state \(D\).

3) \(ECC_{ex}(\_,\ldots, B)\), starting from \(ECC\) state \(B\), visiting state \(C\), and finally terminating in state \(D\).

4) \(ECC_{ex}(\_,\ldots, C)\), starting from \(ECC\) state \(C\), and terminating in state \(D\).

B. Sufficient condition for liveness

As seen in previous Section, termination can be observed for specific assignments. 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, and let the function \(SCS(ECC)\) result in the set of strongly connected subgraphs of \(ECC\) (seen as a graph, with edges for each transition condition). The following generalization is possible:

\[\forall scs \in SCC(ECC), \exists t \in scs, ev(t) = true \quad (5)\]

, i.e., each strongly connected directed subgraph (\(scs\) derived from the \(ECC\)) must have at least one edge \(e\) for which the guard involves an event (\(ev(t)\) holds). This is a safe simplification, assuming the Boolean guard expression on the input/output and local variables always evaluates to true and hence does not contribute to termination. The implication of Boolean guard expressions are further discussed in Section IV-D.

Example, well-formed: Figure 3 depicts all the SCSs of our example \(ECC\). In the example, \(ev(t)\) holds for \{\(D \rightarrow B, B \rightarrow D, D \rightarrow b, B, E \rightarrow F\}\, and satisfies \((a), (b), \) and \((c)\) respectively, hence collectively giving well.

![](image)

Fig. 3. Strongly connected subgraphs of the example \(ECC\).

Example, ill-formed: The ECC example eccill, (Figure 4), amounts to the same set of SCSs, (Figure 3). However in this case the edge \(D \rightarrow B\) does not depend on the event \(b\), hence the there will be no \(ev(t)\) that holds for the SCS \{\(B, C, D\)\}, thus we can conclude ill-formedness.

C. Alternative formulation

From graph theory, it is known that for any directed graph the set of maximal SCCs can be derived in linear time [22], [23], with recent related Coq developments [24], [25]. Since a maximal SCC (\(scc\)) may have inner SCCs, there may be many paths where \(v_i\) to \(v_j\) and \(v_j\) to \(v_i\), \((v_i, v_j \in scc)\), and hence being ill-formed).

In the case of inner SCSs we have to visit each path (of the SCC) and ensure well-formedness for each one. In the extreme,
SCC(ECC) = Q(ECC), (where Q(ECC) is the set of nodes in ECC), we have gained nothing by computing the SCS. (This is not an unlikely situation, but rather the common case/ECC design pattern, where each node is reachable at any time.)

The path problem at hand is related to the complete enumeration of SCSs. An efficient implementation proposed by Johnson has the complexity $O((n + e)(c + 1))$ and space bounded by $O(n + e)$, where there are $n$ vertices, $e$ edges and $c$ elementary circuits in the graph [26]. The number of circuits (strongly connected subgraphs) is (worst case) exponential in $n$, since for a complete graph (with $n > 2$) every permutation will result in a cycle.

Even though it can be noted that the number of ECC vertices may in practice allow for an enumerative approach, we seek an alternative formulation. Intuitively, we can see Equation 5 as being a post-processing of the SCSs under the mapping $ev(t)$. 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:

$$ECC^{pre} = ECC \setminus \{ t \in ECC \mid ev(t) = true \} \quad (6)$$

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

$$SCC(ECC^{pre}) = \emptyset \quad (7)$$

**Example:** Figure 5 shows the $ecc^{pre}$. For the example $SCC(ecc^{pre}) = \emptyset$, i.e., $ecc^{pre}$ has no strongly connected components (cycles).

**Example:** Figure 4, gives the $ecc_{ill}$ (differing to $ecc$ only by having no event condition on $D \rightarrow B$). Figure 6 (a) shows the $ecc_{ill}^{pre}$, derived from $ecc_{ill}$, and (b) shows $SCC(ecc_{ill}^{pre})$ (with the component $\{B, C, D\}$).

D. **Strength**

The given well-formedness condition is sufficient, to guarantee liveness, however it is not a necessary condition.

In the case when algorithms (mutating the data variables) together with the guarded transitions (formulated over the data variables) allows $ECC_{ex}$ to terminate, the scheduling will progress. By design, the presented well-formedness criterion does not take variables into account for the analysis, but the approach could be extended to a weaker (possibly) necessary condition.

A weaker condition however is more complex to formulate, and considerably harder to check at compile time, since it would involve the logic of the algorithms (which is elegantly avoided by the presented strong condition).

Alternative, the termination criterion could be used in a generic form for run-time liveness verification. (However, at the point of fault detection, the system is already ill behaved, and the detected live-lock must be gracefully handled by the system implementation.)

In any case, it can be argued that presented (strong) condition is beneficial to the robustness of the design, since it rejects complex and potentially error prone recurring and iterative ECC patterns. (At this point you may ask yourself wether programming loops by means of variable updates by associated actions seems like good engineering practice?)
V. COQ DEFINITIONS

The BFB denotation from Section III-A can be captured in a straightforward manner by record types and plain definitions in Coq. As an example, Listing 1 shows the record definition of the transition guard expression (line 4) and the constructive evaluation function clear (line 15), that given an event eid and a guard expression guard evaluates to (true|false).

```coq
Definition nodeId_t := nat.
Definition eventId_t := nat.
Record guard_t := mkGuard {
  guard : option guard_t;
  nodeId : nodeId_t;
}.
Fixpoint ecc_ex (s : State_t) : option State_t :=
  match s with
  | (S n', (el, oe)) =>
    match oe with
    | None => None
    | Some s =>
      let n := beq_nat id eid
      | None => None (* not found, return illegal state *)
      | true =>
        let edge_l := filter (from_edge (current s)) edges in
        let edge_t := map (@snd nat guard_target_t) edge_l in
        let edge_l := filter (from_edge (current s)) edges in
        let edge_t := map (@snd nat guard_target_t) edge_l in
        eccentricity to
        end.
  | nil => None (* not deep enough *)
  | Some s =>
    let n := beq_nat id eid
    | None => None (* not found, return illegal state *)
    | true =>
      let edge_l := filter (from_edge (current s)) edges in
      let edge_t := map (@snd nat guard_target_t) edge_l in
      let edge_l := filter (from_edge (current s)) edges in
      let edge_t := map (@snd nat guard_target_t) edge_l in
      eccentricity to
      end.
end.
```

Example: The Listing 2, shows our two running examples eccEx11, as described in Figures 1 and 4 respectively.

```coq
Definition A := 1.
Definition B := 2.
Definition C := 3.
Definition D := 4.
Definition E := 5.
Definition F := 6.

(* Well-formed example *)
Definition my_edges : edges_t :=
  (A, (mkGuard (Some B) true, B)) ;
  (A, (mkGuard (Some E) true, E)) ;
  (B, (mkGuard (Some D) true, D)) ;
  (B, (mkGuard None true, C)) ;
  (C, (mkGuard None true, D)) ;
  (D, (mkGuard None true, B)) ;
  (E, (mkGuard None true, F)) ;
  (F, (mkGuard None true, E)) ;
  nil.

(* I'll-formed example *)
Definition my_edges2 : edges_t :=
  (A, (mkGuard (Some B) true, B)) ;
  (A, (mkGuard (Some E) true, E)) ;
  (B, (mkGuard (Some D) true, D)) ;
  (B, (mkGuard None true, C)) ;
  (C, (mkGuard None true, D)) ;
  (D, (mkGuard None true, B)) ;
  (E, (mkGuard None true, F)) ;
  (F, (mkGuard None true, E)) ;
  nil.
```

Listing 2. Examples ecc and eccEx11 in Coq.

A. ECC execution and cyclic check

In order to define the well-formedness criteria, we first have to define the semantics for ECC execution (ECC_ex in Listing 3). Coq provides a large set of pre-defined datatypes such as lists and pairs, as well as associated functions to make computations over these types. Functions like filtering elements from a list (filter), mapping a given function over all the elements of a list (map), and the projections to obtain the first and second elements of a pair (respectively fst and snd) are examples of such rich set of definitions.

The from_edge function is an helper Boolean function that returns true if an edge source is equal to a given identified id. The ecc_ex function returns an element of the partial type option edge_t, meaning that the return values can be either None, indicating that execution is ongoing after n computation steps, or Some s, where s is the terminating ECC state.

Secondly, we define a cyclic check, essentially computing a maximal SCC by a depth first search (DFS) (Listing 4). Given a starting state eid, returns either the value None – indicating the non-existence of cycles – or Some stack – where stack is a value of type list nodeId_t with the strongly connected nodes, or the current path after n computation steps.

Finally, we can now give a constructive definition well, as presented in Listing 5. The helper function ev returns true if an edge is dependent on an event, while isNone returns true for None (and false otherwise). The former is used to filter out (pre-process) edges with event guards pre_edges, while the latter is used to check that the pre_cycle is indeed free of cycles.
let rec int_to_nat = function
| 0 -> Well.O
| n -> Well.S (int_to_nat (n -1))

let rec nat_to_int = function
| Well.O -> 0
| Well.S n -> 1 + (nat_to_int n)

(* to nat (Coq representation) *)
let ecc_to_int ecc = ...
In this paper, we have showed how a subset of the IEC 61499 standard, the Execution Control Charts (ECCs), can be formalized in the Coq proof assistant. The ECC semantics are fundamental to the dynamic behavior of the Basic Function Block elements, and chosen as an illustrative example to showcase the potential of formal mechanizations within theorem proving tools.

In particular, we study a liveness aspect for ECC execution, defined as the ability for ECC scheduling to progress. To this end, we develop a constructive definition for well-formedness that implies scheduling progression, i.e., liveness. While being constructively defined in Coq, corresponding code (correct by construction) can be automatically extracted. As a proof of concept, we demonstrate how the extracted code can be integrated in the prototype RTFM-4FUN development. Moreover, we have presented and discussed the proof obligations that allow for extraction of certified code. Although the presented work is applied in the context of the IEC 61499 standard, liveness as a well-formedness criterion can be studied for other languages used in the area of industrial control.

In terms of future work, we will seek to complete the proofs, and in this way obtain a truly certified well-formedness check procedure for ECC scheduling progression. Following the same approach, well-formedness can be extended to involve the progress of scheduling (liveness) for the composition of Function Blocks. Another topic of interest is to further study the formalization of the run-time system, in order to establish correctness criteria which may ultimately lead to the construction of a certified run-time system implementation.

ACKNOWLEDGEMENTS

This work was partially supported by National Funds through FCT (Portuguese Foundation for Science and Technology), and the EU ARTEMIS JU funding, within project ARTEMIS/0001/2013, JU grant nr. 621429 (EMC2) and VINNOVA (Swedish Governmental Agency for Innovation Systems) and Svenska Kraftnät (Swedish national grid).

REFERENCES