CISTER
Research Centre in
Real-Time & Embedded
Computing Systems

Conference Paper

Contract Based Verification of IEC 61499

David Pereira*
Luis Miguel Pinho*
Per Lindgren
Marcus Lindner

*CISTER Research Centre

CISTER-TR-161106

2016/07/18

Contract Based Verification of IEC 61499

David Pereira*, Luis Miguel Pinho*, Per Lindgren, Marcus Lindner

*CISTER Research Centre

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail: dmrpe@isep.ipp.pt, lmp@isep.ipp.pt, per.lindgren@ltu.se

http://www.cister.isep.ipp.pt

Abstract

The IEC 61499 standard proposes an event drivenexecution model for component based (in terms of FunctionBlocks), distributed industrial automation applications. However,the standard provides only an informal execution semantics, thusin consequence behavior and correctness relies on the designdecisions made by the tool vendor. In this paper we presentthe formalization of a subset of the IEC 61499 standard inorder to provide an underpinning for the static verification ofFunction Block models by means of deductive reasoning. Specifically,we contribute by addressing verification at the component,algorithm, and ECC levels. From Function Block descriptions, enrichedwith formal contracts, we show that correctness of componentcompositions, as well as functional and transitional behaviorcan be ensured. Feasibility of the approach is demonstrated bymanually encoding a set of representative use-cases in WhyML,for which the verification conditions are automatically derived(through the Why3 platform) and discharged (using automaticSMT-based solvers). Furthermore, we discuss opportunities andchallenges towards deriving certified executables for IEC 61499models.

Contract Based Verification of IEC 61499

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 component based (in terms of Function Blocks), distributed industrial automation applications. However, the standard provides only an informal execution semantics, thus in consequence behavior and correctness relies on the design decisions made by the tool vendor. In this paper we present the formalization of a subset of the IEC 61499 standard in order to provide an underpinning for the static verification of Function Block models by means of deductive reasoning. Specifically, we contribute by addressing verification at the component, algorithm, and ECC levels. From Function Block descriptions, enriched with formal contracts, we show that correctness of component compositions, as well as functional and transitional behavior can be ensured. Feasibility of the approach is demonstrated by manually encoding a set of representative use-cases in WhyML, for which the verification conditions are automatically derived (through the Why3 platform) and discharged (using automatic SMT-based solvers). Furthermore, we discuss opportunities and challenges towards deriving certified executables for IEC 61499 models.**

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. In consequence, the run-time behavior emerges from the specific interpretations of the execution semantics underlying the tool chain at hand. In consequence, correctness can only be argued from a deployment perspective, and not at the model level, with adversative implications to portability, inter-operability and re-use of IEC 61499 models. The standard was first established 2005 and later refined in 2012 [7] with the aim of addressing ambiguities documented in, e.g., [5], [10]. However, issues still remain as is indicated by [16].

Already in [19], the need for formal methods to verify IEC 61499 models was identified. One way to address the issue of the correctness of IEC 61449 models is via static verification. If applied in the early stages of the design process inconsistencies of the specifications involved may be revealed, thus tedious re-iterations involving implementation, testing and debugging can be avoided. Furthermore, a static verification approach that embodies on the principles of *design by contract* [15] facilitates modular and compositional verification and enable an incremental design process allowing safe re-use of specifications (and accompanying implementations) in different settings. This becomes specially appealing for component based models such as IEC 61499. Despite the aforementioned advantages, design by contract based methods and tools have not yet reached the mainstream of industrial software development.

In this paper we outline and advocate an approach along the lines of deductive reasoning with potential to be automated down to a single click solution for verification the composition of Function Blocks that individually satisfy their contracts. We also discuss and demonstrate advanced features regarding Function Block state preservation, state updates and transitional properties, which goes beyond the currently available, state-of-the-art automatic verification methods.

Our approach considers an encoding of the involved IEC 61449 components into the WhyML language of the Why3 program verification framework [4]. The behavior of the original IEC 61499 Function Block is translated into WhyML and enriched with the necessary contracts in the form of pre- and post-conditions, loop variants/invariants, among other program logic constructs rooted in Hoare logic [12], and adopted a posteriori as the base for the design-by-contract paradigm of Meyers [15]. Henceforth, given a IEC 61499 Function block already translated into WhyML, the Why3 platform is capable of generating all the logical verification conditions necessary to ensure the correctness of the Function Block with respect to its specification. Moreover, the Why3 platform provides excellent support for both automatic and interactive theorem provers and therefore allows to discharge the verification conditions in an automatic or assisted way. Our approach is proof enabling and therefore opens up for development of certified implementations. Also, by using a deductive verification approach via Why3, our approach serves as a complement to the currently more adopted technology of model checking, which normally requires higher level models that results from abstracting of the system's concrete behavior.

I think to this end we could state that formal verification can both reduce the time / effort to test based verification, and lead to improved reliability, reusability etc.

II. Background

A. IEC 61499

The IEC 61499 standard [13] 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., [6].

1) Design Elements: In IEC 61499, all applications are built from *Function Blocks* (FBs). There are three types of FBs: *Basic Function Blocks* (BFBs), used to specify general behavior; *Service Interface Function Blocks* (SIFBs), used to interface the environment of a FB network; and finally, *Composite Function Blocks* (CFBs), emerging from a composition of BFBs and/or SIFBs and inner CFBs. In common, all FB types provide an *interface* defining *input events* with associated *input*

ports (data connections), and *output events* with associated *output ports*. The operation of a BFB is defined (in a finite state machine like manner) by its Execution Control Chart (ECC), input/output events, and input/output/local variables. Each state in the ECC implies an ordered set of zero or more *actions* (*algorithms* to execute and *output* events to emit) when visited. An edge in the ECC defines a *transition condition* as either a single input event, a Boolean expression on input/output and local variables, or a combination thereof. The operation and implementation of SIFBs are left undefined in the standard. CFBs provides a hierarchical abstraction not considered in this work. 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 specification of data types in the IEC 61499 standard refers to that of IEC 61131-3 (the programming language annex), which includes basic types (Boolean, sized signed and unsigned integers with sub-ranges, etc.), fixed size strings, records and multi-dimensional arrays. Transition conditions and algorithms follow the language specification of IEC 61131-3, which includes *Structured Text* (ST) among other supported formats. The ST language is an imperative language with heritage to Pascal, and has become the de-facto choice to many industrial developments. The IEC 61499 standard does not exclude other languages for algorithm implementations. Through supporting the common data types of 61131-3, interoperability can be achieved.

The standard IEC 61499 also introduce the notions of *adapters* (for the grouping of event and data connections), *applications* and *sub-applications* (for the grouping of FBs), *resources* (for the scheduling of events), and *devices* for system deployment.

2) Function Block Execution Model: The execution model is an asynchronous, event driven model. A device may provide one or more resource(s), responsible for the scheduling of events. The order of event delivery is undefined, and therefore the execution model is non-deterministic. The IEC 61499 standard defines the ECC execution semantics according to Figure 8 (ECC_{ex}). 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 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 (. . .).*

Taking as example the ECC exhibited in Figure 8, on event delivery (t_1), the associated input data connections are *sampled* to the corresponding input data variables. For a BFB, the ECC transition conditions from the current state are checked in the order given by the occurrence in the underlying ECC representation (s_1). When a transition takes place (t_3), the actions of the target state are sequentially executed, implying potential local/output-variable updates and the generation of output events (s_2). When finished, that is when no more actions to execute, the input event is cleared (t_4), and further transition conditions (from the target state) are inspected (s_1). In this way, s_1 and s_2 are iterated until no further ECC transitions are possible (t_2), and the BFB returns to its idle state (s_0), awaiting for further events.

B. Why3

Why3 is a tool for deductive verification of programs. Why3 provides a rich language called WhyML that allows for users to simultaneously specify and program, with a clear separation between the purely logical part of the specification and the process of generating verification conditions from the actual programming code. Following this approach, Why3 also acts as a font-end to several external theorem provers – both automatic and user-assisted – that are used to discharge the verification conditions generated in a verification process. Furthermore, Why3 also enforces a notion of modular specification by providing users with the means to define theories that are reusable via *cloning* inside larger developments.

III. IEC 61499 Static Verification

In this Section we present verification approaches for a subset of the IEC 61499 standard from three viewpoints: application development by composing already available components; simple function block development, largely re-using ready made algorithms; and, finally, advanced/safety-critical function block development with focus on providing guarantees to state-full and transitional behavior. While advanced and safety-critical development usually requires experience and prior knowledge in the areas of contract-based development and formal verification, composition of FBs and re-use of algorithms can be put at the hands of the non-experts since their verification processes may be highly mechanized.

A. Component Level Verification

In this section we provide and hands-on example of our approach towards component level verification of IEC 61499 models. For that, we assume the following informal specification of a system comprised of:

- Two temperature sensors, where Sensor1 gives measures in Celsius c, in the range $0 \leq c < 10$ and Sensor2, gives measures in Fahrenheit f, in the range $32 \leq f < 50$.
- A simple safety controller that, assumes measures in Celsius c such that $0 \leq c < 10$, and that produce Bang-Bang (Boolean) output (Sensor1 $<$ Sensor2).
- A generic conversion service, converting Celsius to, and from Fahrenheit.

1) IEC 61499 System Model: A possible Function Block Network implementation for the running example is given in Figure 1. The integer data type does not consider any specific IEC 61499/6113-3 encoding. `Sensor1` and `Sensor2` are instances of a generic `FB_Sensor` type. The (service) parameter `DataType` indicates measurement type, and `Min/Max` the value range. The `FB_ConvService` is deployed to convert the output values of `Sensor2` to Celsius. The `FB_Controller` is an instance of the generic Bang-Bang

controller with `Min/Max` defining the allowed control range in Celsius.

The ECC for the `FB_ConvService` is presented in Figure 2 and invokes the corresponding conversion algorithm according to received event. The controller can be implemented in a similar fashion like is shown in Figure 3.

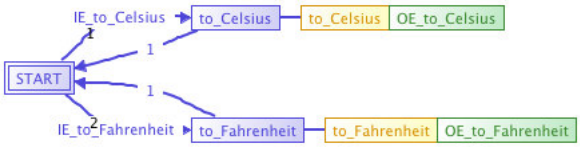

Fig. 2: `FB_ConvService` ECC.

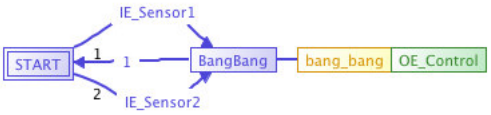

Fig. 3: `FB_Controller` ECC.

2) WhyML Verification Model: We now aim to verify, by means of formal contracts, the compositional soundness at the FB network level. Notice that here all FBs involved are essentially stateless, *i.e.*, state transition conditions and algorithms depend neither on the ECC state, nor on the local/output variables as depicted in Figure 2 and Figure 3. This allows, in this specific case and from a contract perspective, to consider output variables as functions from input variables. Section V provides a discussion on the general case.

With the outset of state independency, we can construct contracts at the BFB/SFB level in a straightforward manner. We utilize the module system of Why3, which facilitates the separation of concerns, and start by defining common data types for the components.

3) WhyML Data: Data types are defined as traditionally supported in functional programming. The `metric_t` type is an algebraic data type whose constructors define an enumeration of the supported temperature domains, while the `val_t` type declaration defined a pair where the left elements is an integer, and the right one is a value of `metric_t`. These types can now be used within the `Data` module, or imported in other models, like is exemplified in the module `Test` where we declare a function instance `test` instance of type `val_t` returning the pair made by the integer value 3 and the metric value `Fahrenheit`.

```
module Data
  use export int.Int
  use export Pair

  type metric_t =
    | Celsius
    | Fahrenheit

  type val_t = ( int, metric_t )
end
```

Notice, the module `Test` is not part of the final system model, it serves just as a mere example demonstrating the type- and module-system of Why3.

```
module Test
  use import Data

  let test () = (3, Fahrenheit)
end
```

4) WhyML ConvService: With the outset of state independency (described in Section III-A2), we can capture the ConvService function block by the `ConvService` functional specification. Notice that here the `div`, imported from `int.ComputerDivision` available in Why3's API, follows the integer division semantics of commonly implemented computer arithmetics.

```
theory ConvService
  use import int.ComputerDivision
  use export Data

  function c_to_f (c : int) : int = (div (9 * c) 5) + 32
  function f_to_c (f : int) : int = div ((f - 32) * 5) 9

  function conv (d : value_t) (f : metric_t) : value_t
  =
    let (dv, dt) = d in
    match dt, f with
      | Celsius, Fahrenheit -> ( c_to_f dv, Fahrenheit )
      | Fahrenheit, Celsius -> ( f_to_c dv, Celsius )
      | (_, _) -> d
    end

  function to_Celsius (d : value_t) : int
  = fst (conv d Celsius)

  function to_Fahrenheit (d : value_t) : int
  = fst (conv d Fahrenheit)
end
```

5) WhyML SensorGen, Sensor1 & Sensor2: We first define a generic sensor model `SensorGen`. This module is parametrized by the constants `min`, `max` and `metric` which will be later instantiated to define the concrete models for Sensor1 and Sensor2. Besides these three parameters, the module `SensorGen` also specifies a predicate that checks if a value of type `val_t` is within the expected bounds, a function `red` that is responsible for reading a `val_t` value and check that it is in range via the `in_range` predicate, and a function `range_of` that determines that calculates between the bound of the range under consideration for sensor data.

```
module SensorGen
  use export Data

  constant min    : int
  constant max    : int
  constant metric : metric_t

  type val_t

  predicate in_range (v : val_t) =
    min <= fst v < max /\ metric = snd v

  val read () : val_t
    ensures  { in_range result }

  function range_of () : int = max - min

end

module Sensor1
  use import Data

  constant s1_min    : int      = 0
  constant s1_max    : int      = 10
  constant s1_metric : metric_t = Celsius

  clone export SensorGen with
```

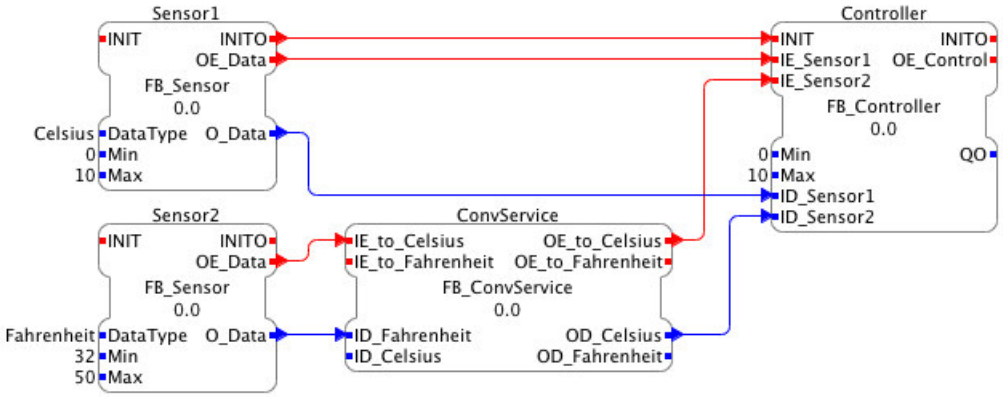

Fig. 1: Function Block Network.

```
    constant min    = s1_min,
    constant max    = s1_max,
    constant metric = s1_metric
end

module Sensor2
  use import Data

  constant s2_min    : int      = 32
  constant s2_max    : int      = 50
  constant s2_metric : metric_t = Fahrenheit

  clone export SensorGen with
    constant min    = s2_min,
    constant max    = s2_max,
    constant metric = s2_metric
end
```

6) WhyML SensorTest: Before we put the complete system together, we demonstrate how verification in Why3 can be put to use. First, we declare the WhyML modules `Sensor1` and `Senson2` that specify each of the sensors classes considered in the system. Then, in the implementation module `SensorTest`, we build sensor instances `S1` and `S2` for the specification modules `Sensor1` and `Senson2`, respectively, and also define a set of assertions that assert that both sensor instances have positive range, that assert that all readings fall within the range of respective sensor, and that assert that values of `S1` is within the range of `S2`. The verification results, using as backend the automatic theorem prover Alt-Ergo [8], is that the first three assertions pass while the forth fails (as expected) as the predicate `in_range` requires equal metrics. The Why3 verification interface and produced results is depicted in Figure 4. Notice that `SensorTest` is just an example to demonstrate the verification process and thus not part of the system model.

```
module SensorTest
  clone import Sensor1 as S1
  clone import Sensor2 as S2

  let test()
  =
    assert { S1.range_of() > 0 /\ S2.range_of() > 0};
    let s1_v = S1.read() in
    let s2_v = S2.read() in
    assert { S1.in_range s1_v};
    assert { S2.in_range s2_v};
    assert { S1.in_range s2_v};
    (s1_v, s2_v)
end
```

7) WhyML ControlGen: The generic Bang-Bang controller specification is captured by the `ControlGen` module. It is parametrized by the range predicate (`range_pred`). It requires that incoming sensor values (`s1` and `s2`) are within the defined range of the controller in order to ensure the result.

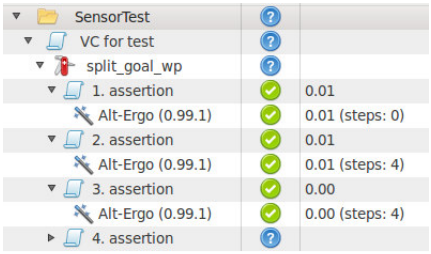

Fig. 4: Verification results in **why3**.

```
module ControlGen
  use export ConvService

  (* the range predicate is a parameter for the module *)
  predicate in_range int

  val control (s1 s2 : value_t) : bool
    requires { in_range (to_Celsius s1) }
    requires { in_range (to_Celsius s2) }
    ensures  { result = (to_Celsius s1 < to_Celsius s2) }

end
```

8) WhyML System: Finally we can put the system together, and verify compositional soundness w.r.t the defined contracts. We define a concrete range predicate and use that to instantiate the `ControlGen` model. The system orchestration passes readings from sensors `S1` and `S2` to the controller instance. The result is scrutinized by an assertion to verify our expectations.

For the given orchestration, all verification conditions hold and are discharged by Alt-Ergo prover through Why3 and therefore we can conclude the composition to be correct. Counter-examples can be straightforwardly devised, *e.g*, by limiting the control range, and/or increasing the sensor range(s). The development is available on request, and all examples will replay using Why3 without the need for any additional libraries.

```
module System
  clone import Sensor1 as S1
  clone import Sensor2 as S2

  (* should work in Celsius range 0 <= c < 10 *)
  predicate range (c:int) = 0 <= c < 10
  clone import ControlGen with predicate in_range = range

  let orchestration () =
    (* take readings from the sensors *)
    let s1_v = S1.read () in
    let s2_v = S2.read () in

    (* present readings to the controller *)
    let bang = control s1_v s2_v in

    (* make sure the controller meets our expectations *)
    assert {
      match bang with
        | True  -> to_Celsius s1_v <  to_Celsius s2_v
        | False -> to_Celsius s1_v >= to_Celsius s2_v
      end
   }
end
```

9) Summary: We have shown how contracts at the component level can be specified for the IEC 61499 standard, and we have demonstrated that the verification conditions for safe compositions can be discharged by automatic solvers. With end-users in mind (*e.g.*, plant operators and system maintainers) mechanization and ease of use are of paramount importance. To this end, the proposed approach is clearly promising, allowing the standard library of IEC 61499 and vendor specific I/O blocks to be designed by expert engineers in the field and made available as trusted, contract carrying pre-verified components, enabling safe orchestration by the end-user. As exemplified, verification goes far beyond traditional type checking of interfaces as functional and logical properties can be captured.

B. Algorithm Level Verification

In the previous section we were dealing with specifications (contracts) at the component (Function Block) level, without any consideration of underlying implementations. Here we will demonstrate how rigorous specifications (and implementations) can be formulated for BFB algorithms. Again we take the outset of a running example, capturing key aspects and demonstrate the feasibility of our proposed approach.

Let us assume an informal specification of `FB_Sort` with the corresponding interface depicted in Figure 5. On the arrival of `ISortArray`, the output integer array `oArray` should take on the sorted values (index 0 being the lowest) of the input integer array `IArray` and the associated `OSortEvent` should be triggered. Moreover, on the arrival of `ISortElem`, the `IElem` integer value should be sorted into the current `oArray`, and `OElem` should take on the overflowing integer value and the event `OSortElem` should be triggered. We may assume a fixed array length of 10, and initial values of all variables to be 0.

1) FB_Variables: In order to formalize algorithm contracts, we need a representation for FB variables. To this end, we utilize a record type declaration as defined in the module `FB_Variables`. The standard library of Why3 provides polymorphic mutable arrays of fixed length. We use the integer type without consideration of specific IEC 61499/61131-3 encoding. The fields `iElem` and `oElem` are declared as mutable,

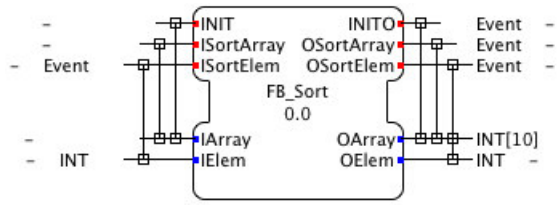

Fig. 5: `FB_Sort` function block interface.

thus allowing them to be updated. The `well_formed` predicate ensures a notion of well-formed arrays w.r.t. the length of the input and output arrays. We define the `mk_variables` contract in order to enforce proper initialization according to the FB specification.

```
module FB_Variables
  use export int.Int
  use export array.Array

  (* we model the variables of the FB as a record *)
  type variable_t = {
    (* input variables     *)
    iArray         : array int;
    mutable iElem  : int;
    (* no local variables  *)
    (* output variables    *)
    oArray         : array int;
    mutable oElem  : int;
  }

  predicate well_formed ( v : variable_t )
  = length v.iArray = length v.oArray /\
    length v.iArray > 0

  val mk_variables (n : int) : variable_t
    requires { 0 < n } (* array length *)
    ensures  {
      let v = result in
        well_formed v /\
        v.iElem = 0 /\ v.oElem = 0 /\
        forall i:int. 0 <= i < n ->
          v.iArray[i] = 0 /\ v.oArray[i] = 0
    }
end
```

2) Specification of Algorithms with Contracts: We can now formulate the specifications for `sort_array` and `sort_elem` as shown below[1].

3) Algorithm Implementations: The Why3 platform ships with a rich standard library and set of examples[2]. Among these examples we find two imperative implementations of in-place sorting of integer arrays (`InsertionSortNaive` and `InsertionSort`, the latter reducing the number of swap operations). Both implementations are certified (pre-verified) to the `sorted` predicate (defined in `array.IntArraySorted`), and can thus safely be re-used in any setting of integer array sorting.

As applied in our example, the user can at a later stage interchangeably chose in between the `FB_SortSpec` which provides a mere specification for high level verification, or `FB_SortNaive` / `FB_SortOpt` which provide refined implementations allowing the extraction of certified code.

[1] A helper module `FB_Gen` is introduced to reduce textual replication.

[2] Additionally there is a large gallery of pre-verified algorithms and data structures [20].

```
module FB_Gen
  use export array.IntArraySorted
  use export array.ArrayPermut
  use export array.ArrayEq
  use export FB_Variables

  let copy_vars (v : variable_t) : unit
    requires { well_formed v }  (* well-formedness *)
    ensures { array_eq v.iArray v.oArray }  (* eq? *)
  =
    blit v.iArray 0 v.oArray 0 (length v.iArray);
end

module FB_SortSpec
  use export FB_Gen

  predicate sort_array_pred ( v ov : variable_t ) =
    sorted v.oArray /\                (* output sorted   *)
    array_eq v.iArray ov.iArray /\    (* input preserved *)
    permut_all (v.iArray) v.oArray    (* data consistent *)

  val sort_array (v : variable_t) : unit
    requires { well_formed v }        (* well-formedness *)
    ensures  { sort_array_pred v (old v) }

  predicate sort_elem_pred  ( v ov : variable_t ) =
    sorted v.oArray /\                (* output sorted   *)
    array_eq v.iArray ov.iArray /\    (* input preserved *)

    (forall a a' : array int.
     split_last_pred a (ov.oArray) v.iElem -> (* input  *)
     split_last_pred a' (v.oArray) v.oElem -> (* output *)
     permut_all a a') /\              (* data consistent *)

    (forall i:int. 0 <= i < (length v.oArray)
      -> v.oElem >= v.oArray[i] )     (* oElem largest   *)

  val sort_elem (v : variable_t ) : unit
    requires { well_formed v }        (* well-formedness *)
    ensures  { sort_elem_pred v (old v) }

end

module FB_SortNaive
  use import FB_SortSpec
  use import InsertionSortNaive

  let sort_array (v : variable_t) : unit
    requires { well_formed v }  (* well-formedness *)
    ensures  { sort_array_pred v (old v) }
  =
    copy_vars v                   (* copy variables *)
    sort v.oArray;          (* sort output in-place *)
end

module FB_SortOpt
  use import FB_SortSpec
  use import InsertionSort

  let sort_array (v : variable_t) : unit
    requires { well_formed v }  (* well-formedness *)
    ensures  { sort_array_pred v (old v) }
  =
    copy_vars v                   (* copy variables *)
    sort v.oArray;          (* sort output in-place *)
end
```

Let us now focus on the `sort_elem` and sketch a possible implementation below. In order to use the pre-verified in-place sorting algorithms, we need to allocate a local array `a` (with values of `oArray` appended with `iElem`). To facilitate development, we define a module `ArrayAux` (not depicted here) that provides singleton arrays, appending a single element (`append_last`) and splitting out last element (`split_last`) together with accompanying predicates and lemmas. As the length of arrays is non-mutable, this implies allocation and copying.

```
module FB_ElemOpt
  use export FB_SortSpec
  use export ArrayAux
  use import InsertionSort

  (* the sort_elem algorithm *)
  let sort_elem_opt (v : variable_t) : unit
    requires { well_formed v }

    ensures  { sort_elem_pred v (old v) }
  =
    (* append iElem to current output array oArray *)
    let a = append_last v.oArray v.iElem in

    sort a; (* sort the array a *)

    (* split in (hd, t), where t is the last element *)
    let (hd, t) = split_last a in

    (* store hd, t in output oArray and eElem *)
    blit hd 0 v.oArray 0 (length hd);
    v.oElem <- t
end
```

For the verification, we introduce ghost variables (and code), which during later extraction are omitted from the generated code[3]. The complete set of verification conditions are discharged by Alt-Ergo (after splitting VCs at the top-level) within 10 seconds on an ordinary i7 based laptop, assigned 2 cores.

4) Summary: We have shown how FB variables can be declared and how contracts for algorithms (operating on the FB variables) can be specified.Moreover, we have demonstrated how implementation refinement can re-use pre-defined/verified data structures and that complex functional behavior (such as sorting) can be automatically discharged through the Why3 platform. Building on the existing Why3 library and the rich set of examples and gallery, new specifications and corresponding certified implementations can be designed for the specific application at hand, and/or stored for future re-use.

C. ECC Level Verification

In the previous Sections, we presented methods that leads us towards highly mechanized verification at the component and algorithm level of the IEC 61499 standard. In this Section we focus on advanced features considering the transitional properties of BFBs. Mastering those, the skilled engineer may provide advanced contract carrying FBs and algorithms for safe re-use.

1) ECC Execution Model: As briefly reviewed in Section II-A, the standard provides an informal execution semantics with the consequence of incompatible run-time environments due to tool dependent interpretations. Moreover, the verification becomes deployment-specific, as a consistent underpinning is lacking. The issue has been acknowledged and various formalization have been proposed. A particular problem to that end is the potential non-termination of ECC transitions. The standard informally requires termination at the algorithm level and thus may be used as an outset for well-formedness. Our proposed modeling in terms of Why3 largely helps to that end, as termination for recursion as well as loops are required in terms of well-founded and strictly decreasing `variant`'s.

However, ECC termination is not explicitly treated by the standard, and is thus only an informal assumption on any

[3]For brevity, verification code and ghost declarations are omitted in the listing.

correct implementation. However, by introducing restrictions to the execution model, the general underlying *halting problem* can be circumvented and termination granted [14]. Based on the observation that transitions involving event conditions can only be taken as the first step of a transition chain, well-formedness can be stated by the non-existence of connected components of the ECC graph after removing all edges with associated event conditions.

In this work, we relax the well-formedness condition of [14] allowing all models for which an upper bound is given to the number of times each ECC node is visited on behalf on a triggering event. While deriving the upper bound in general is an undecidable problem, we foresee restrictions based on well-founded relations to be applicable, but out of scope for this presentation.

The presented modeling and verification approach focus the internal behavior of BFBs, while modeling resource level event dispatching and data variable propagation is left for future work[4].

2) Informal System description: Assume a system consisting of: two valves, each controlled by the events `OE_Open`/`OE_Close` and sensed by the event `IE_Closed` (emitted when valve reaches its closed position), and a (safety) controller `FB_Save` with inputs `IE_Open1/2`, `IE_Close1/2`, and `IE_Closed1/2` that shall:

- close both valves on `IE_INIT`;
- ensure that valve 1 should not be opened unless valve 2 is closed; and
- valve 2 should not be opened unless valve 1 is closed.

3) System Model: As the system is purely event based, we must introduce the local (Boolean) variables `IS_CL1` and `IS_CL2` to hold the (virtual) states of valves 1 and 2 respectively. The safety controller `FB_Safe` component and corresponding ECC are depicted in Figures 6 and 7 respectively.

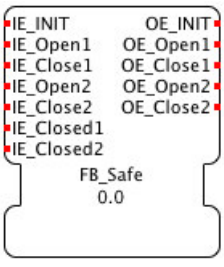

Fig. 6: `FB_Safe` function block interface.

4) Modeling Function Block State: Events and states are modeled as enums. The Function Block state is modeled as a record holding the (mutable) current state, and the (mutable) variables. A complete FB model also hold a map from events to variables, but omitted here for brevity.

[4]For the latter, the IEC 61499 standard is very vague, which calls for further investigations in order to propose a flexible yet analyzable semantics.

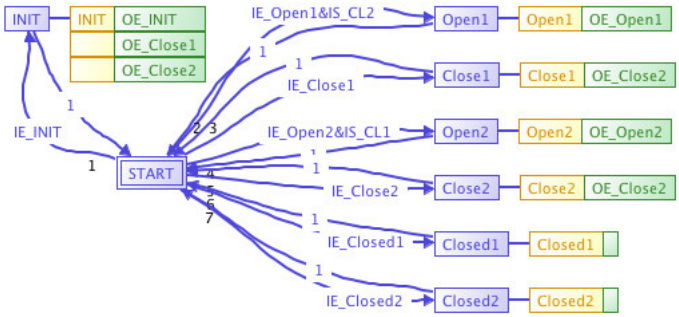

Fig. 7: `FB_Safe` ECC specification.

```
type variable_t = { (* FB variables as a record *)
  mutable is_closed1 : bool; (* local variables *)
  mutable is_closed2 : bool;
}

type i_event_t =              (* events as enums *)
  | IE_Init
  | IE_Close1  | IE_Open1 | IE_Close2 | IE_Open2
  | IE_Closed1 | IE_Closed2

type o_even_t =
  | OE_Open1 | OE_Close1 | OE_Open2 | OE_Close2

type ecc_t =               (* ECC states as enum *)
  | S_Start | S_Init
  | S_Open1 | S_Close1 | S_Open2 | S_Close2
  | S_Closed1 | S_Closed2

type fb_state_t = {      (* FB state as a record *)
  mutable ecc : ecc_t;
          v   : var_t
}

function mk_state () (* initialization contract *)
  ensures {
     result.ecc = S_Start /\
     result.v.is_closed1 = false /\
     result.v.is_closed2 = false
  }
```

5) ECC Actions: The ECC action specification, depicted in Table I, is modelled by the function `ecc_ex_action`, presented below, in a straightforward manner. The algorithms (operating on the variables `s.v`) are inlined for brevity. In a real setting algorithms may be externally defined. The resulting set of output events is presented as a potentially empty (`Nil`) list.

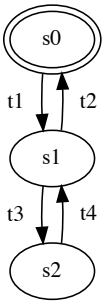

Fig. 8: ECC_{ex}.

State	Algorithm	Output Events
INIT	IS_CL1 := False; IS_CL2 := False;	OE_Close1 OE_Close2
Open1	IS_CL1 := False;	OE_Open1
Open2	IS_CL2 := False;	OE_Open2
Close1		OE_Close1
Close2		OE_Close2
Closed1	IS_CL1 := True;	OE_Close1
Closed2	IS_CL2 := True;	OE_Close2

TABLE I: ECC action specification

```
let ecc_ex_action (s : fb_state_t) : list o_event_t
    (* a *)
    ensures { mem OE_Open1 result -> s.ecc = S_Open1 }
    (* b *)
    ensures { mem OE_Open2 result -> s.ecc = S_Open2 }
  =
    match s.ecc with
      | S_Start   -> Nil
      | S_Init    -> Cons OE_Close1 (Cons OE_Close2 Nil)
      | S_Open1   -> s.v.is_closed1 <- false;
                     Cons OE_Open1 Nil
      | S_Close1  -> Cons OE_Close1 Nil
      | S_Open2   -> s.v.is_closed2 <- false;
                     Cons OE_Open2 Nil
      | S_Close2  -> Cons OE_Close2 Nil
      | S_Closed1 -> s.v.is_closed1 <- true; Nil
      | S_Closed2 -> s.v.is_closed2 <- true; Nil
    end
```

6) ECC Execution: The ECC transition specification (Figure 7), is modeled by `ecc_ex` in a straightforward manner. WhyML requires a termination condition on a strictly decreasing well-founded relation. To this end, `ls` represents the (finite) list of ECC states to visit and `ls'` the remaining states after transition (where the length of `ls'` is smaller than the one of `ls`). This provides a relaxation of the ECC termination condition proposed in [14], as ECC states may be revisited an arbitrary, yet bounded number of times on behalf of a triggering event. In the following, t_i refer to a transition, and s_j to a state of the ECC operation state machine presented in Figure 8 (ECC_{ex}), and as defined in the IEC 61499 standard.

```
let rec ecc_ex
    (ie_opt : option i_event_t)
    (s : fb_state)
    (ls : list ecc_t) : list o_event_t
  variant  { length ls } (* strictly decreasing *)
  requires { mem s.ecc ls }
= let ls' = remove s.ecc ls in
'L:
  let n_ecc_opt = match s.ecc with
    | S_Init            -> Some S_Init
    | S_Init            -> Some S_Init
    | S_Open1           -> Some S_Init
    | S_Open2           -> Some S_Init
    | S_Close1          -> Some S_Init
    | S_Close2          -> Some S_Init
    | S_Closed1         -> Some S_Init
    | S_Closed2         -> Some S_Init
    | S_Start           ->
    match ie_opt with
      | Some IE_Init    -> Some S_Init
      | Some IE_Open1   ->
        (* a' *)
        if s.v.is_closed2 then Some S_Open1 else None
      | Some IE_Open2   ->
        (* b' *)
        if s.v.is_closed1 then Some S_Open2 else None
      | Some IE_Close1  -> Some S_Close1
      | Some IE_Close2  -> Some S_Close2
      | Some IE_Closed1 -> Some S_Closed1
      | Some IE_Closed2 -> Some S_Closed2
      | _               -> None
    end
```

```
  end in
  match n_ecc_opt with
    | None       -> Nil          (* no more transitions *)
    | Some n_ecc ->              (* try transition      *)
      match mem n_ecc ls' with
        | False  -> Nil             (* ECC cycle, abort *)
        | True   ->
          s.ecc <- n_ecc;            (* take transition *)
          let oe = ecc_ex_action s in (* execute action *)
          assert { (mem OE_Open1 oe) ->             (* A *)
            (at s.v.is_closed2 'L) = true };
          assert { (mem OE_Open2 oe) ->             (* B *)
            (at s.v.is_closed1 'L) = true };
          (* iterate recursively and                    *)
          (* return the appended list of output events  *)
          oe ++ (ecc_ex None s ls')
      end
  end
```

Initially the BFB is awaiting for an event (ECC_{ex} is in the idle state s_0). The (resource) scheduler invokes `ecc_ex` with an event `ie_opt` (`Some x:i_event_t`) (t_1). ECC transition conditions are evaluated into `n_ecc_opt` (s_1). If `None` (or the termination condition `!mem n_ecc ls'` is met) no more transitions are possible and the ECC execution terminates (t_2). Else the transition is taken (t_3), and we update the ECC state `s.ecc <- n_ecc`. In (s_2) we execute the (sequence) of actions, collect the resulting output events in `oe`, and return with `oe` appended to the output events resulting from invoking `ecc_ex` recursively (t_4).

7) Static Verification: We can now statically verify the ECC according to its specification, namely that valve 1 should not be opened unless valve 2 is closed (and vice versa). This amounts to assertions (* A *) and (* B *) respectively, in the `ecc_ex`. Focusing the first case, we ensure that the occurrence of an `OE_Open1` implies that `s.v.is_closed1` was `true` at the point when action execution resulted in an `OE_Open1`. Causality is key, and we refer to the precise point of reference by the label `'L`, as defined *before* the action is executed.

However, this alone does not suffice for automatic discharging of the corresponding verification condition, as we need to establish a connection between the `ecc_ex` execution and the `ecc_ex_action` (by means of contracts). To this end, we ensure in `ecc_ex_action` (* a *) that an `OE_Open1` event implies that `s.ecc` is `S_Open1`. By deduction on (* a' *), the verification condition (* A *) can now be discharged by Why3 and we can conclude that valve 1 *will* not be opened unless valve 2 is closed. Analogously, the verification condition of (* B *) is discharged.

8) Summary: We have demonstrated how stateful and transitional behavior of FB ECC execution can be modeled in a straightforward manner. We have also shown that advanced causal properties can be mechanically discharged provided adequate contracts. By further lifting the insurance to the contract of `ecc_ex`, stateful and transitional properties may be verified for the composition of function blocks (as demonstrated in Section III-A).

IV. Towards Certified Implementations of IEC 61499

In the presented work, the encoding of IEC 61499 models in WhyML has been made by hand. It is clear that this is a tedious and error prone process, and that the connection in between the IEC 61499 models and their verification is

ad-hoc. However, we foresee that the encoding is largely mechanizable if enriching the IEC 61499 input format with contract information, following the lines of ACSL [2] and SPARK [1].

To this end, suitable representations for the IEC 61499/IEC 61131-3 data types should be defined in WhyML and deployed for automatic translation. The Why3 standard library `int` provides ranged integers, `floating_point` (according to IEEE-754), `mach.int` (arithmetics for programs, e.g., on `Int32`, `UInt32`, etc.), and so forth, which in effect embody the data types of IEC 61131-3.

At the algorithm level, the IEC 61499 endorses the IEC 61131-3 Structured Text language. With its heritage to Pascal, and relative simplicity, translation into WhyML is foreseeable. To the end of certification through extraction, one could also think of translations from a subset of WhyML into structured text. The latter would enable re-using a subset of existing pre-verified algorithms and data structures already available. To this end, the extraction process in Why3 is highly configurable through a *driver* architecture.

In order to further improve usability, the Why3 platform provides a rich programming API giving access to the internal data structures and functions of Why3. This provides ample opportunities towards tool-integration, bridging the gap in between the FB design and verification processes. To this end, we foresee to develop a verification server that establishes a bridge between the FB IDE (e.g., the open source 4DIAC tool) and the Why3 platform.

V. Related and Future Work

Formal verification of IEC 61499 has been studied mainly from the perspective of model checking, see e.g., the survey [11]. A recent approach utilizing abstract state machines and symbolic model checking is proposed in [17], allowing the verification of non-boolean conditions. Our approach differs by taking a deductive verification approach, which avoids potential state space explosion problems that are common when using model checking. Furthermore, our approach follows the lines of certified programming – allowing guarantees to the functional correctness of the implementation w.r.t its specification – and undertakes a contract based approach to compositional verification. This opens up for compositional and hierarchical verification following the lines of [9]. Initial work in this direction, for compositional verification of IEC 61499 has recently been proposed [18]. However, the possibility to express behaviour at the component level is limited to service sequences in IEC 61499, which defines the set of event sequences acceptable for the function blocks without any notions of timing, data, and state. An extension to behavioural types expressed in terms of extended regular expressions has recently been proposed in [21]. While our approach focus on static verification, the work presented in [21] in targets run-time monitoring. Another related approach to IEC 61499 verification, takes the outset of observers [3], function blocks added to the system model for the purpose of verification, allowing static reachability analysis of erroneous states through model checking.

Our deductive verification approach takes the outset of manually specified contracts at the component, algorithm and ECC levels. However, to manually specify the stateful and transitional behaviour of complex function blocks may become challenging and with limited usability. To this end, and besides the aforementioned future work (Section IV), we focus current and future work on the automatic contract generation from the IEC 61499 models at the component (function block) level. However, as mentioned, the expressiveness of behaviour specification at the component level is very week. To this end, extensions such as behavioural types [21] are promising, allowing succinct and versatile specification for the component interface.

VI. Conclusions

In this work, we have established a foundation for reasoning on a subset of IEC 61499 models by means of contracts. Specifically we have targeted verification at the component, algorithm and ECC levels. The feasibility of the approach has been demonstrated on a set of representative use-cases, for which assertions on compositional soundness, functional correctness and non-trivial safety conditions have been automatically generated and effectively discharged through the Why3 platform. The proposed method provides ample opportunities for mechanization as a majority of the presented encodings in WhyML can be straightforwardly derived from IEC 61499 models. Moreover, the design by contract approach is proof enabling and allows for the extraction of certified implementations. Future work includes further mechanization and integration to IEC 61499 tool-chains, aspects of code certification, and to establish semantics for reasoning on causal and timely properties on IEC 61499 models.

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

[1] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[2] P. Baudin, J.C. Filliâtre, T. Hubert, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI C Specification Language (preliminary design V1.2)*, preliminary edition, May 2008.

[3] Z.E. Bhatti, R. Sinha, and P.S. Roop. Observer based verification of IEC 61499 function blocks. In *9th IEEE International Conference on Industrial Informatics*, pages 609–614, July 2011.

[4] F. Bobot, J.C. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011.

[5] G. Cengic and K. Akesson. On Formal Analysis of IEC 61499 Applications, Part A: Modeling. *IEEE Transactions on Industrial Informatics*, 6(2):136–144, 2010.

[6] J. Christensen, T. Strasser, A. Valentini, V. Vyatkin, A. Zoitl, J. Chouinard, H. Mayer, and A. Kopitar. The IEC 61499 Function Block Standard: Software Tools and Runtime Platforms. In *ISA Automation Week*, 2012.

[7] International Electrotechnical Commission. International Standard IEC 61499: Function Blocks - Part 1, Architecture. Geneva, Switzerland: Int. Electrotech. Commission, 2012.

[8] S. Conchon, E. Contejean, and J. Kanig. CC(X): Efficiently combining equality and solvable theories without canonizers. In Sava Krstic and Albert Oliveras, editors, *SMT 2007: 5th International Workshop on Satisfiability Modulo*, 2007.

[9] W. Dong, Z. Chen, and J. Wang. A contract-based approach to specifying and verifying safety critical systems. *Electronic Notes in Theoretical Computer Science*, 176(2):89 – 103, 2007.

[10] V. Dubinin and V. Vyatkin. On Definition of a Formal Model for IEC 61499 Function Blocks. *EURASIP J. Embedded Syst.*, 2008:7:1–7:10, April 2008.

[11] H.M. Hanisch, M. Hirsch, D. Missal, S. Preusse, and C. Gerber. One Decade of IEC 61499 Modeling and Verification - Results and Open Issues. In *IFAC Symposium on Information Control Problems in Manufacturing*, 2009. Moscow, Russia.

[12] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[13] IEC standards. International Electrotechnical Commission, March 2014.

[14] P. Lindgren, M. Lindner, D. Pereira, and L.M. Pinho. A Formal Perspective on IEC 61499. In *IEEE International Workshop on Distributed Intelligent Automation Systems*. IEEE Computer Society Press, 2015.

[15] B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992.

[16] S. Patil, V. Dubinin, C. Pang, and V. Vyatkin. Neutralizing Semantic Ambiguities of Function Block Architecture by Modeling with ASM. In *Perspectives of System Informatics*, volume 8974 of *Lecture Notes in Computer Science*, pages 76–91. Springer Berlin Heidelberg, 2015.

[17] S. Patil, V. Dubinin, and V. Vyatkin. Formal verification of iec61499 function blocks with abstract state machines and smv – modelling. In *Trustcom/BigDataSE/ISPA, 2015 IEEE*, volume 3, pages 313–320, Aug 2015.

[18] H. Prahofer and A. Zoitl. Verification of hierarchical iec 61499 component systems with behavioral event contracts. In *11th IEEE International Conference on Industrial Informatics*, pages 578–585, 2013.

[19] C. Schnakenbourg, J.-M. Faure, and J.-J. Lesage. Towards IEC 61499 function blocks diagrams verification. In *Systems, Man and Cybernetics, 2002 IEEE International Conference on*, volume 3, pages 6 pp. vol.3–, Oct 2002.

[20] The Why3 Development Team. Why3 gallery of verified programs, May 2016. online: http://toccata.lri.fr/gallery/why3.en.html.

[21] M. Wenger, A. Zoitl, and J.O. Blech. Behavioral type-based monitoring for iec 61499. In *Emerging Technologies Factory Automation (ETFA), 2015 IEEE 20th Conference on*, pages 1–8, Sept 2015.