Compiler for the translation of Hybrid Automata to IEC-61499 for modeling and simulation of Hybrid Industrial Systems - Report

Kawsihen Elankumaran
Research Intern
Department of Electrical and Computer Engineering
The University of Auckland
Auckland, New Zealand
Email: kawsihen15@gmail.com

Abstract—**Safety critical industrial systems have been playing an indispensable role in technologies ranging from huge nuclear power plants to tiny pacemakers. The accuracy and reliability demanded by these systems necessitate accurate modeling and simulation of them before being implemented in real life. The traditional methods such as co-simulation employed in this regard have been long suffering from issues such as timing fidelity due to the lack of rigorous semantics. An excellent way to circumvent these problems is to employ the IEC-61499 standard in modeling both the plant and controller together. This provides numerous advantages with respect to performance, precision and the ability for emulation. In this regard the aim of this project is to develop a compiler that would automatically translate the models of plants and controllers formally described in Hybrid Automata to BlokIDE compliant IEC-61499 function blocks so that they can be reliably simulated for functional correctness.**

I. Introduction

Most real life distributed industrial systems comprise of the plant is continuous domain governed by ordinary differential equations and their controllers in discrete domain. A huge share of those so called Hybrid systems are safety critical in nature where errors could lead to irreversible consequences. Therefore when a new controller is to be developed for the plant it needs to be throughly tested for correctness before being implemented on the plant. Co-simulation has been often used for such purposes. In co-simulation the subsystems constituting a system are modeled and simulated in a distributed manner. But this itself turns out to be a root cause of problems when going for simulation to validate the functional correctness of the developed controller. For example, typically the controller might be modeled in IEC-61499 standard in a software environment and the plant might be modeled in Simulink. Now there is an inevitable need for these two models to communicate to exchange data and commands. Resorting to TCP has usually been the option. The inherent nature of the TCP protocol forbids the reliable use of it to simulate these critical systems. TCP transmissions for instance could be delayed by the transmitting agent as per the protocol. This is however not something that could be anticipated in a real system where signals arrive almost immediately to the controller. There are also good chances that TCP could suffer from race conditions in delivering the signals. This too adds some difficulty in validating the controller. The existing popular technologies such as the Simulink and Stateflow combination used to model the plant and the controller have been reported to have suffered from similar issues. Hence the modeling of these reactive systems require a more reliable tool.

The solution proposed to this problem by A. Malik et al. [1] is to model both the plant and the controller together in IEC-61499 standard. IEC-61499 is an internatioanl standard for the modeling of distributed industrial systems based on the concept of function blocks. The occurrence of delays and races would be impossible when both the plant and controller logically exist together. This will therefore help eliminate the drawbacks of co-simulation and model and simulate a system that closely resembles a real world system in architecture and functionality. Modeling both the plant and the controller together as in IEC-61499 has performance advantages too as illustrated in the work by A. Malik et al. [1]. The code size and execution time has been significantly lower through this approach. Therefore extending the usage of IEC-61499 standard to model the plant as well provides us with obvious advantages.

BlokIDE software is the development environment used in modeling, simulation, analysis and code generation of distributed industrial systems developed by the PRETzel research group at The University of Auckland. Although BlokIDE uses a proprietary XML schema to represent and store models, it is fully compatible with the IEC-61499 standard, allowing users to import and work with models represented in IEC-61499 XML format. Alternatively models in proprietary schema can be also imported and worked with straight. BlokIDE allows manual development of models of system in accordance with the IEC-61499 standard using the graphical interface in which case they would be stored in the proprietary format. Hybrid systems on the other hand are formally expressed as Hybrid Automata (HA). Hybrid Automaton is a mathematical model for expressing systems where digital and analog processes

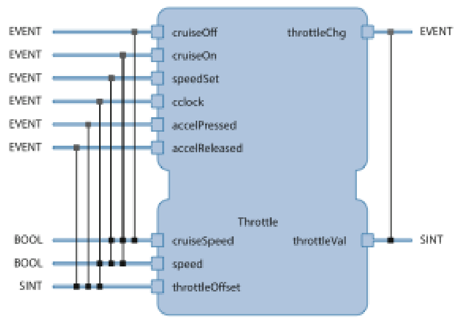

Fig. 1. Example of a Function block from [2]

interact. The HA models of these systems are initially available as Python structures that contain details about their inputs, outputs, algorithms and states. Therefore a developer willing to model, simulate and analyze a system will have to manually transform its Hybrid Automaton Python structure to the BlokIDE compatible XML format before continuing with those essential operations. This is exactly where this project gets in. In this project we implement a compiler that reads the Python structure and automatically convert them into IEC-61499 XML schema compliant XML representation which can readily be imported into BlokIDE and proceeded to simulation and analysis.

II. The IEC-61499 standard

The IEC-61499 standard is an internationally accepted standard for the development of distributed systems. The underlying principle behind the standard is the usage of function blocks as the basic components which are interconnected as networks to form larger complete systems. The three types of function blocks defined by the standard are the Basic function block, Composite function block and Service interface function block. Figure 1 shows an example of a Basic function block. The behavior of these Basic blocks are governed by the Execution control charts (ECC) which are like finite state machines. The states in the ECC are associated with algorithms that execute when the function block is in the particular state i.e. bound by its guard conditions. Transitions occur to other states when the appropriate conditions hold. These interact with other function blocks and the environment through their interface which has event and data ports along their sides. Composite function blocks are essentially a network of Basic function blocks and therefore their behavior is described by the Basic function blocks they enclose. Other higher level terms from the standard are Resources, Devices and Systems.

The book titled Model-Driven Design Using IEC 61499 by L. H. Yoong et al [2] provides a lot of information regarding the elements and the organization of the standard.

IEC-61499 XML Schema

The IEC-61499 standard XML schema is a formal way of representing function blocks and other basic elements of the standard. This software standard allows interoperability among different technologies. Since the schema for the this standard is not properly documented officially, it is worth to have an idea about its structure.

```
<FBType Comment="" Name="">
        <InterfaceList>
                <EventInputs>
                  <Event Comment="" Name=
                      ""/>
                </EventInputs>
                <EventOutputs>
                  <Event Comment="" Name=
                      "">
                    <With Var=""/>
                  </Event>
                </EventOutputs>
                <InputVars/>
                <OutputVars>
                  <VarDeclaration Comment
                      ="" Name="" Type=""/
                      >
                </OutputVars>
        </InterfaceList>
        <BasicFB>
            <ECC>
              <ECState Comment="" Name=""
                  >
                <ECAction Algorithm=""
                    Ouptput=""/>
              </ECState>
              <ECTransition Condition=""
                  Destination="" Source=""
                  />
            </ECC>
            <Algorithm Comment="" Name=""
                >
              <Other Language="" Text=""/
                  >
            </Algorithm>
        </BasicFB>
</FBType>
```

Although this is not complete, its shows how the Interface function block, ECC and Algorithms are represented in IEC-61499 XML standard. The schema is self explanatory. The two main nodes in the tree are the InterfaceList node which represents the interface of the function block in XML and the BasicFB node which encapsulates the ECC and Algorithms. It is also worth to have a glance at the standard schema presented by HOLOBLOC, Inc. on their website [3].

III. Methodology

The compiler is implemented in Python as well. It reads from the Python file containing the HA structures of the system, runs algorithms on it and writes the generated XML output to another file with a .fbt extension which can then be directly consumed by BlokIDE if needed. For the purpose

of illustration, throughout this text we'll use the Water tank element from the Water tank - Burner system on the work by A. Malik et al. [4] as an example. It should be noted that the compiler however is generic and supposed to be working on all Hybrid Automaton structures supplied on the correct format.
The process of this translation is outlined in the work by A. Malik et al. [1]. Given a Hybrid Input Output Automaton to be translated, it is first checked to be well formed, which indicates its constituent differential equations are analytically solvable. Any well formed HIOA can be transformed into IEC-61499. The first step would be to convert them to Synchronous HIOA (SHIOA), a discrete version of HIOA which is more compatible with the discrete IEC-61499 format. Next, a set of algorithms propose the translation of SHIOA to IEC-61499.

For this job of reading the Python models and transforming them into XML, the lxml XML processing library for Python [5] was used. This comparatively more efficient and rich in features than the default ElementTree class from the Python installation. Being a Python binding of a C library, this offers performance advantages too. Python 2 was used for the development of the compiler. The code was developed in a virtual environment keeping the dependencies separately.

A. Input Hybrid Automata

As noted earlier, the HIOA structures are available as Python classes. This is imported at the beginning and the algorithm works on its attributes.

B. Translation of Interface Function Blocks

The original Python HA models lacked any information regarding the Interface Function Blocks in them. However this would be crucial for the development of IEC-61499 blocks. Therefore the structures were modified to include the input and output events, variables and the associations among them. As shown in the listing below ExternalEvents class has two dictionaries as fields each for input and output event-variable associations where the keys of the dictionary are the events and the values are lists that contain the variables with which the particular event is associated. ExternalVars class has two lists containing the input and output variables as fields.

```
extEves = ExternalEvents({ Event("ON") :
    [], Event("OFF") : [] }, { Event("
    update") : [Symbol("x")]})

extVars =  ExternalVars([],[Symbol("x")])
```

Now that we have all the information, it is just a matter of directly transforming them into IEC-61499 XML. The following code does that translation.

```
INTERFACELIST = etree.SubElement(FBTYPE,
    'InterfaceList')

EVENTINPUTS = etree.SubElement(
    INTERFACELIST, 'EventInputs')

for ext_inp_eve, asso_vars in ha_model.
    rest[0].externalInputEvents.iteritems
    ():

        EVENT1 = etree.SubElement(
            EVENTINPUTS, 'Event', Name=
            ext_inp_eve.s , Comment='')

        for asso_var in asso_vars:
                WITH1 = etree.SubElement(
                    EVENT1, 'With', Var=
                    str(asso_var))

INPUTVARS = etree.SubElement(
    INTERFACELIST, 'InputVars')

for ext_inp_var in ha_model.rest[1].
    externalInputVars:

        VARDECLARATION1 = etree.
            SubElement(INPUTVARS, '
            VarDeclaration', Name=str(
            ext_inp_var), Type='BOOL',
            Comment='')
```

Shown above is the listing for the input side only. Iterating through the dictionary, the code builds the XML node for the input events and the variables associated with them. Finally it also build the separate XML node to list down the input variables alone as per the IEC-61499 standard. Note that the same procedure would be required for the output side as well for the output events and the output variables associated with them. The resulting IEC-61499 XML for the interface would be as follows.

```
<InterfaceList>
        <EventInputs>
          <Event Comment="" Name="OFF"/>
          <Event Comment="" Name="ON"/>
        </EventInputs>
        <EventOutputs>
          <Event Comment="" Name="update"
              >
            <With Var="x"/>
          </Event>
        </EventOutputs>
        <InputVars/>
        <OutputVars>
          <VarDeclaration Comment="" Name
              ="x" Type="BOOL"/>
        </OutputVars>
</InterfaceList>
```

This was tested for correctness by loading the generated file into BlokIDE which rendered the output as shown in Figure 2. This was also tested with multiple function blocks from systems such as Train-Gate and Reactor-Controller.

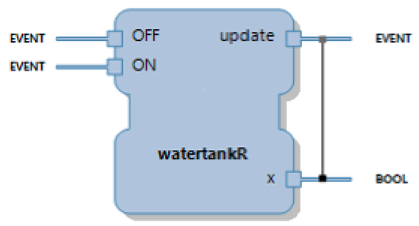

Fig. 2. Rendered Interface function block for watertank HA by BlokIDE from the generated XML

C. Translation of ECC and Algorithms

Translation of the Execution Control Chart and the Algorithms are not straight forward as the interface. The first reason is that the Ordinary Differential Equations describing the behavior of the system are in continuous domain while the IEC-61499 which is by its inception a standard for controllers is in discrete domain. This requires the differential equations to be solved and discretized in time to be used in the translated model. This is the so called SHIOA transformation outlined in that paper. The second issue is that the HIOA structures represents a Mealy state machine while ECC of the IEC-61499 standard are Moore state machines. A distinctive difference between these two is that in Mealy state machines outputs and updates occur along the transitions while in Moore state machines those are associated only with states.

The conversion to SHIOA that includes the analytical solution and discretization of the ODEs was done with the help of the Python SymPy module using the dsolve function [6]. The well formed criteria is assumed since we require analytical solutions and attempt to solve ODEs within the scope of the dsolve function.

The conversion form Mealy machine to Moore machine is accomplished by having an extra state along every transition edge among the existing states and performing the updates as algorithms on those states.

The translation process is carried on as follows according to the algorithm from the paper.

1) Iterate through the existing locations in the HA model and create the XML representation of each of them.

```
for loc in ha_model.locations:

        ECSTATE = etree.SubElement(
            ECC, 'ECState', Name=loc.
            name, Comment='',...)
        ECACTION = etree.SubElement(
            ECSTATE, 'ECAction',
            Algorithm=loc.name+'Algo'
            , Output='')
        ALGORITHM = etree.SubElement(
            BASICFB, 'Algorithm', Name
            =loc.name+'Algo', Comment=
            '')
```

2) For every of those states, extract their ODEs, solve them and discretize them in time.

```
solution_str = solution_str + str(
    dsolve(ode.expr,ode.var)).replace(
    'x(t)␣==','me->x␣=').replace('t','
    me->d*me->k').replace('C1','(float
    )␣me->C1x') + ';\n'
```

This is done using the dsolve function as previously mentioned. The replacements are just to conform the algorithm to the BlokIDE's convention of variables.

3) Attach the algorithms to that state and pin them to the XML tree at the proper place.

```
OTHER = etree.SubElement(ALGORITHM, '
    Other', Language='C' , Text=
    solution_str)
```

4) Create the starting state, add the initialization algorithm and link it to the initial state of the SHIOA.
As the algorithm of the starting state, important variable initializations essential to simulation are made such as the tick variable k to be 0 and the step variable d to be 0.01. It is very important to note that rather than depending on additional flags from the Python model to indicate the initial location of the list of locations, we assume that the first element of the list of locations to be the initial one. Therefore it is important to present the model in that format.

```
init_str = 'me->d␣=␣0.01;\nme->k=0;\n
    ␣me->x=␣' + str(first_loc.odeList
    [0].initValue) + ';'
```

5) Iterate through all the existing Edges/Transitions and for each of them create a new intermediate state in XML.

```
for edge in ha_model.edges:

        ECSTATE = etree.SubElement(
            ECC, 'ECState', Name='
            t_state'+str(i), Comment='
            ',...)

        ECACTION = etree.SubElement(
            ECSTATE, 'ECAction',
            Algorithm='t_state'+str(i)
            +'Algo' , Output='')
```

6) Extract the updates from the edge and make it the algorithm for the newly created state. Also reset the step variable k to 0 in the algorithm.
7) Create transitions in XML to and from the new state to connect everything up. As the condition for the incoming transition use the already existing condition and events from the previously exited edge. Leave the condition for the outgoing to be True always.

```
ECTRANSITION = etree.SubElement(ECC,
    'ECTransition', Source=edge.l1,
    Destination="t_state"+str(i),
    Condition=final_condition,...)

ECTRANSITION = etree.SubElement(ECC,
    'ECTransition', Source="t_state"+
    str(i), Destination=edge.l2,
    Condition="True",...)
```

8) Iterate through the original states and create XML representation for self transitions by extracting the guard conditions from the model.

```
for loc in ha_model.locations:

          for condition_list in loc.
              invariant.itervalues():
                    for condition_element
                        in condition_list
                        :
                              #string
                                  processing
                                  here

          ECTRANSITION = etree.
              SubElement(ECC, '
              ECTransition', Source=loc.
              name, Destination=loc.name
              , Condition=condition_str
              ,... )
```

This completes the main parts of the translation including the interface, execution charts and algorithms.

D. Writing compiled files

Till now, the program has been building up an XML tree according to the standard schema by the above algorithm. Now a new file of extension .fbt is created on the name of the function block and the entire XML tree is written down to that file. This file can be imported by BlokIDE by selecting 'Import IEC-61499 files' from the 'BlokIDE' menu and used on existing projects.

```
fHandle = open('destination'+ ha_model.
    name +'.fbt','w')
fHandle.write('<?xml␣version="1.0"␣
    encoding="UTF-8"?>\n')
fHandle.write(etree.tostring(FBTYPE,
    pretty_print=True))
```

The functional correctness of this algorithm was tested by loading the generated XML into BlokIDE and inspecting the output. The BlokIDE rendered Execution control chart of the watertank function block is shown in Figure 3. Note that initially the elements of the ECC would be scattered around since we initialize the x and y coordinates of them arbitrarily while generating XML. This could be easily arranged by right clicking on the rendered model and selecting a layout from the Auto Arrange option.

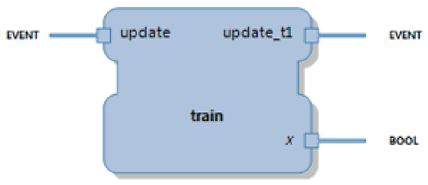

Fig. 4. Rendered Interface function block for train HA by BlokIDE from the generated XML

IV. Validation

The compiler was experimented on the translation of several HIOA models and was successful in the transformation of them to IEC-61499 compliant Basic function blocks in BlokIDE. Figure 4 shows the Interface function block of the train HA from the train-gate system as rendered by BlokIDE from the generated XML and Figure 5 shows the Execution control chart rendered the same way. Comparison of these with the previously manually constructed models reveal good consistency between the two meaning that the compiler was successful in the translation of Hybrid automata to IEC-61499 XML usable by BlokIDE.

V. Issues

There were some issues with BlokIDE which were encountered during the validation processes.

1) IEC-61499 standard XML schema has algorithms for the states inline in XML while the propitiatory schema stores these algorithms in a separate file with .algo extension and references them from the main file. When BlokIDE loads the IEC-61499 file although it picks up the algorithms and shows them on the toolbar, it doesn't create a new .algo file automatically.
2) It appears that there are some XML parsing problems with BlokIDE when it draws the ECC from the XML. For example the XML code *add1 &amp;&amp; (x &gt;=550)* is improperly parsed with unnecessary parentheses.
3) There were also problems in BlokIDE picking up the Cygwin tools. Building the models from BlokIDE seemed to throw compiler errors in generating the executable but externally compiling them using the Make utility worked and produced executables.

The algorithms for the states in IEC-61499 XML have not yet been validated by simulation due to these issues in BlokIDE in loading and compiling IEC-61499 files. This means the code producing them might need some modification to produce executables that can be loaded on a programmable hardware device.

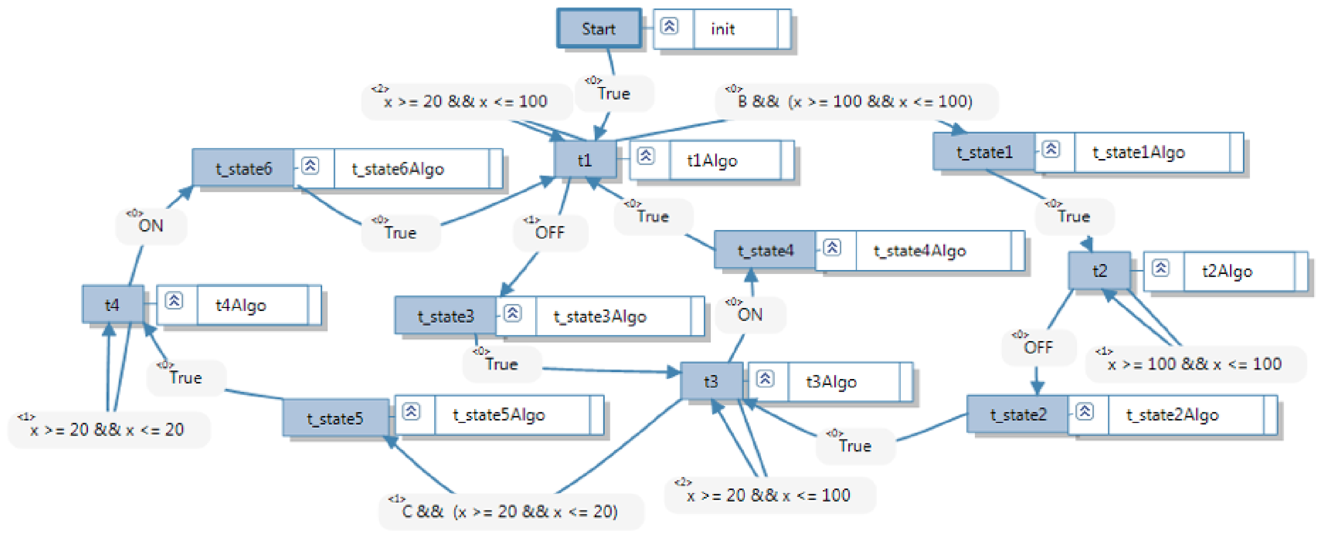

Fig. 3. Rendered Execution control chart for watertank HA by BlokIDE from the generated XML

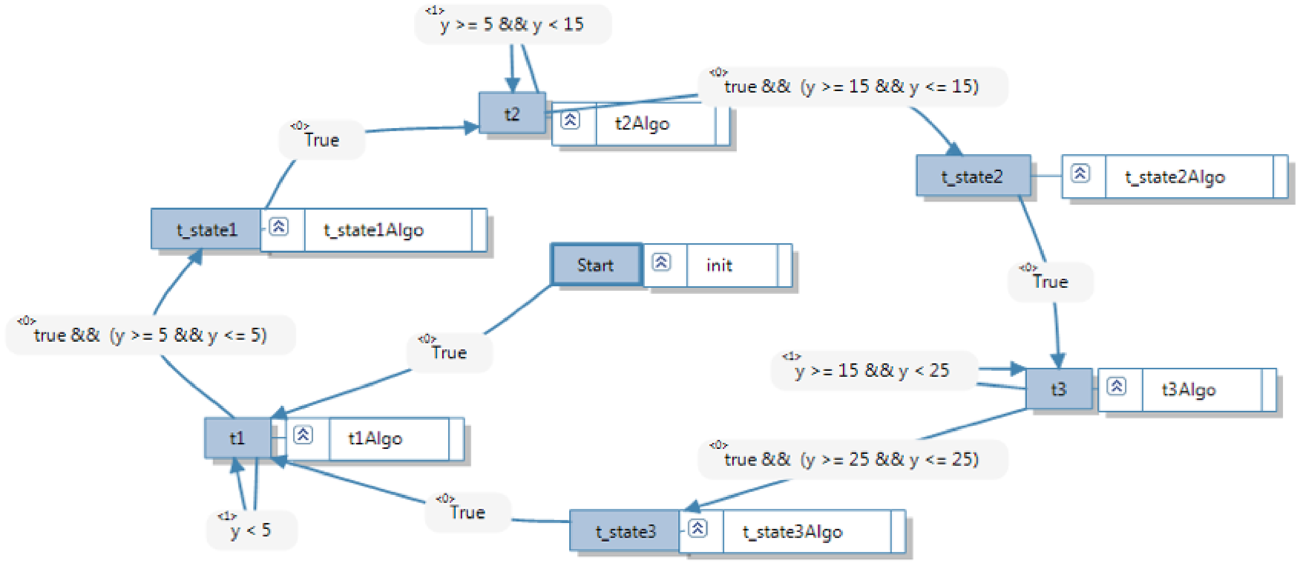

Fig. 5. Rendered Execution control chart for train HA by BlokIDE from the generated XML

VI. Conclusion

With the safety critical industrial systems playing a vital role in every field of engineering, the very nature of the tasks they are involved in demand precise testing of their functionality for their development. The traditional method of development and simulation of them such as co-simulation suffer from various drawbacks in accurately modeling the real time nature of their operation. The idea to circumvent this issue was to model both the plant and the controller in the same IEC-61499 standard. BlokIDE has been the tool used throughout the research to model and simulate the plant-controller system using IEC-61499 standard. However these hybrid systems are generally expressed as Hybrid automata models. This work focuses on the development of an automatic compiler from Hybrid automata to IEC-61499 which would eliminate manual conversion procedures.

The compiler implemented in Python reads form the models and run algorithms to transform them to IEC-61499 XML. The algorithm produces XML for the Interface function block, Execution control chart and the algorithms. The HIOA is turned into SHIOA by solving and discretizing the algorithms. Interface function blocks are created by direct translation. The subsequent process includes turning the Mealy state machine to Moore state machine by generating extra states along every transition. Self transitions are added finally. The correctness of the algorithm is verified by loading the generated XML from selected HIOA models into BlokIDE and comparing them with the manually developed ones as well and with the HIOA itself. Future works along this line could include extending the compiler to translate Composite function blocks and complete systems. For this the corresponding HA models in Python need to be supplied.

References

[1] A. Malik, P. S. Roop, and T. Steger, ”Modelling and Control of hybrid systems using IEC-61499,” Unpublished manuscript.

[2] L. H. Yoong, P.S. Roop, Z.E. Bhatti, and M.M. Kuo, *Model-Driven Design Using IEC 61499.* Springer, 2015.

[3] HOLOBLOC, Inc. Retrieved January 23, 2016, http://www.holobloc.com/xml/LibraryElement.dtd

[4] A. Malik, P. S. Roop, S. Andalam, E. Yip, and M. Trew, ”A synchronous rendering of hybrid systems for designing Plant-on-a-Chip (PoC)” in *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING.*

[5] lxml - XML and HTML with Python. Retrieved Jnuary 23, 2016, http://www.lxml.de

[6] ODE. Retrieved January 23, 2016, http://docs.sympy.org/dev/modules/solvers/ode.html

[7] HOLOBLOC, Inc. FBDK - The Function Block Development Kit. Retrieved January 23, 2016, http://www.holobloc.com/doc/fbdk/