Function Block Diagram to UPPAAL Timed Automata Transformation Based on Formal Models

Doaa Soliman*, Kleanthis Thramboulidis**, Georg Frey*

*Saarland University, Saarbrücken, Germany (e-mail: {doaa.soliman,georg.frey}@aut.unisaarland.de)
**University of Patras, Greece (e-mail: thrambo@ece.upatras.gr)

Abstract: Verification of IEC61131-3 based safety applications is a challenge in the development process of industrial systems. In this paper, we formally describe the set of transformation rules we have defined for the automatic transformation of IEC61131-3 function block based safety applications to UPPAAL timed automata models. These models are used for the verification of the safety application. Both the source and the target domain models have been formally defined and these definitions are used for the formal definition of the transformation rules. We adopted as format of the source models the PLCopen XML specification that is widely accepted by industry. Based on this format and the defined transformation rules a prototype model transformer was developed using Java. The transformer was used on several safety applications to check its functionality and the efficiency of the transformation process.

Keywords: IEC61131-3, UPPAAL, Function Block Diagram, Safety Applications, PLCopen.

1. INTRODUCTION

The IEC61131-3 (2003) set of programming languages is widely adopted by industry and the majority of industrial automation systems are based on these languages. Many of these systems are safety critical and should conform to various safety standards defined by international standard organizations, such as ISO and IEC. This is why safety issues of IEC61131 have already been examined by the research community. Moreover, PLCopen (http://plcopen.org) has developed a library of safety function blocks (SFBs) that can be used for the development of safety applications. Among the challenges that engineers face during the development process of safety applications is the verification of the safety application before implementing it. UPPAAL (http://www.uppaal.org) was selected, in this work, to be used for the verification process of function block diagram (FBD) safety application. The FBD design models of the safety application are transformed to UPPAAL models, which are next imported to the UPPAAL model checker. UPPAAL is a good choice for formal verification of systems that can be modelled as a collection of non-deterministic processes with real valued clocks. This makes the tool suitable for the verification of FBD safety applications built from function blocks (FBs) triggered by timers.

Soliman, Thramboulidis, and Frey (2011) have defined the set of mapping rules for the transformation process based on the meta-models of the source and target domains. Soliman and Frey (2009, 2011) have presented the validation via simulation and verification via model checking based on these models. In this paper, we proceed with the formal definition of the models of the two domains and express the transformation rules based on these formal models. Based on this, we have developed a prototype model transformer using the Java language and we have checked its behaviour on several safety applications. We adopted the PLCopen XML as format of the IEC61131-3 FBD safety applications. This allows the approach to be used by many IEC61131-3 commercial development tools that support the PLCopen XML specification, defined by PLCopen (2009). An example safety application is used through the paper to illustrate the proposed modelling and transformation process.

The remainder of this paper is organized as follows: In Section 2, related work is discussed. Section 3 presents the formal definitions of FBD programs defined by IEC6113-3. UPPAAL timed automata (TA) models are presented in section 4. The transformation rules are formally defined in Section 5. Conclusions are given in the last section.

2. RELATED WORK

Nowadays, there is an increasing interest of automating the verification process of applications in industrial automation. Several research groups have already published the results of their work to this direction. These works are mainly based on the transformation of the FBD models of the safety application to formal models, which are next used for verification purposes with the help of a model checking tool.

Pavovic and Ehrich (2010), present a method for the automated formal verification of PLC software written in FBD using the NUSMV model checker. The FBD should be constructed from basic FBs such as bit operations, comparators and jumps. Moreover, the IL representation of the FB body, which is generated from the corresponding FBD
with the help of the PLC programming tool is the one to be transformed to a NUSMV model. This method cannot be applied for the verification of safety applications built up from PLCopen SFBs, since SFBs are not basic FBs.

Yoo, Cha and Jee (2008) propose a way to formally verify FBD programs. They adopt an IEC61131-3 compliant syntax for the FBD and propose the translation of this specification into Verilog models. The so generated Verilog models are verified using the Cadence SMV model checker. A tool, FBD2V, has been implemented to automatically perform this transformation, and a case study is used to show the effectiveness of the proposed approach. In a similar way with Pavovic and Ehrich (2010), the transformation approach is presented only for systems composed finally of basic FBs, so SFBs of PLCopen are not supported.

A formal verification approach of a safety procedure of a nuclear power plant written in FBD language is presented by Németh and Bartha (2009). The approach is based on the Colored Petri Net (CPN) representation and it is similar to our approach in that it uses a pre-developed library of CPN subnets to represent the FBD specification in an identical structural way. However, the transformation of FBD diagrams to CPN is carried out manually. An automatic transformation process is not supported.

Silva and Barbosa (2008) propose and describe an automatic generation of TA models from FBD models. This allows testing of FBs against their specifications in the case that there is no PLC available to run them. The UPPAAL TRON tool is used to test some simulation scenarios on the generated TA models. The FBD elements that this approach deals with, are logical AND, OR, SR/RS flip-flops and time elements TON, TOFF and TP. The test of the model is evaluated by feeding some simulation traces to UPPAAL TRON tool. Traces are automatically generated from cause/effect matrix specified by experts. This approach cannot be applied to the safety applications that considered in our approach, since it only supports logical and time FBs.

Wardana, Folmer and Vogel-Heuser (2009) present an automatic verification approach to ensure the correctness of continuous function chart (CFC) programs using UPPAAL. As stated in this paper, the absence of a standard for CFC leads to different implementation by vendors. Their formal definitions of CFC are based on Emerson DeltaV tool. Although this approach is similar to ours, i.e., transformation rules are formalized based on formal definitions of both CFC and UPPAAL TA, it cannot be generalized to other implementation tools since it is not based on standards.

We are not aware of any other approaches that automatically transform IEC61131-3 FBD specifications to formal models for their automatic verification, based on formal definition of the source and target domains.

3. FORMAL DEFINITION OF FUNCTION BLOCK DIAGRAM

The FBD language has introduced very early in the industrial automation domain the model-driven development paradigm, as claimed by Thramboulidis (2010). The language has its origins in the field of signal processing. The standard defines that an IEC61131-3 program may contain zero or more function block instances or other language elements defined in it. However, we consider in this approach only programs that consist of one or more function block instances. Moreover, we use only two predefined libraries for the development of safety applications as pointed out in PLCopen directions. The first one is the SFB library, which was defined by PLCopen (2006) specifically for safety applications and contains 20 SFB types. This library is widely accepted and is already supported by many commercial PLC programming tools. The second is a library of Logic gate Function Blocks (LFBs), such as AND, OR, XOR and NOT.

According to IEC61131-3, the FB is defined as an independent, encapsulated data structure with a defined algorithm working on this data. The algorithm is represented by the code part of the FB type. Every FB has a name, a set of input parameters and a set of output parameters that define the interface of its FB instances and a body that defines their behaviour.

Fig. 1. Function block network of a safety application.
Programs, as defined by the standard, can only be instantiated within resources, while function blocks can only be instantiated within programs or other function blocks. The objective of a program is to specify the signal processing required for the control of a machine or a process by a programmable controller system. In this section, we formally define programs defined by the FBD language, i.e., FBD programs, with the objective to use this definition for the transformation process of the FBD representation of the safety application to UPPAAL TA representation. The FBD program shown in Fig. 1, which is part of a safety application, is used in the following as an example.

**Definition 1 (FBDProgram):** an FBD program is defined as a tuple FBDProgram = < Name, IP, OP, AV, Body > where:

- **Name:** the name of the FBDProgram that is defined by its developer,
- **IP:** a set of input variables that represent monitored parameters of the controlled machine or process, IP = \{ip_1, ip_2, ..., ip_n\},
- **OP:** a set of output variables that represent controlled parameters of the machine or process, OP = \{op_1, op_2, ..., op_m\},
- **AV:** a set of access variables, which are used as named variables that can be accessed by some of the communication services specified in part 5 of IEC61131. AV = \{av_1, av_2, ..., av_m\},
- **Body:** the body defines the behavior of the FBD program instances and its formal definition is given by the next definition.

**Definition 2 (FBDProgramBody):** The Body of the FBDProgram is defined as a tuple:

FBDProgramBody = < FBNs, VPBody-I, VPBody-O, VPBody-L > where:

- **FBN:** a set of function block networks,
- **VPBody-I:** a set of input variables entering the FBD program body, VPBody-I = \{pbvi_1, pbvi_2, ..., pbvi_m\},
- **VPBody-O:** a set of output variables leaving the FBD program body, VPBody-O = \{pbvo_1, pbvo_2, ..., pbvo_n\}, and
- **VPBody-L:** a set of local variables, VPBody-L = \{pbvl_1, pbvl_2, ..., pbvl_p\}.

For example, for the FBD program body of Fig. 1:

\[V_{\text{FBody-I}} = \{\text{SI}_1\_1, \text{SI}_1\_2, \ldots, \text{SI}_3\_2, \text{SI}_3\_4, \text{SI}_3\_6\},\]
\[V_{\text{FBody-O}} = \{\text{SO}_1\_1, \text{SO}_1\_3, \text{SO}_1\_7\},\]
\[V_{\text{FBody-L}} = \{\text{Reset}, \text{Automatic}, \ldots, \text{SLS}_M\_1, \text{M}_2\}.\]

A screen shot of the declaration part of the example safety application.

**Definition 4 (FBI):** An FB instance is defined as a tuple FBI = < InstanceName, FBTypeName, EO> where:

- **InstanceName:** user defined name of the specific instance,
- **FBTypeName:** the name of the corresponding FB type,
- **EO:** Execution order. It is an integer that defines the execution order of the FB instance. The execution order usually appears in the middle of every instance FB, as shown in Fig. 1.

FBIs of type LFB have no instance names and they have only one output port. An example of LFB instance is the AND FB shown in Fig. 1, which is defined as FBI_5=<-, AND, 5>.

**Definition 6 (C):** Connections is defined as C = {SFBC, IC, LC, LogicC, OC} where:

- **SFBC** is a set of direct connections between OPs of SFBs with IPs of SFBs. It is defined as: SFBC = \{InstanceName_{op} \rightarrow InstanceName_{ip}, \ldots\}, e.g.:
- **IC** is a set of connections that involves input variables elements in V_{\text{FBody-I}}. The input variable pbvi_i \in V_{\text{FBody-I}}
may be connected to an input port $ip_k \in IP$ of FBs or local variable $pbvl_i \in VPBody-L$. It can be defined as:

$$IC_m: pbvi \rightarrow InstanceName.ip,$$

where $IC_m$ is a logic function and "$\rightarrow$" means "and/or", e.g.:

$$ANDC: SF_EmergencyStop_1.S_EStopOut & SF_Equivalent_1.S_ESPE_Out \rightarrow SF_SafetyRequest_1.S_SafetyActive$$

$OC: is a set of output connections between op $\in OP$ of SFBs and an output variable $pbvo_ij \in VPBody-O$. It is defined as:

$$OC_m: InstanceName.op \rightarrow pbvo_ij.$$  

4. UPPAAL TIMED AUTOMATA MODELS

The UPPAAL system is simple in structure. It consists of three main parts which are: declarations, TA models and system declarations. In the declarations part all used variables are globally declared to allow synchronization via shared variables between TA models. TA models are separately created. The execution of these models is specified in system declarations part.

The UPPAAL TA system presented in this paper is formalized based on the work of Alur and Dill (1994). In the following the UPPAAL system is defined to facilitate the transformation process. This means that all parts, which are not used by the transformation process, are not included in our definition. A complete definition and the semantics of UPPAAL TA networks are given in Soliman and Frey (2009).

Definition 7 (UPPAALSys): UPPAAL system can be defined as a tuple $UPPAALSys = \langle Declarations, TAModels, SystemDeclarations\rangle$ where:

- **Declarations** is a set of triples $\langle type, variableName, value\rangle$ where:
  - type is defined as bool, int, const bool, const int, or clock,
  - variableName represents the name of variables used within TAModels,
  - value is considered initial value or constant according to the type parameter.

Fig. 3 shows a side of Declarations part of FBN in Fig.1. The left part of figure declares the variables used within SF_Equivalent_1 SFB instance. The right part declares the input, output and local variables used within the FBD body.

$\text{Fig. 3. Declarations part in the UPPAAL system (part of).}$

$\text{TAModels is a set of all TA models used in UPPAAL system. Every TA model has an associated name, a set of locations, with at least one initial location, and edges allowing transition to and from different locations in model.}$

Definition 8 (TA): A UPPAAL timed automaton $TAModel$ is a tuple ($L, I_0, E, C, Init, Inv, TL$), where:

- $L$ is a finite set of locations,
- $I_0$ is the initial location,
- $E$ is the set of the edges defined by $E \subseteq L \times Guard \times Sync \times Update \times L$, where Guard is the set of conditions and time constraints allowed in guards, Sync is a set of synchronization actions. It is never used in our approach, so I skip its definition, and Update is a set of sequences of assignment actions of the form $x_i := e_i, \ldots, x_n := e_n$, where $e_i, \ldots, e_n$ are integer expressions and clock resets,
- $V = Vbool \cup Vint$ denotes the set of Boolean and integer variables,
- $C$ denotes the set of real-valued clocks $C \cap V = \Phi$,
- $Init \subseteq Update$ is a set of assignments that assigns the initial values to variables,
- $Inv: L \rightarrow Inv(C, Vint)$ is a function, that assigns an invariant to each location. Inv(C, Vint) is the set of invariants over clocks $C$ and integer variables $Vint$, and $TL: L \rightarrow \{0, u, c\}$ the function, that assigns the type (ordinary, urgent or committed) to each location. In ordinary location, clocks can be incremented, representing time delay, until the invariant or guard constraints are reached. The system cannot delay if there is a process in an urgent or committed location. The transitions via the outgoing edges of a committed location have priority.
TA model of SFB instance SF_Equivalent_1 in Fig. 4 is defined as:

\[ L = \{ \text{Idle}, \text{Init}, \text{Wait}_A, \text{Error}_1, \text{Error}_2, \text{Wait}_B, \text{Error}_3, \text{From}_A, \text{Safety}_\text{Output}_E \} \]

\[ L_0 = \{ \text{Idle} \} \]

\[ E = \{ E_1, E_2, \ldots \} \]

where Sync is not defined on all edges. An example on edges is \( E_1 = \{ \text{Idle}, \text{Activate}_E, \ldots, \text{Ready}_E = 1, \text{Init} \} \).

\[ V = \{ \text{Activate}_E, \text{S}_\text{Channel}_A, \text{S}_\text{Channel}_B, \text{DiscrepancyTime}_E, \text{Ready}_E, \text{S}_\text{Equivalent}_\text{Out}_E, \text{Error}_E \} \]

\[ V_{\text{int}} = \{ \text{DiscrepancyTime}_E \} \]

\[ C = \{ \text{DT}_E \} \]

\[ \text{Init} = \{ \text{Ready}_E = 0, \text{S}_\text{Equivalent}_\text{Out}_E = 0, \text{Error}_E = 0 \} \]

\[ \text{Inv} = \{ \text{Wait}_A \} = \{ \text{Wait}_B \} = \{ \text{From}_A \} = \{ \text{DiscrepancyTime}_E = 0 \} \]

\[ T_L \] for all locations is \( \{ 0 \} \).

As shown in Fig. 4, all variables have suffix “_Eq_1”, where “_Eq” is representing SFB type name “SF_Equivalent” defined by PLCopen (2006) and “_1” is representing the first instance of this SFB in FBD program. In that way, we can use more than one instance of similar types without confusion. Moreover, that allows synchronization via shared variables declared in global declarations part of UPPAAL. More details on formalization and verification of PLCopen SFB library can be found in Soliman and Frey (2009, 2011).

The left hand side of Fig. 5 shows all TA model names resulting from the transformation of the FBD program, part of which is shown in Fig. 1, to UPPAAL TA models. There are four types of TA models: SFBs TA, Connections TA, R_TRIG/F_TRIG TA and INPUTs TA. SFBs and Connections TA capture the SFBs instances and all direct/indirect connections in FBD. R_TRIG/F_TRIG TA are associated to some SFBs IPs. Their function is to reset rising/falling edges detectors after every execution cycle. INPUTs TA are to allow altering external input variables that are physically connected to input modules of safety PLC. The right hand side of Fig. 5 shows examples of TA models.
SystemDeclarations in which execution order is defined by assigning priorities to TA models. In the systemDeclaration part of the transformed UPPAAL TA models, that are given below (Fig. 6), input connections IC1 to IC11 TA models have the highest priority then SF_Equivalent_1/SF_Equivalent_2 and so on.

**Fig. 6. Priorities on system of TA models.**

The execution flow is defined in the FBD program according to the IEC61131-3 and consequently in the generated UPPAAL TA models.

5. TRANSFORMATION RULES

In this section we present our transformation rules that are based on the above defined formal models of the FBD and UPPAAL notations.

**Transformation Rule 1 (Declarations mapping Rule):** The objective of this rule is to transform the declarations part of the FBD to UPPAAL artefacts. Variables defined in the externalVars element of PLCopen XML are declared as global in UPPAAL XML, and input/output variables of each externalVars element of PLCopen XML are declared as global in UPPAAL XML.

- For each FBi:= <InstanceName_i, FBTimeTypeName, EO> where FBT:= <FBTimeTypeName, IP, OP, B>:
  - In Declarations part of UPPAALSys, declare input parameters IP_i and output parameters OP_i where, i=1,2,...,n and n is the number of instances (see left side of Fig. 3).
- For each pbvi:= V_PBody_i, pbvi:= V_PBody_i and pbvi:= V_PBody_i of FBDProgramBody, Insert corresponding pbvi, pbvi:= V_PBody_i and pbvi:= V_PBody_i in Declarations (see Fig. 3 right side).

**Transformation Rule 2 (FBI mapping Rule):** The objective of this rule is to map FBIs to UPPAAL TA models.

- For each SFB instance <InstanceName, FBTimeTypeName, EO>, the corresponding TA model is inserted to UPPAAL with TA model name:= InstanceName using a pre-defined SFB TA model library (see Fig. 5 left side).

**Transformation Rule 3 (Connections mapping Rule):** The objective of this rule is to map the connections. For each connection in C = {FBC, FBIC, FBIC, LC, OC}, a TA model in UPPAAL is inserted.

- For each SFBCm:InstanceName,op→InstanceName, ip:
  - Insert a TA model with name ICm where: L= L0={ICm}, E={E1}={ICm, pbvi:= ip, -, pbvo:= ipm, ICm}, V= { pbvi, ip }, C= { }, Init= { }, No invariant, and TL = {u} (C6 in Fig. 5).
- For each ICm: pbvi→InstanceName,ip:
  - Insert a TA model with name Cm. This TA has one initial urgent location and self-edge where: L= L0={Cm}, E= {E1}={Cm, pbvi:= ip, -, pbvo:= ICm}, V= { pbvi, ip }, C= { }, Init= { }, No invariant, and TL = {u} (IC4 in Fig. 5).
- For each ICm: pbvi→ pbvo:
  - Insert a TA model with name ICm where: L= L0={ICm}, E= {E1}={ICm, pbvi:= ipm, -, pbvo:= pbvo, ICm}, V= { pbvi, pbvo }, C= { }, Init= { }, No invariant, and TL = {u}.
- For each LCm: pbvi→ InstanceName,ip,:
  - Insert a TA model with name Cm where:
    - L= L0={Cm}, E= {E1}={Cm, opk:= ipk, -, ipk:= pbvl-i, Cm}, V= { pbvi-i, ipk }, C= { }, Init= { }, No invariant, and TL = {u}.
- For each LCm: pbvi→ pbvo,:
  - Insert a TA model with name ICm where: L= L0={ICm}, E= {E1}={ICm, pbvi-j:= pbvo-k, -, pbvo-k:= pbvi-j, ICm}, V= { pbvi-j, pbvo-k }, C= { }, Init= { }, No invariant, and TL = {u} (IC4 in Fig. 5).
- For each LogicCm: (InstanceName,opk, pbvi-k, pbvo-k Const.)→ InstanceName,ip/pbvo/pbvi:
  - Insert a TA model with name LogicCm where:
    - L= L0={LogicCm}, E= {E1}={LogicCm, f (opj, pbvi-k, pbvl-i,Constn)→ ipp/pbvo-q/pbvl-r, ipp/pbvo-q/pbvl-r:= f ( opj, pbvi-k, pbvl-i,Constn ), V= { opj, pbvi-k, pbvl-i,Constn }, V= { opj, pbvi-k, pbvl-i,Constn ), V= { opj, pbvo-q/pbvl-r }, C= { }, Init= { }, No invariant, and TL = {u} (ANDC1 in Fig. 5).
- For each OCm: InstanceName,op→ pbvo:
  - Insert a TA model with name OCm where: L= L0={OCm}, E= {E1}={OCm, opj:= pbvo-k, -, pbvo-k:= opj, OCm}, V= { opj , pbvo-k }, C= { }, Init= { }, No invariant, and TL = {u} (OC1 in Fig. 5).

**Transformation Rule 4 (Rising/Falling Edge Triggers mapping Rule):** There are a set of input ports FBT.ip that need to reset an R_TRIG/F_TRIG signals associated to them. For example, the rising trigger edge of the Reset input of SF_SafetyRequest, which is called R_TRIGatReset, is a pulse signal that goes to true when Reset is true and
immediately reset to false after executing the SFB. These rising/falling signals have formal names of \( R\_TRIGatipn/FTRIGatipn \), specified by PLCopen (2006). Their names are transformed to \( R\_TRIGipn\_Type\_Name_n/ F\_TRIGipn\_Type\_Name_n \) in the UPPAAL system, e.g., \( R\_TRIGReset\_SR_1 \), where \( Type\_Name \) represents a SFB type and \( n \) is the instance number.

- For each \( R\_TRIGatip_i \) related to FBI,
  Insert a TA model with name \( R\_TRIGipn\_FBTypeName_n \), where: \( L=\{ R\_TRIG \}, E=\{ E1 \}=\{ - \}, R\_TRIGipn, -, R\_TRIGipn_n=0, - \}, V=\{ R\_TRIGipn \}, C=\{ \}, Init=\{ \}, No invariant, and TL = \{ u \} (\( R\_TRIGReset\_SR_1 \) in Fig. 5).

- For each \( F\_TRIGatip_i \),
  Insert a TA model with name \( F\_TRIGipn\_FBTypeName_n \), where \( L=\{ F\_TRIG \}, E=\{ E1 \}=\{ \}, F\_TRIGipn, -, F\_TRIGipn_n=0, - \}, V=\{ F\_TRIGipn \}, C=\{ \}, Init=\{ \}, No invariant, and TL = \{ u \}.

**Transformation Rule 5 (INPUTs mapping Rule):** To allow validation against simulation scenarios using the UPPAAL simulator, external variables in \( V_{PBVi-1} \) of FBDDProgramBody are allowed to be changed by user.

- For each \( pbpi \) in \( V_{PBVi-1} \),
  Insert TA model with name \( INPUTn \), where: \( L=\{ L0=\{ - \}, E=\{ E1 \}=\{ \}, pbpi, pbpi_n=\{ - \}, V=\{ pbpi \}, C=\{ \}, Init=\{ \}, No invariant, and TL = \{ u \} \).

**Transformation Rule 6 (Execution Flow mapping Rule):** The objective of this rule is to specify the execution flow of the UPPAAL model by defining priorities on TA models.

- Based on EO defined in FBI=\{<Instance\_Name, FBTypeName, E0=\},
  Assign priorities of TA models in UPPAAL SystemDeclarations part (see Fig. 6).

6. CONCLUSIONS

In this paper, formal models of FBDDs and UPPAAL TA for an automated transformation process have been presented. Based on these models we have defined transformation rules. A model-to-model transformation tool was developed using Java. The prototype transformation tool was used to test several real safety applications, and that proved its applicability. An example safety application was used as a case study through the paper to facilitate understanding of the models and the transformation rules.

ACKNOWLEDGEMENT

The authors would like to thank Theodoros Chatzigiannakis for the development of the prototype transformation tool.

REFERENCES


