Control Engineering Practice 19 (2011) 929–946

Contents lists available at ScienceDirect

Control Engineering Practice

journal homepage: www.elsevier.com/locate/conengprac

Verification and validation of safety applications based on PLCopen safety function blocks

Doaa Soliman*, Georg Frey

Chair of Automation, Saarland University, University Campus Building A5.1, 66123 Saarbrücken, Germany

ARTICLE INFO

Article history:
Received 30 November 2009
Accepted 5 January 2011
Available online 17 February 2011

Keywords:
Timed automata
Safety function block
IEC 61508
IEC61131-3
Verification and validation
Model-checking

ABSTRACT

Functional Safety is a major concern in the design of automation systems today. Many of those systems are realized using Programmable Logic Controllers (PLCs) programmed according to IEC 61131-3. PLCopen – as IEC 61131 user organization – semi-formally specified a set of software function blocks to be used in safety applications according to IEC 61508. In the presented work, formal models in the form of timed automata for the safety function blocks (SFBs) are constructed from the semi-formal specifications. The accordance of the formalized blocks to the specification is verified using model checking. Furthermore, their behaviour is validated against specified test cases by simulation. The resulting verified and validated library of formal models is used to build a formal model of a given safety application – built from SFBs – and to verify and validate its properties.

1. Introduction

Programming Logic Controllers (PLCs) are increasingly being used to implement safety functions for safety critical systems. One of the preferred programming languages in this area is Function Block Diagram (FBD) according to IEC 61131-3 (John and Tiegelkamp, 2001). There are many research projects in the field of verification and implementation of function blocks libraries according to this standard, e.g. Völker and Krämer (2001) or Song, Koo, and Seong (2004). However, firstly, safety issues are not addressed in IEC 61131-3. The third part of IEC 61131 deals only with five programming languages used by PLCs without distinguishing between control/logic signals and safety signals. For example, a Boolean safety related signal is declared as BOOL according to IEC 61131-3, but for a safety function it should be SAFEBOOL according to PLCopen. This is to differentiate between safety related signals and other signals. Secondly, PLC programming software packages provide only function block libraries dealing in general with communication, mathematical operations, logic, and so on. As a step for building safety applications in IEC 61131-3 FBD according to IEC 61508 (Bell, 2005), PLCopen (2006) specifies a set of safety function blocks (SFBs). Moreover, to demonstrate the usability of the defined SFBs in real life safety applications, PLCopen (2008) developed a user guideline. Several manufacturers of IEC 61131 programming tools have already implemented libraries according to this specification.

Recently, many researches represented formalization of existing PLC programs. As noted in Bani Younis, and Frey (2003) the reason of this formalization was either re-implementation or verification & validation (V&V) of an existing control programme. Nowadays, because of safety aspects, formal models are needed in the beginning of the design process. Depending on those types of models, formal V&V and tests by simulations can be achieved before implementation.

The main goal of the presented work is to ease the V&V process of a safety application built up from the PLCopen SFB library in one of the commercially available tools. This means, a verified, validated and certified SFB library should be implemented for use by the safety engineers. Moreover, a trusted transformation from the implemented safety application to a formalized one for verification is needed. This concludes our aim into several points. The first one, introduced by Soliman and Frey (2009) and extended in this paper, is to have a tool which allows verification of safety application built up from the PLCopen SFB library. The second point – also discussed in the following – is to build a well-tested SFB library in accordance with PLCopen specifications for implementation. The motivation to implement this library is discussed in the next section. The third point – under construction – is to automatically transform the implemented safety application built up from the well-tested SFB library to the formal model using the verified formal SFBs in the verification tool. Finally, this transformation tool will be generalized to be used by many commercially available tools.

The rest of this paper is organized as follows: In the next section the motivation of implementing SFB library in Instruction List (IL) programming language is explained. In Section 3,

* Corresponding author. Fax: +49 681 302 57599.
E-mail addresses: doaa.soliman@aut.uni-saarland.de, doaa_ib@yahoo.com (D. Soliman), georg.frey@aut.uni-saarland.de (G. Frey).

0967-0661/$ - see front matter © 2011 Elsevier Ltd. All rights reserved.
doi:10.1016/j.conengprac.2011.01.001

verification approach of a safety application is described. A description of PLCopen specifications is illustrated by one single SFB example and one complete safety application in Section 4. To test the PLCopen approach of implementing a SFB library, this library is implemented in IL in Section 5. The usage of this library is demonstrated by a safety application example in the same section. Section 6 describes the formalization of SFBs in details using an example from the PLCopen library. The use of the formalized SFBs for verification of is shown in Section 7 utilizing the safety application example used in implementation. Section 8 shows the required transformation tool tasks. The paper concludes with a short summary and an outlook on further work.

2. Motivation to implement SFB library

Many commercially available PLC programming tools are nowadays supporting IEC 61131-3 standard. However, according to PLCopen organization, there are only five companies that have a certified PLCopen-SFB-Library. None of these five companies supports the exportation of a project programmed in FBD according to PLCopen XML standard. This PLCopen XML standard is another scheme published by PLCopen in 2005. This XML scheme is widely accepted by many companies inside and outside Europe and they started implementing these XML specifications. According to the feedback from these companies, an improved PLCopen XML scheme is released in 2008. The aim of publishing this XML scheme is to allow an interface platform to exchange projects between different tools. This interface is the one used to allow automatic transformation of a safety application into UPPAAL model checker in the presented work. UPPAAL is an integrated tool environment for modelling, validation and verification of real-time systems modelled as networks of timed automata. However, the near future will witness the supporting of PLCopen XML scheme by many commercial tools.

To summarize the motivation of our work, one should give first an overview of the qualifications needed in PLC programming tool to transform a safety project into one readable form by UPPAAL.

Fig. 1 describes the transformation process of a safety application from PLC code to UPPAAL. This process is discussed in the next section. As seen in the figure, there are three qualifications needed to automatically transform a safety application to an UPPAAL XML format. At the moment, these qualifications are not easy to be found in one PLC programming tool. However, many companies are planning to support all these features in the near future.

To find an alternative PLC programming tool that supports the entire features needed for transformation, the idea of implementing an SFB library raised. According to the required qualifications, OpenPCS programming tool is chosen to implement the SFB library. Although it takes us time and effort, it has some advantages. One advantage is to get an overview about the internal construction of the function blocks. Moreover, individual test of SFBs makes one more familiar with them. However, one of the drawbacks of that alternative solution is that exporting PLCopen XML in OpenPCS tool is still under construction. That adds an effort to get the required PLCopen XML scheme.

3. Approach of verification of a safety application

The presented approach is based on the development process given in Fig. 1. To build a safety application an engineer is assumed to build his software using a library of SFBs that was implemented according to the PLCopen (2006) specifications. To

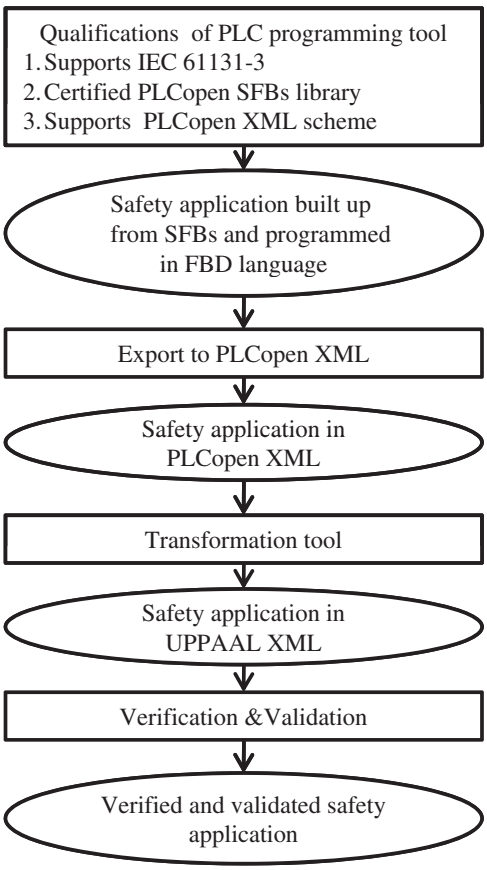

Fig. 1. Proposed development process for verified safety applications using the PLCopen SFBs.

do so, the specification for the safety application is read and interpreted by the engineer and the solution is implemented in a PLC programming tool. Now it has to be verified that the implemented application fulfils the original specification. To this end, the process described in Fig. 1 is proposed.

The result implemented safety application in FBD language is then transformed to UPPAAL XML to be verified. Following the structure (interconnection of SFBs) of the implemented application a formal model of the safety application is built from a library of pre-formalized SFBs. On this model, properties derived from the original specification can be verified. In this paper, the transformation of the implemented safety application in a PLC programming tool to a formalized one in a model checker tool is carried out manually. However, the tool responsible of the automatic transformation is under construction as discussed in Section 8.

The main part of the presented work is to provide the library of pre-formalized SFBs. This process is shown in the right (shaded) part of Fig. 2. Starting from the PLCopen specification a library of SFBs is built using timed automata in the UPPAAL tool. To validate the temporal behaviour of the SFBs, simulations of the models are performed. Furthermore specified properties are formalized using temporal logic. By using model checking – see e.g. Bérard et al. (2001) – it is then verified that these properties hold on the timed automata. On the other side (left part of Fig. 2) it is assumed that the manufacturer of a programming tool starts from the same specification and builds a set of implemented SFBs. The conformance of these SFBs to the PLCopen specification is normally certified by some organization like TÜV (German organization that certifies the safety of products of all kinds).

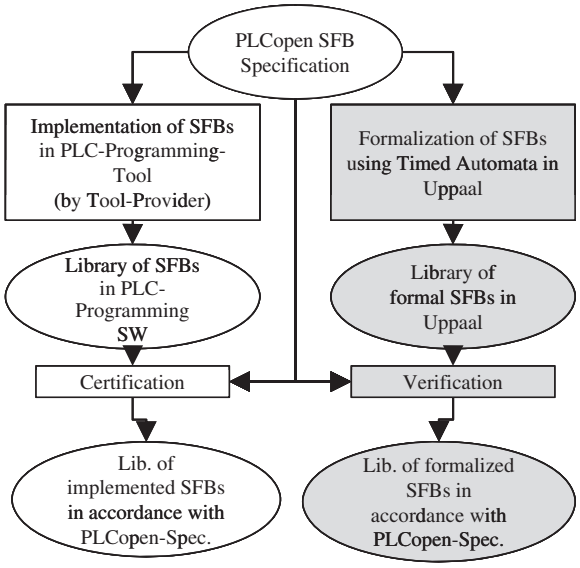

Fig. 2. Process of deriving the safety function block libraries for implementation (left) and verification (right) from the PLCopen specification.

In the following it is assumed that the implemented SFBs are actually behaving like described in the PLCopen (2006) specification.

4. Description of PLCopen specifications

The PLCopen (2006) specification contains the description of 20 SFBs. All those descriptions follow the same structure that is illustrated in the following subsection using the SFB SF-Equivalent as an example. Main parts of this description are:

- A graphical description of the internal states and behaviour using a state diagram.
- A list of properties described in natural language.
- Timing diagrams describing the temporal behaviour for some specific scenarios.

4.1. Complete description of one SFB: SF_Equivalent

This SFB is used with 1oo2 (one out of two) safety switches. It indicates the status of two safety inputs, e.g. sensors. Depending on the safety output of this SFB "*S_EquivalentOut*", a safe or unsafe state of the system is considered. In other words, if the two inputs are equivalent, it means both are true or both are false. And, this leads to safe or unsafe state, respectively. Else, if they are not equivalent for more than a specified time, it leads to an error state.

The Input/Output variables declarations of the block are shown in Fig. 3. Moreover, an interface description (*Name*, *Data Type*, *Initial Value and Function*) of every variable is tabled.

The textual description of this SFB according to PLCopen is "This function block converts two equivalent SAFEBOOL inputs to one SAFEBOOL output with discrepancy time monitoring. Both input Channels A and B are interdependent. The function block output shows the result of the evaluation of both channels. If one channel signal changes from TRUE to FALSE the output immediately switches off (FALSE) for safety reasons. The discrepancy time is the maximum period during which both inputs may have different states without the function block detecting an error.

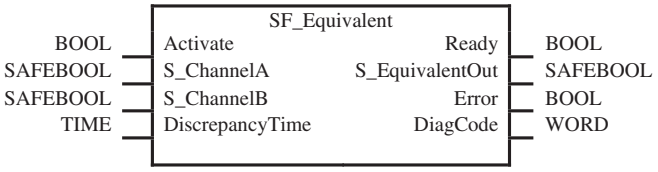

Fig. 3. Interface description of SFB *SF_Equivalent*.

Discrepancy time monitoring starts when the status of an input changes. The function block detects an error when both inputs do not have the same status once the discrepancy time has elapsed."

In the state diagram shown in Fig. 4, the inputs are used as transition actions and the outputs are updated according to every state. The time input is assumed a preset value of a timer. The state diagram is split into three regions (shown by dashed lines) according to the output variables status in every region. For example, the *Ready* output is false only in *Idle* state, and the *S_EquivalentOut* output is true only in *Safety output enabled* state. The *Error* output is true only in error states.

The *DiagCode* output which declared as WORD varies to indicate the active state. The symbol "*S_*" is given as prefix to safety related signals. Therefore, variables with such a prefix have SAFEBOOL declaration type as shown in Fig. 3. The numbers in small circles around every state indicate the priority in transitions where zero is the highest priority. Hidden transitions with highest priority 0 and action *NOT Activate* from all states to *Idle* state are not shown in the state diagram for readability reasons.

Additionally, the temporal behaviour of the SFB is described for specific scenarios in a timing diagram (Fig. 5). This timing diagram is split into two parts, the upper part describes the changes in the inputs of the SFB and the lower describes the response of the outputs. In the *Start* operation, all outputs are false by default until the SFB is activated (*Activate=true*). As a response to this activation the *Ready* output goes immediately to true and the *Normal* operation begins. In normal operation, the status of the two input channels (*S_Channel A and S-ChannelB*) is observed. If both channels go to true consecutively within a specified time (*DiscrepancyTime*), the *S_EquivalentOut* output goes to true indicating a safe state. Contrarily, if both input channels go to false within discrepancy time, the output goes to false indicating an unsafe state.

A discrepancy timer is used to measure the time in which both input channels have different status. Thereafter, it resets without response when both input channels have the same status. However, if the discrepancy time is elapsed and both input channels have different status, then the *Error* output goes to true (not shown in this timing diagram). A diagnostic code output (*DiagCode*) indicates the active internal state of the SFB in hexadecimal numbers.

Finally, a description and output setting of every state are tabled. This table defines the output variables' status for each state in the state diagram, and is used later in modelling and verification.

4.2. Complete description of a safety application

The PLCopen (2008) safety specification contains a set of examples. For every example, a block diagram is available showing the interconnection between SFBs and also showing the safety inputs and outputs. A description of safety functions and declaration of the used variables is also given. Additionally, in some safety examples, a timing diagram is given.

In the following, the first example from this specification "*Emergency stop with safe stop & equivalent monitoring*" is used. Fig. 6 shows the proposed solution for this problem implemented

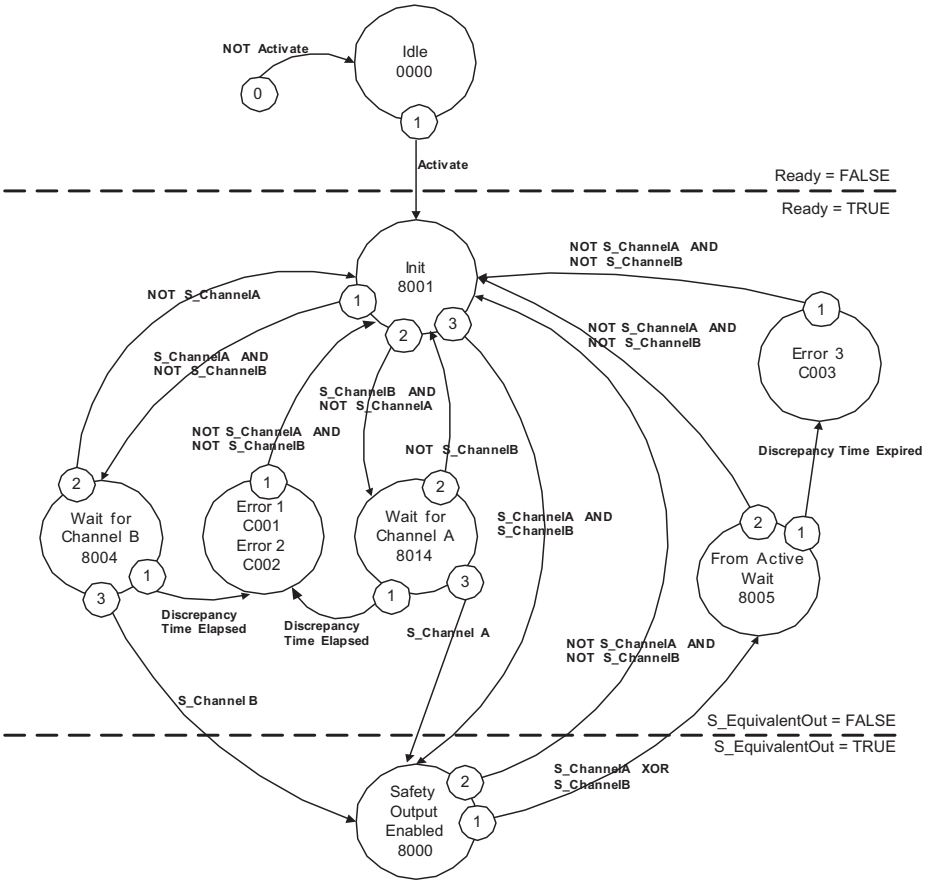

Fig. 4. State diagram for SFB *SF_Equivalent*.

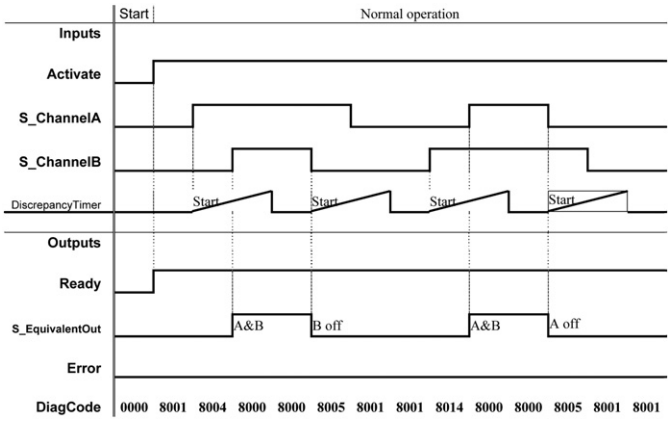

Fig. 5. Part of the timing diagram of SFB *SF_Equivalent*.

in function block diagram by connecting SFBs directly or via simple logic gates. This example uses the following SFBs:

- *SF_Equivalent* as described previously in Section 4.1.
- *SF_EmergencyStop* handles the Emergency Stop condition for the application, and ensures the correct restart.
- *SF_ESPE* handles the output signal switching device of electro-sensitive protective equipment (ESPE), e.g. light curtain, for the application, and ensures the restart inhibit.
- *SF_SafeStop1* initiates and monitors a stop of a drive system in accordance with stop category 1.

This safety application has three safety inputs and one safety output as shown in Fig. 7. The two inputs *S1_S_EStopIn_1* and *S1_S_EStopIn_2* are connected to the emergency stop pushbutton through 1oo2 switch. If both go true within specified discrepancy time, it indicates that the switch is released and no emergency stop is requested. This information passes via *SF_Equivalent* and *SF_EmergencyStop* to *SF_SafeStop1*. Similarly, the input *S2_S_ESPE_In* is connected to ESPE, and its status passes via *SF_ESPE* to *SF_SafeStop1*. According to PLCopen (2006) specifications, *SF_SafeStop1* initiates a controlled stop of an electrical drive in accordance with stop category 1—with power available to the machine actuators to achieve the stop and then removal of the power when the stop is achieved.

Fig. 8 shows the timing diagram for *SF_SafeStop1* in case of normal operation and safety request. If there is no stop request (*S_StopIn*=*true*), then it indicates a normal operation and the safety output *S_Stopped* goes to false. Else, in case of false *S_StopIn*, it waits for acknowledgment from the drive system within a specified monitoring time. This acknowledgement signal informs the safety application that the motor is stopped. When received, *S_Stopped* output goes to true and brings the system to a safe state.

Some of the safety functions achieved by this solution, according to PLCopen, are as follows:

- Issuing the emergency stop (via *SF_EmergencyStop*) or interrupting the light beam in the light curtain (via *SF_ESPE*) stops the drive in accordance with stop category 1.
- The stop of the electrical drive within a predefined time is monitored (via *SF_SafeStop*1).
- The Safe Status of the drive is indicated by the *S_Stopped* variable, connected to the functional application.

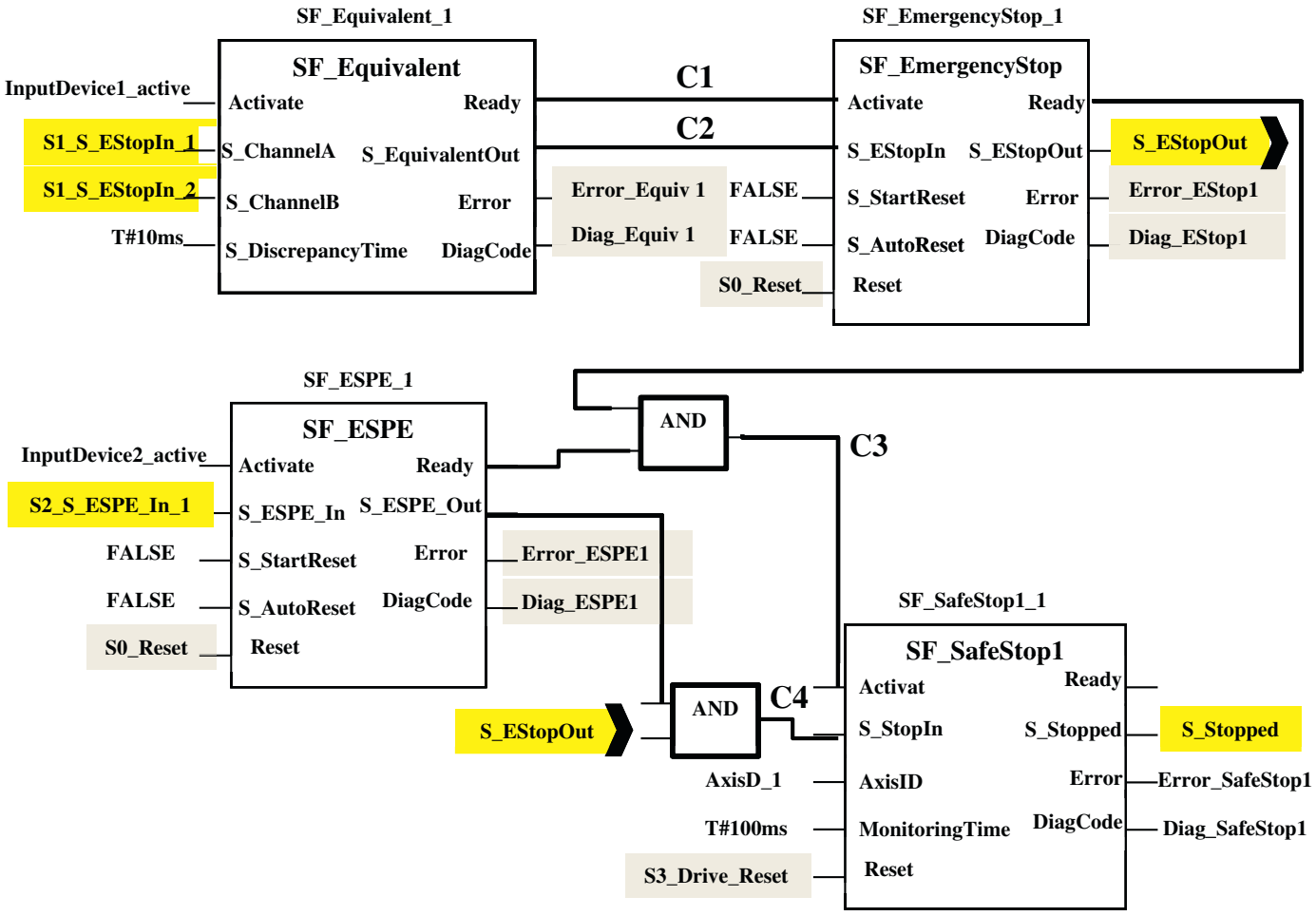

Fig. 6. Safety application to be verified.

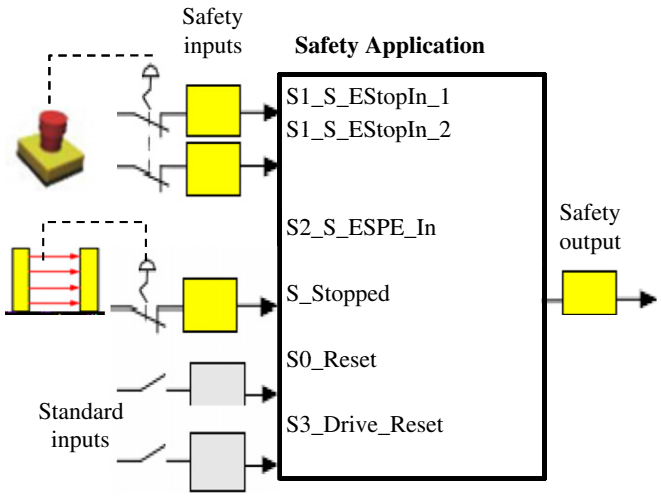

Fig. 7. Graphical overview of the safety application interface.

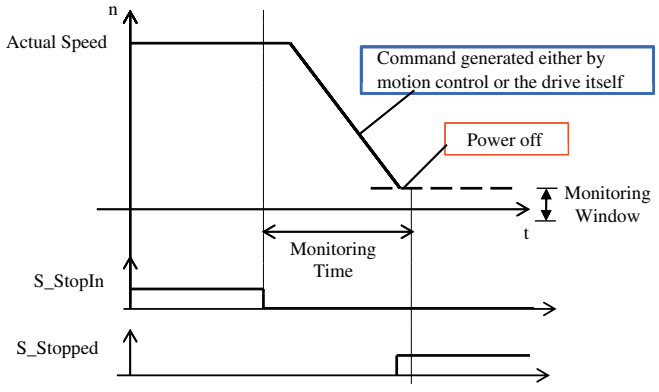

Fig. 8. Process timing diagram for *SF_SafeStop*1.

- If the stop is performed by the Emergency Switch, a manual reset is required (via *SF_EmergencyStop*).
- If a monitoring time violation is detected (via *SF_SafeStop*1), manual error acknowledge is required to allow a reset.
- The two channel connectors of the emergency stop are monitored. An error is detected when both inputs do not have the same status once the discrepancy time has elapsed (via *SF_Equivalent*).
- The functional stop in this example is performed as a safe stop issued from the functional application. A restart interlock for this stop is not necessary.

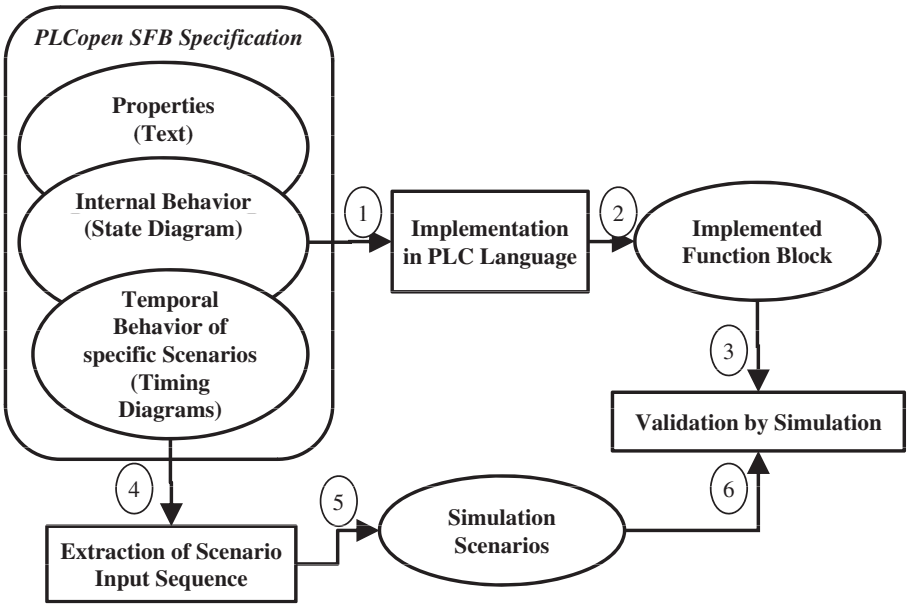

Fig. 9. Detailed description of the process of implementing a SFB.

5. Constructing the safety function block library for implementation

The idea here is to have an accessible and verified SFB library for two reasons:

- To get an overview about the internal construction of function blocks, the external connection of them in FBD editor and the execution process of a FBD application. This overview allows comparing the model and execution process of an application in UPPAAL to its corresponding in a real application.
- To transform the safety application built up from those SFBs to XML format according to PLCopen Scheme. This scheme is another standard published by PLCopen organization. The PLCopenXML format is the one to be transferred automatically to a readable system by UPPAAL.

Since there is no available commercial tool supporting all the features needed for this purpose, as indicated in Section 2, the SFB library is implemented directly in the following.

5.1. Implementation of SFBs in a PLC programming tool

For the implementation OpenPCS programming software from infoteam is used. The software and user manuals are available for free on the website http://www.infoteam.de. It consists of a set of tools including editors for all five languages of IEC 61131-3. The FBD editor is the tool for reuse of predefined functionality encapsulated in function blocks. It supports also a windows simulation tool named SmartSIM. This tool is used instead of a real hardware controller to test the implementation.

The process to implement an SFB in a PLC programming tool is shown in Fig. 9. First of all the Input/Output variables of the SFB are declared according to the interface description in the PLCopen document. Then, the state diagram is transferred into an IL code. Depending on this implemented function block and the input sequence scenario extracted from the timing diagram, an online simulation is executed to validate the temporal behaviour using the simulator.

This process is described for the SFB *SF_Equivalent* introduced in Section 2. And also applied to all other SFBs defined by PLCopen (2006).

5.2. The approach of converting a state diagram to IL code

Fig. 10 shows the required procedures to obtain a tested and verified SFB in IL programming language. Initially, the Input/Output variables of *SF_Equivalent* SFB shown in Fig. 3 are declared using declaration window in IL editor.

As shown in Fig. 11, the declaration type SAFEBOOL is given to a safety-related Boolean input or output as mentioned before. This type of declaration is suggested by PLCopen and supported by OpenPCS to distinguish between the safety related signals and the standard ones.

Moreover, a variable name is given and declared for every state (Fig. 12) and every timer of the state diagram. Next, the transitions are numbered according to the outgoing and ingoing states, e.g. *Tnm* means the transition coming out from state *n* into state *m*.

IL is a type of assembly language for PLCs. To implement a state diagram, each state is assigned a Boolean variable. Based on these assignments the behaviour can be implemented component-wise, i.e. a code-segment for each state and each transition. This implementation is similar to the one presented for Petri Nets in Frey (2000). The difference is that in this approach only one transition can be fired in each execution cycle. Hence, after firing of a transition the other transition code segments are not evaluated (jump to the end of transition codes). Furthermore, it has to be assured that priorities assigned in the PLCopen state diagrams are modelled correctly.

Now, the state diagram is ready to be transformed to IL code by three steps. First, the three timers declared previously, are called as follows:

```
CAL T_1 (IN:=S_3, PT:=DiscrepencyTime)
CAL T_2 (IN:=S_5, PT:=DiscrepencyTime)
CAL T_3 (IN:=S_7, PT:=DiscrepencyTime)
```

The timer is set and reset according to IN input, which actually is a state status. This means, it is not needed to reset the timer,

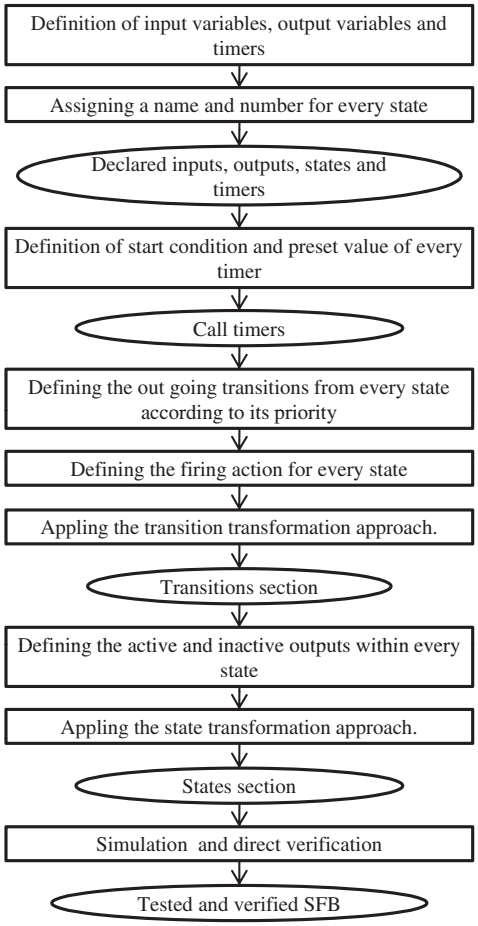

Fig. 10. Procedures of transforming a state diagram to IL.

```
VAR_INPUT
      Activate            : BOOL ;
      S_ChannelA          : SAFEBOOL ;
      S_ChannelB          : SAFEBOOL ;
      DiscrepencyTime     : TIME ;
END_VAR
VAR_OUTPUT
      Ready               : BOOL ;
      S_EquivalentOut     : SAFEBOOL ;
      Error               : BOOL ;
      DiagCode            : WORD ;
END_VAR VAR
 S_1   : BOOL := TRUE ;  (* Idle *)
 S_2   : BOOL := FALSE ;(* Init *)
 S_3   : BOOL := FALSE ;(* Wait for Channel A *)
 S_4   : BOOL := FALSE ;(* Error 1, Error 2 *)
 S_5   : BOOL := FALSE ;(* Wait for Channel B *)
 S_6   : BOOL := FALSE ;(* Error 3 *)
 S_7   : BOOL := FALSE ;(* From Active Wait *)
 S_8   : BOOL := FALSE ;(* Safety Output enabled *)
 T_1   : TON;            (* Timer for S3 --> T34 *)
 T_2   : TON;            (* Timer for S5 --> T54 *)
 T_3   : TON;            (* Timer for S7 --> T76 *)
END_VAR
```

Fig. 11. Declarations of *inputs*, *outputs*, *states* and *timers*.

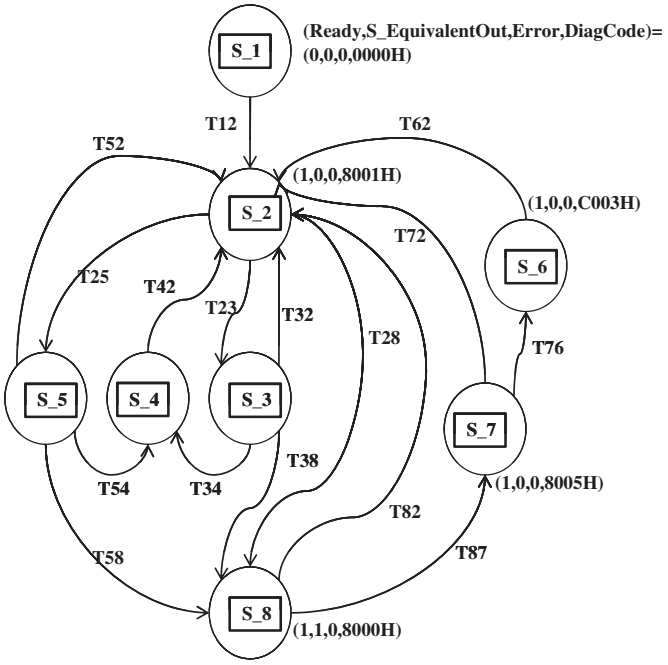

Fig. 12. Notation of *states*, *transitions*, and *outputs* on the state diagram.

```
(******  Transition T51 ******)
      LD    S_5
      ANDN Activate
      R     S_5
      S     S_1
JMPC State_1
(******  Transition T54 ******)
      LD    S_5
      AND   T_2.Q (* t(S5)>= DiscrepencyTime *)
      R     S_5
      S     S_4
JMPC State_4
(******  Transition T52 ******)
      LD    S_5
      ANDN S_ChannelA
      R     S_5
      S     S_2
JMPC State_2
(******  Transition T58 ******)
      LD    S_5
      AND S_ChannelB
      R     S_5
      S     S_8
JMPC State_8
```

Fig. 13. Converting transitions with and without timers to IL.

because it resets automatically when its input state is deactivated.

Second, all transitions are transferred to IL taking into account the priority numbers defined on every transition. For simplicity, the state S_5 (*Wait for ChannelB*) and its outgoing transitions are considered to be transferred to IL. In this state, it is assumed that the system is waiting for *ChannelB* to be activated while *ChannelA* is active. As shown in Fig. 13, the highest priority transition (T51) is fired if the *Activate* input of the SFB is false. Else, the next transition (T54) is fired if the *DiscrepancyTime* is expired and so on until one transition is fired. The idea of implementing a transition is to load the pre-state and ANDing it with the transition actions and/or timer output. If the result is true, then the pre-state is reset and the post-state is set. Next, a conditional jump is executed to

assign the SFB outputs. However, if no transition is fired the execution will be ended.

Finally, the states are transferred to IL code. One example of state S_5 is shown in Fig. 14. The state IL code is executed only when one ingoing transition to this state is fired. Whenever state execution is carried out, it ends unconditionally (*JMP End*). As shown in Fig. 12, the SFB outputs are updated according to the active state. In state S_5, the outputs status is (*Ready*, *S_EquivalentOut*, *Error*, *DiagCode*)=(*1*, *0*, *0*, *8004H*).

By this systematic way, all timers, transitions and states are coded and ready for simulations by SmartSIM tool.

To simulate the SFB IL code, it is needed to be called in FBD editor, compiled and transferred to the simulator tool. The simulator supports setting the input variables and observing the output variables. It is also possible to watch all the declared variables inside the function block. By using this simulator, the input sequences extracted from the timing diagram are applied to the input variables of the SFB, and then the output variables are observed. The Control Data Analyzer, supported by OpenPCS, hands the programmer the possibility to plot variables over time. The screenshot shown in Fig. 14 illustrates a part of the simulation. The simulation results assured that the Input/Output variables are acting like those in the timing diagram given by PLCopen (cf. Fig. 15). However, this validation is in no way complete, since only specific scenarios are tested.

```
(****** State Wait for Channel B ******)
State_5: LD  S_5
     S Ready
     R S_EquivalentOut
     R Error
     LD 16#8004
     ST DiagCode
JMP End
```

Fig. 14. Converting states to IL.

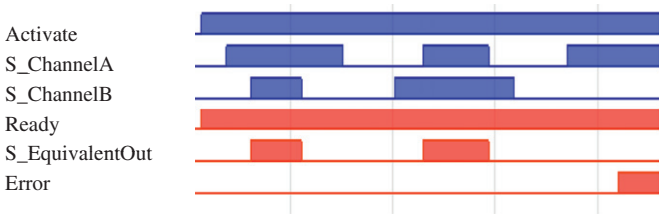

Fig.15. Simulation results of *SF_Equivalent*.

5.3. Implementation of a safety application

Using the FBD editor, the four SFBs are inserted and declared. The safety Input/Output variables are also declared to allow connecting to the SFBs. The FBD shown in Fig. 16 is a direct implementation of the safety application in Fig. 6.

To execute the safety application, it has to be compiled and sent to the simulator tool. Many scenarios of input sequences are extracted from the safety functions which are defined previously

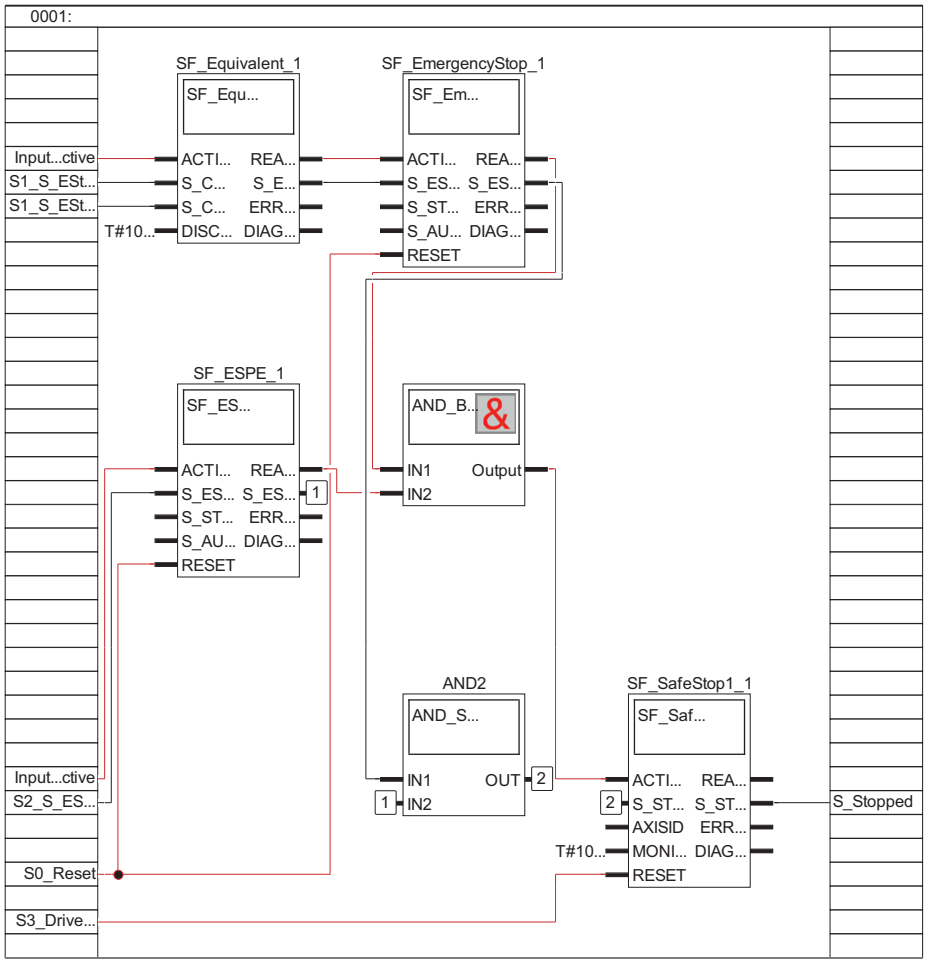

Fig. 16. Implemented safety application in FBD.

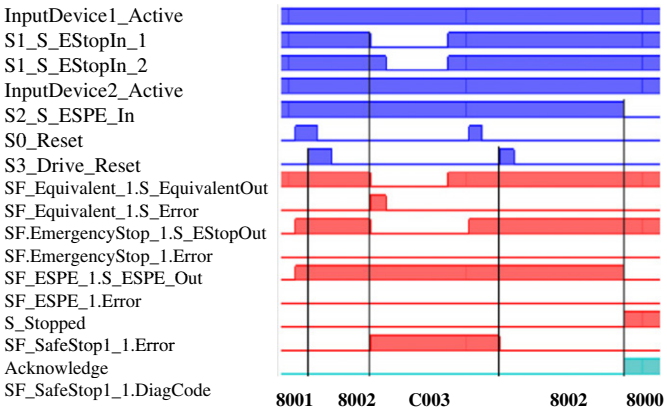

Fig. 17. Simulation results of the safety application.

in Section 4.2. Thereafter, they are applied to the safety application to allow observing the output variables response.

A screenshot of the timing diagram of the input and output variables is shown in Fig. 17. The simulation results showed that the safety functions are successfully achieved for the test-cases. Again, this is not a complete coverage of all possible scenarios and hence no guarantee for correct safe behaviour.

The reason of choosing IL language is to allow direct verification of individual SFB using [mc]square model checker as shown in Biallas, Frey, Kowalewski, Soliman, and Schlich (2010). However, due to the complexity, this approach is only feasible as long as a few simple blocks are connected. Furthermore, SFB libraries provided by PLC software vendors are normally compiled to protect intellectual property. In this case model checking on IL level is not possible since the IL code is not accessible. For these reasons the V&V approach presented in the following is proposed.

6. Constructing the safety function block library for verification

6.1. State diagrams

Generally, a state diagram is used to give an abstract description of the behaviour of an SFB. As shown before in Fig. 4, the state diagram used in this paper is a graph with the following elements:

- *State* is a finite set of vertices represented by circles and labelled with unique words written inside them.
- *Start State*: an element of the *States* set. It always labelled *Idle*.
  Input$=BV_{in}\cup TC$ is a finite set of input labels where BV_{in} is the set of Boolean input variables and *TC* is the set of time constant inputs. Boolean variable inputs are used in event conditions on transitions. But, Time constant inputs are considered time constraints.
  Output$=BV_{out}\cup WV$ a finite set of output labels where BV_{out} is the set of Boolean output variables and *WV* is a Word output variable known as *DiagCode*. The *DiagCode* output is not used in transformation, since the state label acts as a substitute to it. *Outputs* are updated with every state.
- *Transition* is a set of edges with conditions on them. An edge is an arrow directed from the source-state toward the target-state. Only one transition can be taken of a state. Thus, a priority number is assigned to possible parallel transitions from every state where a lower number means high priority.
- *Condition* is a set of conditions on transitions. Conditions are divided into two types. They are event and time conditions. An event condition is a statement of logical relation between BV_{in}. Only the logical operators OR, AND, XOR and NOT are allowed. Time conditions are time constraints defined by *TC* restrictions.

6.2. Timed automata

Timed automata were introduced by Alur and Dill (1991) as a formal notation to model the behaviour of real-time systems. The definition provides a simple way to annotate state-transition graphs (state diagrams) with timing constraints using finitely many real-valued clock variables.

The states of the graph are called *locations*, and transitions are called *edges*. A clock can be reset to zero simultaneously with any edge firing. At any instant, the reading of clock equals the time elapsed since the last time it was reset. With each edge and location a clock constraint is associated.

As mentioned in Larsen, Pettersson, and Yi (1997), the basis of the UPPAAL model is the notation of timed automata developed by Alur and Dill (1994) as extension of classical finite-state automata with clock variables. A system in UPPAAL is modelled as a network of timed automata (Behrmann, David, & Larsen, 2004) extended with variables, guards and invariants that might involve both clocks and variables, notion of urgency (urgent and committed locations, urgent channels), etc. However, there are several formal definitions of UPPAAL timed automata. For the translation in this article, the formal description presented by Bortnik, Fronczak, and Rooda (2007) is chosen. As it covers most of the features of UPPAAL timed automata that are implemented in the tool. A slight modification is performed to suit our purpose. For example, the set of variables is split into two sets to differentiate between Boolean and time variables. For simplicity, some symbols names are changed and other unused symbols are neglected.

6.2.1. Formal definition of UPPAAL timed automaton

Definition 1. A UPPAAL timed automaton A is a tuple $(L, l_0, E, V, C, Init, Inv, T_L)$, where:

- L is a finite set of locations,
- l_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 which includes actions and co-actions. An action send over a channel h is denoted by $h!$ and its co-action *receive* is denoted by $h?$ The τ_h-action is an internal action which cannot synchronize and does not have a co-action, and
  - *Update* is a set of sequences of assignment actions of the form $x_l := e_l, \ldots, x_n := e_n$, where $e_l, \ldots, e_n$ are integer expressions and clock resets,
- $V = V_{bool} \cup V_{int}$ denotes the set of Boolean and integer variables,
- C denotes the set of real-valued clocks ($C \cap V = \emptyset$),
- $Init \subseteq Update$ is a set of assignments that assigns the initial values to variables,
- Inv: $L \rightarrow \mathcal{I}nv(C, V_{int})$ is a function, that assigns an invariant to each location. $\mathcal{I}nv(C, V_{int})$ is the set of invariants over clocks C and integer variables V_{int}, and
- T_L: $L \rightarrow \{o, u, c\}$ the function, that assigns the type (ordinary, urgent or committed) to each location. 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.

Definition 2. A network of timed automata *NA* is a tuple $(\overline{A}, \bar{l}_0, V', C', H, T_H, Init')$, where $\overline{A}=(A_1,\ldots,A_n)$ is a vector of n timed automata $A_i=(L_i, l_i^0, E_i, V_i, C_i, Init_i, Inv_i, T_i^L)$ for $I\le i\le n$. $\bar{l}_0=(l_1^0,\ldots,l_n^0)$ is the initial location vector, V' and C' are the sets of global (shared) variables and clocks, respectively, $(C'\cap V'=\emptyset)$, and H is a set of channels ($V'\cap H=\emptyset$ and $C'\cap H=\emptyset$). The function T_H: $H\to\{o, u, b\}$ assigns the type (ordinary, urgent or broadcast) to each channel. In case $H=\emptyset$, function T_H is undefined and is then informally denoted by $\emptyset$. Init' is the set of assignments that assigns the initial values to the global variables.

6.2.2. UPPAAL semantics

The description of the UPPAAL timed automata semantics is given below. It is based on Bortnik et al. (2007) description with slight changes as a result to the modification on timed automaton definitions.

$\alpha'=upd(\alpha)$ is used to denote how the valuation is changed due to assignment (update). Note, that if there are more than one assignment on the edge, they are performed sequentially. For example, if *upd* is of the form $\{x:=3, y:=x+I\}$, and $\alpha(x)=0$, $\alpha(y)=I$, then $\alpha'(x)=3$, $\alpha'(y)=4$.

The state of the timed automaton is defined by the location and valuation $s=\langle l, \alpha\rangle$. The initial valuations $\alpha_0(c)=0$ for all $c\in C$, $\alpha_0(x)=Init(x)$ for all $x\in V$. Let $\alpha\vDash Inv(l)$ denote that α satisfies $Inv(l)$. The set of possible valuations denoted $\mathbb{A}$.

A vector of locations $\bar{l}$ is a n-tuple. With, $\bar{l}[l'_i/l_i]$ we denote a vector where the ith element l_i of $\bar{l}$ is replaced by $l_{i'}$. Similarly, with $\bar{l}[l'_i/l_i, l'_j/l_j]$ we denote a vector where the ith element l_i of $\bar{l}$ is replaced by l'_i and the jth element l_j of $\bar{l}$ is replaced by l'_j.

The semantics of a network of timed automata is defined as a transition system $(S, s_0, \to)$, where $S=(L_I\times\ldots\times L_n)\times\mathbb{A}$ is the set of states, $s_0=\langle \bar{l}_0\ \alpha_0\rangle$ is the initial state, and $\to\subseteq S\times S$ is the transition relation defined by the following rules:

- Simple action: A simple action $(\bar{l}, \alpha)\xrightarrow{upd}(\bar{l}[l'_i/l_i], \alpha')$ can be performed, if there exists an edge $l_i\xrightarrow{g,ch,upd} l_{i'}\in E_i$, $l_i\in\bar{l}$ such that:
  - The guard g is satisfied in α
  - The invariants are satisfied in α'.
  - The source location is committed or there is no other committed locations in $\bar{l}$.
  - $\alpha'=upd(\alpha)$.
- Synchronization action: A synchronization action $(\bar{l}, \alpha)\to^h(\bar{l}[l'_i/l_i], \alpha')$ can be performed, if for $i\ne j$ there exist $l_i\to^{gi,h!,ai} l_i\in E_i$ $l_i\in\bar{l}$ and $l_j\to^{gj,h?,aj} l_{j'}\in E_j$, $l_j\in\bar{l}$ such that:
  - both guards g_i and g_j are satisfied in α.
  - Invariants will be satisfied in α'.
  - One or both of the processes is in the committed location or there is not any other committed location in $\bar{l}$.
  - The valuation $\alpha'=upd_j(upd_i(\alpha))$.
- Delay: A delay $((\bar{l}, \alpha)\to^t(\bar{l}, \alpha')$ can be performed, if all following holds:
  - There is not any process in the committed location.
  - There is not any process in the urgent location.
  - The synchronization over an urgent channel is not possible.
  - All the invariants remain satisfied during delay.
  - The valuation α' is defined as follows, $\forall_c\in C$: $\alpha'(c)=\alpha'(c)+t$ and $\forall_x\in V$: $\alpha'(x)=\alpha(x)$.

6.3. Approach to transform a state diagram to timed automaton

Depending on the previous definitions of state diagram and timed automaton, the following approach is given to transform a state diagram to a corresponding timed automaton.

- L=State,
- l_0=Start State,
- E=Transition, where:
  - *Guard* is the set of event and time conditions allowed in guards. Event conditions are of the form $bv_{in}^i\sim bv_{in}^j$ where bv_{in}^i, $bv_{in}^j\in BV_{in}$ where $i\ne j$ and $\sim\in\{\&,|,\ \hat{}\ ,\ !\}$. And, time conditions are of the form $c==tc$ where $c\in C$ and $tc\in TC$,
  - *Sync* is a set of co-actions h? over channels h. Where $h\in H$ and H is the set of channel names assigned to elements of BV_{in}. Channels are only used in simulation. That is to synchronize between the automaton transformed from a state diagram and input variables automata. *Sync* is not considered a part of the transformation, and
  - *Update* is a set of sequences of assignment actions and clock resets. Actions are of the form $BV_{out}:=b$ where $b=0$ or 1 and clock resets are of the form $c:=0$.
- $V=V_{bool}\cup V_{int}$ where $V_{bool}=BV_{in}\cup BV_{out}$ and $V_{int}=TC$,
- C denotes the set of real-valued clocks. For every $tc\in TC$ there is a corresponding clock $c\in C$,
- *Init* is the set of assignments of the form $bv_{out}:=0$ where $bv_{out}\in Bv_{out}$,
- $Inv: L\to\mathcal{I}nv(C, V_{int})$ is a function of the form $c\le tc$ where $tc\in TC$, and
- $T_L: L\to\{o\}$.

Next part is considered a direct application to this approach.

6.3.1. Timed automaton in UPPAAL

Our goal is to apply this definition of timed automata to the state diagram described by PLCopen (2006). In other words, the aim is to transfer the state diagram to a timed automaton in UPPAAL. The *SF_Equivalent* state diagram described in Fig. 4 is taken as an example to illustrate this operation. This leads to the following results:

- L={*Idle, Init, Wait_for_A, Error1_Error2, Wait_for_B, Error3, From_Active_Wait, Safety_Output_Enabled*},
- l_0={*Idle*},
- V={*Activate_Eq, A, B, DT, Ready_Eq, S_EquivalentOut, Error_Eq*} where V_{int}={*DT*} denotes DiscrepancyTime,
- C={*D*} where *D* is a clock symbol representing *DT*,
- *Init*={*Ready:=0, S_EquivalentOut:=0, Error_Eq:=0*},
- *Inv*({*Wait_for_A*})=*Inv*({*Wait_for_B*})=*Inv*({*From_Active_Wait*})=$D\le DT$ and the invariant of all other locations is $D>0$,
- T_L for all locations is {*o*},

To define the set of edges $E=(l, Guard, Sync, Update, l')$ in the timed automaton of the safety function block, a part of the state of Fig. 4 and its corresponding timed automaton in UPPAAL are taken as illustration in Fig. 18.

In Fig. 18 left, the state diagram changes its state from Init to *Wait for Channel A* in case of a true *S_ChannelB* and false *S_ChannelA*. It keeps a wait state until either an expiration of specified discrepancy time, or a change in *S_ChannelA*/*S_ChannelB* status. Taking into account the priority number defined with every transition. In the right side of the figure, the clock *D* gets set to zero every time the system switches from location {*Init*} to location {*Wait_for_A*} on input event (*B &(!A)&Activate_Eq*).

The invariant ($D\le DT$) associated with the location {*Wait_for_A*} ensures that time can elapse in this location only as long as its invariant stays true. The value of D is checked on another transition from {*Wait_for_A*} to {*Error1_Error2*} for the constraint ($D==DT$). According to this description *Edge1* and *Edge2* can be represented by (*Init, B&(!A)&Activate_Eq*, $\emptyset$, *D:=0, Wait_for_A*) and (*Wait_for_A, (D==DT)&&Activate_Eq*, $\emptyset$, *Error_Eq:=0, Error1_Error2*),

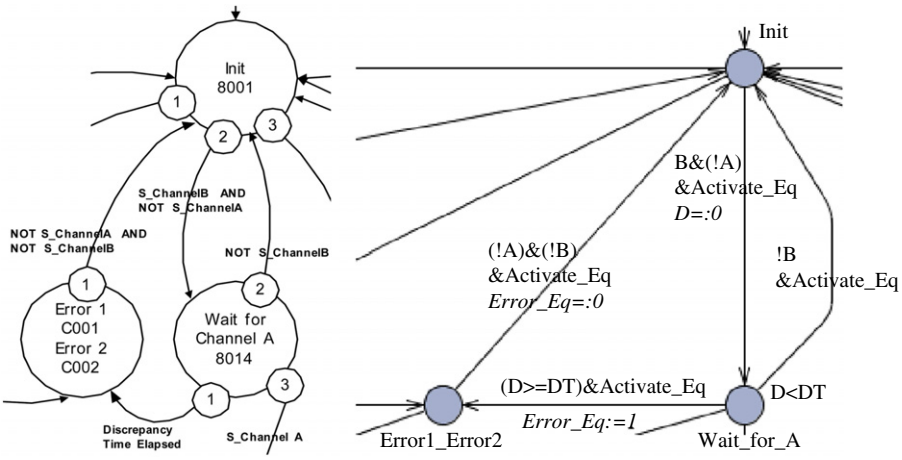

Fig. 18. A part of the state diagram of Fig. 4 (left) and its corresponding timed automaton (right).

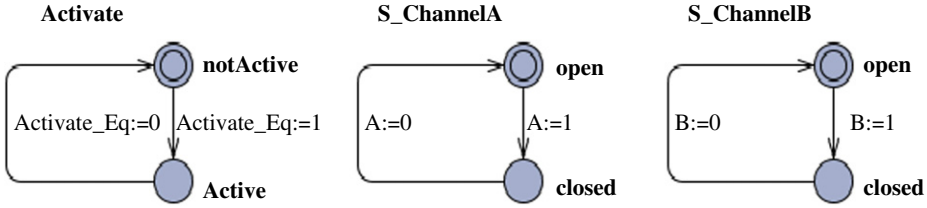

Fig. 19. Timed automata for *Activate S_ChannelA and S_ChannelB* input variables of SF_Equivalent.

respectively. The label (*Activate*) is ANDed to the event to tackle the priority problem. The other edges can be represented in the same way leading to a definition of the set of edges:

$E=\{edge1, edge2,\ldots,edgeN\}$ where N is the number of edges in the timed automaton.

However, there are two possibilities of state change in the previous timed automaton. The first is due to the elapse of time in a location (depends on the invariant) and the second is due to a true event within time constraint (depends on the guard). To explain this, some sample transitions in *SF_Equivalent* timed automaton are given:

Generally, the timed automaton starts in its initial location *Idle* at time 0 with all its clocks (actually one clock D) and output variables (BV_{out}) initialized to 0. According to the semantics of UPPAAL timed automata, the state defined as $s=(State\ label, D, Ready\_Eq, S\_EquivalentOut, Error\_Eq)$. As time advances, the value of the clock changes reflecting the elapsed time. In the above sample path, the clock D takes the values 0, $\tau1$, $\tau1+\tau2$, 0, DT, $DT+\tau4$ … in different states. Where, τi is an unspecified amount of time and $i=1, 2,\ldots$. The notation above the arrow represents the active condition on edge and the advance in time. The automaton stays in state *Idle* for undetermined amount of time $\tau1$ until the input *Activate_Eq* goes true. There, the automaton changes its state to Init after time $\tau2$. After time $\tau3$ at active input event *B&(!A)&Activate* the timed automaton changes its state from location *Init* to *Wait_for_A* and resets D to 0. At clock reset, it restarts counting time with respect to the time of reset. If the current value of the clock satisfies $D==DT$, and *Activate_Eq* is true, the state changes to *Error1_Error2* and the *Error_Eq* output goes true and so on.

6.3.2. System of timed automata in UPPAAL

A network of timed automata is built by combining several parallel automata synchronized with one another. This network is a product of timed automata as introduced in Alur (1999). The product construction is presented so that a complex system can be defined as a product of component systems.

In UPPAAL, a state of the system is defined by the locations of all automata, the clock constraints and the values of the discrete variables. Every automaton may fire an edge separately or synchronize with another automaton, which leads to a new state (Behrmann et al., 2004).

As an application to Definition 2 on *SF_Equivalent*, additional three timed automata templates are defined. Let A_1 be the timed automaton translated from the state diagram of *SF_Equivalent*. Additionally, A_2, A_3 and A_4 are timed automata for *Activate*, *S_ChannelA* and *S_ChannelB* input variables. All labels and clocks are declared globally in the system to allow communication between all templates.

Fig. 19 shows the three timed automata. Every one consists of two locations and two edges between them. It simply describes the unconditional true/false variation of the input variable. On every edge, a new value of the input variable can be assigned. Hence, *NA* is a timed automaton for the SFB *SF_Equivalent* where:

- $\overline{A}=(A_1,A_2,A_3,A_4)$,
- $\bar{l}_0=l_1^0\times l_2^0\times l_3^0\times l_4^0=\{A_1.Idle\}\times\{A_2.notActive\}\times\{A_3.open\}\times\{A_4.open\}$,
- $V'=V_1\cup V_2\cup V_3\cup V_4=V_1, C'=C, H=\emptyset$ and $Init'=Init$.

Actually, the synchronization here depends on shared variables which are declared globally. However, to allow simultaneous change to more than one edge in different automata, synchronization by sending messages should be programmed. This is achieved by defining channels between automata. A channel name can be declared by the data type chan name. It can be used on edges to send or receive a message as described next.

The size of a transition system is exponential in number of concurrent components. Thus, the number of component should be minimized to avoid state explosion. Therefore, the extra input variables automata are only used for simulation. At verification,

these input variables are specified within the formal property. However, in complex safety applications with large number of SFBs automata, another idea is used. We show later how to exploit some features of UPPAAL to attenuate the state explosion problem.

6.4. Temporal logic

Temporal logic is a form of logic specifically tailored for statements and reasoning which inform the notation of order in time (Bérard et al., 2001). It is used by model checkers to query whether a property is satisfied on the system or not. The main purpose of a model checker is to verify the model w.r.t. a requirement specification. Like the model, the requirement specification must be expressed in a formally well-defined and machine readable form. Several temporal logics exist for this purpose in the scientific literature, and UPPAAL uses a simplified version of Computation Tree Logic (CTL) (Behrmann et al., 2004).

6.4.1. Computation Tree Logic syntax and semantics

CTL, first produced by Emerson and Clarke (1980), is a branching temporal logic used in model checking. The formal syntax and semantics for CTL is given in Clarke, Emerson, and Sistla (1986). This syntax is given below. Let *AP* be the underling set of *atomic propositions*.

Every atomic proposition $a \in AP$ is a CTL formula.

- If ϕ_1 and ϕ_2 are CTL formulas, then so are $\neg\phi_1$, $\phi_1 \wedge \phi_2$, $AX\phi_i$, $EX\phi_i$, $AF\phi_2$, $EG\phi_1$, $A[\phi_1 U \phi_2]$ and $E[\phi_1 U \phi_2]$.
- The symbols $\neg$ and $\wedge$ are negation and conjunction. *A*, *E*, *X*, *U*, *F* and *G* are temporal operators for *all*, *exists*, *next*, *until*, *future* and *global*, respectively.

The semantics of CTL formulas are defined w.r.t a transition system associated to timed automata. The transition system is defined previously in Section 6.2. Formally, a CTL structure is a triple $M=(S, R, P)$, where:

- S is a finite set of states.
- R is a binary relation on $S(R \subseteq S \times S)$ which gives the possible transitions between states and must be total; that is $\forall x \in S \exists y \in S[(x,y) \in R]$.
- P: $S \rightarrow 2^{AP}$ assigns to each state the set of atomic propositions true in that state.

A *path* is an infinite sequence of states $(s_0, s_1, s_2, \ldots)$ such that $\forall i[(s_i, s_{i+1}) \in R]$. For any structure M and state $s_0 \in S$, there is an infinite computational tree with root labelled s_0 such that $s \rightarrow t$ is an arc in the tree iff $(s, t) \in R$.

6.4.2. Computation Tree Logic semantics of SF_Equivalent

In order to apply this definition to SF_Equivalent SFB, Fig. 12 is used to define the CTL structure. Let the set of atomic proposition *AP* is the *Input*/*Output* variables alphabet (V_{bool}) defined previously in Section 6.2, and moreover {*Discrepancy Time Elapsed*}. In other words.

- $AP = V_{bool} \cup \{Discrepancy\ Time\ Elapsed\}$ and
- $S=\{S\_1, S\_2, \ldots\ldots, S\_8\}$,
- R is the total transitions taken into account the transitions from any state to itself and the transitions from every state to the initial state.

To define *P*, sample examples of some states will be taken. In the initial state *S*_1, all input and output variables are false. Therefore, no atomic proposition is true in this state. In state *S*_2, the input *Activate* and the output *Ready* are true. So, $P_{s\_2}=\{Ready, Activate\}$. Finally, in state *S*_4, the inputs *Activate* and *S_ChannelB*, and the outputs *Ready* and *Error* are true. Moreover, {*Discrepancy Time Elapsed*} is also true. Hence, $P_{s\_4}=\{Activate, A\_Channel\ B, Ready, Error, Discrepancy\ Time\ Elapsed\}$, and so on.

According to this definition, the corresponding tree for the initial state *S*_1 is obtained and shown in Fig. 20.

6.4.3. CTL-formula and UPPAAL model checker

UPPAAL model checker can be used to determine the satisfaction of a given real time property with respect to a timed automaton. If the property is not satisfied, a diagnostic trace can be generated that indicates how the property may be violated. UPPAAL also provides a simulator that allows a graphical visualization of possible dynamic behaviours of a system description, i.e. a symbolic trace. The diagnostic trace, generated in case a property is violated, can be fed back to the simulator so that it can be analysed (Katoen, 1999). As shown in Fig. 21, depending on timed automata constructed from the state diagram and CTL-formulas formalized from textual properties, the satisfaction of these properties can be determined.

Table 1 shows CTL formulas supported by UPPAAL and their classifications. Like in CTL, the query language consists of path formulas and state formulas. State formulas describe individual states, whereas path formulas quantify over paths or traces of the model. Path formulas can be classified into reachability, safety and liveness. For example, the property $E<>p \equiv E\ (true\ U\ p)$ is a

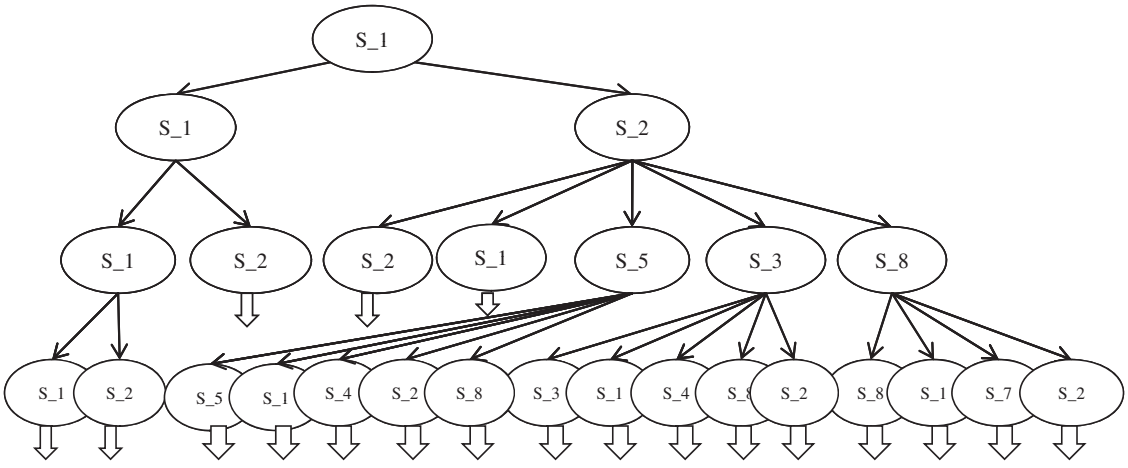

Fig. 20. A corresponding computation tree for *SF_Equivalent* SFB.

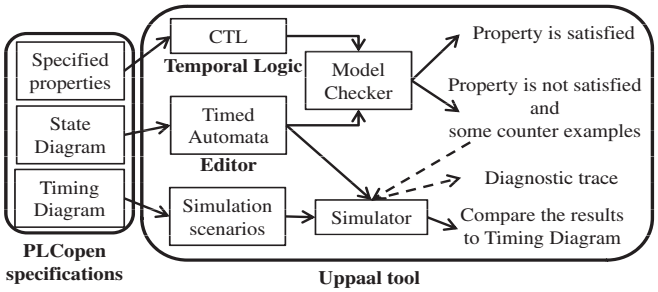

Fig. 21. Model checking and simulation in UPPAAL.

Table 1
Properties supported by UPPAAL.

Name	UPPAAL property	CTL operators	Path formula classification
Possibly	$E<>p$	EF p	Reachability
Invariantly	$A[]p$	AG p	Safety
Potentially always	$E[]p$	EF p	Safety
Eventually	$A<>p$	AF p	Liveness
Leads to	$p-->q$	AG (p imply AF q)	Liveness

path formula, where p is a state formula. This property is pronounced "p holes potentially". And, it means, starting from the initial state there exist some paths where p eventually holds. UPPAAL additionally defines a logical operator called *imply*, where p *imply* q is equivalent to $\neg pvq$.

The formalization of the textual property " if *Activate* input turns true, *Ready* output turns eventually true" into CTL formula is *AG* (*Activate imply AF Ready*). It can be expressed in UPPAAL as liveness property as follows:

Activate—> Ready.
: However, it is known, from state diagram in Fig. 4, that *Activate* and *Ready* state formulas (atomic propositions), hold in all states unless the initial state. Hence, we can assume that property as safety one, and express it as follows:

A[] Activate imply Ready.
: In the next section, the formalization of the textual properties, expressed by PLCopen, into UPPAAL CTL-formulas are described.

6.5. Approach for formalization of SFBs in UPPAAL

As mentioned previously, UPPAAL allows analysis of networks of timed automata with binary synchronization. It contains three main parts. First, timed processes are described using a *graphical editor*. Second, systems can be simulated by a *graphical simulator*. Finally, specified properties can be verified using *the verifier*.

To formalize an SFB, the process depicted in Fig. 21 is followed. First, the state diagram is transferred to a timed automaton using the graphical editor of UPPAAL. Depending on this timed automaton and the input sequence scenario extracted from the timing diagram, a simulation is executed to validate the temporal behaviour using the graphical simulator. Thereafter, the textual properties are translated into temporal logic formulae. Using the verifier, the properties of the safety function can be verified.

Actually, the use of symbolic model checking in combination with timed automata would be sufficient to verify the behaviour of the formalized SFBs. However, the additional use of simulation is useful for two distinct reasons. First, to validate specific scenarios presented in the specification it is the more direct approach. Second, and much more important, it allows users not familiar with formal models to assess the results directly.

In the frame of the presented work, all twenty SFBs defined by PLCopen have been formalized according to this procedure. As an example, the following subsection describes the process in detail for the SFB *SF_Equivalent*.

6.5.1. Formalization of SF_Equivalent

To formalize *SF_Equivalent* the general procedures shown in Fig. 22 are followed.

First of all, the Input/Output variables are declared as shown in Fig. 23. These variables are declared globally to allow sharing by other automata.

Second, the state diagram of the SFB is graphically translated to the state transition timed automaton shown in Fig. 24. Additionally, the three automata shown in Fig. 19 of the Boolean input variables are constructed to give the facility to vary these inputs while implementing the simulation.

As it appears in the state diagram of Fig. 4, the outgoing transitions from every state are numbered according to priorities where 0 is the highest priority. In translating the state diagram of PLCopen in a timed automaton in UPPAAL the problem of covering priorities arises. However this problem can be easily solved by extending the transition condition of a low priority condition with the negation of a high priority transition. A detailed analysis

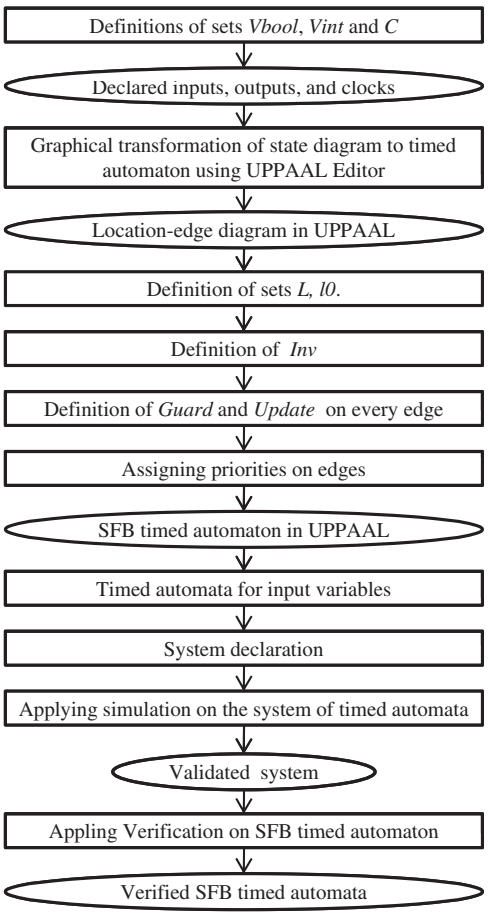

Fig. 22. Detailed procedures for modelling, validation and verification of SFB.

performed for all SFBs shows that this extension is not always necessary.

Prior to simulation, one should be sure that the output variables vary correctly with every state location of the SFB automaton. This can be achieved by querying about the status of these variables in every state using the verifier. For example, in location *Error*3, both *Ready* and *Error* are true and *S_EquivalentOut* is false. This safety property is converted to a CTL formula in UPPAAL as follows:

A[] SF_Equivalent.Error1_Error2 imply *(Ready_Eq)&(!S_EquivalentOut)&(Error_Eq)*

Which means that, starting from the initial state of the computation tree, for all paths globally, if *Error*1_*Error*2 state is reached then [(*Ready_Eq*) & (*!S_EquivalentOut*) & (*Error_Eq*)] becomes true in the current and future states of the computation tree. Another query which is, *is it possible for Error output variable to be true outside the two error states Error*1_*Error*2 *and Error* 3*?* The answer must be *no*. If we translate this query to CTL it becomes:

E < > !(SF_Equivalent.Error1_Error2|SF_Equivalent.Error3) & Error_Eq

This property is not satisfied assuring the expected result. Similar quires about the output variables status in all stated are verified. In that way, the programming errors can be found and fixed. Next to this pre verification process, the simulation can be begun.

Thirdly, using the automata (Fig. 24), simulation in UPPAAL validated the timing diagrams from the PLCopen specification (Fig. 5). Fig. 25 shows the response of *SF_Equivalent* automaton to the change in input automata.

```
// VAR_INPUT
 bool S_ChannelA, S_ChannelB, Activate_Eq;
 clock D;                 //DiscrepancyTime clock
 const int DT=10;     //DiscrepancyTime (10ms)
 //VAR_OUTPUT
 bool Ready_Eq, S_EquivalentOut, Error_Eq;
```

Fig. 23. Declarations of Input/Output variable and clocks.

To clear the difference between synchronization by shared variables (Fig. 25) and by message passing, channels between input automata and *SF_Equivalent* automaton are declared as follows:

chan Aclosed, Aopen, Bclosed, Bopen; and
broadcast chan Activate_Eq, notActivate_Eq.

The simulation results after channels addition is shown in Fig. 26. After simulation, deadlock-freeness of the system is checked with *A[]* not deadlock.

Fourthly, the accordance of the automaton to the textual functional description is checked by verification. Taking a part of this description mentioned above, for example, *"If one channel signal changes from TRUE to FALSE the output immediately switches off* (*FALSE*) *for safety reasons*". A corresponding safety CTL formula is derived:

A [] S_EquivalentOut & (A^B) imply *!S_EquivalentOut* where ^ is the logic operator *XOR*.

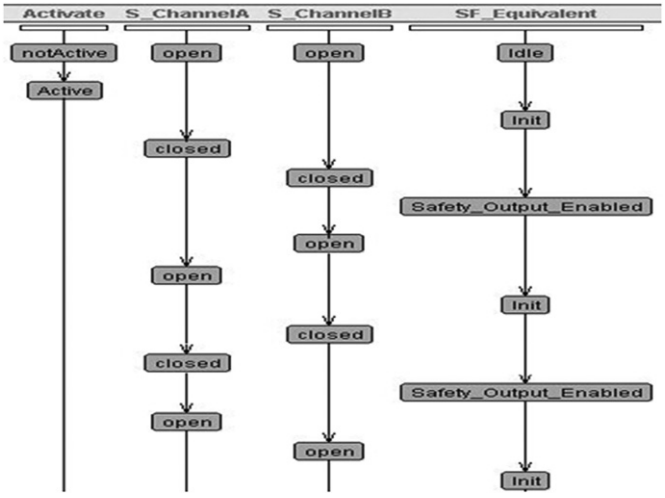

Fig. 25. Simulation scenarios extracted from timing diagram in Fig. 5 with shared variables synchronization between automata.

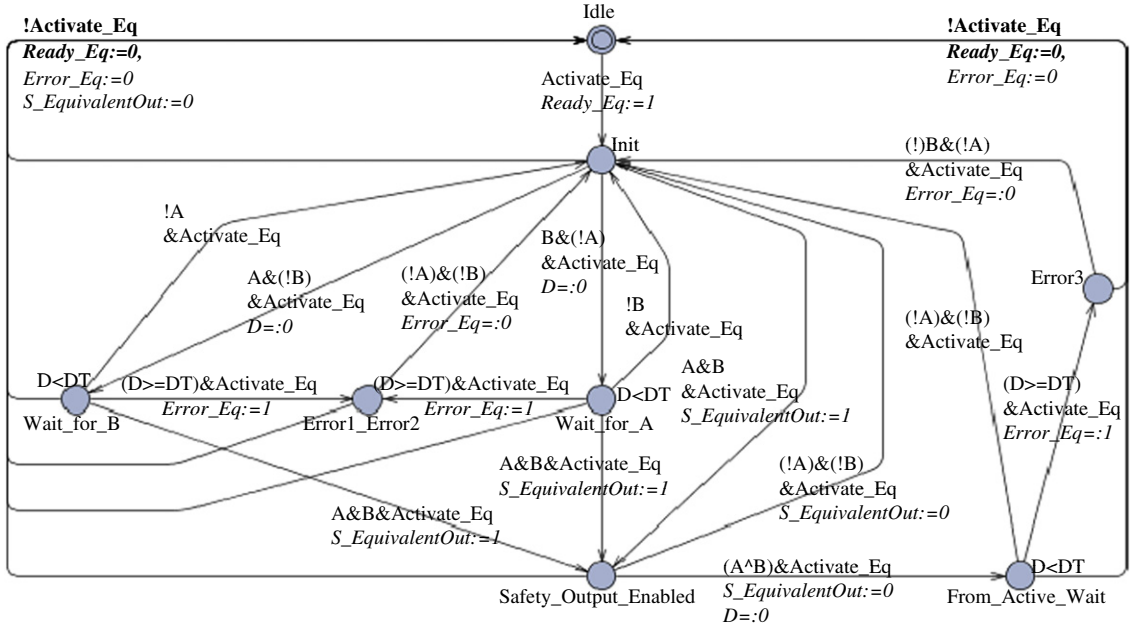

Fig. 24. UPPAAL timed automaton for state diagram in Fig. 4.

Another property is "the function block detects an error when both inputs do not have the same status once the discrepancy time has elapsed".

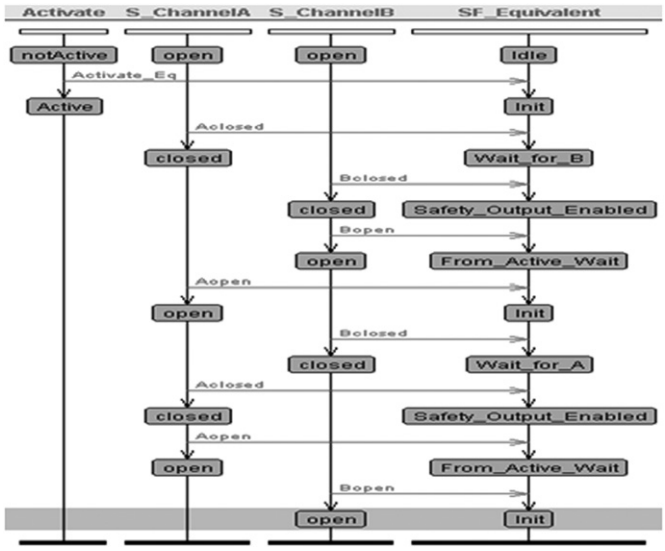

Fig. 26. Simulation scenarios extracted from timing diagram in Fig. 5 with message passing synchronization between automata.

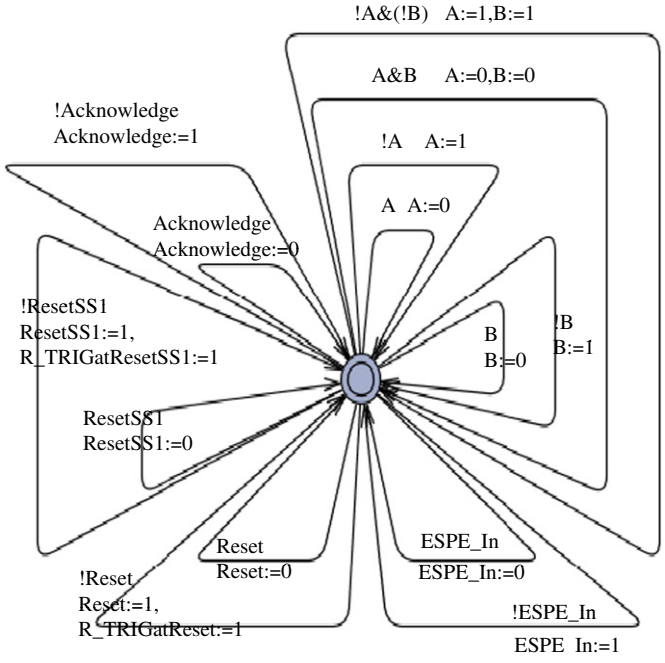

Fig. 27. Timed automaton for input variables of safety applications.

A[] (A^B) && (D >=DT) imply *Error_Eq*

These properties and all other textual properties are satisfied assuring the functionality of the SFB.

7. Verification of a safety application

The verification approach is now applied using the models introduced in the previous section to an example taken from the second part of the PLCopen (2008) specification.

The UPPAAL tool has the ability to import constructed timed automata and combine them in a new system. Hence, the four SFB and the required input timed automaton are imported easily in one system. The next step is to achieve the connections according to Fig. 6.

7.1. Input automaton

To avoid complexity which leads to state explosion, only one input timed automata for all inputs and reset signals is used. As shown in Fig. 27, it consists of one initial location and many self edges. Initially, two edges are needed for every Boolean input variable. Since the system is built to respond directly to a single change in input variable, it is not possible for simultaneous changes in input variables. To overcome this drawback, extra edges are added for simultaneous changes in more than one variable. For example, when both input channels of *SF_Equivalent* change their status simultaneously.

7.2. Connection automata

To avoid internal modification inside every SFB timed automaton, the connections are achieved by new timed automata. As shown in Fig. 6, there are two direct connectors from *SF_Equivalent* to *SF_EmergencyStop* (C1, C2). And, another two indirect connectors from *SF_EmergencyStop* and *SF_ESPE* to *SF_SafeStop*1 (C3, C4). Fig. 28 shows these four connectors timed automata.

Every connector timed automaton is constructed from one initial-urgent location and one self-loop-edge. The edge coming out from an urgent location can have discrete guards and arbitrary updates, but no guards over clocks. This is because, the time is not allowed to pass in urgent locations. In other words, the clocks in all SFB automata are not affected by the execution of connectors' automata.

The guard on the edge of one connector automaton is activated only when variables on both terminals of the connectors are unequal. In case of inequality, the variable on the output terminal is updated. Taking the first connection automaton as an example, if the guard *Ready_Eq* changes, then the guard *Ready_Eq XOR Activate_EStop* goes true and the edge is fired. This leads to updating *Activate_Estop*, which achieves the connection.

7.3. Variable evaluation and execution flow

And now, there is a question which might arise: Is that timed automata model in UPPAAL really representing the safety

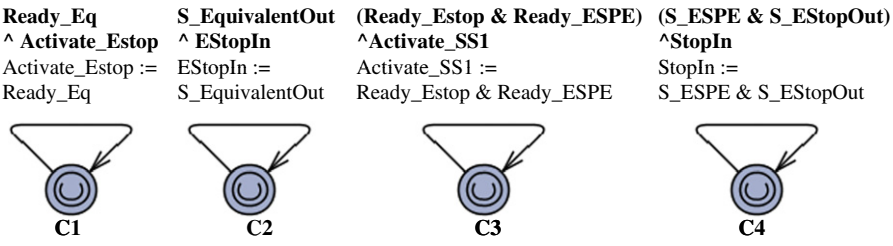

Fig. 28. Timed automata for connectors between SFBs.

application implemented in FBD? To answer this question, one should first represent what is stated in literature about programme variables evaluation and execution flow in both SFB and UPPAAL.

As stated in John and Tiegelkamp (2001), a FBD programme is evaluated FB by FB, from top to bottom. A single FB is evaluated by the following rules:

- Evaluate all inputs of a FB element before executing the element.
- The evaluation of a FB element is not complete until all outputs of this element are evaluated.
- The evaluation of a FBD programme is not complete until all outputs of all elements of the network are evaluated.

Moreover, the sequence of evaluation is carried out from left to right. That is because the input variables are designed to be at left hand side and the output variables are at right. However, a feedback data flow is also possible taking into account the default initial value for first time evaluation.

According to the evaluation process defined above, the execution flow of the defined safety application in Fig. 6 and its corresponding implementation in Fig. 16 is restricted. The execution flow should start at *SF_Equivalent* until all possible active transitions are executed, then *SF_ESPE*, *C*1, *C*2, *SF_EmergencyStop*, top AND function, down AND function and lastly *SF_SafeStop*1.

Actually, UPPAAL has another concept different from execution flow evaluated above. As stated in Behrmann et al. (2004), a system is modelled in UPPAAL as a network of several timed automata in parallel. The model is further extended with bounded discrete variables that are part of the state. These variables are used as in programming languages: they are read, written and are subject to common arithmetic operations. A state of the system is defined by the locations of all automata, the clock constraint and the values of the discrete variables. Every automaton may fire an edge separately or synchronized with another automaton, which leads to a new state.

That means if we import the SFBs and connectors automata to construct a new system, the simulation is carried out with randomly execution flow. However, the simulation process shows that it leads finally to the expected state. Instead of random execution flow in UPPAAL, a controlled execution flow can be used. That can be achieved by assigning priorities to automata of the system. Automata priorities are specified on the system using the separator '$<$' to define a higher priority for automata to its right. The automata priorities are defined as follows:

system *SF_SafeStop1* $<$ *C4* $<$ *C3* $<$ *SF_EmergencyStop* $<$ *C2* $<$ *C1* $<$ *SF_ESPE* $<$ *SF_Equivalent*.

7.4. Validation

To validate that the system is acting like the one introduced in FBD programme, some simulation scenarios are applied to the system of timed automata. This scenario is extracted from textual description of safety functions which are carried out by this safety application. The *Inputs* automaton shown in Fig. 27 is used to assign inputs to the safety application. According to the evaluation process stated in John and Tiegelkamp (2001), *Inputs* automaton should take the highest priority in execution, but that is not possible. This is because *Inputs* automaton runs unconditionally. In other words, it has always active transitions, which means no lower priority automaton gets the chance for execution. To tackle this problem, *Inputs* automaton is given the lowest priority taking into account the initial conditions needed to start execution.

As an example for simulation scenario extracted from text the next text is given. "*The safety application is in normal operation state if all safety input variables are true, taking into account the required activation and reset input variables*". The simulation scenario describing that textual specification is shown in Fig. 29. This simulation scenario is obtained by automatic simulation mode to validate the execution flow of timed automata system. Actually, the *Inputs* automaton is not used in this simulation to save space. Instead, input variables status required are declared as initial values.

Starting from *SF_Equivalent* at right side of Fig. 29, the timed automata are executed sequentially with respect to priorities until reaching *Operation_Mode* state in *SF_SafeStop*1.

According to simulation results, it is shown that the system is functioning as expected in PLCopen (2008).

7.5. Verification

Some of the verified properties are as following:

Property 1. *The safety application is in normal operation state if all safety input variables are true, taking into account the required activation and reset input variables.*

A[] Activate_Eq & Activate_ESPE & A & B & ESPE_In & R_TRIGatReset & R_TRIGatResetSS1 imply SF_SafeStop1.Operation_Mode (Property is satisfied).

Property 1 is describing the safety function which validated above in Fig. 29. It is satisfied assuring the functionality of this property.

Property 2. *After the activation of SF_Equivalent and SF_ESPE, the SF_SafeStop will keep activated.*

A[] Activate_Eq &Activate_ESPE imply Ready_SS1	(Property is satisfied)
E$<>$ ((!Activate_Eq)|(!Activate_ESPE)) & Ready_SS1	(Property is not satisfied)

The first CTL formula in Property 2 is satisfied which means that, for all paths of the computation tree, if the inputs *Activate_Eq* and *Active_ESPE* are true, then the output *Ready_SS*1 will be always true. The second one is not satisfied assuring that the output *Ready_SS*1 never be true if one of the activation inputs is false.

Property 3. *Issuing the Emergency stop (via SF_EmergencyStop) or interrupting the light beam in the light curtain (via SF_ESPE) stops the drive in accordance to stop category 1.*

A[] ((!A)|(!B)|(!ESPE_In)) & Acknowledge imply S_Stopped	(Property is satisfied)
E$<>$ S_Stopped & !((A&B)|ESPE_In)	(Property is not satisfied)

That means, if there is a safety request to stop the drive by missing one of the safety inputs (*A*, *B* or *ESPE_In*), and the control system gives acknowledgement of stopped drive, then a safety output *S_Stopped* is issued to indicate safety stop. The other one indicates that it is not possible for the drive to stop while no safety request is needed.

Property 4. *The stop of the electrical drive within a predefined time is monitored (via SF_SafeStop1).*

A [] (!StopIn) & Acknowledge && (T$<$MT) imply S_Stopped	(Property is satisfied)
E$<>$ (!StopIn) & (!Acknowledge) imply S_Stopped	(Property is not satisfied)

That means, if an acknowledgement signal (Acknowledge$=$1) is received within specified monitoring time ($T<MT$), a safety stop is monitored (*S_Stopped*$=$1).

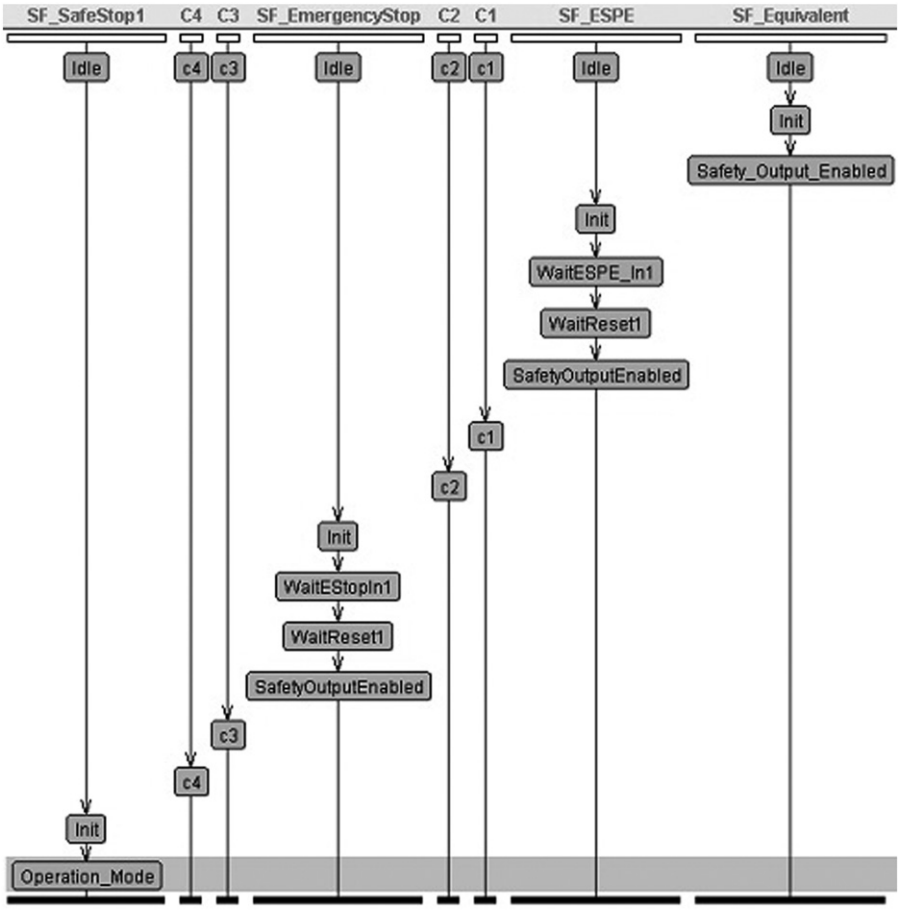

Fig. 29. Simulation scenario of the safety application.

All other properties are verified in the same manner. As conclusion to the verification process of this safety application, it is said to be working as expected.

7.6. State explosion problem

As stated by Henzinger (1992), the main practical limitation of model checking algorithms is caused by the size of the state graph, which growth exponentially with the number of parallel component in a system. So, to reduce the complexity of a system composed of many parallel processes in UPPAAL, system priority is suggested.

To authors' knowledge, using system priority in that way in UPPAAL attenuates the state explosion problem. This is because only one automaton is executed at a time. However, there are two drawbacks in using system priority. Firstly, using input variables automata is not possible in according with execution flow defined in FBD. To overcome this drawback, the input variables are defined within the verified property. Secondly, channel synchronization is not allowed with system priority.

7.7. Performance of the verification approach

In general it has to be noted thus safety applications do not face as many complexity problems as, e.g. control applications since they are normally much smaller and dedicated to provide single specific functions. The verification approach presented here for a safety application is achieved in two steps. The first step is local verification of the application components. These components are SFBs and connections, which are verified previously. The goal is to check properties of the components first. Therefore, we can assume that the internal structure of the components is correct, before we go on to verify global properties in the second step. This means that problems detected in system verification are only related to the interconnection of SFBs. Which in turn leads to an easier localization of errors.

8. Tasks of transformation tool

As a result to this work, there are two SFB libraries available. The first library of implemented SFBs in accordance with PLCopen is used in implementing a safety application in FBD. This safety application is exported to PLCopen XML format as shown in Fig. 30.

The result PLCopen XML is the required interface to the transformation tool. This XML file contains all the information needed to transform it to UPPAAL XML format.

The transformation tool – currently under construction – has four tasks. The first one is to transform the Input/output variables declaration of the safety application to global declaration in UPPAAL XML. The second task is achieved using the library verified previously in UPPAAL. It is to transform all SFBs presented in PLCopen XML to their corresponding automata in UPPAAL XML. The third task is to formalize a connection automaton for every direct or indirect connection between SFBs. According to the position of every SFB and the order of every connection, the execution flow is extracted from PLCopen XML. Depending on this execution flow the system priority is.

After transformation, UPPAAL XML file can be read by UPPAAL tool. Whenever, simulation and verification process started taking into account the specification of safety application.

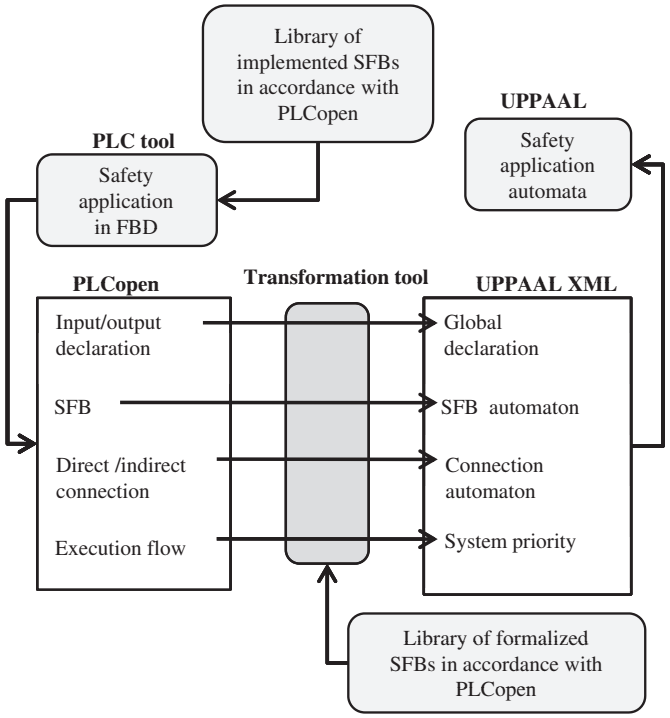

Fig. 30. Transformation tool tasks.

9. Summary and outlook

The major contribution of this paper is to formalize an approach for the verification of safety applications built from PLCopen safety function blocks (SFBs). This approach was presented and illustrated using an example. It has to be made clear, that the presented approach relies on the assumption that the implementation of the SFBs actually follows the specification of PLCopen in all details. This assumption is normally verified by certification of SFBs by independent organizations like TÜV. In the presented work for test purposes SFBs without certification have been used. The SFBs have been implemented directly in IL and verified using Model Checking in Biallas et al. (2010).

Future work in this project will concentrate on automating the model building step. Based on the PLCopen XML interchange format for PLC applications the formal model will be built from the pre-defined automata automatically. Given an automated process the application to more complex applications in co-operation with practitioners is planned. Another open question is the needed support in property formulation. Currently, the formalization of properties for model checking can only be done be experienced specialists. To tackle this problem specification methods commonly known in the safety domain such as for example cause and effect charts will be evaluated during the practical application of the verification tool for their potential to be used as intermediate formalization step.

References

Alur, R., & Dill, D. (1991). A theory of timed automata. In *Proceedings of the REX workshop "Realtime theory in practice"*.

Alur, R., & Dill, D. (1994). Automata for modelling real-time systems. *Theoritical Computer Science*, *126*(2), 183–236.

Alur, R. (1999). Timed automata. In *Proceedings of the 11th International conference on computer-aided verificatio*n, LNCS 1633 (pp. 8–22), Springer-Verlag.

Bani Younis, M., & Frey, G. (2003). Formalisation of existing PLC programs: a survey. In *Proceedings of CESA*, Lille, France.

Behrmann, G., David, A., & Larsen, G. (2004). A tutorial on UPPAAL, UPPAAL web site, Documentation, Tutorials.

Bell, R. (2005). Introduction to IEC 61508. In *ACS workshop on tools and standards Sydeny. Conference in research and practice in information technology*, UK Crown, Vol. 55.

Bérard, B., Bidoit, M., Finkel, A., Laroussinie, A., Petit, A., & Petrucci, L., et al. (2001). *Systems and software verification, model-checking techniques and tools*. Germany: Springer.

Biallas, S., Frey, G., Kowalewski, S., Soliman, D., & Schlich, B. (2010). Formale Verifikation von Sicherheits-Funktionsbausteinen der PLCopen auf Modell- und Code-Ebene. In *Proceedings of the 11th Fachtagung Entwurf komplexer Automatisierungssysteme (EKA 2010)*, Magdeburg, May 2010.

Bortnik, E. M., Fronczak, J. M., & Rooda, J. E. (2007). *Translating X models to UPPAAL timed automata. SE Report*. The Netherlands: Eindhoven University of Technology.

Clarke, E., Emerson, E., & Sistla, A. (1986). Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, *8*(2), 244–263.

Emerson, E., & Clarke, E. (1980). Characterizing correctness properties of parallel programs using fixpoints. In *Proceedings of the seventh international colloquium on automata, languages and programming* (pp. 169-181). New York: Springer Verlag.

Frey, G. (2000). Automatic implementation of petri net based control algorithms on PLC. In *Proceedings of the American control conference ACC 2000* (pp. 2819–2823), Chicago (IL).

Thomas A. Henzinger (1992). Symbolic model checking for real-time systems. In *Proceedings of the seventh annual IEEE symposium on logic in computer Science (LICS)* (pp. 394–406), USA.

John, K. & Tiegelkamp, M. (2001). IEC 61131-3: Programming Industrial Automation System: Concepts and Programming Languages, Requirements for Programming Systems, Aids to Decision-Making Tools.

Katoen, J. (1999). *Concepts, algorithms, and tools for model checking*. Lecture Notes in Mechanised Validation of Parallel Systems, Nurnberg, Germany.

Larsen, K., Pettersson, P., & Yi, W. (1997). UPPAAL in a nutshell. *International Journal of Software Tools for Technology Transfer*, *1*, 134–153.

PLCopen (2006). *Part 1: Concepts and Function Blocks. TC5, Safety Software Technical Specification, Version1.0*. Germany: PLCopen.

PLCopen (2008). *Part 2: User Examples. TC5, Safety Software Technical Specification, Version1.0*. Germany: PLCopen.

Soliman, D., & Frey, G. (2009). Verification and validation of safety applications based on PLCopen safety function blocks using timed automata in UPPAAL. In *Proceedings of the second IFAC workshop on dependable control of discrete systems (DCDS)* (pp. 39–44), Bari, Italy.

Song, M., Koo, S., & Seong, P. (2004). Development of a verification method for timed function blocks using ESDT and SMV. In *Proceedings of the IEEE international symposium HASE'04*.

Völker, N., & Krämer, B. (2001). Automated verification of function block-based industrial control systems. *Science of Computer*, *42*, 101–113.