A graph based semantics for Logical Functional Diagrams

Aziz Sfar, Dina Irofti, Madalina Croitoru

A graph based semantics for Logical Functional Diagrams

Aziz Sfar[1,2], Dina Irofti[2], and Madalina Croitoru[1]

[1] GraphIK, INRIA, LIRMM, CNRS and University of Montpellier, France
[2] PRISME Departement, EDF R&D, France

Abstract. In this paper we place ourselves in the setting of formal representation of functional specifications given in logical diagrams (LD) for verification and test purposes. Our contribution consists in defining a formal structure that explicitly encodes the semantics and behavior of a LD. We put in a complete transformation procedure of the non-formal LD specifications into a directed state graph such that properties like oscillatory behavior become formally verifiable on LDs. We motivate and illustrate our approach with a scenario inspired from a real world power plant specification.

Keywords: System Validation · Functional Specifications · Logic Functional Diagram · Graph based Knowledge Representation and Reasoning.

1 Introduction

A power plant is a complex system and its functional behavior is described, for each of its subsystems, using logical diagrams. The logical diagrams are coded and uploaded into the controllers. During the power plant life-cycle (around 60 years and even more), the controllers' code needs to be updated and verified. Engineers generate scenarios in order to verify the new code. However, the scenarios generation is far from being a simple procedure because of the system's complexity. Indeed, the power plant contains a few hundred subsystems, and the behavior of each subsystem is described in a few hundred pages of logical diagrams. Knowing that a logical diagram page contains on average 10 logic blocks, a quick calculation shows that a power plant behavior can be described by a few hundred of thousands of logic blocks. Another nontrivial problem for scenario generation for such systems is caused by the loops existing between the logic blocks, i.e. the input of some logic blocks depends on their outputs.

Logical diagram specifications lack the formal semantics that allow the use of formal methods for properties verification and test scenarios generation. Done through manual procedures, these tasks are tedious. In this paper, we tackle the problem of lack of semantics of logical diagram specifications. To solve this problem, we propose a formal graph model called the Sequential Graph of State/Transition ($SGST$) and we define a transformation method of logical diagrams into the proposed graph. On the $SGST$, we show how to formally verify that the functional behavior described by the logical diagram specification is deterministic. In fact, the specification model is supposed to provide a

description of the expected behavior of the controller. If the expected behavior itself is non-deterministic, then test generation based on that behavior does not make sense. This problem can be generated by the presence of loop structures in the logical diagrams that may prevent the behavior (i.e the expected outputs) from converging. The convergence property has to be verified before getting to test generation. Verifying this property directly on the logical diagram, which is a mix of logical blocks and connections presented in a non-formal diagram, is not easy to achieve. This task is possible in theory, as the logical diagrams can be reduced to combinatorial circuits. In literature, a combinatorial circuit [6] is a collection of logic gates for which the outputs can be written as Boolean functions of the inputs. In [10] it is shown that a cyclic circuit can be combinatorial, and a method based on binary decision diagrams is proposed to obtain the truth table of the circuit. The problem of where to cut the loops in the circuits and how to solve this loops has also been addressed in other studies [11], and applied in particular on the Esterel synchronous programming language [5]. Another algorithm for analysis cyclic circuits based on minimising the set of input assignments to cover all the combinatorial circuit has been proposed in [7]. Identifying oscillatory behavior due the combinatorial loops in the circuit has also been studied (see [2] and references therein). However, all studies cited here are mainly based on simulation rather than formal verification on models. The focus of these works is entirely dedicated to the verification of the cyclic behavior of the circuits and not to test purposes. Yet, several studies have already been published for the matter of both formal properties verification and test sequences generation. For instance, in their survey [4] Lee and Yannakakis address the techniques and challenges of black box tests derived from design specifications given in the form of finite state machines (Mealy machines). In [12], the author extends the test sequences generation to timed state machines inspired from the theory of timed automata [1]. These results and many others (such as formal verification of properties [8]) are applicable on state/transition graphs and can by no means be directly used on logical diagrams. In order to take advantage of the already established techniques, we focus our study on transforming logical diagrams into formal state graphs. Prvosot [9], has proposed transformation procedures of Grafcet specifications into Mealy machines, allowing the application of the previously mentioned formal methods of verification and test generation. However, Grafcets and logical diagrams are completely different representation models. A model transformation of logical diagrams into state graphs has been conducted by Electricté de France (EDF) [3] for cyclic behavior verification purposes. We inspired our work from both [9] and [3] to develop a formal state graph representation of the exhaustive behavior encoded in the logical diagram, the $SGST$. The proposed graph allows the verification of the cyclic behavior (called convergence in this paper) and potentially the formal verification of other properties. It also provides the ground to obtain the equivalent Mealy machine on which the existing formal test generation results can be applied.

This paper is organized as follows. The second section introduces the Logical Di-

agram specification with an example. A formal definition of the proposed $SGST$ is given in Section 3. Section 4 details the model transformation procedure from logical diagrams to $SGST$ graphs. In Section 5 we show how the behavior convergence property could be formally verified on the $SGST$. A discussion and a conclusion are given in the last section.

2 Motivating example and preliminary notions

2.1 Logical diagrams

Logical diagrams are specification models used to describe control functions in power plants. They contain a number of interconnected logic blocks that define how a system should behave under a set of input values.

Figure 1 illustrates a logical diagram extracted from a larger real world controller's logic specification in a power plant. It has five inputs (denoted by i_1 to i_5), one output (denoted o_1) and logic blocks: either blocks corresponding to logic gates or status blocks (corresponding to memory and on-delay timer blocks described below). The gates in Figure 1 are: two NOT gates followed by two AND gates and two OR gates. They correspond to the conjunction ($\cdot$), disjunction ($+$), and negation ($\bar{}$) Boolean operators, respectively (e.g. the output of an OR gate with two inputs is equal to 0 if and only if both inputs are equal to 0 etc.). The on-delay timer block gives the value 1 at its output if its input maintains the value 1 for 2 seconds; 2 seconds being the characteristic delay θ of the timer shown in the T block in Figure 1. The memory block is a set (E) /reset (H) block: if the E input is equal to 1, then the output is equal to 1; if the H input is equal to 1, then the output of the block is 0. If both E and H inputs are equal to 1, the output is equal to 0 since the memory in this example gives priority to the reset H over the set E. This priority is indicated in the block symbol by the letter p. A 0 at both inputs keeps the output of the memory block at the same last given value.

The timer and the memory are blocks whose outputs not only depend on the values at their inputs, but also on their last memorized status. In this paper we call them ***status blocks***. Each of them possesses a finite set of status values and evolves between them. A status block output value {0,1} is associated to each possible status. In the case of the example of Figure 1, the memory block M_2 has two possible status values M_1 and M_0 where the status M_1 gives a logic value of 1 at the output of the block M_2 and M_0 status corresponds to the logic value 0. The on-delay timer block T_1 has 3 statuses denoted TD_0, TI_0 and TA_1, where the associated block output values are 0, 0 and 1, respectively. We also note on this example the presence of a loop structure (containing the block T_1, an OR block and the memory block M_2).

More formally, a logical diagram specification is composed of I, the set of inputs of the diagram, O, the set of outputs of the diagram and B, the set of the logic blocks of the diagram. The **logic blocks** B connect the outputs O to the inputs I and define the function that relates them. $B = B^S \cup B^{LG}$, namely:

- the logic gates B^{LG}: these are the AND, OR and NOT blocks in the diagram. Each of them is equivalent to a Boolean expression over its entries using the Boolean operators ($+$), ($.$) and ($\bar{}$) for AND, OR and NOT, respectively.

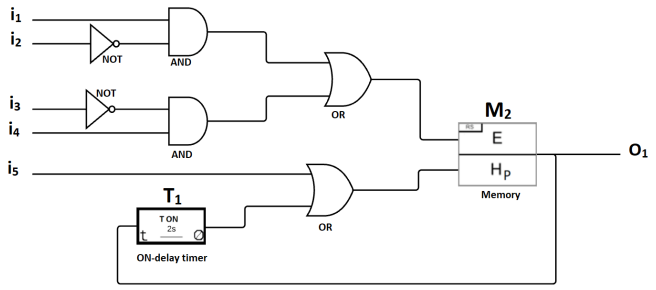

Fig. 1. Example of a logical diagram specification.

– the status blocks B^S: these are blocks that have a status that evolves between a set of values. The evolution of a status of a block $b^s \in B^S$ depends on the values at its entries and the last value of its status. A logic value at the output of the status block is associated to each of these status values.

Definition 1 (P_{status} set). *We denote by P_{status} the set of all the possible status values of the blocks B^S of the logical diagram.*

We note that the status values that a block $b^s \in B^S$ can take are in a subset of P_{status}. Some insights are given in the following example.

Example 1 (Illustration of P_{status} set on the motivating example given in Figure 1). For an on-delay timer status block (such as the block denoted T_1 in Figure 1), $P^{T_{ON}}_{status} = \{TD\_0, TI\_0, TA\_1\}$; for a memory status block (such as M_2 block in Figure 1), $P^{M}_{status} = \{M\_0, M\_1\}$. The associated block output logic value of a status value 'S_X' is indicated in its name by the numeric 'X'. The P_{status} set for the example illustrated in Figure 1 is $P_{status} = P^{T_{ON}}_{status} \cup P^{M}_{status}$.

Logic variables $Vars$. Logic gates B^{LG} in the diagram can be developed into Boolean expressions over logic variables $Vars$ by substituting them with their equivalent Boolean operator. Basically, we end up having outputs O and entries of B^S blocks that are equal to Boolean expressions on $Vars$.

Definition 2 ($Vars$ and Exp_{Vars} sets). *We define $Vars = I \cup O_{B^S}$ by the set of logic variables, that includes I, the set of input variables of the logical diagram and O_{B^S}: the set of output variables of status blocks B^s in the diagram.*

We denote by Exp_{Vars} the infinite set of all possible Boolean expressions on logic variables in $Vars$. For example, $(o_{b^s_k} + i_k) \in Exp_{Vars}$. In the reminder of this paper, we will use the following mathematical notations on sets. Let A be a set of elements:

$A^k = \overbrace{A \times A \times ... \times A}^{k}$ is the set of all ordered k-tuples of elements of A. Given e =(a_1,...,a_k)$\in$ A^k, we denote $e(i)$ the i^{th} element of e, i.e. $e(i) = a_i$. Given e =(a_1,...,a_k)$\in$ A^k, we denote $ord_e(a_k) = k$ the order k of a_k in e.

2.2 Test generation for logical diagrams

Let us explain how these diagrams are supposed to be read and subsequently implemented in a physical system (i.e. the logic controller[3]). The diagrams are evaluated in evaluation cycles repeated periodically. Within each evaluation cycle the status blocks B^S are sequentially evaluated in accordance to a defined order ω while logic gates are evaluated from left to right.
The logic specification diagrams are implemented using a low level programming language into logical controllers. In order to check the conformity of the code with respect to the diagram, test beds are generated. The tests function in a black box manner: we check the conformance of the observed output values to the expected ones for different input values.
As one can see, even for a simple diagram like the one given in Figure 1, finding an exhaustive testing strategy is not obvious. A simple solution for scenario test generation is through simulations of the diagram for each and every possible combination on the inputs $i_1...i_5$. This poses practical difficulties for two main reasons. On one hand, manual exhaustive test generation is a tedious, time-consuming task that has to be done to hundreds and hundreds of logical diagram specifications uploaded on logic controllers. On the other hand, a loop structure in the logical diagram could cause oscillation problems. This means that logic values that circulate in a loop could keep changing indefinitely when passing through the blocks of the loop. This is a non desired phenomenon as it might prevent the controller's outputs from converging for a fixed set of input values. To overcome these difficulties, we propose (1) a graph state model called ***sequential graph of state/transition*** ($SGST$) and (2) a transformation procedure of the logical diagrams into the $SGST$. In this new graph, the nodes represent the states of the logical controller. The edges are labelled with the Boolean conditions over logic variables $Vars$. For instance, using the procedure we propose in this paper, we obtain for the logical diagram shown in Figure 1 the corresponding sequential graph of state/transition given by Figure 2. Throughout this paper, the logical diagram in Figure 1 will be our case study.

3 The sequential graph of state/transition (*SGST*)

A *sequential graph of state transition* ($SGST$) is a combinatorial structure that explicitly represents all the possible evaluation steps within evaluation cycles of the logical diagram by the controller.
Formally, the **Sequential Graph of State/Transition** (SGST) is an oriented graph defined by the couple (N, E) where N is the set of nodes and $E \subseteq N \times N$ is the set of directed edges. Nodes and edges of the graph are both labeled using the labeling functions l_N and l_E, respectively.

Definition 3 (l_N function). *For a given set of status blocks B^S, the **labeling function of the nodes of the** $SGST$ **graph** l_N is defined as l_N: $N \mapsto (P_{status})^L$, where $L = Card(B^S)$. This function assigns, for each status block $b^s \in B^S$ in the logical diagram, a status value to the node $n \in N$ in the $SGST$.*

[3] We refer to the implemented logical diagram specification as a logic controller.

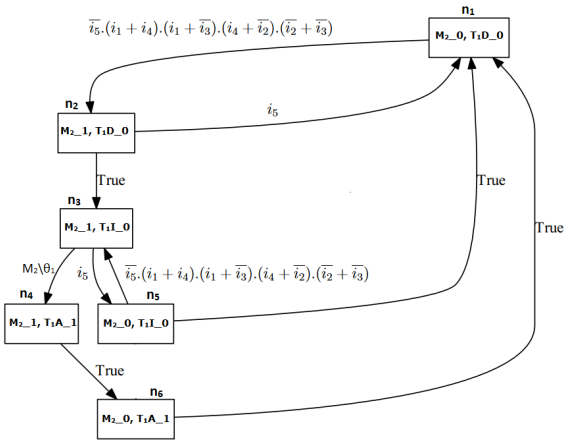

Fig. 2. The *SGST* corresponding to the non-formal logical diagram given by Fig. 1, obtained using our transformation procedure.

Definition 4 (l_E **function).** *For a given set of logic variables* $Vars$*, the* ***labeling function of the edges of the*** $SGST$ ***graph*** $l_E : E \mapsto Exp_{Vars}$*, assigns a logical expression including logic variables from* $Vars$ *to* $e \in E$ *in the* $SGST$*.*

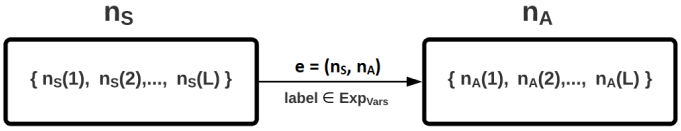

Fig. 3. SGST graph representation: example of two states n_S and n_A linked with a transition e. An edge e of the $SGST$ graph links two states, from the starting node n_S to the arrival node n_A. The edge e is labelled with a Boolean expression *label* from the Exp_{Vars} set. The starting and arrival nodes are labelled with a set of L status values, where L is the total number of status blocks in the logical diagram, i.e. $Card(B^S) = L$. The set Exp_{Vars} contains Boolean expressions on logic variables in $Vars$. The set $Vars = I \cup O_{BS}$ contains both the input variables I of the logical diagram, and the output variables O_{BS} of status blocks B^S in the logical diagram.

Definition 5 ($Eval_{logic}$ **function).** *We define* $Eval_{logic} : P_{status} \mapsto \{0, 1\}$ *as the* ***logic evaluation function*** *that returns the equivalent logic value of a status value. In a node* $n \in N$*, the logic value at the output of a status block* b_i^s *is* $o_{b_i^s} = Eval_{logic}(n(i))$ *where* $n(i)$ *is the status value of the block* b_i^s *in the node* n*.*

Some notions from Definitions 2–5 are illustrated in Figure 3. We remind that a logic value at the output o_{b^s} is associated to each status value of b^s assigned to a node $n \in N$. Therefore, a node n containing the status values of all blocks B^S encodes the logic values at each of their outputs.

Definition 6 (n^{Seq} **logical sequence).** *For a given set of status blocks* B^S*, we define* n^{Seq} *of a node* $n \in N$ *as the logical sequence on status blocks output*

variables o_{B^S}. It is a logical expression associated to the set of status values in N. This expression uses only the conjunction operator AND ($\cdot$) and the complement operator ($\overline{\ }$) and involves all the status output variables $o_{b^s} \in O_{B^S}$ of status blocks $b^s \in B^S$:

$$n^{Seq} = \prod_{k=1}^{L} (Eval_{logic}(n(k)) \cdot o_{b_k^s} + \overline{Eval_{logic}(n(k))} \cdot \overline{o_{b_k^s}});\ whereL = Card(B^S).$$

Example 2 (Illustration of n^{Seq} on the motivating example given by Figure 1). In the $SGST$ example of Figure 2, the node n_1 encapsulates the status values $\{M_2\_0,\ T_1D\_0\}$. These status values correspond to the logic values 0 and 0 at the outputs of the blocks M_2 and T_1, respectively. The logical sequence n_1^{Seq} of the node n_1 is $n_1^{Seq} = (\overbrace{Eval_{logic}(M_2\_0)}^{0} \cdot o_{M_2} + \overbrace{\overline{Eval_{logic}(M_2\_0)}}^{1} \cdot \overline{o_{M_2}}) \cdot (\overbrace{Eval_{logic}(T_1D\_0)}^{0} \cdot o_{T_1} + \overbrace{\overline{Eval_{logic}(T_1D\_0)}}^{1} \cdot \overline{o_{T_1}}) = \overline{o_{M_2}} \cdot \overline{o_{T_1}}$.

4 *SGST* construction

Given a logical diagram D, we consider I_D (the set of input variables of D) and B_D (the set of logic blocks of D). As explained in the previous sections, B_D=$B_D{}^S \cup B_D{}^{LG}$, where $B_D{}^S$ is the set of status blocks of D and $B_D{}^{LG}$ is the set of logic gates of D. The $SGST$ graph of the diagram D is denoted $SGST_D : (N_D, E_D)$ and is constructed as follows.

4.1 Building the *SGST* nodes

We will construct one node for each possible combination of status values between the status blocks. Let us start by defining the set of all the possible combinations of status values of blocks $b^s \in B_D^S$. Let n_T be the number of on-delay timer blocks and n_M the number of memory blocks in the logical diagram D. We define the set of all combinations of status values as $C_{status} = (P^M_{status})^{n_M} \times (P^{T_{ON}}_{status})^{n_T}$. The number of nodes of the $SGST_D$ is $Card(N_D) = Card(C_{status}) = 3^{n_T} \times 2^{n_M}$; $Card(P^{T_{ON}}_{status}) = 3$, $Card(P^M_{status}) = 2$. To each of these nodes we attribute a combination of status values using the l_N function: $\forall n_i \in N_D$, $l_N(n_i) = c_i$ where $c_i \in C_{status}, i \in \{1..Card(N_D)\}$. In the example of Figure 1, $P^{T_{ON}}_{status} = \{TD\_0, TI\_0, TA\_1\}$, $P^M_{status} = \{M\_0, M\_1\}$, C_{status} =$\{(M\_0, TD\_0), (M\_1, TD\_0), (M\_1, TI\_0), (M\_1, TA\_1), (M\_0, TI\_0), (M\_0,\ TA\_1)\}$. $Card(C_{status})$=6, the $SGST_D$ has therefore six nodes n_1 to $n_6 \in N_D$ labeled $c_1 ... c_6 \in C_{status}$, as shown in Figure 2.

4.2 Building the *SGST* edges

Edges that link the nodes in the graph are labelled with a logical expressions over $Vars$. If for a set of logic values of $Vars$, a Boolean expression that labels an edge starting from a node n_S and arriving to a node n_A is True, a change of status values of a B^S block in the diagram takes place. The $SGST_D$ of a logical

diagram D is developed to represent the evolution of states of the diagram in a formal model. The way this evolution works is defined by the evaluation process of the diagram by the controller. This evaluation is done in periodic cycles:

1. Reading and saving the values of all input variables i_k, where $k \in \mathbb{N}$.
2. Running a **sequential** evaluation algorithm on status blocks: at this point, each status block is evaluated, one after another, in accordance to the logic values at their entries and their last evaluated status value. The logic values at the entries of status blocks are obtained by the evaluation of logic gates connected to these entries in a left to right direction. We denote by ω the order of the evaluation sequence of status blocks.
3. Evaluating outputs. Outputs o_k are Boolean expressions of input variables I_D and status blocks output variables O_{B^S}.

We build the edges of the $SGST_D$ that link the nodes following the sequential evaluation of status blocks that we just established. This sequential evaluation dictates that only one status block is evaluated at a time. In other words, status blocks are not evaluated simultaneously. The result of evaluation of a status block is used in the evaluation of the next status block in the ordered sequence ω. This is translated in the graph by building edges that only connect nodes that have the same status values for all status blocks except for one. We call these nodes neighboring nodes.

Proposition 1. *Let n_1 and n_2 be two nodes of n_D and $l_N(n_1) = (\mu_1, \mu_2, ..., \mu_L)$ and $l_N(n_2) = (\lambda_1, \lambda_2, ..., \lambda_L)$ be their status values. n_1 and n_2 are two neighboring nodes and can possibly be linked by an edge in the $SGST_D$ if $\exists c \in \{1, ..., L\}$, with $L = Card(B_D^S)$, satisfying the following two conditions:*

- *$\forall k \in \{1, ..., L\} \backslash c$, $\mu_k = \lambda_k$; we note that $\mu_k = n_1(k)$ is the status value of b_k^s in n_1 and $\lambda_k = n_2(k)$ is the status value of b_k^s in n_2 where b_k^s are status blocks in B_D^S.*
- *$n_1(c) = \mu_c \neq \lambda_c = n_2(c)$ where $b_c^s \in B_D^S$ is the only status block that changes value from μ_c in node n_1 to λ_c in node n_2.*

Roughly speaking, Proposition 1 tells us that two nodes in the $SGST$ graph can be neighbours only if all their status values are identical except one. To conclude, an edge of the graph is equivalent to a change of the status value of a single status block between two neighboring nodes n_S and n_A linked by that edge. We refer to this change of value as an evolution *evol* and we define $EVOL_{b^s}$ as the set of all evolution possibilities of $b^s \in B_D^S$ between its status values $P_{status}^{b^s}$.

Definition 7 (*evol* tuple). *An **evolution** evol $\in EVOL_{b^s}$ is defined by the tuple (s_i, s_f, C_{evol}), with:*

- *s_i: the initial status value of the evolution evol; $s_i \in P_{status}^{b^s}$*
- *s_f: the final status value of the evolution evol; $s_f \in P_{status}^{b^s}$*
- *C_{evol}: the evolution condition; this is a Boolean expression deducted from the logical diagram. The evolution from s_i to s_f can only occur if this expression is True. We note that $C_{evol} \in Exp_{Vars}$.*

In order to construct the edges of the $SGST_D$, first the Boolean expressions of the status block entries have to be calculated (A). Second, the evolution sets $EVOL_{b^s}$ of every status block b^s have to be determined using the calculated expressions (B). Finally, edges of the graph are constructed based on the determined evolution sets of status blocks (C).

A. Developing the logical expressions at the entries of status blocks: We remind that the status value of a block $b^s \in B_D^S$ is calculated based on its previous status value and the logic values at the entries of the block. The logic values at these entries are obtained by evaluating the elements connected to them. We develop these connections into Boolean expressions. The evaluation of a Boolean expression associated to an entry of a status block gives the logic value of that entry. These expressions are developed as follows:
Let $x_{b_k^s}$ be an entry of a status block $b_k^s \in B_D^S$. $x_{b_k^s}$ could be connected to one of the following elements:

- An input $i_j \in I_D$: in this case the logic value of this entry is equal to the logic value of the input variable $x_{b_k^s} = i_j$;
- The output of a status block b_j^s, with $j \neq k$: here, the entry takes the logic value of the output of the block b_j^s denoted by $o_{b_j^s}$. Then, $x_{b_k^s} = o_{b_j^s}$;
- The output of a logic gate b_{LG}: we denote by $o_{b_{LG}}$ the output of the logic gate b_{LG}; then, $x_{b_k^s} = o_{b_{LG}}$. The output $o_{b_{LG}}$ of the logic gate b_{LG} can be developed into a Boolean expression that uses the logic operator of the block b_{LG} over its entries. Entries of b_{LG} that are connected to the output of another logic gate are further developed into Boolean expressions and so on. This recursive development continues through all the encountered logic gates and stops at logic inputs I_D and status block outputs O_{B^S}.

Example 3. The logical diagram of Figure 1 has two status blocks T_1 and M_2 with outputs denoted o_{T_1} and o_{M_2}, respectively. The block T_1 has a single entry x_T directly connected to the output o_M: $x_T = o_{M_2}$. The block M_2 has two input terminals denoted E and H. The entry H is connected to the output of an 'OR' gate that we call or_1, $H = or_1$. The variable or_1 can be developed into the following expression: $or_1 = i_5 + o_{T_1}$. The expression of the entry H is therefore $H = i_5 + o_T$ Similarly, the input terminal E is connected to an 'OR' logic gate $E = or_2$. We denote by x_1 and x_2 the input terminals of this 'OR' gate. $or_2 = x_1 + x_2$. Both x_1 and x_2 are connected to logic gates. They are therefore developed into Boolean expression in their turn. Following this process, we obtain $E = i_1 \cdot \overline{i_2} + \overline{i_3} \cdot i_4$.

B. Building the evolution sets $EVOL_{b^s}$ of every status blocks $b^s \in B_D^S$: The evolution possibilities of each status block are determined by the nature of the status block (i.e. memory blocks or timer blocks). Knowing the Boolean expressions at the entries of a block $b^s \in B_D^S$ we define the algorithms that construct all the evolution possibilities $EVOL_{b^s}$ of the block: Algorithm 1 corresponds to the evolution set construction for memory blocks, and Algorithm 2 constructs the evolution set for a timer block. We note that, in the case of

Algorithm 1 Evolution construction algorithm for status blocks of memory type

Input: Boolean expressions (E,H) ▷ E and H are the two entries of the memory block
Output: evolution set $EVOL_{b^s}$; Reminder: $evol \in EVOL_{b^s}, evol = (s_i,\ s_f,\ C_{evol})$
for all $(s_i,\ s_f) \in P^M_{status} \times P^M_{status}$ **do**
  if $(s_i,\ s_f) = (M\_1,\ M\_0)$ **then**
    $C_{evol} \leftarrow H$
  else if $(s_i,\ s_f) = (M\_0,\ M\_1)$ **then**
    $C_{evol} \leftarrow E \cdot \overline{H}$
  end if
  $evol \leftarrow (s_i,\ s_f,\ C_{evol})$
  add $evol$ to $EVOL_M$
end for

Algorithm 2 Evolution construction algorithm for status blocks of timer type

Input: Boolean expressions X ▷ X is the entry of the block
Output: evolution set $EVOL_{b^s}$; $evol \in EVOL_{b^s}, evol = (s_i,\ s_f,\ C_{evol})$
for all $(s_i,\ s_f) \in P^{T_{ON}}_{status} \times P^{T_{ON}}_{status}$ **do**
  if $(s_i,\ s_f) = (TD\_0,\ TI\_0)$ **then**
    $C_{evol} \leftarrow X$
  else if $(s_i,\ s_f) = (TI\_0,\ TD\_0)$ **then**
    $C_{evol} \leftarrow \overline{X}$
  else if $(s_i,\ s_f) = (TI\_0,\ TA\_1)$ **then**
    $C_{evol} \leftarrow X \cdot X/\theta$
  else if $(s_i,\ s_f) = (TA\_1,\ TD\_0)$ **then**
    $C_{evol} \leftarrow \overline{X}$
  end if
  $evol \leftarrow (s_i,\ s_f,\ C_{evol})$
  add $evol$ to $EVOL_M$
end for

timers, in addition to the logic value at the entry of the block, the status and output value of timer blocks also depend on time. After receiving a stimulus (i.e. a rinsing or falling edge), a timer changes its status automatically after a time period during which the stimulus action is maintained. In the case of an on-delay timer with a characteristic delay θ, if its input X is set to 1 for a period $\Delta_t > \theta$, the timer goes to the activated status TA giving the value 1 at its output instead of 0 in its deactivated status TD. We introduce another logic variable $X/\theta \in Vars$ such that: $\begin{cases} X/\theta = 1 & \text{if X holds the value 1 for a period } t > \theta \\ X/\theta = 0 & \text{otherwise} \end{cases}$

Example 4. Applying Algorithms 1 and 2 on the diagram example of Figure 1, we obtain the evolution sets of the two status blocks in the diagram. For the timer T_1, we obtain $T_1 = \{evol_1,\ evol_2,\ evol_3,\ evol_4\}$, with:
$evol_1 = (T_1D\_0,\ T_1I\_0,\ O_{M_2})$; $evol_2 = (T_1I\_0,\ T_1D\_0,\ \overline{O_{M_2}})$;
$evol_3 = (T_1I\_0,\ T_1A\_1,\ O_{M_2} \cdot O_{M_2\setminus\theta})$; $evol_4 = (T_1A\_1,\ T_1D\_0,\ \overline{O_{M_2}})$.
For the memory M_2, we obtain $EVOL_{M_2} = \{evol_1,\ evol_2\}$, with:

$evol_1 = (M_2\_0,\ M_2\_1,\ \overline{O_{T_1}} \cdot \overline{e_5} \cdot (i_1 + i_4) \cdot (i_1 + \overline{i_3}) \cdot (i_4 + \overline{i_2}) \cdot (\overline{i_2} + \overline{i_3}))$;
$evol_2 = (M_2\_1,\ M_2\_0,\ O_{T_1} + e_5)$.

C. Building the edges E_D of the $SGST_D$:
Having built the nodes of the graph and determined all the evolution sets $EVOL_{b^s}$ of status blocks B_D^S, we build the edges that connect these nodes. We remind that the controller evaluates its status blocks in an ordered sequence ω. In other words, status blocks are not evaluated simultaneously; they are evaluated one at a time. A node in the $SGST_D$ encapsulates the status values of all the blocks B_D^S. Two nodes in N_D can have one or many different status values. The sequential evaluation of the controller is reproduced in the graph by building edges that only link neighboring nodes that have the exact same status values of all blocks B_D^S except for one (see Proposition 1). An edge linking two neighboring nodes corresponds to an evolution $evol \in EVOL_{b^s}$ of a single status block b^s. We propose Algorithm 5 for building the edges of the graph $SGST_D$ of a logical diagram D. For each node $n_k \in N_D$ of the $SGST$, the algorithm generates all the possible outgoing edges corresponding to all the evolution possibilities of all status blocks B_D^S from the node n_k. Algorithms 3 and 4 are used in Algorithm 5 for neighboring nodes recognition and nodes logical sequences generation.

Algorithm 3 Test whether n_S and n_A are neighboring nodes (see Proposition 1)

```
Input: n_S, n_A ∈ N_D
Output1: AreNeighbors ∈ {True, False}
Output2: c the index of the status block whose status value is changed from n_S to n_A
AreNeighbors ← True
differences ← 0                ▷ number of different status values between n_S and n_A
for all k = 1 to Card(B_D^S) do
    if n_S(k) ≠ n_A(k) then
        differences ← differences + 1
        if differences > 1 then
            AreNeighbors ← False
            break loop
        end if
        c ← k                  ▷ c is the index of the block that changes status from n_S to n_A
    end if
end for
```

5 Reasoning with the *SGST*

In this section, we show how the convergence of the expected behavior of the controller described by its logical diagram could be verified using the equivalent $SGST$ of that diagram. The $SGST$ is composed of a set of nodes and edges that reproduce the information encoded in the logical diagram in a formal and explicit description. A ***node*** in the $SGST$ corresponds to a possible state of the controller, i.e. a possible combination of status values. An ***edge*** corresponds to an evolution of a single status block. That is a change of the status value of a

Algorithm 4 Build n_S^{seq} the logical values sequence of outputs $O_{B_D^S}$ equivalent to status values in n_S (see Definition 6)

Input: $n_S \in N_D$
Output: n_S^{seq}
$n_S^{seq} \leftarrow True$
for all $k = 1$ to $Card(B_D^S)$ **do**
$n_S^{seq} \leftarrow n_S^{seq} \cdot (Eval_{logic}(n_S(b_k^s)) \cdot o_{b_k^s} + \overline{Eval_{logic}(n_S(k))} \cdot \overline{o_{b_k^s}})$
end for

block $b^s \in B_D^S$. The outgoing edges E_{n_j} of a node $n_j \in N_D$ in the $SGST_D$ graph of a diagram D, are all the theoretical evolution possibilities of all the status blocks from the node n_j. In practice, only one of these outgoing edges $e \in E_{n_j}$, is **traversed** depending on the values of the input variables I_D of the diagram. A ***traversal of an edge*** (n_S, n_A) is the effective transition of the controller's state from the node n_S to the node n_A by running the correspondent status block evolution of the traversed edge.
We remind that a **full evaluation cycle** of the logical diagram is held periodically by the controller. In each evaluation cycle of the diagram, status blocks are evaluated one after another according to an order ω until each and every block $b^s \in B^S$ is evaluated once and only once.
In the $SGST$, for a set of input values I_v a full evaluation cycle corresponds to a chain of successive edges traversed one after another in respect to the order of evaluation ω. In some cases, many successive evaluation sequences ω may have to be run to finally **converge to a node**. However in other cases, even after multiple evaluation sequences, this convergence may never be reached; Traversal of edges could be endlessly possible for a set of input values I_v.
The convergence of status values is the property that we are going to study in the rest of this paper. If we consider the real world case of power plants, the convergence property has to be verified on the logical diagrams before implementing them in the controllers. The non convergence of the evaluation cycles of the diagram for a set of input values I_v leads to the physical output signals of the controller alternating continuously between 0 and 1 which is a non-desired phenomenon. In the $SGST$ graph, this corresponds to a circuit of nodes being visited over and over again indefinitely. We will define trails and circuits in the graph then propose a formal criteria of behavior convergence on the $SGST$.

5.1 Traversal of the *SGST*: Trails

In the $SGST$, nodes are visited by traversing the edges that link them. A sequence of visited nodes in the graph is called a trail τ and is defined as follows:

Definition 8 (Trail τ). *For a given $SGST_D = (N_D, E_D)$, a **trail** $\tau \in (N_D)^k$, with $k \in \mathbb{N}$, is an ordered set of nodes (n_1, n_2,..., n_k) where each pair of successive nodes n_i and n_{i+1}, with $i \in \{1..k-1\}$, are neighboring nodes.*

A trail is therefore a series of state changes along neighboring nodes in the $SGST$ graph. From the $SGST$ we can form an infinite number of trails. However, only a finite subset of these trails could be effectively traversed in practice. This is due

Algorithm 5 Edge construction Algorithm

Input1: $EVOL_{b^s}$ $\forall b^s \in B_D^S$
Input2: the set of nodes N_D of the $SGST_D$
Output: the set of edges E_D of the $SGST_D$; $e \in E, e = (n_S, n_A, label)$
for all $(n_S, n_A) \in N_D \times N_D$ **do**
  AreNeighbors, c =Algorithm3(n_S, n_A) ▷ c is the index of the status block whose status value is changed from n_S to n_A
  if $AreNeighbors = True$ **then**
    n_S^{seq}=Algorithm4(n_S)
    for all $evol \in EVOL_{b_c^s}$ **do** ▷ b_c^s is the block that changes status from n_S ▷ to n_A; $EVOL_{b_c^s}$ is the evolution set of b_c^s
      if $s_i(evol) = n_S(c)$ and $s_f(evol) = n_A(c)$ **then** ▷ Find the evolution of b_c^s ▷ that corresponds to ▷ $n_S(c) \mapsto n_A(c)$
        $expression = C_{evol}(evol)$
        if $expression \wedge n_S^{Seq} \neq False$ **then** ▷ check if the expression of the evolution is contradictory ▷ to the logic values of status blocks outputs $O_{B_D^S}$ ▷ given by the sequence n_S^{Seq} of n_S
          $e = (n_S, n_A)$
          $l_E(e) = C_{evol}(evol)$
          add e to the set of edges E_D
          break loop
        end if
      end if
    end for
  end if
end for

to the order ω of the evaluation of status blocks. We call trails that are conform to the order ω **determined trails**. These trails correspond to the progressive traversal of **viable edges** in the $SGST$ for a set of input values I_v.

Definition 9 (Viability of an edge). *Let $e = (n_S, n_A) \in E_D$ be an edge in the SGST linking the start node n_S to the arrival node n_A. The edge e is said to be viable for a set of input values I_v if the Boolean expression label(e) is True for the values I_v. The traversal of the edge e changes the state of the controller from n_S to n_A by changing the status value of a single block $b^s \in B_D^S$. We denote by $b^s_{n_S \mapsto n_A}$ the status block b^s whose value was altered by going from n_S to n_A.*

A node n_k visited in the middle of a determined trail τ, can have multiple outgoing edges in the $SGST$ that are viable at the same time for a set of input values I_v. Only one of the viable edges is traversed in accordance to the edge traversal determination rule $(n_{k-1}, n_k) \mapsto n_{k+1}$: the next viable edge (n_k, n_{k+1}) to be traversed is the one that alters the status value of the block $b^s_{n_k \mapsto n_{k+1}}$ of the lowest order in ω after $b^s_{n_{k-1} \mapsto n_k}$ the block whose status value changed from n_{k-1} to n_k.

Proposition 2 (edge traversal determination rule). *Let $\omega \in (B^S)^L$, be an order of the evaluation sequence of status blocks, $L = Card(B^S)$. Let us suppose*

that for a set of input values I_v, the controller is placed in the state of node n_k, coming from the previous node n_{k-1}; between these two nodes, the status value of the block $b^s_{n_{k-1}\mapsto n_k}$ has changed. Let N_{next} be the set of all the reachable nodes from n_k by the viable edges e =(n_k,n_{next}), with $n_{next} \in N_{next}$.
Then, the next node $n_{k+1} \in N_{next}$ to be effectively visited in the trail is satisfying: $ord_\omega(b^s_{n_{k-1}\mapsto n_k}) < ord_\omega(b^s_{n_k\mapsto n_{k+1}}) < ord_\omega(b^s_{n_k\mapsto n_{next}})$ $\forall n_{next} \in N_{next}\backslash\{n_{k+1}\}$, where $\forall n_j \in N_D$; $\forall (n_k, n_j) \in E_D$ then $ord_\omega(b^s_{n_k\mapsto n_j})$ is the order of evaluation of the block $b^s_{n_k\mapsto n_j}$ in ω; $b^s_{n_k\mapsto n_j}$ is the status changing block from n_k to n_j.

Example 5. Let us consider a logical diagram with three status blocks of memory type $B^S = \{M_1, M_2, M_3\}$; the corresponding $SGST$ graph is illustrated in Figure 4, and has four nodes and three edges. We fix the set of inputs values $\{i_1, i_2, i_3\}$=$\{1, 1, 1\}$. We suppose that the last visted node is n_k coming from n_{k-1}. We note that, from the node n_k, both edges e_2=(n_k, n_u) and e_3=(n_k, n_v), labeled i_2 and i_3, respectively, are viable for the input values $\{1, 1, 1\}$. The following

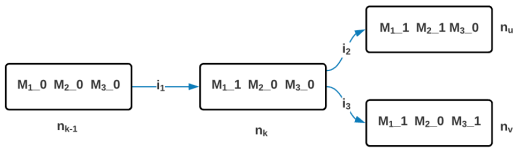

Fig. 4. Trail building in an SGST graph. Two possible trails are valid for the same input values I_v in this example, depending on the order of evaluation of the three status blocks, M_1, M_2, M_3.

node of the trail τ=(n_{k-1}, n_k) is determined in accordance with Proposition 2. We first consider ω =(M_1, M_2, M_3) the order of evaluation of the three status blocks. The last traversed edge is e_1=(n_{k-1}, n_k) with a change on the status value of the block M_1 of order $ord_\omega(M_1) = 1$ in the evaluation sequence ω. Edge e_2 alters the status value of the block M_2 of order $ord_\omega(M_2) = 2$ while the edge e_3 alters the status value of the block M_3 of order $ord_\omega(M_3) = 3$. Since $ord_\omega(M_1) = 1 < ord_\omega(M_2) = 2 < ord_\omega(M_3) = 3$, the next status block to be evaluated after M_1 is M_2, so the next node in the trail τ is $n_{k+1} = n_u$. In this case, τ=(n_{k-1}, n_k, n_u). However, if we consider ω =(M_3, M_2, M_1), i.e. $ord_\omega(M_3) = 1$, $ord_\omega(M_2) = 2$, $ord_\omega(M_1) = 3$, then the last evaluated block M_1 in the trail (n_{k-1}, n_k) is of order 3 which is the last order in the evaluation sequence ω. For the same input values $\{i_1, i_2, i_3\}$=$\{1, 1, 1\}$, the next block to be evaluated from n_k is this time M_3 of the order 1, which corresponds to edge e_3=(n_k, n_v). In this case, τ=(n_{k-1}, n_k, n_v).

We make the assumption that the initial node of a determined trail is a permanent node. A **permanent node**, unlike a transitional node, is a node in which the controller's state can remain permanently for a certain set of input values.

Definition 10 (permanent node). *Let $n_k \in N_D$ be a node, and $E_{n_k} \subset E_D$ the set of outgoing edges from the node n_k. We say that the node n_k is a* ***permanent***

node *if* $\exists\ I_v$*, a set of input values, satisfying the holding on condition of the node* n_k*:* $C_{Hold} = \prod_{e_i \in E_{n_k}} \overline{label(e_i)}$.

Example 6. For the $SGST$ graph given in Figure 2, node n_2 has two outgoing edges labeled i_5 and $True$. The holding on condition of node n_2 is $C_{Hold} = \overline{i_5} \cdot \overline{True} = False$. This condition is False for any set of input values I_v; thus, n_2 is not a permanent node. The node n_1 has only one outgoing edge labeled $\overline{i_5} \cdot (i_1 + i_4) \cdot (i_1 + \overline{i_3}) \cdot (i_4 + \overline{i_2}) \cdot (\overline{i_2} + \overline{i_3})$. The holding on condition of node n_1 is

$$C_{Hold} = \overline{\overline{i_5} \cdot (i_1 + i_4) \cdot (i_1 + \overline{i_3}) \cdot (i_4 + \overline{i_2}) \cdot (\overline{i_2} + \overline{i_3})} = i_5 + i_2 \cdot i_3 + i_2 \cdot \overline{i_4} + i_3 \cdot \overline{i_1} + \overline{i_1} \cdot \overline{i_4},$$

and can be satisfied for certain sets of input values, e.g. $\{i_1,\ i_2,\ i_3,\ i_4,\ i_5\} = \{0, 0, 0, 0, 1\}$. Thus, node n_1 is a permanent node. From the permanent node n_1, and for an order of evaluation $\omega = (M_2,\ T_1)$, a possible determined trail that could be effectively traversed would be $\tau_1 = (n_1,\ n_2,\ n_3,\ n_4,\ n_6,\ n_1)$ for the order $\omega = (M_2,\ T_1)$ and the set of input values $\{i_1,\ i_2,\ i_3,\ i_4,\ i_5\} = \{1, 0, 0, 1, 0\}$.

5.2 Formal verification of the convergence property in the *SGST*

In practice, we say that a signal converges if its periodic evaluation by the logic controller gives a constant value over a long time range during which the input signals I remain constant. A non convergent Boolean signal is a signal that keeps oscillating between 0 and 1 over multiple evaluation cycles of the logic controller while input values are unchanged. In the $SGST$, oscillating Boolean signals correspond to an indefinite visiting of the same subset of nodes over and over again. This causes an indefinite change of status values, which results in its turn to an indefinite change of logic values at the output of status blocks.

Definition 11 (Circuits in the $SGST$**).** *We define a* ***circuit*** *in a SGST graph as a finite series of nodes* $(n_1, n_2, \ldots, n_m)$ *such that the consecutive nodes* n_k *and* n_{k+1} *are neighboring nodes and* $n_1 = n_m$.

However, a determined trail in the $SGST$ graph could contain a circuit of nodes without necessarily traversing it indefinitely. Indeed, a trail could correspond to a one-time traversal of a circuit to leave it as soon as it visits the same node twice, as shown by Example 7.

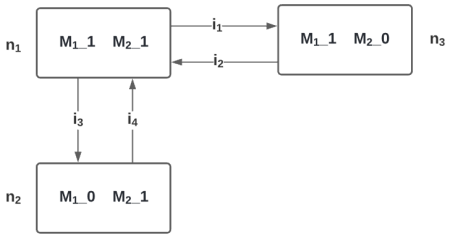

Fig. 5. Example of multiple circuits in an $SGST$ graph.

Example 7. We consider the $SGST$ graph given by Figure 5. The status blocks of the $SGST$ are B^S=$\{M_1, M_2\}$. It contains three possible circuits (n_1, n_2, n_1), (n_1, n_3, n_1) and $(n_1, n_2, n_1, n_3, n_1)$. We suppose the evaluation order ω=(M_2, M_1). We fix a set of input values $\{i_1, i_2, i_3, i_4\} = \{1, 1, 1, 0\}$. Using Proposition 2, we obtain the trail τ=(n_1, n_3, n_1, n_2), starting from the permanent node n_1. We can observe that τ contains the circuit (n_1, n_3, n_1), but this circuit is quit to the node n_2. However, if we fix the set of input values at $\{i_1, i_2, i_3, i_4\} = \{1, 1, 0, 0\}$, and start from node n_1, we obtain the trail τ =(n_1, n_3, n_1, n_3) that is equivalent to an indefinite traversal of the circuit (n_1, n_3, n_1).

Generally, if for an input I_v the progressive calculation of a the nodes of a trail τ results in visiting twice the same successive neighboring node couple (n_k, n_{k+1}), then τ corresponds to a circuit of nodes that can be indefinitely visited and the outputs of the blocks whose status values are changed in that trail are oscillating.

Definition 12 (Convergence property in a trail). *We denote by T^{ω}_{SGST} the set of all determined trails in the SGST that can be effectively traversed for an evaluation order ω. A trail $\tau = (n_1, n_2, ..., n_m) \in T^{\omega}_{SGST}$ is **convergent** if $e_k \neq e_j$ $\forall e_k = (n_k, n_{k+1})$, $e_j = (n_j, n_{j+1})$ two couples of neighboring nodes in τ.*

Definition 13 (Convergent logical diagram). *We say that a logical diagram D is convergent for all the sets of input values if all the determined trails of its $SGST_D$ are convergent.*

We propose a method for searching all the determined trails T^{ω}_{SGST} for an evaluation order ω. Trails are determined by giving their symbolic Boolean condition of traversal instead of the sets of input values. This means that a determined trail $\tau \in T^{\omega}_{SGST}$ is defined by the sequence of its nodes $\tau = (n_1, n_2, ..., n_{m-1}, n_m)$ and its traversal condition $C_\tau = \prod_{k \in \{1..m-1\}} label(e_k = (n_k, n_{k+1}))$.
Starting from each permanent node in the $SGST$ we calculate all the possible trails based on the trail determination rule for an order ω (Proposition 2). From each node we explore all the possible outgoing edges by negating the condition labels of edges alternating the blocks of the least order. Each label of an explored edge is added to C_τ. For instance, let us suppose that a trail reaches a node n_k coming from n_{k-1} and that n_k that has two outgoing edges e_1=(n_k, n_u) and e_2=(n_k, n_v). $\tau = (n_1, n_2, ..., n_{k-1}, n_k)$, $C_\tau = \prod_{j \in \{1..k-1\}} label(e_k = (n_j, n_{j+1}))$.

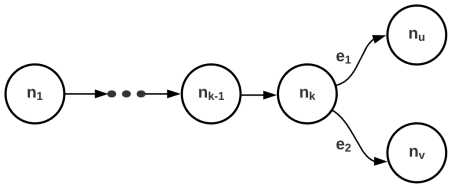

Fig. 6. Trail traversal condition update.

We suppose that $ord_\omega(b^s_{n_{k-1} \mapsto n_k}) < ord_\omega(b^s_{n_k \mapsto n_u}) < ord_\omega(b^s_{n_k \mapsto n_v})$, for the order ω. Since the status block altered by e_1 is of a lower order than the one

altered by e_2, if $label(e_1) = True$ then e_1 is the next movement in τ, but if $label(e_1) = False$ and $label(e_2) = True$ then the next movement in τ is e_2. Thus, two determined trails τ_1 and τ_2 can branch off from the determined trail τ at n_k such that $C_{\tau_1} = C_\tau \cdot label(e_1)$ and $C_{\tau_2} = C_\tau \cdot \overline{label(e_1)} \cdot label(e_2)$. Both new trails continue the course and branch off to more possible trails at each bifurcation. Path exploration of a trail can stop in one of the following scenarios:

- If the last encountered node n_k is a permanent node. Here, τ is a determined trail that puts the controller in the state of the node n_k starting from the state of the initial node n_1 for all the input values I_v that satisfy C_τ.
- If for the last encountered node n_k, the update of the traversal condition $C_\tau \cdot label(e = (n_{k-1}, n_k))$ is False. This means that the trail is not possible due to a contradiction of the condition labels of the graph edges crossed by the trail.
- If the last two couple of nodes (n_{k-1}, n_k) have already been visited in τ. In this case τ corresponds to a circuit of nodes that can be effectively traversed an infinite number of times for the inputs values I_v satisfying condition C_τ.

Example 8. The $SGST$ example of Figure 2 has only one permanent node n_1. Starting from n_1, only two determined trails are possible in the case of the evaluation order $\omega = (T_1, M_2)$:
$\tau_1 = (n_1, n_1)$, $C_{\tau_1} = i_5 + i_2 \cdot i_3 + i_2 \cdot \overline{i_4} + i_3 \cdot \overline{i_1} + \overline{i_1} \cdot \overline{i_4}$, $\tau_2 = (n_1, n_2, n_3, n_4, n_6, n_1, n_2)$, $C_{\tau_2} = \overline{i_5} \cdot (i_1 + i_4) \cdot (i_1 + \overline{i_3}) \cdot (i_4 + \overline{i_2}) \cdot (\overline{i_2} + \overline{i_3})$. τ_2 does not converge meaning that for any set of input values I_v that satisfies C_{τ_2} the nodes of τ_2 are visited indefinitely which causes oscillating signals in the controller.

6 Discussion

In this paper, we proposed a formal model, called the $SGST$ graph, representing the possible states of a controller programmed using a logical diagram. We show how to transform the logical diagram into the corresponding $SGST$ graph, and how to verify the convergence property, i.e. verify that the controller does not have undesired oscillatory behavior.

For a logical diagram as simple as the one of Figure 1, there are 2^5 possible input values. If we were to test the convergence manually for this diagram, 2^5 simulation cases have to be run and each simulation may have to go through many steps before deciding whether or not the output values converge. However, by building the $SGST$ of the diagram, the verification of the convergence property is cut down to finding the two determined trails in the graph and verifying if these trails are a closed circuit of successive nodes. If a trail is indeed a circuit of nodes, then finding the input values for which the logical diagram does not converge is a simple calculation of logical input combinations that satisfy the Boolean condition of the traversal of that trail in the $SGST$ graph. Thus, the $SGST$ comes in with all the advantages of a formal graph, i.e. one can use proved techniques for property verification (such as convergence, considered in this paper, but also potentially other properties).

Making sure that the behavior described by the logical diagram converges is crucial for test generation and for the overall verification and validation procedure. However, this is not the sole goal of transforming logical diagrams into *SGST* graphs. We developed the *SGST* to take a step in the application of the existing formal testing methods on logical diagram. For the time being, generating tests derived from logical diagram specifications of power plants logical controllers is still handled manually or simulation-based. So, we designed the *SGST* to provide an explicit formal representation of the exhaustive behavior evolution steps between possible states of the controller described by a logical diagram. A test scenario is a sequence of these steps modeled with edges in the *SGST*. Therefore, the test sequences generation could be transposed into the application of existing graph traversal techniques such as the Chinese postman tour [13]. The existing test generation ([4, 12]) and selection techniques are based on finite state machines specifications. We consider the *SGST* to be an important intermediate step to move from non-formal diagrams to state machines.

Transforming *SGST* graphs into Mealy machines and implementation of the formal results in the literature to test controllers described with logical diagram specifications will be studied in future work.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical computer science **126**(2), 183–235 (1994)
2. Fayyazi, M., Kirsch, L.: Efficient simulation of oscillatory combinational loops. In: Proceedings of the 47th Design Automation Conference. pp. 777–780 (2010)
3. Jean-françois Hery, J.c.L.: Stabilité de la spécification logique du contrôle-commande - méthodologie et mise en œuvre. Tech. rep., EDF R&D (2019)
4. Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines-a survey. Proceedings of the IEEE **84**(8), 1090–1123 (1996)
5. Lukoschus, J., Von Hanxleden, R.: Removing cycles in esterel programs. EURASIP Journal on Embedded Systems **2007**, 1–23 (2007)
6. Malik, S.: Analysis of cyclic combinational circuits. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **13**(7), 950–956 (1994)
7. Neiroukh, O., Edwards, S., Song, X.: An efficient algorithm for the analysis of cyclic circuits. vol. 2006, p. 6 pp. (04 2006). https://doi.org/10.1109/ISVLSI.2006.18
8. Peled, D., Vardi, M.Y., Yannakakis, M.: Black box checking. In: Formal Methods for Protocol Engineering and Distributed Systems, pp. 225–240. Springer (1999)
9. Provost, J., Roussel, J.M., Faure, J.M.: Translating grafcet specifications into mealy machines for conformance test purposes. Control Engineering Practice **19**(9), 947–957 (2011)
10. Riedel, M.D.: Cyclic combinational circuits. California Inst. of Technology (2004)
11. Shiple, T.R., Berry, G., Touati, H.: Constructive analysis of cyclic circuits. In: Proceedings ED&TC European Design and Test Conference. pp. 328–333 (1996)
12. Springintveld, J., Vaandrager, F., D'Argenio, P.R.: Testing timed automata. Theoretical computer science **254**(1-2), 225–257 (2001)
13. Thimbleby, H.: The directed chinese postman problem. Software: Practice and Experience **33**(11), 1081–1096 (2003)