Constraint Programming for Logic controller Synthesis

Mathieu Roisin, Pierre-Alain Yvars, Bernard Riera

▶ **To cite this version:**

Mathieu Roisin, Pierre-Alain Yvars, Bernard Riera. Constraint Programming for Logic controller Synthesis. 2024 10th International Conference on Control, Decision and Information Technologies (CoDIT), Jul 2024, Vallette, Malta. pp.1843-1848, 10.1109/codit62066.2024.10708360 . hal-04914353

HAL Id: hal-04914353

https://hal.science/hal-04914353v1

Submitted on 10 Feb 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Constraint Programming for Logic controller Synthesis

Mathieu Roisin[1], Pierre-Alain Yvars[1] and Bernard Riera[2]

***Abstract*—This article deals with the use of a constraint satisfaction problem (CSP) modeling with a constraint programming (CP) solver to support the synthesis of logical controllers for Programmable Logic Controller (PLC). In this work, manufacturing systems are considered and are seen as Discrete Event Systems (DES) with logical inputs (sensors) and logical outputs (actuators). The controller is seen as a scheduler of operative (or functional) independent tasks. The methodology is based on the definition of constraints. The solver will indicate, based on the previous PLC variables state, the possible tasks which could be activated. In addition, it is proposed a solution to get only one solution, enabling, in the future, a possible implementation in a PLC.**

I. INTRODUCTION

Logic controllers are used in a very large number of systems that often are critical systems. Today, Programmable Logic Controllers (PLC) are the most popular hardware in the industrial world to support logic controller programs. The operation of an industrial programmable logic controller can be broken down into several sequential steps: data acquisition from sensors (using a I/O memory map), program execution which calculates the state of actuators, actuators update. This process repeats in a continuous loop (PLC scan time), allowing the PLC to effectively control and automate industrial processes according to the specific needs of the application.

To improve the reliability of logic controllers, many formal methods have been proposed over the last thirty years [3]. The method presented in this paper belongs to the class of formal synthesis approaches of which goal is to produce automatically a correct by construction logic controller by using the specifications and the properties to be satisfied. In this paper, to achieve this goal, we propose to use a constraint programming (CP) approach. The main idea is to express the properties from the specification that the controller must necessarily satisfy to support the design of the logical controller having to be programmed in the PLC. The main idea is, from the definition of constraints, knowing the cyclic behavior of a PLC, to use a CP solver, to know the possible actions being able to be performed at the current state. To do this, we propose a taxonomy of constraints to be satisfied by a logic controller and then a formalization using a problem modelling language associated with a CP solver on mixed domains. The approach is illustrated by a case study of the discrete control [1] of a manufacturing system [2].

[1] ISAE-Supmeca, Quartz, EA7393, 3 rue Fernand Hainaut, 93407 Saint Ouen Cedex, France (e-mail : mathieu.roisin@isae-supmeca.fr, pierre-alain.yvars@isae-supmeca.fr)

[2] CReSTIC, University of Reims Champagne-Ardenne Reims, Moulin de la Housse, Chem. des Rouliers, 51100 Reims, France (e-mail : bernard.riera@univ-reims.fr)

The paper is organized as follows. Section 2 describes the different works that already exist. In section 3, constraint programming is detailed. In section 4, the methodology to model the problem is explained. In section 5, we use the methodology on a study case. In section 6, concluding remarks are discussed.

II. RELATED WORKS

A. Logic Controller Design

Designing logical controllers is a complex task that requires great rigor. Indeed, dysfunctions of controlled systems can be dangerous for operators and the environment. This is why it is essential to apply rigorous and proven design methodologies [3] to ensure the safety and reliability of control systems. The Evaluation Assurance Level (EAL), ISO 15408 [4] standard provides a recognized framework for evaluating the security of critical applications.

B. Algebraic synthesis for logic controllers

Research on algebraic function has led to the definition of a way to formalize each requirement and determine whether the set of requirements is coherent [5]. A logic filter was then built using this approach [6] in order to secure existing PLC programs. This has led to the development of an algebraic synthesis and logic filter approach to the control of cyber-physical manufacturing systems [7] that can synthesize a logic controller.

C. Integrating a solver inside a PLC

The synthesis of a logic controller can be equivalent to a Boolean Satisfaction Problem (SAT). Integration of a SAT solver inside a PLC has been achieved [6]. Nevertheless, the performance achieved with the proposed algorithms may not be satisfactory enough for complex industrial applications. The development of an efficient local search algorithm that can be implemented in an API therefore remains an open problem.

III. CONSTRAINT PROGRAMMING

A. Constraint Satisfaction Problem

A Constraint Satisfaction Problem (CSP) is defined by a triplet (X, D, C) such that [8]:

- X = $\{x_1, x_2, \dots, x_n\}$ is a finite set of *n* variables which we call constrained variables.

- D = $\{d_1, d_2, \dots, d_n\}$ is a finite set of *n* variable value domains of X such that: $\forall\ i \in \{1, \dots, n\}, x_i \in d_i$

- C = $\{c_1, c_2, \ldots, c_p\}$ is a finite set of p constraints, such that $\forall\ i\ \in \{1, \ldots, p\}, \exists!\ X_i\ \subseteq X\ /\ c_i\ (X_i)$

A constraint is any type of mathematical relation (linear, quadratic, non-linear, logical…) covering the values of a set of variables. A CSP is declarative and constraints can be defined in extension as well as in intension.

The variable domains can be:

- Discrete: in the form of sets of possible values.
- Continuous: in the form of intervals on real numbers

Solving a CSP boils down to instantiating each of the variables of X while meeting the set of problem constraints C.

B. Solving a discrete CSP

In the case of logic controller synthesis, we are interested in discrete CSP, also known as CSP on finite domains [9]. A solution to a finite-domain CSP is an assignment of a value to each variable such that the constraints are satisfied whenever possible. Otherwise, the CSP has no solution, especially if the CSP is over-constrained. The main way to solve CSP is called Constraint Programming. The resolution principle alternates the choice of a non-instantiated variable and a value for this variable in its domain and the propagation of this choice through the problem's constraint network C. Propagation algorithms (called *propagate()* in Algorithm 1) allow the solver to reduce the domain of each variable in such a way that values that do not satisfy the constraints are removed. CP is a global and admissible method: on the one hand the domains of the variables are iteratively reduced until finding the solution(s) and on the other hand, all the given solutions necessarily respect the constraints of C. Moreover, a CSP is a-causal and the whole set of solutions can be generated via constraint programming solving mechanisms. Finally, if a problem has no solution, this means that the CSP becomes inconsistent and this is detected by the resolution algorithm (cf Algorithm 1).

Algorithm 1: Solving a discrete CSP

```
function CSPSolve (X, D, C, stack=[D])
solution ← false
while (not solution ∧ X ← pop(stack)) do
    s ← propagate()
    if s then
        if (∀x_i ∈ X , x_i ∈ d_i is a singleton))
            then  solution ← true
            else Choose x_i ∈ X , x_i ∈ d_i that is not a singleton
                Choose one v_ij ∈ d_i
                push(stack, {d_1, ..., d_i\{v_ij}, ..., d_n})
                push(stack, {d_1, ..., {v_ij}, ..., d_n})
        end if
    end if
end while
if solution then return D else  return ∅
end function
```

The advantages of constraint programming compared with solution engines based on the SAT problem [10] are:

- the possibility of directly using non-Boolean domains (integers, floats)
- the possibility of mixing Boolean, integers and float variables on a same constraint
- the existence of algorithms dedicated to particular sub-problems also called global constraints.

C. Global constraints

A global constraint is a union of simple constraints. The advantage is twofold: better expressiveness and more efficient propagation thanks to a propagation method specific to each global constraint. Many global constraints have been defined and developed by the CP community [11, 12]. A global constraint that we will be using in the remainder of this article is the table constraint, also known as the catalogue constraint [13]. The table constraints is used to make a set of variables subject to compliance with combinations of values expressed in the form of a set of tuples. Very efficient propagation algorithms dedicated to this type of constraint exist in the literature. They can be used to remove tuples that have become impossible during the solving process. In term of solving, table constraints are more efficient than a flat representation of the list of tuples as a disjunction of conjunctions.

CSP on finite domains can be used to deal with Boolean variables and Boolean algebra. The Boolean operators *and*, *or*, *not* can be represented in {0,1} as shown in Table 1. Note that several representations are possible for the '*and*' and '*or*' operators. Moreover, table constraints can theoretically encode any constraints and more precisely any constraints on binary domains.

TABLE 1: EQUIVALENCE BETWEEN BOOLEAN DOMAIN OPERATOR AND FINITE DOMAIN OPERATOR

Operator	Boolean variables in {false, true}	Integer variables in {0, 1}
x and y	$x * y$	$x * y$, $\min(x, y)$
x or y	$x + y$	$\min(1, x + y)$, $\max(x, y)$
not x	$\bar{x}$	$1 - x$
x xor y	$x * \bar{y} + \bar{x} * y$	$\lvert x - y \rvert$

Several libraries exist for constraint programming on finite domains. These include the free libraries Choco [14] or ACE [15] with Java and Gecode [16] with C++. These libraries require mastery of a host programming language such as C++ or Java, depending on the case. Some initiatives propose to separate modelling and resolution by providing a flat CSP modeling language and a resolution tool. Examples include minizinc [17] and pyCSP3 [18]. A few rare projects offer a structured modelling language combined with a resolution tool on mix domains (finite domains and intervals). These include the DEPS language [19] and the DEPS Studio IDE[20]. The work described in this article has been implemented in DEPS and under the DEPS Studio IDE and solver.

IV. TAXONOMY OF DESIGN CONSTRAINTS FOR LOGIC CONTROL SYSTEM SYNTHESIS

In this paper, we consider that a manufacturing system controller must schedule operative tasks. In other words, the logic controller must authorize or not the activation of tasks, which could be in two different states: idle or busy.

For the rest of the article, we will assume these hypotheses:

- Operative tasks are independent from each other.
- During its execution, a task does not require any external conditions.

- If a task is authorized to start, it starts.
- Achievement of one or several operative tasks are necessary conditions to authorize an operative task.
- An operative task stops only when it reaches its final condition.
- The system can be expressed with discrete values.

All variables in this section are Boolean variable with the usual equivalence $\{false, true\} \Leftrightarrow \{0, 1\}$

One can notice that task authorization conditions and task final conditions are events.

A. State representation

A manufacturing system has uncontrollable and controllable variables.

Uncontrollable variables are sensors and observers. We will note the state of those variables with the following notation:

- Si: State of the sensor i
- Oi: State of the observer i

Controllable variables in this system are tasks. We will note the state of a task with the following notation:

- Ti: State of the task i (1 for busy and 0 for idle)

Considering the PLC feature, we use and note the previous state of a variable we will use the prefix "$p_$" in front of the variable. For example: p_Ti is the value of the variable Ti in the previous state.

B. Structural invariants of the synthesis problem

By its very nature, the field of automatic engineering introduces a set of structural invariants internal to each task. A task begins when it is authorized and ends when it reaches his final condition. A task cannot be authorized and reached his final condition at the same time. The final condition of a task has value 0 if the task is not active.

So, we defined 2 variables for each task:

- Aut_Ti: Authorization to start task i
- $Final_Ti$: the task i is achieved.

The relation between the previous state of a task and the current state of a task is defined by (1):

$$Ti = Aut_Ti \text{ or } p_Ti \text{ and not } Final_Ti \quad (1)$$

An authorization of a task depends on different variables that will depend on the last and current state of the problem.

We can identify different variables that will impact the value of the authorization:

- $Init_Ti$: Initial condition of the Task i. This variable has a value of 1 if all the requirements that imply sensors or observers allow the task to be authorized.
- $Sync_Ti$: Synchronization condition of the Task i. This variable has a value of 1 if all the necessary tasks to authorize the task i has been performed before.
- $Inco_Ti$: Incompatibility condition of the Task i. This variable has a value of 1 if all the requirements that imply the mutual exclusion between tasks allow the task i to start.
- $Prio_Ti$: Priority conditions of the Task i. This variable has a value of 1 if all priority expressions allow the task i to start. This condition is useful only if we want to have a deterministic solution.

In the case of only one solution is desired, a task is authorized as soon as possible if the initial, synchronization, safety, and priority conditions are satisfied, and the task was not active in the last state as in (2):

$$Aut_Ti = Sync_Ti \text{ and } Init_Ti \text{ and } Inco_Ti \text{ and } Prio_Ti \text{ and not } p_Ti \quad (2)$$

Otherwise, if all the possible solutions for task activation are expected, we would have (3):

$$Aut_Ti \leq Sync_Ti \text{ and } Init_Ti \text{ and } Inco_Ti \text{ and } Prio_Ti \text{ and not } p_Ti \quad (3)$$

In this case, if an authorization could have a value of 1, we will have 2 different solutions, one with authorization value 1 and another with authorization value 0.

C. Task synchronization

To define the constraints, we propose to fill in a "Synchronization Table" (Table 2). This one contains all the necessary information: initial condition, previous tasks, subsequent tasks, final conditions and incompatibility with tasks. The behavior of each task, because the tasks are independent, can be easily modelled using for instance a Petri net or a Grafcet [21]. From these task models initial and final conditions are known. We suppose in this paper, that tasks do not need more information to be authorized.

TABLE 2: EXEMPLE OF SYNCHRONIZATION TABLE

Task	Initial Condition	Previous Tasks	Sub sequent tasks	Final Condition	Incompatibility with task
i	$Init_Ti$	Task j	Task k	End_Ti	Task y
x	$Init_Tx$	Task y and Task z	Task y and Task m	End_Tx	Task z and Task m
a	$Init_Ta$	Task b or Task c	Task b or Task c	End_Ta	

To determine if a task has all its previous tasks performed, tokens are used. When a task is performed, a token for each subsequent task is created. Each task that starts (i.e. authorized) will consume each token that allows this task to start.

We define a variable for each token:

- GTi_j: Token created by task i for the task j
- GTi_jk: Token created by task i for the task j or k

Therefore, the synchronization condition of a task is a combination of tokens that allows the task to start (is 1 if all the tokens needed to start the task are available, otherwise it is 0).

The evolution of the value of a token can be expressed by (3) with the task i that uses the token and the task y that creates the token:

$$GTy_i = Final_Ty \text{ or } p_GTy_i \text{ and not } Aut_Ti \quad (4)$$

D. Priority between tasks

The domain of solutions that satisfies the requirements can have multiple degrees of freedom. Thus, for a unique previous state, we can have multiple possible valid current states. The number of possible valid current states is two to the power of

the number of degrees of freedom. At least one degree of freedom is created by each incompatibility condition and token that can be consumed by more than one task. These degrees of freedom involve tasks in the constraint that created that degree. Some of these degrees of freedom may already be constrained by the synchronization table. However, without analyzing the problem, we cannot be sure that all of these degrees of freedom are constrained. If we want a deterministic solution (i.e. one solution), we need to constrain each of these degrees of freedom. To constrain them, we defined an equation for the value of priority of each task. The way we constrain each of them depends on how we want the system to function.

V. CASE STUDY

This section is a proof of concept to show in more detail how the proposed methodology works, as well as to demonstrate its effectiveness and current limitations. The model is implemented in the DEPS Studio IDE with the DEPS language.

A. Study system

The study system is a system for packaging liquids in bottles (Fig. 1). This problem is taken from [2]. The operating part of this system consists of a turntable around which four separate stations are located:

- At station 1: A robot manipulator is used to load bottles onto the turntable from an empty bottle feed chute.
- At station 2: The bottles are filled.
- At station 3: The bottles are capped and sealed.
- At station 4: The same robot manipulator is used to unload the bottles from the turntable into an evacuation chute.

To ensure the smooth operation of this production system, the system is divided into 6 operational tasks as listed in Table 3.

Fig. 1. Study system [2]

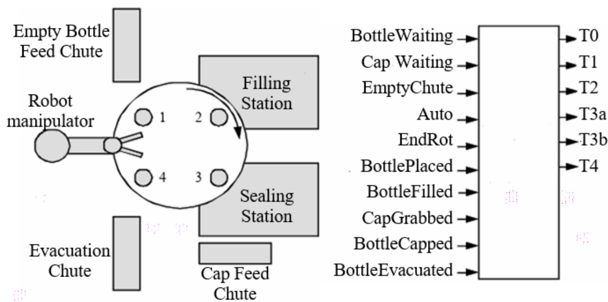

TABLE 3: TASKS DESCRIPTION

Task	Component	Description
0	Turntable	Rotate the rotary table by a quarter turn
1	Robot manipulator	Place bottle on the turntable at station 1
2	Filling station	Fill the bottle at station 2
3a	Sealing station	Grab a cap at station 3
3b	Sealing station	Cap the bottle at station 3
4	Robot manipulator	Evacuate bottle from turntable at station 4

The available sensors in this system are:

- Auto: Automatic mode is activated.
- EndRot: Turntable has finished his rotation.
- BottleStandby: A bottle is waiting to be grabbed and loaded on the turntable.
- BottlePlaced: A bottle is place on the turntable at station 1.
- BottleFilled: The bottle at station 2 is full.
- CapWaiting: A cap is waiting to be grab.
- CapGrabbed: A cap is grabbed at station 3.
- BottleCapped: The bottle at station 3 is capped.
- EmptyChute: The evacuation chute is empty.
- BottleEvacuated: The bottle at station 4 is evacuated in the evacuation chute.

Table 4 indicates for each task initial conditions, previous tasks, subsequent tasks, final conditions, and incompatibility with other tasks.

TABLE 4: SYNCHRONIZATION TABLE

Task	Initial Condition	Previous tasks	Sub sequent tasks	Final Condition	Incompatibility with task
0	Auto	1, 2, 3b and 4	1, 2, 3b and 4	EndRot	1, 2, 3b, 4
1	Bottle Standby	0	0	Bottle Placed	0, 4
2		0	0	Bottle Filled	0
3a	Cap Waiting	3b	3b	Cap Grabbed	3b
3b		0 and 3a	0 and 3a	Bottle Capped	0, 3a
4	Empty Chute	0	0	Bottle Evacuated	0, 1

B. DEPS Model creation

The model is implemented in the DEPS Studio IDE with the DEPS language. Each of the variables that will appear in this section is initialized as Boolean variable in {0,1}.

1) Structural invariants of the synthesis problem

We have a total of six tasks that we need to set up the invariant expression between variable of these tasks. Invariants expressions are the same as explain in part IV. These expressions are:

- The one that defined the value of the state of the task with the previous state and the authorization and the final condition that is (1).
- The second one is the expression of the authorization value in function of the previous state of the task and the synchronization, initial, incompatible and priority condition that is (2).

Fig. 2 shows how these expressions are integrated for the task 0 in the DEPS Studio IDE.

Fig. 2. DEPS Invariants expression for variable for task 0 of the system

```
(*-------------------Problem invariant expression--------------------*)
T0=Aut_T0 or (p_T0 and not (Final_T0));
AutT0=Sync_T0 and Init_T0 and Prio_T0 and Inco_T0 and not (p_T0);
```

2) Task synchronization

To implement the synchronization requirement between tasks, we use the tokens to create relationships between synchronization conditions and tokens. Fig.3 show how we implement this expression for the synchronization of the task 0 in DEPS Studio IDE.

Fig. 3. DEPS Synchronization condition for task 0 of the system

```
Sync_T0=p_GT1_0 and p_GT2_0 and p_GT3b_0 and p_GT4_0;
```

Then we need to create the relation that will make token change. Therefore, we create a relation between tokens, the previous state of the token, task authorization and final condition. To implement how tokens change at each PLC cycle in DEPS we decide to use an extension constraint, named as catalog. This takes as arguments, variables that this constraint must constrained and the data table describing the different possible tuples (Fig.4). This table is equivalent to (4).

Fig. 4. DEPS Token Creation and Usage table for extension constraint

```
Table TokenModification
Attributes
Token : Boolean ;
p_Token : Boolean ;
AutT : Boolean ;
FinalT : Boolean ;
Tuples
[0,0,0,0],
[1,0,0,1],
[0,0,1,0],
[0,0,1,1],
[1,1,0,0],
[1,1,0,1],
[0,1,1,0],
[1,1,1,1]
End
```

Then we can constrain each token using a catalog constraint to follow the constraints of that table. Fig. 5 shows the constraint created in the DEPS for each token created by the end of the task 0.

Fig. 5. DEPS Catalog constraint to change token's value created by task 0.

```
(*Token Created by T0*)
Catalog([GT0_1, p_GT0_1, Aut_T1, Final_T0 ], TokenModification);
Catalog([GT0_2, p_GT0_2, Aut_T2, Final_T0 ], TokenModification);
Catalog([GT0_3b, p_GT0_3b,Aut_T3b,Final_T0], TokenModification);
Catalog([GT0_4, p_GT0_4, Aut_T4, Final_T0 ], TokenModification);
```

3) Initial, final and incompatibility conditions

Using the synchronization table, we set the value of each initial condition and final condition (Fig.6.).

Fig. 6. DEPS model of system Initial Condition and final Condition

```
(*--------------------------------FinalsOfTasks--------------------------------*)
Final_T4=BottleEvacuated and p_T4;  Final_T0=EndRot and p_T0;
Final_T3a=CapGrabbed and p_T3a;     Final_T2=BottleFilled and p_T2;
Final_T3b=BottleCapped and p_T3b;   Final_T1=BottlePlaced and p_T1;
(*-------------------------------Initial Condition-------------------------------*)
Init_T0=Auto;                       Init_T1 =BottleWaiting;
Init_T2 =1;                         Init_T3a =CapWaiting;
Init_T3b =CapWaiting;               Init_T4 =EmptyChute;
```

Then we express the incompatibility requirements (Fig.7.). All incompatibility requirements are defined in the Task Synchronization table. There are three incompatibility conditions:

- Task 0 with task 1,2,3b and 4
- Task 1 with task 4
- Task 3a with task 3b

We will use a suffix "_ni" to indicate the sub-variable number i of a variable A such that the variable A is equal to the conjunction (*and* operator) between each A_ni.

Fig. 7. DEPS Incompatibly Requirements between task 1 and 4

```
(*T1 incompatible with T4, Create 1 Degree of Liberty*)
T1 and T4=0; (*To tolerate invalid initial state and return no solution*)
Inco_T1n2=not (T4);                 Inco_T4_n2=not (T1);
Inco_T1=Inco_T1_n1 and Inco_T1_n2;
IncoT4=Inco_T4_n1 and Inco_T4_n2;
```

4) Priority between tasks

Each of these incompatibility requirements creates one degree of freedom for a total of three. If we want to have a deterministic behavior, we must constrain these degrees of freedoms. By analyzing the order in which tasks are performed, two of the three degrees of freedom are already constrained. However, we will still constrain these degrees, even if they are already constrained, to show that the method works without analyzing the order in which the task is performed.

Here is how we will constrain them:

- T1 has priority over T4 (From T1 incompatible with T4).
- T1 and T2 and T3b and T4 has priority over T0 (already constrained) (From T1 and T2 and T3b and T4 incompatible with T0).
- T3b has priority over T3a (already constrained) (From T3a incompatible with T3b).

To do this, we use the equation derived from the requirement that created this degree of freedom. We replace the initialization value of this variable with its priority value, and we replace the value of the other task that constrains this value with the value of the authorizations of those tasks without the impact of that requirement (Fig. 8).

Fig. 8. DEPS System Task Priority

```
(* T4 cannot start if T1 can start*)
PrioT4=not (SyncT1 and PrioT1 and InitT1 and IncoT1n1);

(*T0 cannot start if T1 or T2 or T3b or T4 can start*)
PrioT0=not ((Sync_T1 and Prio_T1 and Init_T1 and Inco_T1_n2) or
(Sync_T2 and Init_T2 and Prio_T2) or (Sync_T3b and Prio_T3b and
Init_T3b and Inco_T3b_n2) or (Sync_T4 and Prio_T4 and Init_T4 and
Inco_T4_n2));

(* T3a cannot start if T3b can start*)
PrioT3a=not (Sync_T3b and Prio_T3b and Init_T3b and Inco_T3b_n1);

(*Each Priorty variable that has not been given a value are equal to 1*)
PrioT1=1;              PrioT2=1;              PrioT3b=1;
```

C Result

If the system has incoherent requirements, then the solver will immediately return no solution. By setting the previous state of the system and the value of each sensor, the solver will give us the next state of the system. If the initial state violates at least one requirement the solver return no solution. If we use

the priority with coherent requirements and a valid initial state, the solver will always give us only one valid solution (Table 5). The time to compute this solution is less than 1 millisecond on an Intel Core I7- 2.8GHz – 16 Go Ram computer configuration. This model allows to reverse the resolution. By fixing the current state of the system, the solver will give us all the previous states of the system that will resolve into that next state.

TABLE 5: RESULT TABLE FOR A PARTICULAR PREVIOUS STATE

Task	Previous state of task	Current state of task
0	0	0
1	0	1
2	1	0
3a	0	0
3b	0	1
4	0	0

Token	Previous state of token	Current state of token	Sensor	Current state of sensor
GT0_1	1	0	Endrot	0
GT0_2	0	0	Auto	1
GT0_3b	1	0	BottleCapped	0
GT0_4	1	1	BottlePlaced	0
GT1_0	0	0	BottleWaiting	1
GT2_0	0	1	BottleEvacueted	0
GT3a_3b	1	0	BottleFilled	1
GT3b_3a	0	0	CapWaiting	1
GT3b_0	0	0	EmptyChute	1
GT4_0	0	0	CapGrabbed	1

VI. CONCLUSION

We have presented in this paper a new requirement typology for designing logical controllers using constraint programming. Each requirement can be expressed by an equation or a truth table with a variable in the range {0,1} at the engineer's convenience. The whole requirements can be solved with a CP solver that handles both types of constraints. Using the priority condition, the proposed method allows us to obtain a unique safe solution if needed. The implementation on the case-study shows the viability of the methodology to create a constraint based logical controller.

Currently, this work is limited to using Boolean domain for the variable. Each task is represented by a set of Boolean variables. Future prospects will study the capability of representing the state of each task and the state of actions on a task by integer variables. Then, the number of variables will be reduced which will make the model smaller and more readable. This work paves the way for structured model constraint base programming, where the automation engineer does not need to write equations, but instead, creates instances of models and establishes links between these models. This will require the development of design patterns and will allow for a more modular approach to the synthesis of logical controllers. The approach will be tested on other case studies.

VII. ACKNOWLEDGMENT

This research was funded by l'Agence Nationale de la Recherche (ANR) for the Digital Twins for Cyber-Physical Systems project (ANR-23- CE10-0010-01). The authors would like to thank the ANR. They also thank the DEPS Link non-profit organization for the availability of the DEPS Studio software.

REFERENCES

[1] C. G. Cassandras and S. Lafortune, *Introduction to discrete event systems*. Boston, MA: Kluwer Academic Publishers. 1999.

[2] Y. Hietter, "Synthèse algébrique de la loi de commande d'un système à évènement discrets logique", Phd thesis, ENS Cachan, Cachan, 2009.

[3] J. Zaytoon and B. Riera, "Synthesis and implementation of logic controllers – A review", Annual Reviews in Control, vol. 43, pp. 152–168, 2017, doi: 10.1016/j.arcontrol.2017.03.004.

[4] ISO, "ISO/IEC 15408-1:2022 Information security, cybersecurity and privacy protection". Geneva, Switzerland, 2022. Accessed: Feb. 15, 2024. [Online]. https://www.iso.org/standard/72891.html Available

[5] Y. Hietter, J. M. Roussel, J. J. Lesage, "Algebraic synthesis of dependable logic controllers", IFAC-PapersOnLine, vol. 41, pp. 4132–4137, ENS Cachan, Cachan, 2008.

[6] R. Pichard, N. Ben Rabah, V. Carre-Menetrier and B. Riera "CSP solveur for Safe PLC Controller: Application to manufacturing systems", IFAC-PapersOnLine, vol. 49, pp. 402–407, Université de Reims Champagne-Ardenne, Reims, 2016.

[7] T. Ranger, A. Philippot, B. Riera "Algebraic Synthesis of Safety Logica Filter on Manufacturing Systems", IFAC-PapersOnLine, vol. 55, pp. 169–174, Université de Reims Champagne-Ardenne, Reims, 2022.

[8] U Montanari. "Networks of constraints: fundamental properties and applications to picture processing". Information Science, 7:95–132, mar 1974.

[9] E. P. K. Tsang. "Foundations of constraint satisfaction. Computation in cognitive science". Academic Press,1993.

[10] N.Een, M.Sheeran, "SAT-solving in practice, proc of 9th International Workshop On Discrete Event System", *WODES*, 2008.

[11] N. Beldiceanu, "Introduction to the special issue on global constraints", Constraints, 12(1):1–2, mar 2007.

[12] N. Beldiceanu, M. Carlsson, and J. X. Rampon. "Global constraint catalog", online (https://sofdem.github.io/gccat/gccat/index.html), 2014.

[13] K. M. Thanh, "Algorithms for table constraints and soft-regular constraints", Phd Thesis, Louvin University, 2019. https://dial.uclouvain.be/pr/boreal/fr/object/boreal%3A222553

[14] X Lorca, C Prud'homme, and Jean-Guillaume Fages. "Choco3 documentation"(https://usermanual.wiki/Document/userguide331.860520537.pdf),

[15] C. Lecoutre, "ACE , a generic constraint solver", 2023. https://doi.org/10.48550/arXiv.2302.05405

[16] C. Schulte, G. Tack, and M. Z. Lagerkvist, "Modeling and programming with gecode" online (https://www.gecode.org/doc-latest/MPG.pdf), 2019.

[17] N. Nethercote, P.J. Stuckey, R. Becket, S. Brand, G.J. Duck, and G Tack. "Minizinc: Towards a standard cp modelling language". In Proceedings of the 13th International, 2007.

[18] PyCSP3 . https://pycsp.org/

[19] P.A. Yvars and L. Zimmer, "DEPS: A model- and property-based language for system synthesis problems", International Journal of Software and Systems Modeling (SoSyM), 2023.

[20] P.A. Yvars and L. Zimmer, "Integration of constraint programming and model-based approach for system synthesis", *2021 IEEE International Systems Conference,* 2021.

[21] IEC 60848, "GRAFCET specification language for sequential function charts", 3rd edition, 2012.