Cetratus: Live Updates in Programmable Logic Controllers

Imanol Mugarza and Juan Carlos Mugarza

May 23, 2021

Cetratus: Live Updates in Programmable Logic Controllers

1st Imanol Mugarza
Ikerlan Technology Research Centre
Basque Research and Technology Alliance (BRTA)
Mondragon, Spain
imugarza@ikerlan.es

2nd Juan Carlos Mugarza
Automation Area
University of Mondragon
Mondragon, Spain
jcmugarza@mondragon.edu

***Abstract*—Manufacturing companies are facing new market demands, mostly driven by global competition and digitalization. In this context, more efficient, flexible, adaptable and evolvable mechatronic and manufacturing systems are required, which enable quick adjustments to the production in order to address (all these) market changes. However, production idle times due to such re-configurations and adaptations might be costly. In this paper, a live updates concept for Programmable Logic Controllers (PLCs) is presented. The proposed design employs a Petri net runtime engine, in which the executed functional program (the Petri net model with its interpretation) is updated while running, without system shutdown and restart being needed. To this end, a quarantine-mode execution and monitoring approach is used for the new PLC program functional validation. A reconfigurable Vernadat machine case study is also presented.**

***Index Terms*—dynamic software updates, live patching, live updates, PLC, Cetratus, availability**

I. Introduction

In the current digitalization and globalization era, higher efficiency, flexibility and productivity is wished in order to reduce time-to-market and production costs. Traditional systems, such as Dedicated Manufacturing Lines (DML) and Flexible Manufacturing Systems (FMS), are able to produce goods at high production rates, usually with some degree of low flexibility and customization [1]. In spite of that, with the aim of dynamically responding to market changes in a cost-efficient manner, different strategies, methods and schemes, denoted Reconfigurable Manufacturing Systems (RMS), have been proposed [1], [2].

A RMS is a class of production systems which it is able to rapidly change and evolve in order to adjust its production functions, capabilities and capacity [3]. These manufacturing systems may adapt its functionality to suit new products and therefore enable the production of high variety of products. These reconfiguration might be driven by new products to be manufactured, changes in current products or changes to the production rates of the products.

Nevertheless, production downtimes due to such reconfigurations might be costly. As stated by *H. Haddou-Benderbal et al.* [4], [5], the system might go through frequent reconfigurations. The configuration change time, which represents the time required to a given machine to change the configuration, is an RMS system metric and variable, used to compute and determine optimal system layout and production process. By reducing such duration in all machines, the manufacturing system responsiveness and reactivity is improved, manufacturing capability increased and, therefore, overall costs reduced. Actually, as expressed by *Y. Koren* [1], if those machines have not been originally designed to provide reconfigurability capabilities, such process will prove costly, lengthy, and thus, impractical. One of the technology enablers for RMS is reconfigurable software [1].

In this paper, a live updates concept for industrial Programmable Logic Controllers (PLCs) is proposed, which enables zero downtime reconfiguration capabilities and therefore, reduces the manufacturing system reconfiguration times and costs. The presented approach is based on the *Cetratus* framework [6], [7], originally devised for safe and secure industrial control systems, for example railway [7] or smart energy [8] systems. Even though the proposed framework was focused on the incorporation of leading-edge security mechanisms, any other types of software components could be updated. To this end, a quarantine-mode based execution and monitoring approach is enforced. In this work, in the context of RMSs, the executed functional program is adapted.

This paper is organized as follows: After this introduction, background information about Petri nets and the design of the proposed solution is presented. Following, the live update process used for the machine reconfiguration is described. Afterwards, a case study, presenting an implementation of the introduced design and method on a real PLC, is presented and discussed. Lastly, conclusions and future lines are drawn.

II. Background

A Petri net (PN), introduced in 1962 by *Carl Adam Petri* [9], is described as $N = < P, T, Pre, Post >$, where P and T are the sets of *places* and *transitions* and *Pre* and *Post* are the $|P|$ x $|T|$ sized, natural valued, *incidence matrices*. For example, $Post[p, t] = w$ means that there is an arc from t to p with weight or multiplicity w. When all weights are one the net is defined as *ordinary*. Petri nets methods allow model formal proof and validation analysis [10].

A *marking* is a $|P|$ sized, natural valued, vector. A marked net or P/T system is a pair $< N, M_0 >$, where M_0 is the initial marking. A transition t can be fired at marking M if

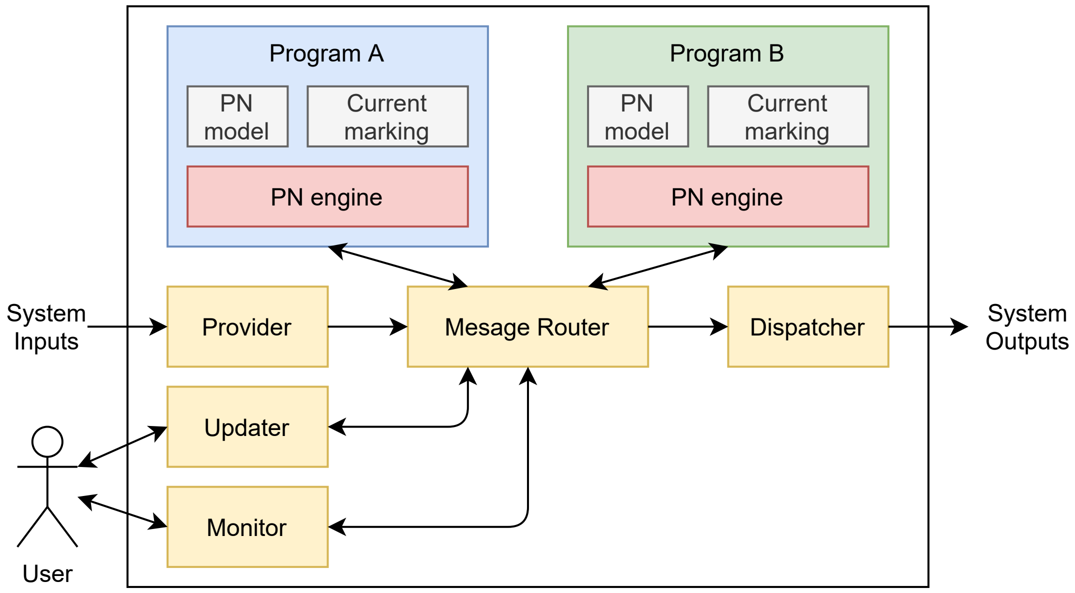


Fig. 1: Proposed PLC software architecture

$M \geq Pre[P, t]$ condition is satisfied. The firing of transition t yields then a new marking M'. This is computed by Equation 1, where C is the *token - flow* matrix of the net, given by $C = Post - Pre$, and θ is the firing vector, representing t transition firing. This equation is also known as the *state equation* of the system.

$$M_{i+1} = C * \theta + M_i \tag{1}$$

As discussed by *A. Giua* and *F. DiCesare* [11], Petri nets were already been used in different automation applications back in 1993. For this purpose, a Petri net interpreter, denoted *PN engine* in this work, was implemented. This software element continuously computes the *state equation* shown in Equation 1 using the Petri net model specified in the C matrix.

Nowadays, as described by *Karatkevich et al.* in their book *Design of Reconfigurable Logic Controllers* [12], one of the most spread Petri net-based language for industrial PLCs is Sequential Function Chart (SFC), also known as GRAFCET diagrams, which is widely supported by diverse PLC programming environments. This high-level graphical programming language is defined in the IEC 61131-3 international standard. Usually, a SFC program interpreter engine is also employed to run the SFC application specified by the engineer.

Moreover, ladder diagrams written in ladder logic, a graphical programming language originated from the design and construction of relay racks as used in process control and manufacturing, could also be translated to common Petri nets [13], [14]. Consequently, the proposed solution in this paper might directly be applied to such logic programs written in SFC and Ladder programming languages. Once the Petri net is created from the SFC and Ladder specifications, such PN model and the current stored marking should be substituted in the PLC for machine reconfiguration.

III. Design

The industrial controller software design is divided in two parts. Figure 1 shows the live updates enabled PLC architecture. On the one hand (at the top of Figure 1), the *PN runtime engine* and application-specific *PN model* and current state, denoted *marking*. The *PN runtime engine* is a PN execution library, which computes new system states according to Equation 1. In addition, as observed, two containers are allocated for the PLC program, A and B respectively. These are alternatively switched as the primary program, while the secondary is used for the quarantine-mode execution and monitoring.

On the other hand, (at the bottom of Figure 1 shown in yellow color) the *Cetratus* framework components are depicted, which are reusable and generic building blocks which enable safe life updates. The *Message Router* enables the data exchange among all the PLC program and other system components. In addition, message duplication and re-directions tasks are also performed. The *Provider* and *Dispatcher* provide system input and outputs management services, acting as wrappers to the underlying I/O drivers, such as for Digital Inputs (DI), Digital Outputs (DO) or fieldbus communications. The *Provider*, *Dispatcher* and *Message Router* provide the foundations for I/O virtualized environment. Finally, the PLC program dynamic updating process is handled by the *Updater* component, in which an indirection handling table is used. This module accomplishes the required PN model update and current marking transformations. The *Monitor* gathers and provides new PLC program execution monitoring data. The execution footprint, i.e. timing and functional behaviour is also measured.

In order to prevent any propagation of faults through the system, independence of execution both in the spatial and time domain between the PLC programs (both A and B containers) and *Cetratus* framework components is needed. Any possible program updating error is also kept under control.

In contrast to common PLC program update, in which a new application logic is supplied to the system and executed at boot, a state transformation function is necessary for a dynamic software update [6], [8]. In any time instance, the executed *PN model* and the *current marking* stored in the container shall be compatible each other. Thus, in case a new *PN model* is supplied to the system, a process to adjust and update such marking is also necessary. Equation 2 shows the *current marking* transformation function, which is executed by the *Updater* SW component shown in Figure 1 when the new PLC program is dynamically instantiated.

$$M_{new} = ST_{matrix} * M_{old} + ST_{base} \tag{2}$$

For the computation of the new marking M_{new}, in addition to the old M_{old} marking, the ST_{matrix} marking state transformation matrix and the ST_{base} marking state transformation base vector, shall be specified. These parameters are supplied by the user in the PLC program update package. Software update description and meta-data information is also commonly provided.

In case no changes are performed to the marking, neither at the compositional (length of the vector) and contents (stored PN marking data) level, the ST_{matrix} matrix would be the unitary matrix ($ST_{matrix} = I$) and ST_{base} a vector initialized with zeros ($ST_{base} = \vec{0}$) of the same length as the *current marking*. In this case, the new *PN model* would use the original marking. Therefore, the execution of the state transformation process would be not needed. It has to be pointed out that the length of the new marking M_{new} might differ from the original M_{old} one.

Therefore, the dynamic PLC program update package, to be supplied to the PLC by the *Updater*, should include the following:

- PLC program meta-data information
- PN model, described by the C matrix (see Equation 1)
- PN marking state transformation parameters, composed by (see Equation 2):
  - ST_{matrix} PN marking state transformation matrix
  - ST_{base} PN marking state transformation base vector

IV. Live updates

As described previously, two containers are created in the system (see Figure 1), A and B, which are alternatively employed for the currently executed PLC program and the quarantine-mode execution. This information is managed through the $PRIMARY$ state variable, which indicates which indicates the container in which the stable PLC application component version is located. Figure 2 shows the state-transition diagram. As can be observed, the A PLC program is initially selected on system boot.

Figure 3 shows the live patching process. Initially, the new PLC program package is supplied to the system by

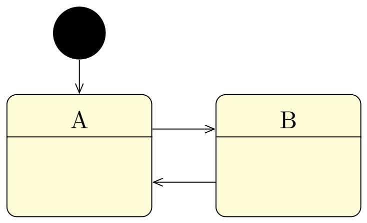


Fig. 2: PRIMARY state transition diagram

the *Updater*. This user may be the system operator or a Manufacturing Execution System (MES). The *Updater* may also abort the updating process at any time.

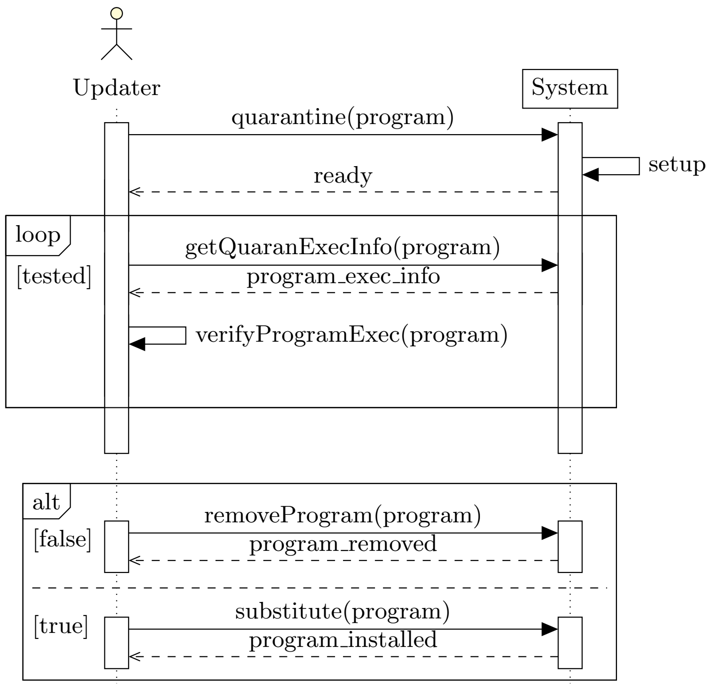


Fig. 3: Quarantine-mode based live patching sequence diagram

Initially, the primary PLC program is only executed, A or B depending on the PRIMARY state variable shown in Figure 2. At this point, the *Message Router* redirects system I/O data back and forth from the corresponding PLC program container.

On the live update request, the *Updater* module first checks which of the containers holds the primary application component by examining the PRIMARY state variable. The new PLC program is then instantiated in the secondary container. In this *setup* procedure, shown in Figure 3, two tasks are being accomplished. First, denoted *code transformation*, the new PN model is loaded into the system. Secondly, the current PN state is updated. This process is also known as *state transformation* [6], [7]. For this purpose, the old PN marking is wrapped and transferred to the secondary container. At this point, the Equation 2 is executed and a new marking, compatible with the new PN model, is computed.

As stated by *Wahler et al.* [15], the state of the program, in this case, the PN marking, has to be transformed within the same execution cycle. If this task is not accomplished in such execution cycle, an outdated, and probably corrupted, marking would be encountered by the PN model. In order to avoid this situation, a $stateTransTimeOut$ timeout notification is raised if the *Updater* module had not enough time to complete the task. The dynamic PLC program updating process is also aborted. The time needed to execute this process will depend on the computation capabilities of the PLC.

Once the *setup* phase is accomplished, the quarantine-mode execution starts, in which both the old PLC program and the new one are simultaneously executed. For this purpose, input information gathered by the *Provider* component is delivered to both programs. Nevertheless, only the outcome produced by the stable PLC program (specified by the PRIMARY state variable) is supplied to the *Dispatcher* by the *Message Router* and set as system output. At the same time, PLC program monitoring data is collected by the *Monitor* component. This information is then provided to the user in the *program_exec_info* message (in the $getQuaranExecInfo$ call shown in Figure 3). The *Updater* will continuously request, though polling, quarantine-mode PLC program execution information and verify the correctness of its execution.

Lastly, the substitution of the program is performed and the PRIMARY variable conmutated. Although, the user may abort the current dynamic updating process if the new program does not provide enough trustworthiness or fulfill the expected behaviour or correctness. This will depend on the monitored and examined data obtained in the quarantine-mode execution and monitoring phase (the $tested$ loop in Figure 3).

V. Implementation

The presented concept in this paper is implemented in an OMRON PLC, programmed in Structured Text (ST) programming language. This equipment is connected to a Machine Configuration Tool (MCT) through the Message Queuing Telemetry Transport (MQTT) protocol [16]. This protocol is widely used for collecting monitoring data from sensors and devices, for example in smart meters [17] or photovoltaic plants [18].

The MCT, which is developed using the Node-RED flow-based development tool, allows the reconfiguration of the machine. On the one hand, the tool collects and visualizes the quarantine-mode monitoring data captured from the PLC. The system variables are also received, specifically the PRIMARY (see Figure 2) and the STATUS variables. The last one indicates if a program update is under way or not.

On the other hand, MCT provides, aligned with the live patching process shown in Figure 3, the system updating features. These are:

1) Load: A new PLC program is loaded into the system, which is then automatically executed in quarantine-mode.
2) Remove: The PLC program under test is discarded.
3) Substitute: The PRIMARY variable is switched.

VI. Case Study

In this section, a reconfigurable Vernadat [19] machine case study is presented. Vernadat is a manufacturing system composed by two machines, which share a common intermediate store, with limited capacity. In this temporary buffer, the initially produced parts by the first machine are stored. These are then refined by the second machine. Loading and unloading tasks are accomplished by a robot. The Vernadat system uses infeeding and an outfeeding conveyors for raw and finished parts respectively. Figure 4 shows the Vernadat system.

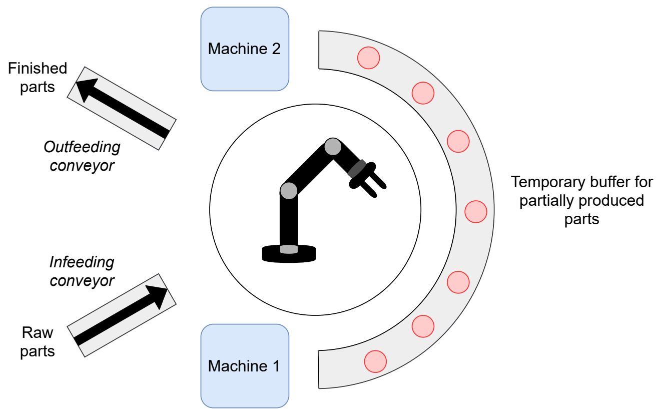


Fig. 4: Reconfigurable Vernadat system [19]

Regarding the temporary buffer for partially produced parts, seven different parts can be simultaneously be stored, in which initially, three parts are allocated. The system makes use of several mechatronical sensors, actuators and control signals for both machines and the robot. Table I provides a description of each signal.

Type	Name	Description	Element
Input	PE	Part ready for processing	Infeeding conveyor
	FCM1	Part loaded	Machine 1
	FPM1	Part processed	
	EB	Part ready	
	SB	Part ready for storing in	Temporary buffer
	FCM2	Part loaded into	Machine 2
	FPM2	Part processed by	
	PS	Finished part in	Outfeeding conveyor
Output	CM1	Part loading order into Machine 1 from infeeding conveyor	Robot
	PM1	Part processing order	Machine 1
	DM1	Move part from Machine 1 to temporary buffer	Robot
	CM2	Move part from temporary buffer to Machine 2	
	PM2	Part processing order	Machine 2
	DM2	Part unloading from Machine 2 and placement into outfeeding conveyor	Robot
	LM1	Apply lubricant	Machine 1
	LM2	Apply lubricant	Machine 2

TABLE I: Input and outputs signals of the Vernadat system

The PLC programs (the initial one and the consecutive updates) are modelled using the PIPE tool [20], [21], an open-source platform independent Petri Nets editor, simulator and analysis tool. In this case study, two consecutive program updates are performed in the reconfigurable Vernadat system. Following, firstly, the initial program being executed, and afterwards, the applied program updates are described. After that, the obtained results are shown.

A. Initial Program

Figure 5 shows the initial logical program of the Vernadat system, as well as its initial marking. As can be noted, initially, 13 places and 10 transitions are defined.

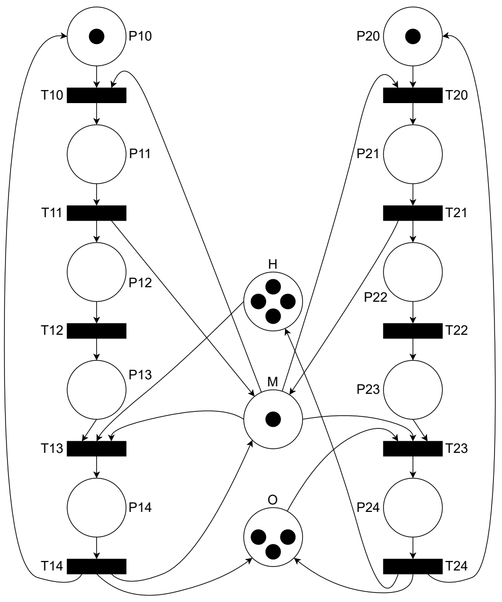


Fig. 5: Initial Vernadat system Petri net [19]

As far as its interpretation is concerned, the defined places and transitions in the Petri net are mapped to the system input and output signals. Nevertheless, it has to be mentioned that these signals might no be necessarily be linked to the PN model. This model might also make use of internal variables and functional blocks, such as timers, counters and/or comparators. Table II shows the mapping of PN transitions and places to system input and outputs signals. PN transitions are linked to system inputs while PN places to system outputs. In the initial program, the LM1 and LM2 output signals are nor assigned and neither used.

Transition	Input	Place	Output
T_{10}	PE	P_{11}	CM1
T_{11}	FCM1	P_{12}	PM1
T_{12}	FPM1	P_{14}	DM1
T_{13}	EB	P_{22}	CM2
T_{20}	SB	P_{21}	PM2
T_{21}	FCM2	P_{24}	DM2
T_{22}	FPM2	...	LM1
T_{23}	PS	...	LM2

TABLE II: Initial mapping of PN transitions and places to input and output signals

B. Programs Updates

In this section, the defined program updates and system reconfigurations are described. These reconfigurations can also include modifications to the PN interpretation. Two different updates are performed: *Buffer size change* and *New phase*.

1) U.1 - Buffer size change: In the first program update, the size of the temporary buffer is increased. For this, the current marking is updated following Equation 2. The state transformation parameters are given by Equation 3. Through this update, the buffer capacity in place H is increased by 2. No modification is performed to the PN model itself.

$$ST_{matrix} = I_{13x13}; \quad ST_{base}^{T} = (2\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0); \tag{3}$$

2) U.2 - New processing stage: In the next update, a new processing stage is created, for which an updated Petri Net program is loaded to the system. Figure 6 shows the updated Vernadat PLC program. As it can be observed, the P_{30} and P_{31} places, as well as $T30$ and $T31$ transitions are added, modelling machine lubrication operations and their finishing, respectively.

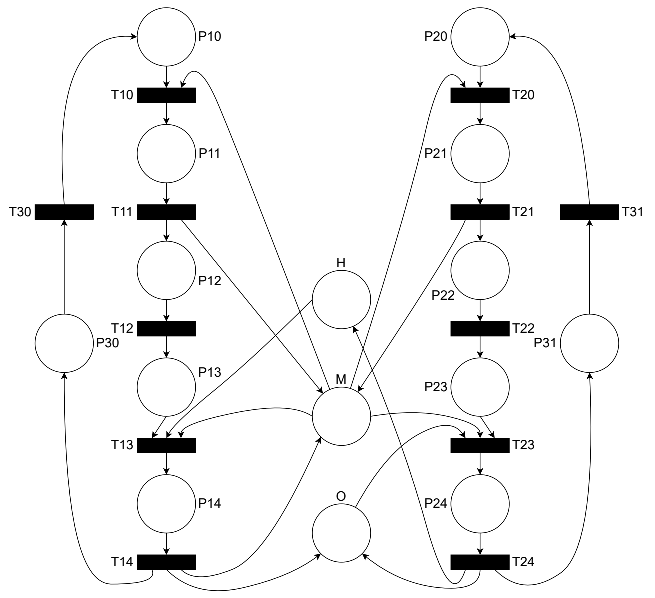
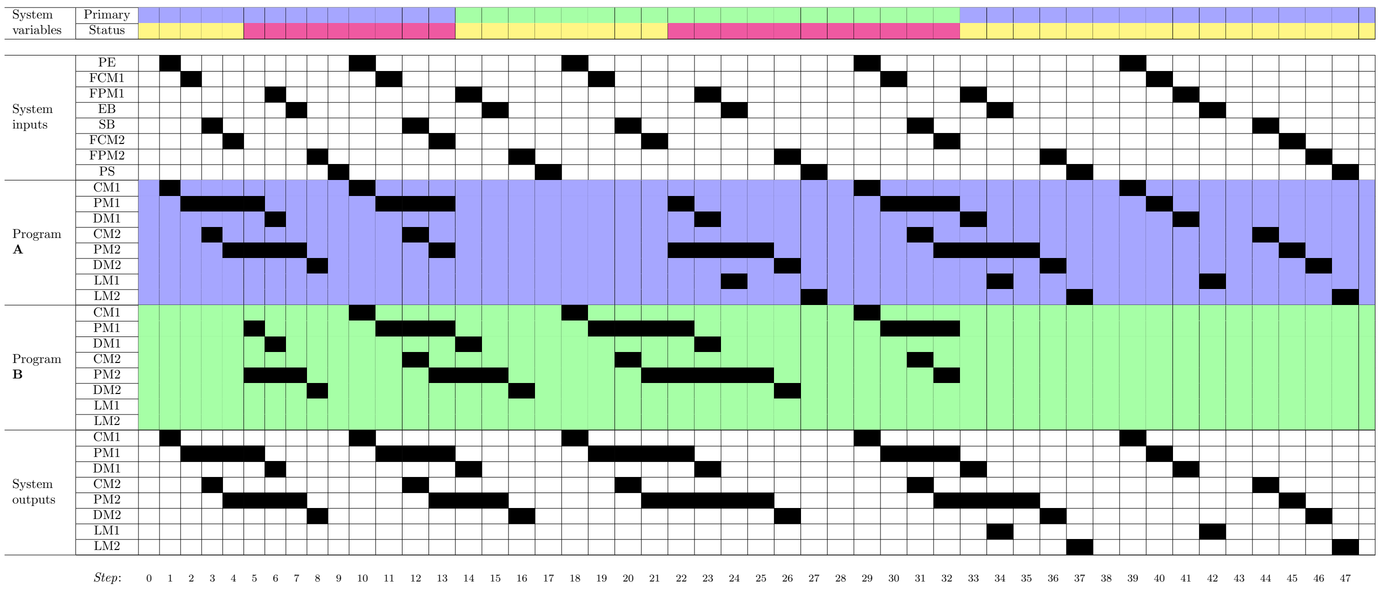


Fig. 6: Updated Vernadat system Petri net

In P_{30} and P_{31} places LM1 and LM2 outputs, not assigned in the initial program, are activated. In addition, in each of the places, a timer is activated, which after a time period (3s) transitions T_{30} and T_{31} are fired, finishing the lubrication tasks.

Equation 4 shows the state transformation parameters for the U.2 update. As it can be observed, no modification is performed on the content, just the length of the current marking is increased.

$$ST_{matrix} = \begin{pmatrix} I_{13x13} \\ O_{2x13} \end{pmatrix}; \quad ST_{base}^{T} = \vec{0}_{1x15}; \tag{4}$$



🟦	Program A	🟨	Executing
🟩	Program B	🟥	Quarantine

Fig. 7: Case study experiment results

C. *Results*

The previously described live updates are carried out in the OMRON PLC. Figure 7 shows the results of the experiment. The time diagram shows the *Primary* and *Status* system variables, outputs produced by both the *A* and *B* program containers and system inputs and outputs.

Initially, following the state machine shown in Figure 2, the *Program A* is executed and established as primary program. The outputs computed by it are then settled as system outputs. Meanwhile, no program is allocated in the *B* container. Then, the *U.1* program update is initialized in step 5. Both programs are then simultaneously executed, although, outputs from program *A* are still determined as system outputs. The substitution is performed at step 13. As observed in Figure 2, the execution of *Program A* is then halted.

Regarding *U.2* program update, the program update is set up at step 22. In contrast to the previous *U.1* update, the execution of a full Vernadat processing cycle is required for its validation, which is accomplished at step 32. Right after, program *A* is, once more, defined as the *Primary*.

As far as timing properties are concerned, Table III shows the obtained PLC execution processing times. As can be noted, the proposed PLC architecture and dynamic updating framework introduce minor overheads.

Task	Processing Time (μs)
Initial program loading	1749.7
Single program execution	531.1 (average)
New program setup	2115.8
Quarantine-mode execution	578.4 (average)

TABLE III: PLC scan cycle timings

VII. Conclusions and Future Work

In the new Industry 4.0 concept, higher productivity, flexibility and efficiency is requested. In this sense, highly configurable, flexible and evolvable mechatronic and manufacturing systems are necessary. Such capabilities will allow real-time production adjusting and tuning to satisfy market needs.

In this paper, a live updates approach for PLCs based on the *Cetratus* framework is proposed. Thought the proposed system, the executed functional program is dynamically updated. The presented framework enables a zero downtime manufacturing process, in which the manufacturing system is reconfigured to efficiently fit market needs.

In the scope of this work, the correctness verification of the new program is manually realized, by examining by hand the quarantine-mode execution data. This process is error-prone and not scaleable. For future work, the MCT could be integrated or connected to a digital twin. This would enable the automatic comparison of the digital twin simulation and the quarantine-mode information, and therefore, the automatic validation of the new program. The integration of a RMS and a digital twin was analyzed by *K.A. Kurniadi* [22]. In this work, the ST_{matrix} and ST_{base} transformation function parameters have manually been specified. The MCT tool should also incorporate automatic state transformation function generation features.

Acknowledgment

This work has partially funded by the European Union's Horizon 2020 research and innovation programme (grant agreement No 871465 (UP2DATE)).

References

[1] Y. Koren, "General rms characteristics. comparison with dedicated and flexible systems," in *Reconfigurable manufacturing systems and transformable factories*, pp. 27–45, Springer, 2006.

[2] Y. Koren and M. Shpitalni, "Design of reconfigurable manufacturing systems," *Journal of manufacturing systems*, vol. 29, no. 4, pp. 130–141, 2010.

[3] M. Bortolini, F. G. Galizia, and C. Mora, "Reconfigurable manufacturing systems: Literature review and research trend," *Journal of Manufacturing Systems*, vol. 49, pp. 93–106, 2018.

[4] H. H. Benderbal, M. Dahane, and L. Benyoucef, "A new robustness index formachines selection in reconfigurable manufacturing system," in *2015 International Conference on Industrial Engineering and Systems Management (IESM)*, pp. 1019–1026, IEEE, 2015.

[5] H. H. Benderbal, M. Dahane, and L. Benyoucef, "Modularity assessment in reconfigurable manufacturing system (rms) design: an archived multi-objective simulated annealing-based approach," *The International Journal of Advanced Manufacturing Technology*, vol. 94, no. 1-4, pp. 729–749, 2018.

[6] I. Mugarza, J. Parra, and E. Jacob, "Cetratus: Towards a live patching supported runtime for mixed-criticality safe and secure systems," in *2018 IEEE 13th International Symposium on Industrial Embedded Systems (SIES)*, pp. 1–8, IEEE, 2018.

[7] I. Mugarza, J. Parra, and E. Jacob, "Cetratus: A framework for zero downtime secure software updates in safety-critical systems," *Software: Practice and Experience*, 2020.

[8] I. Mugarza, A. Amurrio, E. Azketa, and E. Jacob, "Dynamic software updates to enhance security and privacy in high availability energy management applications in smart cities," *IEEE Access*, vol. 7, pp. 42269–42279, 2019.

[9] C. A. Petri, "Kommunikation mit automaten," 1962.

[10] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[11] A. Giua and F. DiCesare, "Grafcet and petri nets in manufacturing," in *Intelligent Manufacturing:*, pp. 153–176, Springer, 1993.

[12] A. Karatkevich, A. Bukowiec, M. Doligalski, and J. Tkacz, *Design of Reconfigurable Logic Controllers*, vol. 45. Springer, 2016.

[13] X. Chen, J. Luo, and P. Qi, "Method for translating ladder diagrams to ordinary petri nets," in *2012 IEEE 51st IEEE Conference on Decision and Control (CDC)*, pp. 6716–6721, IEEE, 2012.

[14] J. Lee and J. S. Lee, "Conversion of ladder diagram to petri net using module synthesis technique," *International Journal of Modelling and Simulation*, vol. 29, no. 1, pp. 79–88, 2009.

[15] M. Wahler, S. Richter, and M. Oriol, "Dynamic software updates for real-time systems," in *Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades*, pp. 1–6, 2009.

[16] V. Lampkin, W. T. Leong, L. Olivera, S. Rawat, N. Subrahmanyam, R. Xiang, G. Kallas, N. Krishna, S. Fassmann, M. Keen, *et al.*, *Building smarter planet solutions with mqtt and ibm websphere mq telemetry*. IBM Redbooks, 2012.

[17] M. Elsisi, K. Mahmoud, M. Lehtonen, and M. M. Darwish, "Reliable industry 4.0 based on machine learning and iot for analyzing, monitoring, and securing smart meters," *Sensors*, vol. 21, no. 2, p. 487, 2021.

[18] P. de Arquer Fernández, M. Á. F. Fernández, J. L. C. Candás, and P. A. Arboleya, "An iot open source platform for photovoltaic plants supervision," *International Journal of Electrical Power & Energy Systems*, vol. 125, p. 106540, 2021.

[19] F. DiCesare, G. Harhalakis, J.-M. Proth, M. Silva, and F. Vernadat, *Practice of Petri nets in manufacturing*. Springer, 1993.

[20] P. Bonet, C. M. Lladó, R. Puijaner, and W. J. Knottenbelt, "Pipe v2. 5: A petri net tool for performance modelling," in *Proc. 23rd Latin American Conference on Informatics (CLEI 2007)*, 2007.

[21] N. J. Dingle, W. J. Knottenbelt, and T. Suto, "Pipe2: a tool for the performance evaluation of generalised stochastic petri nets," *ACM SIGMETRICS Performance Evaluation Review*, vol. 36, no. 4, pp. 34–39, 2009.

[22] K. A. Kurniadi, S. Lee, and K. Ryu, "Digital twin approach for solving reconfiguration planning problems in rms," in *IFIP International Conference on Advances in Production Management Systems*, pp. 327–334, Springer, 2018.