



Università
Ca'Foscari
Venezia

Master's Degree Programme

in Data Management and Analytics (LM-18 – Computer Science)

Final Thesis

Software Verification of PLC Programs

Supervisor

Prof. Agostino Cortesi

Assistant supervisor

Prof. Pietro Ferrara

Graduand

Sara Ferro

Matriculation Number 858974

Academic Year

2019 / 2020

Acknowledgments

I want to thank the most important people who have helped me to reach this important moment in my life.

I begin by thanking the Prof. Agostino Cortesi above all, because He gave me the opportunity to do this project; I thank him for his help, support, encouragement and helpfulness He has shown during these months of work.

Then I want to thank also Prof. Pietro Ferrara for his assistance and for the advices He gave me.

A special thanks goes to Manali Chakraborty who has collaborated with me in carrying out this project and together we were able to achieve the main goal of this work.

I want to thank all the professors that have made me grow up in these years.

My biggest thanks goes to my family that support me and believe in me every day, and also who gave me the opportunity to study.

A final thanks goes to my friends and University colleagues who supported and helped me in times of difficulty.

Abstract

Programmable Logic Controllers (PLC) play an important role in Industrial Control Systems, as they manage the actions of physical tools by collecting data from input devices and sending commands to output devices.

In this thesis, we introduce a formal framework for software verification of the robustness of PLC programs. In particular, (i) we identify external vulnerabilities based on dynamic user interactions, (ii) we define the semantics of Structured Control Language (SCL) and the semantics of Timed Automata (TA), (iii) we provide a set of transformation rules to transform a program written in SCL to a Timed Automaton, and (iv) we show their correctness with respect to the corresponding semantics. By applying these transformation rules, we can apply Model Checking tools (namely UPPAAL) to verify robustness properties of PLC source code.

Keywords Programmable Logic Controller (PLC), Structure Control Language (SCL), Timed Automata (TA), Robustness, UPPAAL, Industrial Control Systems (ICS).

Contents

Acknowledgments	i
Abstract	ii
List of figures	iv
List of tables	v
Introduction	1
Chapter 1: PLC – Programmable Logic Controller	4
1.1 – Vulnerabilities of PLC programs	5
1.2 – Model Checking and Verification in PLC programs	8
1.3 – Problem Statement	10
Chapter 2: SCL – Structure Control Language	11
2.1 – Formal Syntax of SCL	13
2.2 – Concrete SCL Semantics	15
Chapter 3: TA – Timed Automata	16
3.1 – Formal Syntax of TA	16
3.2 – Concrete TA Semantics	17
Chapter 4: Transformation Rules from SCL to TA	19
4.1 – Example	21
4.2 – Correctness	22
Chapter 5: UPPAAL	26
5.1 – Robustness Properties	28
5.2 – File .xta Semantics	30
5.3 – Transformation Rules from SCL to .xta File	32
Chapter 6: Experimental Results	38
Conclusions	56
References	57

List of figures

Figure 1 – General view	2
Figure 2 – PLC functioning	5
Figure 3 – SCL program: square of the sum of first n numbers	12
Figure 4 – A simple SCL program	21
Figure 5 – UPPAAL interface	26
Figure 6 – Graphical representation of robustness properties	29
Figure 7 – Physical ambient	38
Figure 8 – Timed Automata in UPPAAL	46
Figure 9 – Error: out of memory	47
Figure 10 – Memory usage	48
Figure 11 – First part of the automaton	50
Figure 12 – Second part of the automaton	52

List of tables

Table 1 – TA model corresponding to the example of figure 421

Table 2 – Transformation rules from SCL to .xta file33

Table 3 – From a simple example in SCL to the .xta file to the TA35

Introduction

In modern industries, attempts made to automate processes are increasingly frequent, and Programmable Logic Controllers (PLCs) are an enabling technology to achieve this.

A Programmable Logic Controller is a device used to automate industrial processes, receiving inputs from physical devices, such as valves or sensors, processing them, making decisions based on the program installed on it, and sending commands to the output devices it controls, such as motors.

These controllers can automate specific processes, machines, or production lines. The advantages of adopting PLC are the ability to reprogram, change sequences, extend lines, create replicas of machines and processes, all while we can collect and communicate vital data.

Being a crucial device in an Industrial Control System (ICS) and being often highly user interactive and input dependent, PLCs are often threatened by cyber attacks; consequently, physical devices might become unsafe and not always reliable, and so we have to introduce the concept of robustness.

Robustness is the ability of a program to operate even under abnormal or adverse conditions and events. Given an unexpected or erroneous input, a robust PLC should still give an “acceptable” output. So, it is very important to build a robust PLCs able to achieve a certain kind of tolerance for input values.

The objective of this thesis is to design and evaluate a framework for the robustness verification of PLC systems, i.e. that the system results into an acceptable output even when external attacks compromise data provided by dynamic interaction. In particular, we formalize a semantics-based methodology to automatically derive Timed Automata (TA) based models from code written in SCL language, in order to model-check robustness conditions expressed by temporal logic formulas.

The methodology used to reach our goal starts from a PLC program written in the Structure Control Language (SCL). We then transform it into a Domain Specific Language (DSL) using Xtext (open-source software framework for developing domain specific languages). Starting from this DSL code, we generate timed

automata based models, creating the .xta files, to feed UPPAAL (a toolbox for verification of real-time systems); such process can be executed with the help of Acceleo. Finally, to achieve our objective and so in order to verify the robustness of our system, using the automaton in UPPAAL we check if some properties are satisfied or not.

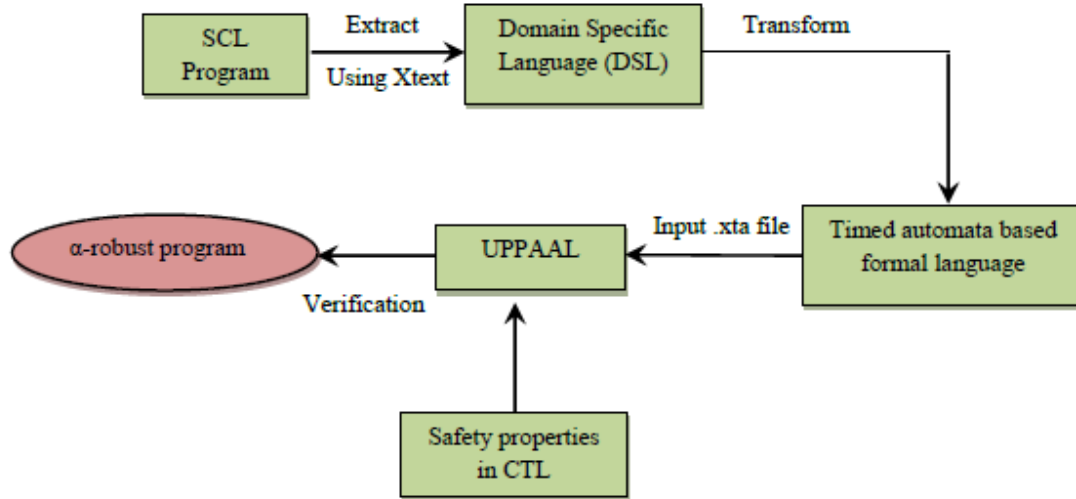


Figure 1 – General view

The thesis is structured in the following way. In the first chapter we will introduce PLCs, what they are, what they are used for, then we will present their vulnerabilities and possible attacks that can be made to them; we present the problem statement, that given a program P we want that it gives a correct output either with acceptable or unacceptable inputs. In the second chapter we introduce what the SCL language is, and how it is structured; we define the formal syntax of SCL and the concrete SCL semantics. In the third chapter we explain what Timed Automata (TA) are, we define their grammar and the concrete TA semantics. In chapter 4 we provide a set of transformation rules to pass from a SCL program to a timed automata-based model, and we give also theoretical proves of correctness for these translation rules. In chapter 5 we explain what is UPPAAL, how it is structured and what it permits us to do; we present the five robustness types of properties that can be checked with UPPAAL, we define the semantics of .xta files, and we list the transformation rules to pass from a program

written in the SCL language to a .xta file; in this chapter we provide also a simple example, where we show starting from a PLC program written in SCL how we transform it into a .xta file, and how an automaton is automatically created. In the last chapter we report the experimental results, giving insights of how our approach deals with some examples; our focus in this chapter is to transform a specific PLC program written in SCL to an .xta file, create automatically the timed automata, and demonstrate some properties on such automaton using UPPAAL in order to check the robustness of the system taken in consideration.

1 PLC – Programmable Logic Controller

A Programmable Logic Controller is a computer, used in automatic engineering or industrial automation, to automate electromechanical processes, such as controlling factory machinery in assembly lines or mechanical attractions. PLCs use programmable memory to store instructions implementing certain functions, such as logical operations, action sequences, time specifications, counters, and calculations for control by analog or digital I/O modules on different types of machines and of processes. A PLC typically has three main components, namely, an embedded operating system, control system software, and analogical and digital inputs/outputs. The field of application of PLCs is very wide and includes various kinds of industries, such as automotive, aerospace, construction, etc. PLCs are commonly found in Supervisory Control and Data Acquisition (SCADA) systems as field devices. Because they contain a programmable memory, PLCs allow a customizable control of physical components through a user programmable interface [3].

One of the advantages of the PLCs is that, thanks to them, it is possible to carry out operations in real time (such as monitoring the productivity of a machine or the operating temperature, automatically starting or interrupting a process, generating alarms in case of malfunction, etc.), due to their reduced reaction time. In addition, they are devices that adapt easily to new tasks due to their flexibility when programming them, thus reducing additional costs when preparing projects. They also allow immediate communication with other types of controllers and computers and even allow network operations. They can be easily programmed through various programming languages. However, they have certain drawbacks such as the need to have qualified technicians to take care of their proper functioning.

In modern industries, more and more attempts are made to automate processes, PLCs are a key mechanism to achieve this aim; in fact, they allow first of all to eliminate the presence of humans in certain jobs, perhaps even dangerous, and also to speed up certain processes. Today we are surrounded by such automation, as well as in industries, we also find them in traffic lights, lighting

management in parks, gardens and shop windows, automatic doors control and even in the control of household devices such as windows, air conditioning, etc. A PLC works by receiving information from sensors and input devices, processing the data and controlling actuators and output devices according to the logic of the installed programs. The flow of a PLC is illustrated, in a simplified way, in the following figure:

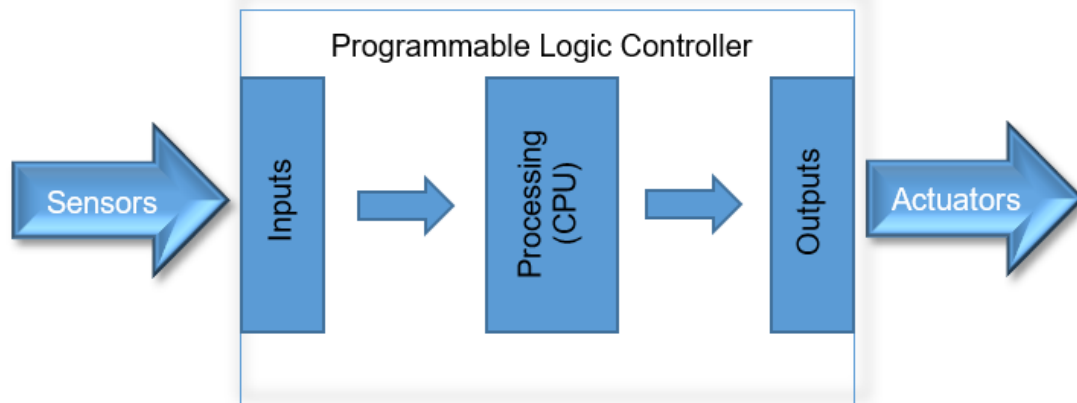


Figure 2 – PLC functioning

1.1 Vulnerabilities of PLC programs

We have already said that PLCs are a crucial device in ICS and they are often input dependent and user interactive, then PLCs have to deal with cyber security attacks. In this section we will see the main vulnerabilities of PLC programs.

Generally, the attacks on PLCs can be classified in:

- Access Control Attacks: steal the user credentials, pretend to be the user to make passive attacks;
- Firmware Modification: can provide an adversary with complete control over an industrial control device and any physical system components that come under its purview;
- Control Flow Attacks: redirect the flow of execution of the statements, instructions and functionalities of a program;
- Configuration Modification: change the default configuration of a device;
- Communication Channel Attacks: intercept the communication between

two or more devices in order to be able to capture critical data.

In [4], the authors proposed three different types of attacks on PLC registers: replay attack, man-in-the-middle attack and S7 Authentication Bypass Attack. They explored the Siemens PLC access control vulnerability by reading and writing the PLC's intermediate register data to achieve the effect of abnormal communications. In the PLC architecture, the CPUs execute the results of the program into the intermediate registers. Thus rewriting the values of intermediate registers can affect the ongoing process in PLC. They attacked the Siemens S7 series controllers, such as S7-200, S7-300, S7-400, S7-1200 and so on.

In [3], authors carried out a security analysis of the most common PLC access control mechanism, namely, password-based access control. They showed how passwords are stored in PLC memory, how they can be intercepted in the network, how they can be cracked, etc. As a consequence of these vulnerabilities, they could carry out advanced attacks on ICS system setup, such as replay, PLC memory corruption, etc.

Firmware alteration is another type of attack in PLC. In this domain, several works explained different methods of how to perform this type of attacks, and how disruptively it can affect PLC security. Authors of the [5] performed a version number update of the PLC by exploiting the firmware. First, they searched the firmware for locations that referenced the version number. Then using reverse engineering, they inspected the disassembled information and modified the version number bytes appropriately. They also calculated the correct checksum values for modified data and updated the new firmware binary file. The utility then revalidated the binary to confirm that the checksum values were correctly updated.

Another work on firmware modification is reported in [6], where the authors implemented a stealthy attack on firmware, by manipulating the input and output lines. The firmware acts as an intermediate level between the main control section of the PLC and the outer world. The inputs towards PLC's control logic passes through the firmware layer, as well as the outputs from the PLC. Thus, the attackers gain the insight knowledge of these communications. In particular, the PLC's control of input and output lines, and the connection between the

firmware and control logic programs, by using reverse engineering to the PLC, and provide fake information to the outer world.

In [7], the authors performed another firmware modification attack by exploiting the shortcomings of ICS in PLC security that does not consider the dynamic changes of memory contents as well as control flow. They developed a rootkit on the CODESYS PLC runtime to intercept I/O operations of the payload program. When the payload wants to read or write a certain I/O pin, interrupt handler installed by the attacker is called first, within which the attacker can reconfigure the I/O pins or modify values to be read/written.

Firmware attacks typically require detailed knowledge on target PLC's hardware components and reverse-engineering of its firmware because PLCs are closed-source embedded devices [8]. An attacker needs to install the rootkit on PLCs either via the built-in remote firmware update mechanism or by loading it via the JTAG interface [6]. For firmware updates protected by cryptographic means (e.g., certificate in the X.509 standard), it is hard to install a modified version of the firmware on the PLC. Alternatively, an attacker can load modified PLC firmware via JTAG interface. However, such an approach will require physical access to the PLC and possibly to disassemble it.

In a very recent work [9], authors proposed a runtime monitoring to develop runtime behaviour models from control system specifications to detect PLC payload attacks. Payload attacks are much easier to implement than firmware attack. It can be easily done if an attacker gain access to the PLC. The proposed solution in this paper can effectively detect the payload attacks. However, it suffers from memory overhead and execution time overhead.

In [10], a bump-in-the-wire device, called PLC guard, is introduced to intercept the communication between an engineering workstation and a PLC, allowing engineers to review the code and compare it against previous versions. Features of the PLC guard include various levels of graphical abstraction and summarization, which makes it easier to detect malicious code snippets.

In [11], an external runtime monitoring device (e.g., a computer or an Arduino microcontroller board) sits alongside the PLC, monitors its runtime behaviours (e.g., inputs, outputs, timers, counters), and verifies them against ICS

specifications converted from a trusted version of the PLC payload program and written in interval temporal logic. It is shown that functional properties of payload program can be verified against ICS specifications, but the types of payload attacks that can be detected by this approach remain to be explored.

In [12] and [13], a trusted safety verifier is introduced as a bump-in-the-wire device that automatically analyses payload program to be downloaded onto a PLC and verifies whether critical safety properties are met using linear temporal logic. However, linear temporal logic implicitly assumes that states of the systems are observed at the end of a set of time intervals. In the case of PLC payload program, snapshots of system states are taken at the end of each program scan cycle. As a result, real-time properties that do not span multiple program scan cycles cannot be checked by the trusted safety verifier. For example, a legitimate payload program is required to energize its output immediately when a certain input pin is energized. An attacker can inject malicious code and prolong the program scan cycle to cause real-time property violation while evading code analytics based on linear temporal logic.

1.2 Model Checking and Verification in PLC programs

With the recent advances in safety critical systems and the increasing complexity of safety parameters, researchers are compelled to pay more attention on the formalization and verification of PLC programs, so that they can verify the PLC programs against various safety parameters before execution.

Authors of [16] and [17] presented a model transformation process for IEC 61131-3 Function Block Diagrams (FBD) to timed automata in UPPAAL for automated verification of safety parameters. They took the PLC Open XML specification of FBD and transformed to the UPPAAL based XML format for timed automata.

Authors of [18] have applied formal methods to perform the verification of PLC programs written in the IL (Instruction List) language. This method consists in applying symbolic model checking techniques in the framework of PLC programs. The specific elements of their approach are:

- the choice of a significative fragment of the IL language, allowing to write

some simple programs;

- a sharp transition system-based operational semantics of this fragment;
- a coding of these transition systems into the input language of a model checker (like Cadence SMV);
- the use of the LTL linear temporal logic to write behavioural properties.

Although based on simple and well-known concepts, this approach allows to prove or reject, in a completely automated way, the correctness of IL programs of a non-trivial size. A similar study on the validation of PLC programs has already been presented for LD (Ladder Diagrams) programs in [19].

A specialized group in CERN laboratory has been working on PLC for the last 5-6 years. They have published quite a few papers on model checking and also built a tool (PLCVerif) for verifying PLC programs based on various model checking techniques [30].

Generally, every PLC program has a very simple life cycle, consisting of:

1. Scan
2. Input
3. Process
4. Output
5. Reset

As discussed in this section, the majority of these attacks in PLC are external, and target the process cycle of the PLC. However, PLC is often used in highly secure environment with several protective measures to secure it from outside world. Still, it can get affected from internal attacks, which may or may not be intentional. Besides, PLCs are widely used in automated safety critical systems, where a tiny failure can have disruptive effects. Thus, it is always better to check the robustness of a PLC program before executing.

There exist some papers in the domain of static analysis, formal methods and model checking on PLC programs. Generating models from PLC code and then verify it using some model checking tools have been proposed in many papers. Still, PLC programs are not fully secure yet, specially from the internal attacks. Besides, analysing the dependability between the inputs, which may lead

to a cascading failure for an erroneous input is still an open research area. Also, PLC is often used in safety critical systems. Hence, to define and ensure the acceptable and unacceptable outputs for a given set of inputs is highly important.

Hence, in this thesis, we introduce a framework to transform a PLC program written in SCL programming language to Timed Automata based format acceptable by UPPAAL. Then we verify the robustness of that model for every user given input value.

1.3 Problem Statement

Given a program P , where Σ is the set of all program states, the semantics of P can be expressed as:

$$\cup \{ \langle \delta_0^i, \dots, \delta_n^i, \dots \rangle : i \in I, \delta_h^i \in \Sigma \}$$

where, I is the set of user dependent input values, and δ_h^i represents the h^{th} program memory state depending on the input value i .

We further assume that $\Sigma_E \subseteq \Sigma$ and $\Sigma_\Delta \subseteq \Sigma$, where Σ_Δ is the set of acceptable final states and Σ_E is the set of erroneous states that are properly caught.

We also assume that $I = I_A \cup I_U$, where I_A is the set of acceptable inputs and I_U the set of unacceptable inputs, possibly due to an internal attack.

Our objective is to check the robustness of P i.e., we want to make sure that if P runs with $i \in I_U$ then considering the execution:

$$P(i) = \langle \delta_0^i, \dots, \delta_n^i, \dots \rangle$$

- 1) either $P(i)$ gets into Σ_E , i.e., $\exists k \geq 0 : \delta_k^i \in \Sigma_E$

or

- 2) $P(i)$ is finite and its last element, belongs to Σ_Δ and it yields to an output value τ which is anyway acceptable to the user, i.e., the presence of an erroneous input has effect on the overall compilation.

2 SCL – Structure Control Language

Our work is based on SCL (Structured Control Language), which is a high-level textual programming language based on PASCAL [21]. A program in SCL can call programs in other PLC languages and programs in other PLC languages can call programs in SCL. SCL can be structured as a sequence of various blocks, such as:

- Organization Blocks (OB): determine the structure of the program. The organization block for normal program execution on PLCs is determined in OB1. This block determines the cyclic semantics of the PLCs, and it represents the interface between main system and PLC.
- Function Blocks (FB): are functions which can also store data between function calls; can be called by OB and other FBs. Has internal memory.
- Functions (FC): correspond to functions we know from programming; can be called by OB and FB with its parameters and has no memory.
- Data Blocks (DB): are used for storing and sharing data, helping to store user data.
- User-defined Data Types (UDT): are used to define complex data types and used for storing user defined data types.

In addition to high-level language elements, SCL also includes language elements typical of PLCs such as inputs, outputs, timers, bit memory, block calls, etc. In other words, SCL complements and extends the STEP 7 programming software and its programming languages Ladder Logic and Statement List [20].

The programs we use in the experiments start with a function block, to which we will refer as the main function block. These programs can have calls to other function blocks, functions, data blocks and data types. All functions and function blocks in SCL can have variables of different types. Input variables get values from the calling block. For the topmost function block, the input variables get values from the input ports. Output variables are used to return values to the calling block. For the topmost function block, the output variables contain the values that are sent to the output ports. In-output variables are a combination of input variables and output variables, these variables get values from the calling

block, or input ports, and return values to the calling block, or sent them to the output ports. Static variables can be used within the blocks. Function blocks have access to memory, therefore they can keep the values of static variables after the program has returned to the calling block. This also makes possible for these variables to have an initial value. A function has no memory, therefore static variables in a function have no initial values and do not keep their values after the program has returned to the calling block.

SCL uses control statements to take care of selective instructions and repetition instructions. The control statements we use are: IF, ELSEIF, ELSE, and WHILE. SCL also supports case distinction, loops and jump statements. For conditional expressions the standard Boolean operators can be used. The predefined data types we use are: BOOL, INT, UINT, WORD, ARRAY, STRUCT, TIME, and REAL. Other predefined data types are dates, chars, timers and doubles. The data types ARRAY and STRUCT do not have a specified size, because the size varies per specification.

A simple example of a SCL code is:

```
FUNCTION TEST : INT /*square of the sum of first n numbers */
VAR_INPUT
    n : INT;
END_VAR

VAR_OUTPUT
    SUM : INT;
    SQUARE : INT;
END_VAR

VAR
    x:INT;
END_VAR

BEGIN
    n:=5;
    x:=0;
    SUM:=0;
    SQUARE:=0;

    WHILE x<n DO
        SUM:=SUM+x;
        x:=x + 1;
    END_WHILE;

    IF SUM <= 181 THEN
        SQUARE := SUM * SUM; // Calculates function value
    ELSE
        SQUARE := 32767; // Set to maximum value in the event of overrun
    END_IF;

END_FUNCTION
```

Figure 3 - SCL program: square of the sum of first n numbers

2.1 Formal Syntax of SCL

According to the IEC-61131-3 standard [23], every PLC program consists of one or many POU (Programming Organization Unit). These POUs are the smallest executable units of each PLC program and can be of several types (as termed by the SIEMENS Simatic-STEP7): Organization Block, Function Block, Function, Data Block and User-defined Data Type.

A SCL program can be defined as a list of *statements*, and each statement can be defined as a collection of *keywords* and *expressions*, terminating by a ';'. While, statements are the basic elements of a SCL program, a *block* is a basic executable unit in a SCL program. In this thesis we generate a timed automata based model for each block of a SCL program. Now, we'll define the semantics of a simple block for a program written in SCL.

Generally, the statements within a block can be roughly categorized in five sections, as:

- Block start statements: the start statements are consisting of a unique keyword for each type of blocks following by the name of that block;
- Block attribute statements: attribute statements can be of two types: Block attributes and System attributes for blocks;
- Declaration statements: the declaration section must contain all specifications required to create the basis for the code section, for example, definition of constants and declaration of variables and parameters.
- Code statements: the code section is introduced by the keyword BEGIN and terminated with END_*, where '*' represents the type of that particular block.
- Block end statements: it is similar to the start statements, but it has only keywords for each block.

We can define the semantics of an SCL, according to IEC 61131-3, as:

$keywords \in \text{Keywords}$

$\text{Keywords} = \{\text{BEGIN, FUNCTION, END_FUNCTION, FUNCTION_BLOCK, END_FUNCTION_BLOCK, ORGANIZATION_BLOCK, END_ORGANIZATION_BLOCK, DATA_BLOCK, END_DATA_BLOCK, VAR, VAR_TEMP, VAR_IN_OUT, VAR_INPUT, VAR_OUTPUT}\}$

$x, y \in \text{Var}$ (variables)

$n \in \text{Num}$ (numbers)

$t \in \text{Type}$ (datatype of variables)

$l \in \text{Lab}$ (labels)

$a \in \text{Exp}_A$

$b \in \text{Exp}_B$

$S \in \text{stat}$

$op_A \in A_{op}$ Arithmetic operator

$A_{op} = \{+, -, *, \%, /\}$

$op_B \in B_{op}$ Boolean operator

$B_{op} = \{\text{AND, OR, XOR, NOR}\}$

$op_R \in R_{op}$ Relational operator

$R_{op} = \{\leq, <, >, \geq, =, \neq\}$

$a ::= x | n | a_1 \ A_{op} \ a_2$

$b ::= x | n | \text{true} | \text{false} | b_1 \ B_{op} \ b_2 | a_1 \ R_{op} \ a_2 \mid \text{not } b$

$S ::= x := a \mid$

$S_1; S_2 \mid$

$\text{if } [b] \text{ then } S_1 \text{ else } S_2 \mid$

$\text{while } [b] \text{ do } S$

2.2 Concrete SCL Semantics

The operational semantics of SCL can be described as:

$$\frac{n \in \mathbb{N}}{\langle n, \Sigma \rangle \xrightarrow{\delta} n} \quad (1)$$

$$\frac{x \in Var}{\langle x, \Sigma \rangle \xrightarrow{\delta} \Sigma(x)} \quad (2)$$

$$\frac{keywords \in Keywords}{\langle keywords, \Sigma \rangle \xrightarrow{\delta} \Sigma(keywords)} \quad (3)$$

$$\frac{\langle a_1, \Sigma \rangle \xrightarrow{\delta} v_1 \quad \langle a_2, \Sigma \rangle \xrightarrow{\delta} v_2}{\langle a_1 A_{op} a_2, \Sigma \rangle \xrightarrow{\delta} v_1 A_{op} v_2} \quad (4)$$

$$\frac{}{\langle \text{TRUE}, \Sigma \rangle \xrightarrow{\beta} \text{TRUE}} \quad (5)$$

$$\frac{}{\langle \text{FALSE}, \Sigma \rangle \xrightarrow{\beta} \text{FALSE}} \quad (6)$$

$$\frac{\langle a_1, \Sigma \rangle \xrightarrow{\delta} v_1 \quad \langle a_2, \Sigma \rangle \xrightarrow{\delta} v_2}{\langle a_1 R_{op} a_2, \Sigma \rangle \xrightarrow{\beta} v_1 R_{op} v_2} \quad (7)$$

$$\frac{\langle b_1, \Sigma \rangle \xrightarrow{\beta} vb_1 \quad \langle b_2, \Sigma \rangle \xrightarrow{\beta} vb_2}{\langle b_1 B_{op} b_2, \Sigma \rangle \xrightarrow{\beta} vb_1 B_{op} vb_2} \quad (8)$$

$$\frac{\langle a, \Sigma \rangle \xrightarrow{\delta} v}{\langle T \ x = a, (\Phi, \Sigma) \rangle \rightarrow (\Phi \cup [x \rightarrow T], \Sigma \cup [x \rightarrow v])} \quad (9)$$

$$\frac{\langle S_1, (\Phi, \Sigma) \rangle \rightarrow (\Phi', \Sigma') \quad \langle S_2, (\Phi', \Sigma') \rangle \rightarrow (\Phi'', \Sigma'')}{\langle S_1; S_2, (\Phi, \Sigma) \rangle \rightarrow (\Phi'', \Sigma'')} \quad (10)$$

$$\frac{\langle b, \Sigma \rangle \xrightarrow{\beta} \text{TRUE} \quad \langle S_1, (\Phi, \Sigma) \rangle \rightarrow (\Phi', \Sigma')}{\langle \text{if } b \text{ then } S_1 \text{ else } S_2, (\Phi, \Sigma) \rangle \rightarrow (\Phi', \Sigma')} \quad (11)$$

$$\frac{\langle b, \Sigma \rangle \xrightarrow{\beta} \text{FALSE} \quad \langle S_2, (\Phi, \Sigma) \rangle \rightarrow (\Phi', \Sigma')}{\langle \text{if } b \text{ then } S_1 \text{ else } S_2, (\Phi, \Sigma) \rangle \rightarrow (\Phi', \Sigma')} \quad (12)$$

$$\frac{\langle b, \Sigma \rangle \xrightarrow{\beta} \text{TRUE} \quad \langle S; \text{while } b \text{ do } S, (\Phi, \Sigma) \rangle \rightarrow (\Phi', \Sigma')}{\langle \text{while } b \text{ do } S, (\Phi, \Sigma) \rangle \rightarrow (\Phi', \Sigma')} \quad (13)$$

$$\frac{\langle b, \Sigma \rangle \xrightarrow{\beta} \text{FALSE}}{\langle \text{while } b \text{ do } S, (\Phi, \Sigma) \rangle \rightarrow (\Phi, \Sigma)} \quad (14)$$

3 TA – Timed Automata

A timed automaton [15] is essentially a finite automaton (that is a graph containing a finite set of nodes or locations and a finite set of labelled edges) extended with real-valued variables. Such an automaton may be considered as an abstract model of a timed system. The variables model the logical clocks in the system, that are initialized with zero when the system is started, and then increased synchronously with the same rate. Clock constraints (i.e., guards on edges) are used to restrict the behaviour of the automaton. A transition represented by an edge can be taken when the clocks values satisfy the guard labelled on the edge. Clocks may be reset to zero when a transition is taken.

3.1 Formal Syntax of TA

In order to define the syntax of TA, we first define:

$x_t, y_t \in \text{Var}$ (timed variables)

$t \in \text{Clk}$ (clock)

$n \in \text{Num}$ (numbers)

Γ is the environment mapping from variables to numbers.

$g \in G$ (set of guards or clock constraints)

$g ::= x_t \mid \text{true} \mid \text{false} \mid x_t \text{ } R_{op} \text{ } x_t$ (guard)

$a, z \in \Delta$ (set of actions)

$a ::= s := y$ (assign)

$y ::= x_t \mid n \mid x_t \text{ } A_{op} \text{ } x_t$ (assign)

$z ::= x_t \mid n \mid ! \mid ? \mid \text{true} \mid \text{false} \mid x_t \text{ } R_{op} \text{ } x_t \mid x_t \text{ } R_{op} \text{ } n$ (sync)

$A_{op} = \{ +, -, *, \%, / \}$

$R_{op} = \{ \leq, <, >, \geq, =, \neq \}$

Then, a timed automaton TA can be defined as a tuple (L, l_0, E, S) where:

- L is a finite set of locations (or nodes);
- $l_0 \in L$ is the initial location;
- $E \subseteq \langle L \times G \times \Delta \times L \rangle$ is the set of edges;

- $S: Var \rightarrow Clk \rightarrow Num$ is the set of states, that returns the values of variable at a particular time.

3.2 Concrete TA Semantics

The semantics of a timed automaton is defined as a transition system where a state or configuration is a pair of the current location and the current values of variables at that time, i.e., $\langle l, S \rangle$.

There are two types of transitions between states. The automaton may either delay for some time (a delay transition) or follow an enabled edge (an action transition).

The transitions can be defined as:

$$\frac{n \in \mathbb{N}}{\langle n, \Gamma \rangle \xrightarrow{\tau} n} \quad (15)$$

$$\frac{x \in Var}{\langle x, \Gamma \rangle \xrightarrow{\tau} \Gamma(x)} \quad (16)$$

$$\frac{\langle x_1, \Gamma \rangle \xrightarrow{\tau} u_1 \quad \langle x_2, \Gamma \rangle \xrightarrow{\tau} u_2}{\langle x_1 A_{op} x_2, \Gamma \rangle \xrightarrow{\psi} u_1 A_{op} u_2} \quad (17)$$

$$\frac{\langle x_1, \Gamma \rangle \xrightarrow{\tau} u_1 \quad \langle x_2, \Gamma \rangle \xrightarrow{\tau} u_2}{\langle x_1 R_{op} x_2, \Gamma \rangle \xrightarrow{\tau} u_1 R_{op} u_2} \quad (18)$$

$$\frac{x \in Var, t \in Clk}{\langle x, t, s \rangle \xrightarrow{\alpha} s(x_t)} \quad (19)$$

$$\frac{\langle l, g, \delta, l' \rangle \in E, s(g_t) = true, s' = s(\delta_{t+1})}{\langle l, s, t \rangle \rightarrow \langle l', s' \rangle} \quad (20)$$

where $\langle l, s, t \rangle$ is the current location and state of a TA at time t , and if the guard conditions are true, i.e., if $g_t = true$ at time t , then the transition will be made at $\langle l', s' \rangle$. Also, for this transition, the action can be either a(assign) or z(sync), or both.

$$\frac{\exists \langle l, g, \Delta, l' \rangle \in E}{s(g, t) \rightarrow true} \quad \frac{\nexists \langle l, g, \Delta, l' \rangle \in E}{s(g_t) \rightarrow false} \quad (21)$$

The value of $s(g_t)$ is true, if there exists an edge between l and l' , at time t . Otherwise $s(g_t)$ is false.

$$\frac{s(g_t) = true}{\langle s(g_t), t + 1, s \rangle \xrightarrow{f} s(g_{t+1})} \quad (22)$$

$$\frac{\delta \in \Delta}{\langle s(\delta_t), t + 1, s \rangle \xrightarrow{f} s(\delta_{t+1})} \quad (23)$$

$$\frac{s(g_t) = false, s' = s(\delta_{t+1})}{\langle l, s, t \rangle \rightarrow \langle l, s', t + 1 \rangle} \quad (24)$$

while the guard is not satisfied for an edge, the TA stays in the same location l .

However, the state s will change from s to s' , where $s'(\delta_{t+1}) = s(s(\delta_t), t + 1)$.

4 Transformation Rules from SCL to TA

The transformation function considers one statement of SCL program at a time and produces its corresponding transition in a timed automata model. As we already described earlier a timed automata can be described as a 4-tuple $\langle L, l_0, E, S \rangle$. We can represent the transformation between SCL programs into TA models using the function $\Pi: \langle inst_{SCL}, A, l_{in} \rangle \rightarrow \langle A', l_{fin} \rangle$, where $inst_{SCL}$ is a SCL program, and A is the timed automata corresponding to the SCL program, and l_{fin} is the final location or ending node of the TA.

The Π function considers change of states during the execution of SCL program and maps it to its corresponding equivalent transition in the TA model.

Now, A can be represented as a 4-tuple $\langle L, l_0, E, S \rangle$ where L is the set of nodes or locations in the TA, $l_0 \in L$ denotes the starting node of the TA, E is the set of edges, that can further be described as $\langle L \times G \times \Delta \times L \rangle$, $e \in E = (l, g, \delta, l')$, where e is an edge or transition in TA from current location l to next location l' , with guard g and actions δ .

Given a timed automata A , the transformation rules from a SCL statement to a corresponding TA transition can be described as, $\Pi \langle inst_{SCL}, A, l_{in} \rangle \rightarrow \langle A', l_{fin} \rangle$. Now, the TA model will start at the receiving of *BEGIN* keyword in the SCL program. The variable declaration parts will be added in the corresponding *.xta* file.

Hence, at the beginning, when the automata is empty, $A = (L = \{\emptyset\}, l_0 = \varepsilon, E = \{\emptyset\}, S)$. The translation rules are describes here:

$$\Pi \langle BEGIN, \varepsilon, \varepsilon \rangle \rightarrow \langle L \cup \{l_{fin} = new(node)\}, l_0 = l_{fin}, \{\emptyset\}, S, l_{fin} \rangle \quad (25)$$

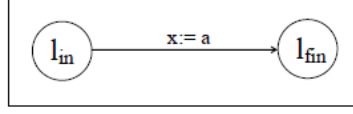
$$\Pi \langle END_FUNCTION_BLOCK, A, l_{fin} \rangle \rightarrow \langle A, l_{fin} \rangle \quad (26)$$

$$\Pi \langle END_ORGANIZATION_BLOCK, A, l_{fin} \rangle \rightarrow \langle A, l_{fin} \rangle \quad (27)$$

$$\Pi \langle END_FUNCTION, A, l_{fin} \rangle \rightarrow \langle A, l_{fin} \rangle \quad (28)$$

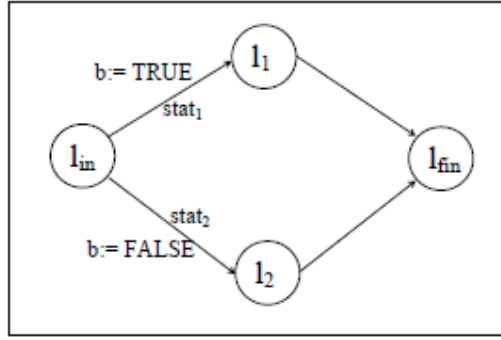
$$\Pi \langle END_DATA_BLOCK, A, l_{fin} \rangle \rightarrow \langle A, l_{fin} \rangle \quad (29)$$

$$\Pi\langle x := a, A, l_{in} \rangle \rightarrow \langle L \cup \{l_{fin} = \text{new}(\text{node})\}, l_0, E \cup \{(l_{in}, \text{TRUE}, x := a, l_{fin})\}, S, l_{fin} \rangle \quad (30)$$

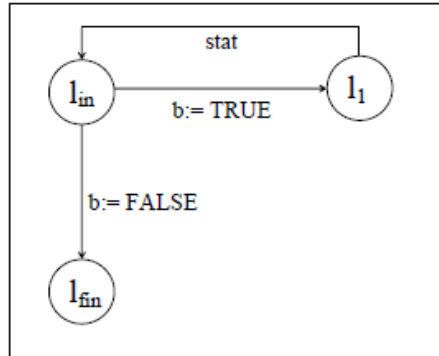


$$\frac{\Pi\langle \text{stat}_1, A, l_{in} \rangle \rightarrow \langle A', l_{fin} = \text{new}(\text{node}) \rangle \quad \Pi\langle \text{stat}_2, A', l_{in} \rangle \rightarrow \langle A'', l'_{fin} = \text{new}(\text{node}) \rangle}{\langle \text{stat}_1, \text{stat}_2, A, l_{in} \rangle \rightarrow \langle L \cup \{(l_{fin}, l'_{fin})\}, l_0, E \cup E', S \cup S', l'_{fin} \rangle} \quad (31)$$

$$\begin{aligned} \Pi\langle (\text{IF } b \text{ THEN } \text{stat}_1 \text{ ELSE } \text{stat}_2 \text{ ENDIF}), A, l_{in} \rangle \rightarrow & \langle L \cup \{l_1 = \text{new}(\text{node}), l_2 = \text{new}(\text{node}), \\ & l_{fin} = \text{new}(\text{node})\}, E \cup \{(l_{in}, b ::= \text{TRUE}, \text{stat}_1, l_1), (l_{in}, b ::= \text{FALSE}, \text{stat}_2, l_2), \\ & (l_1, \text{TRUE}, \epsilon, l_{fin}), (l_2, \text{TRUE}, \epsilon, l_{fin})\}, S \cup \{S[l_1], S[l_2]\}, l_{fin} \rangle \end{aligned} \quad (32)$$



$$\Pi\langle (\text{WHILE } b \text{ DO } \text{stat}), A, l_{in} \rangle \rightarrow \langle L \cup \{(l_1 = \text{new}(\text{node}), l_{fin} = \text{new}(\text{node}))\}, l_0, E \cup \{(l_{in}, b ::= \text{TRUE}, \epsilon, l_1), (l_{in}, b ::= \text{FALSE}, \epsilon, l_{fin}), (l_1, \text{TRUE}, \text{stat}, l_{in})\}, S \cup \{S[l_1]\}, l_{fin} \rangle \quad (33)$$



4.1 Example

Let's consider the following small part of SCL code:

```

FUNCTION_BLOCK Test

VAR_INPUT
    x : INT;
    y : INT;
END_VAR

VAR_OUTPUT
    out : INT;
END_VAR

BEGIN
    out := 0;
    IF x > y THEN
        out := x/y;
    ELSE
        out := y/x;
    END_IF;
END_FUNCTION_BLOCK

```

Figure 4 – A simple SCL program

the corresponding transformation to a timed automata is:

Table 1 – TA model corresponding to the example of figure 4

SCL instructions	Transformation Rules	TA Model
BEGIN	$\Pi(BEGIN, \varepsilon, \varepsilon) \rightarrow \langle l_0, l_0, \{\emptyset\}, S, l_0 \rangle$	10
out := 0;	$\Pi\langle out := 0, \langle l_0, l_0, \emptyset, S, l_0 \rangle, l_0 \rangle \rightarrow \langle \{l_0, l_1\}, l_0, \{l_0, TRUE, out := 0, l_1\}, S, l_1 \rangle$	10 11
IF x > y THEN out := x/y; ELSE out := y/x; END_IF;	$\Pi\langle IF\ x > y\ THEN\ out := x/y;\ ELSE\ out := y/x;\ END_IF;\ \langle \{l_0, l_1\}, l_0, \{l_0, TRUE, out := 0, l_1\}, S, l_1 \rangle \rightarrow \langle \{l_0, l_1, l_2, l_3, l_4\}, l_0, \{l_0, TRUE, out := 0, l_1\}, (l_1, (x > y) == TRUE, out := x/y, l_2), (l_1, (x > y) == FALSE, out := y/x, l_3), (l_2, TRUE, \varepsilon, l_4), (l_3, TRUE, \varepsilon, l_4)\}, S, l_4 \rangle$	

4.2 Correctness

In this section we prove the correctness of the transformation function Π introduced at the beginning of the chapter 4. The transformation function $\Pi(\text{inst}_{\text{SCL}}, A, l_{\text{in}}) \rightarrow \langle A', l_{\text{fin}} \rangle$ takes a SCL instruction inst_{SCL} and a timed automata model A , with the entry location point l_{in} and produces a modified timed automata model A' , where l_{fin} is the exit location point.

In a SCL program, a trace is a (possibly infinite) list of states and statements starting from an initial state $S_0 = \emptyset$: $\langle S_0, stm_1, S_1, \dots, S_{n-1}, stm_n, S_n, \dots \rangle$.

In Timed Automata, a path is a sequence of nodes (with a corresponding state) and edges starting at the initial node l_0 : $\langle (l_0 S'_0), e_1, (l_1 S'_1), e_2, \dots, (l_m S'_m), \dots \rangle$; we go from one node to another by traversing an edge with associated constraints and actions.

In SCL, variables are splitted into input and output variables. This corresponds in TA just to adding to the initial state all the variables. The state of the output of the program corresponds to the state of the corresponding variables in the final state at the end of execution in the automaton. So, when we get to the final node in the automaton, we just project the state over the output variables.

We aim to prove that given any initial trace segment $\langle S_0, stm_1, S_1, \dots, S_{n-1}, stm_n, S_n \rangle$ there is a corresponding path $\langle (l_0 S'_0), e_1, (l_1 S'_1), \dots, (l_{\text{in}} S'_{\text{in}}), e', (l_{\text{fin}} S'_{\text{fin}}) \rangle$ such that if we assume that after traversing the path $\langle (l_0 S'_0), e_1, (l_1 S'_1), \dots, (l_{\text{in}} S'_{\text{in}}) \rangle$ we get to a state $S'_{\text{in}} = S_{n-1}$, then there is a path $\langle (l_{\text{in}} S'_{\text{in}}), \dots, (l_{\text{fin}} S'_{\text{fin}}) \rangle$ such that the final state $S'_{\text{fin}} = S_n$.

The proof relies on structural induction, by considering the different type of statements and the corresponding transformation rules.

Basic case: BEGIN. At the beginning of the SCL program, the BEGIN statement is executed, where the state of variables is empty. In correspondence to the trace $\langle S_0 = \emptyset, \text{BEGIN}, S_0 \rangle$ we can easily recognize in the automaton the empty path $\langle l_0 \emptyset \rangle$.

Assignment. If the last statement of the SLC trace is an assignment $x := a$, the SCL semantics rule (9) applies:

$$\frac{\langle a, S_{n-1} \rangle \xrightarrow{\delta} v}{\langle T \ x=a, (\Phi, S_n) \rangle \rightarrow (\Phi \cup [x \rightarrow T], S_{n-1} \cup [x \rightarrow v])}$$

By the transformation rule of the assignment (30), a node l_{fin} and an edge e' were introduced in the automaton, with an action on the edge e' that corresponds to the assignment of value a to x .

Let us consider in the automaton the path $\langle (l_{in}S'_{in}), e', (l_{fin}S'_{fin}) \rangle$. By applying the rules (19), (20) and (23) of the TA semantics, in S'_{fin} the values of the variables different from x are the same as in S'_{in} , whereas the value of x is $S'_{in}(a)$. Then, $S'_{fin} = S_n$ as desired.

Conditional Statement. If the statement is "IF b THEN $stat_1$ ELSE $stat_2$ " and in S_{n-1} the condition b is true, by the SCL semantic rule (11) we have:

$$\frac{\langle b, S_{n-1} \rangle \xrightarrow{\beta} \text{TRUE} \quad \langle stat_1, (\Phi, S_{n-1}) \rangle \rightarrow (\Phi', S_n)}{\langle \text{if } b \text{ then } stat_1 \text{ else } stat_2, (\Phi, S_{n-1}) \rangle \rightarrow (\Phi', S_n)}$$

and we know that S_n is the result of the application of the statement $stat_1$. Recall from the translation rule (32) of the "if then else" statement, that three nodes and four edges were introduced in the automaton. As we assume that S'_{in} is equal to S_{n-1} , then in S'_{in} b is true.

Consider in the automaton the following path: $\langle (l_{in}S'_{in}), e'_1, (l_1S'_{l_1}), e''_1, (l_{fin}S'_{fin}) \rangle$. By applying the rules (20) and (22) of the TA semantics, we have that S'_{l_1} is equal to S_n and as e''_1 doesn't make any action, S'_{fin} is equal to S'_{l_1} . Thus, $S'_{fin} = S_n$, as desired.

The proof in the case of b being false is similar.

While-loop Statement. If the statement is "WHILE b DO stm", by the SCL semantics rules (13) and (14) we have:

$$\frac{\langle b, S_{n-1} \rangle \xrightarrow{\beta} \text{TRUE} \quad \langle stm; \text{while } b \text{ do } stm, (\Phi, S_{n-1}) \rangle \rightarrow (\Phi', S_n)}{\langle \text{while } b \text{ do } stm, (\Phi, S_{n-1}) \rangle \rightarrow (\Phi', S_n)}$$

$$\frac{\langle b, S_{n-1} \rangle \xrightarrow{\beta} \text{FALSE}}{\langle \text{while } b \text{ do } stm, (\Phi, S_{n-1}) \rangle \rightarrow (\Phi, S_{n-1})}$$

By the semantics of SCL, if the evaluation of b in S_{n-1} is false, we know that we do not enter into the while-loop, and no change occurs in the variables state, so S_n is equal to S_{n-1} . Otherwise, if b is true in S_{n-1} , then S_n is obtained by applying the statement stm and then calling the while statement again.

By the transformation rule (33) of the while statement, two nodes and three edges were introduced in the automaton.

If in the SCL trace b is false in state S_{n-1} , by inductive hypothesis S'_{in} is equal to S_{n-1} , and then in S'_{in} the condition b is false too. In the automaton we consider the path $\langle (l_{in}S'_{in}), e'', (l_{fin}S'_{fin}) \rangle$. In other words, on the SCL side when b is false we exit the while, and this corresponds in the TA to move to the final node l_{fin} by the edge e'' that has no action. By applying the rule (24) of the TA semantics we have that S'_{fin} is equal to S'_{in} , so $S'_{fin} = S_{n-1}$ and we know that $S_{n-1} = S_n$, so $S'_{fin} = S_n$ as required.

Consider now the other case, when in the SCL state S_{n-1} b is true. By inductive hypothesis, S'_{in} is equal to S_{n-1} , where b is true too. Let assume that in the SCL program we iterate the while loop m times, where m can be finite or infinite, this correspond in the TA to cross the path:

$$\langle (l_{in}S'_k), e', (l_1S'_k), e'_1, (l_{in}S'_{k+1}), e', (l_1S'_{k+1}), e'_1, (l_{in}S'_{k+2}), \dots, (l_{in}S'_{k+m-1}), e'_{m-1}, (l_{in}S'_{k+m}) \rangle$$

At the m^{th} iteration, by inductive hypothesis the parallelism among what happens in the while loop in the SCL and what happens in the three states corresponding to the three nodes in the automaton is maintained. The path $\langle (l_{in}S'_k), e', (l_1S'_k), e'_1, (l_{in}S'_{k+1}), e', (l_1S'_{k+1}), e'_1, (l_{in}S'_{k+2}), \dots \rangle$ corresponds to the iteration of the body of the while loop in the SCL program.

Focusing on the last iteration of the while in the SCL program, in the TA

this corresponds to the path $\langle (l_{in}S'_{m-1}), e', (l_1S'_{m-1}), e'_1, (l_{in}S'_m), e'', (l_{fin}S'_{fin}) \rangle$. By applying the rules (20) and (22) of the TA semantics, S'_{in} is updated to S'_{m-1} , and crossing the edge e'_1 , makes S'_m equal to the effect of applying stm . Therefore, S'_m after the statement stm is equal to S_n . As $S'_{fin} = S'_{in} = S'_m$, we get $S'_{fin} = S_n$ as desired.

Ending Statement. As the END statement does not have any impact on the variables state, it corresponds to an empty transition applied to the final state of the TA.

■

5 UPPAAL

UPPAAL is a tool for modelling, validating and verifying real-time systems, that can be represented with timed automata. It is developed by Uppsala University (Sweden) and Aalborg University (Denmark).

In UPPAAL we can identify three main sections:

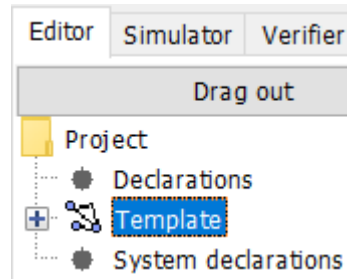


Figure 5 – UPPAAL interface

- Editor: in the editor we can create the system that we want to analyse. The system can be composed of one or more timed automata (Template), and we can synchronize them. The automaton of a template consists of locations and edges, a template may also have local declarations and parameters. In this section we have also to declare the variables that are used in the automata (the types of variables can be int, bool, clock, chan, array and record types can be defined over these and other types.). We can also define structure data types, functions and channels (in the subsection Declarations). Then in the System declaration part we list one or more processes to be composed into a system.

The locations of timed automata are graphically represented as circles and are connected by edges. Each template must have one initial location that is marked by a double circle. We can also put a location as urgent or committed: urgent locations freeze time (time is not allowed to pass when a process is in an urgent location), committed locations, as the urgent ones, freeze time, but also if any process is in a committed location, the next transition must involve an edge from one of the committed locations. Edges, that connect locations, contain: *selections*, that non-deterministically give to an identifier a value in a given range; *guards*, if

there is a guard in an edge then we can go from the initial location of this edge to the final one if and only if the evaluated guard gets to true; *synchronization*, processes can synchronize over channels, this is done by use synchronization labels that are of the form *ready?* in one automaton, that synchronized with *ready!* in another automata; *updates*, used to change a value of a variable.

- Simulator: gives the possibility to examine all the possible dynamic executions of our automaton or of our system composed by more than one timed automaton. The simulator section is composed by 4 parts: the one on the left is called *simulation control* that allows us to perform step-by-step the simulation of our system; the middle part is called *variables* because it shows the values of the data and clocks variables in a specific moment in the automaton; the upper right part is called *processes*, where are represented our automaton and we can see the progressing execution, where the current node and the next edge that will be traversed are coloured by red; the lower right panel is called *message sequence chart* and it shows the trace.
- Verifier: allows to check safety and liveness properties, so that we can study the robustness of our system. First of all we have to insert a new query, we write the query (that represents a property) and we check it, and the verifier will say us if the property is satisfied (green circle beside the query will appear) or not (red circle will appear beside the property). We can have 5 types of properties that will be described later in another section.

We choose UPPAAL because it is a simple tool that supports various features of model checkers. One of the limitations we met was that there is not the possibility to read input values in UPPAAL, then we tried to search for others tool that give us the possibility to do the same things or more than UPPAAL. Some of other tools are: DREAM, TAPAAL, BLAST, CPAchecker, ROMEO, etc.

At the end, our choice remained UPPAAL because it is the simplest and most intuitive, and the overhead required to start using the tool is minimal. Moreover,

none of the other tools permit us to overcome the limitation problem that we have found with UPPAAL, that is that we are not able to read input values. We overcome this problem by using the select in the edges, that permits us to use non-deterministic values.

5.1 Robustness Properties

Given a Timed Automata that represents an SCL program, we want now to define some robustness properties and prove by using UPPAAL that these robustness properties are satisfied by the program. UPPAAL is used to validate and verify real-time systems. The idea is to represent a SCL program by an automaton, simulate and verify various properties on it through some queries. The query language consists of state formulas and path formulas. A state formula is an expression that can be evaluated for a particular state in order to check a property (e.g. a deadlock), without looking at the behaviour of the model. Path formula quantify over paths of execution and ask whether a given state formula p can be satisfied in any or all the states along any or all the paths [15].

We can identify three types of path formulas (path properties):

- Reachability properties ($E <> p$):
are the simplest ones; they ask whether a given state formula, p , possibly can be satisfied by any reachable state.
 $E <> p$ = "there exists a path where p eventually holds".
- Safety properties ($E[]p$ and $A[]p$):
are of the form: something good is invariantly true.
 $E[]p$ = "there exists a path where p always holds";
 $A[]p$ = "for all paths, p always holds".
- Liveness properties ($A <> p$ and $p \rightarrow q$):
are of the form: something will eventually happen.
 $A <> p$ = for all paths, p will eventually hold;
 $p \rightarrow q$ = whenever p holds, q will eventually hold.

In the following figure we depict the graphical representation of the 5 types of properties:

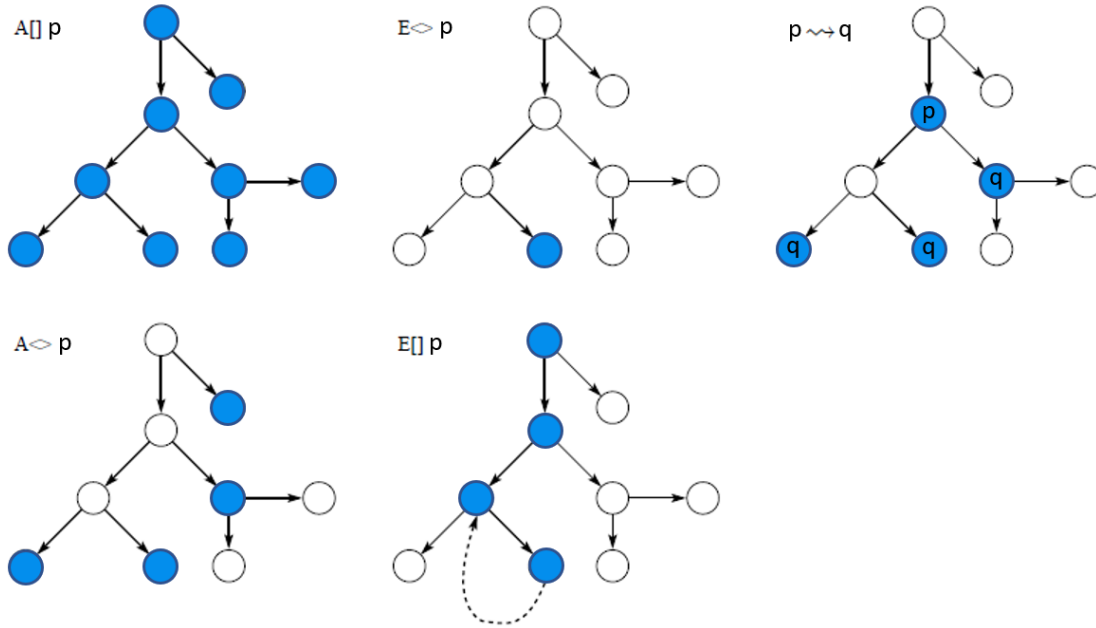


Figure 6 – Graphical representation of robustness properties

We can express one property in terms of another one:

$$A[]p = \text{not } E<>\text{not } p$$

$$A<>p = \text{not } E[]\text{not } p$$

$$p \rightarrow q = A[](p \text{ imply } A<>q)$$

As we already mentioned in the problem statement in chapter 1, let Σ_Δ be the set of acceptable final states and Σ_E be the set of erroneous states; and let I_A be the set of acceptable inputs and I_U be the set of unacceptable inputs. Then, as we trying to verify the robustness of our program, two things are most important:

- The program will always lead to a correct output state for a valid input:
 $A[]\Sigma_\Delta(I_A)$;
- The program will never lead to a correct output state for an erroneous input, it means that the program will always lead to an erroneous output for an erroneous input: $A[]\Sigma_E(I_U)$.

5.2 File .xta Syntax

In this section, we'll define the formal syntax for .xta based files, using Xtext and Acceleo in Eclipse, which are used as an input file in UPPAAL, and SIEMENS SCL programming language.

XTA. The BNF (Backus-Naur Form) syntax for .xta files, as described in the UPPAAL [22] reference manual is expressed here:

```
XTA ::= <Declaration>* <Instantiation>* <System>

Declaration ::= <FunctionDecl> | <VariableDecl> | <TypeDecl> |
<ProcDecl>
Instantiation ::= ID ASSIGNMENT ID '(' <ArgList> ')' ';'
System ::= 'system' ID '(' ID)* ';'

ParameterList ::= '(' [ <Parameter> ( ',' <Parameter> )* ] ')'
Parameter ::= <Type> [ '&' ] ID <ArrayDecl>*

FunctionDecl ::= <Type> ID <ParameterList> <Block>

ProcDecl ::= 'process' ID <ParameterList> '{' <ProcBody> '}'
ProcBody ::= (<FunctionDecl> | <VariableDecl> | <TypeDecl>)*
<States> [<Commit>] [<Urgent>] <Init> [<Transitions>]

States ::= 'state' <StateDecl> (',' <StateDecl>)* ';'
StateDecl ::= ID [ '{' <Expression> '}' ]

Commit ::= 'commit' StateList ';'
Urgent ::= 'urgent' StateList ';'
StateList ::= ID (',' ID)*
Init ::= 'init' ID ';'

Transitions ::= 'trans' <Transition> (',' <TransitionOpt>)* ';'
Transition ::= ID '->' ID <TransitionBody>
TransitionOpt ::= Transition | '->' ID <TransitionBody>
TransitionBody ::= '{' [<Guard>] [<Sync>] [<Assign>] '}'

Guard ::= 'guard' <Expression> ';'
Sync ::= 'sync' <Expression> ('!' | '?') ';'
Assign ::= 'assign' <ExprList> ';'

TypeDecl ::= 'typedef' <Type> <TypeIdList> (',' <TypeIdList>)*
';'
```

TypeIdList ::= ID <ArrayDecl>*

BNF for variable declarations:

```
VariableDecl ::= <Type> <DeclId> (',' <DeclId>)* ';'
DeclId ::= ID <ArrayDecl>* [ ASSIGNMENT <Initialiser> ]
Initialiser ::= <Expression> | '{' <FieldInit> (',' <FieldInit>)* '}'
FieldInit ::= [ ID ':' ] <Initialiser>
ArrayDecl ::= '[' <Expression> ']'
Type ::= <Prefix> ID [ <Range> ] | <Prefix> 'struct' '{'
<FieldDecl>+ '}'
FieldDecl ::= <Type> <FieldDeclId> (',' <FieldDeclId>)* ';'
FieldDeclId ::= ID <ArrayDecl>*
Prefix ::= ( [ 'urgent' ] [ 'broadcast' ] | ['const'] )
Range ::= '[' <Expression> ',' <Expression> ']'
```

BNF for statements:

```
Block ::= ' ' ( <VariableDecl> | <TypeDecl> )* <Statement>* ' '
Statement ::= <Block>
| ';'
| <Expression> ';'
| 'for' '(' <ExprList> ';' <ExprList> ';'
<ExprList> ')' <Statement>
| 'while' '(' <ExprList> ')' <Statement>
| 'do' <Statement> 'while' '(' <ExprList> ')' ';'
| 'if' '(' <ExprList> ')' <Statement>
[ 'else' <Statement> ]
| 'break' ';'
| 'continue' ';'
| 'switch' '(' <ExprList> ')' '{' <Case>+ '}'
| 'return' ';'
| 'return' <Expression> ';'

Case ::= 'case' <Expression> ':' <Statement>*
| 'default' ':' <Statement>*
```

BNF for expressions:

```
ExprList ::= <Expression> ( ',' <Expression> )*
Expression ::= ID
```

```

| NAT
| 'true' | 'false'
| ID '(' <ArgList> ')'
| <Expression> '[' <Expression> ']'
| '(' <Expression> ')'
| <Expression> '++' | '++' <Expression>
| <Expression> '--' | '--' <Expression>
| <Expression> <AssignOp> <Expression>
| <UnaryOp> <Expression>
| <Expression> <Rel> <Expression>
| <Expression> <BinIntOp> <Expression>
| <Expression> <BinBoolOp> <Expression>
| <Expression> '?' <Expression> ':' <Expression>
| <Expression> '.' ID>

AssignOp ::= ASSIGNMENT | '+' | '-' | '*' | '/' | '%' |
'|=' | '&=' | '≐' | '<<=' | '>>='
UnaryOp ::= '-' | '!'
Rel ::= '<' | '<=' | '==' | '!=' | '>=' | '>'
BinIntOp ::= '+' | '-' | '*' | '/' | '%' | '&' | '|' | '^'
| '<<' | '>>'
BinBoolOp ::= '&&' | '||'
ArgList ::= [ <Expression> ( ',' <Expression> )* ]

```

5.3 Transformation Rules from SCL to .xta File

We use the grammar defined above to parse a SCL program using Xtext. After that, we have written a simple java code relying on Acceleo, which transforms the SCL program into a .xta file.

The source code for a block consists of the following sections:

- Block start with specification of the block (absolute or symbolic)
- Block attributes (optional)
- Declaration section (differs from block type to block type)
- Code section in logic blocks or assignment of current values in data blocks (optional)
- Block end.

Detailed transformation rules for the code section: the code section contains statements, which are executed when a code block is called. These statements are used for processing data or addresses, or for setting individual initialization values in data blocks.

Each individual statement belongs to one of the following types:

- **Assignment Statement:** used to assign the result of an expression or the value of another variable to a variable;
- **Control statement:** used to repeat statements or groups of statements or to branch within a program;
- **Subroutine call:** used to call functions or function blocks.

The transformation rules to pass from a SCL code to a file .xta are listed in the following table:


Table 2 – Transformation rules from SCL to .xta file

Statement Type	SCL	.xta
Initial Statement	BEGIN	States ::= 'state' <StateDecl> StateDecl ::= START Init ::= 'init' START ';' ; Transitions ::= 'trans' <Transition>
Assignment Statement	variable ':=' expression	StateDecl ::= current_state ['{' <Expression> '}'] Expression ::= <Expression> AssignOp <Expression> Expression ::= ID(=variable) ':=' <expression>
If Statement	'IF' expression1 'THEN' statement_list1 { 'ELSEIF' expression2 'THEN' statement_list2 } ['ELSE' statement_list3] 'END_IF'	States ::= (',' <StateDecl>) StateDecl ::= IF Transition ::= current_state '->' ID (ID = IF) States ::= (',' <StateDecl>) StateDecl ::= THEN Transition ::= IF '->' ID <TransitionBody> (ID = THEN) TransitionBody ::= '{' [<Guard>] [<Assign>] '}' Guard ::= 'guard' <expression1>; Assign ::= 'assign' <statement_list1>; States ::= (',' <StateDecl>) StateDecl ::= ELSEIF Transition ::= THEN '->' ID <TransitionBody> (ID = ELSEIF) TransitionBody ::= '{' [<Guard>] [<Assign>] '}' Guard ::= 'guard' <expression2>; Assign ::= 'assign' <statement_list2>; States ::= (',' <StateDecl>) StateDecl ::= ELSE Transition ::= IF '->' ID <TransitionBody> (ID = ELSE) TransitionBody ::= '{' [<Guard>] [<Assign>] '}' Guard ::= 'guard' <'!' expression1>; Assign ::= 'assign' <statement_list3>;
For Statement	'FOR' control_variable ':=' for_list 'DO' statement_list 'END_FOR' control_variable ::= identifier	States ::= <',' <StateDecl> StateDecl ::= FOR Transition ::= current_state '->' ID <TransitionBody> (ID = FOR) TransitionBody ::= '{' [<Assign>] '}' Assign ::= 'assign' <control_variable = expression1>; States ::= (',' <StateDecl>)

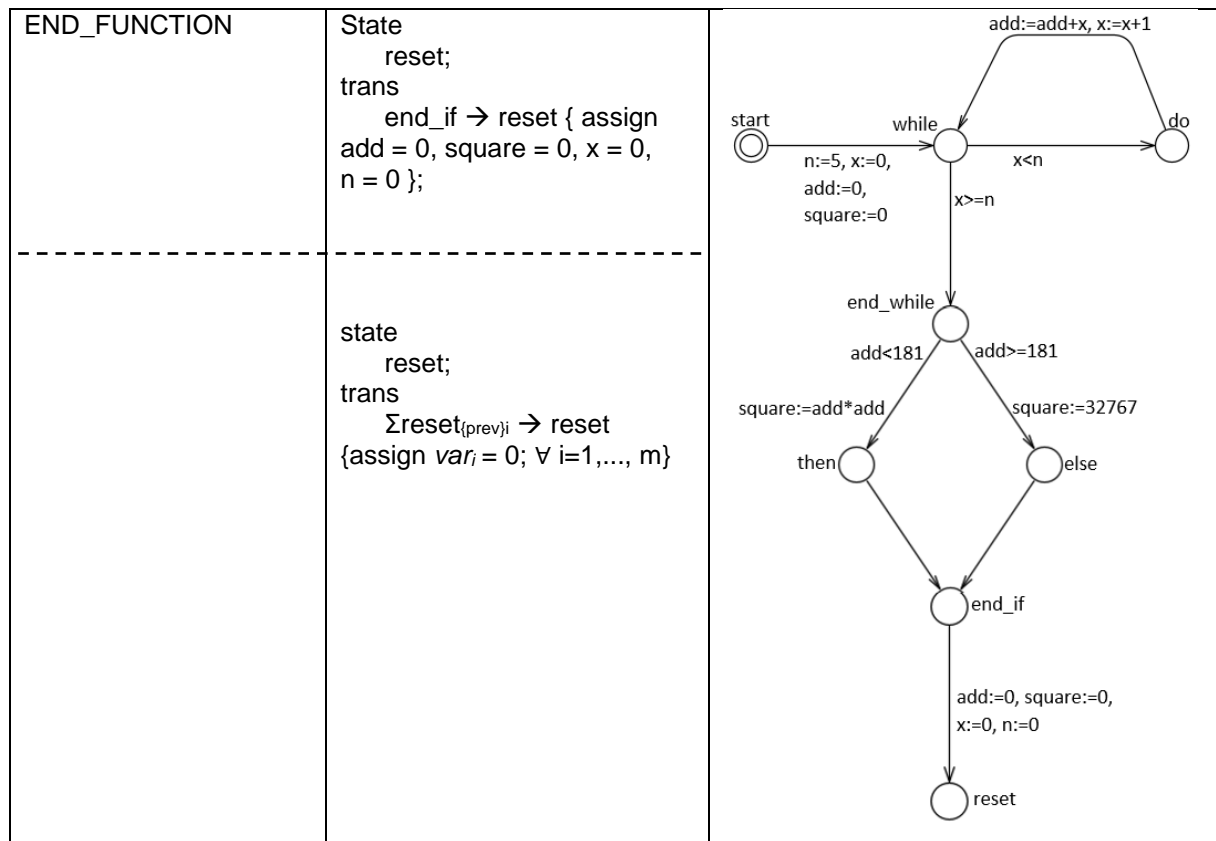
	for_list ::= expression1 'TO' expression2 ['BY' expression3]	<pre> StateDecl ::= DO_FOR Transition ::= FOR'->' ID <TransitionBody>(ID = DO_FOR) TransitionBody ::= '{' [<Guard>] [<Assign>] '}' Guard ::= 'guard' <control_variable <expression2>';' Assign ::= 'assign' <control_variable = control_variable <expression3>';' States ::= (',' <StateDecl>) StateDecl ::= END_FOR Transition ::= DO'->' ID <TransitionBody>(ID = END_FOR) TransitionBody ::= '{' [<Assign>] '}' Assign ::= 'assign' <statement_list1>';' Transition ::= END_FOR'->' FOR Transition ::= FOR'->' END_FOR<TransitionBody> TransitionBody ::= '{' [<Guard>] '}' Guard ::= 'guard' <control_variable >= expression2>';' </pre>
While Statement	while_statement 'WHILE' expression 'DO' statement_list 'END_WHILE'	<pre> States ::= <',' StateDecl> StateDecl ::= WHILE Transition ::= current_state'->' ID (ID = WHILE) States ::= (',' <StateDecl>) StateDecl ::= DO_WHILE Transition ::= WHILE'->' ID <TransitionBody>(ID = DO_WHILE) TransitionBody ::= '{' [<Guard>] '}' Guard ::= 'guard' <expression1>';' Transition ::= DO_WHILE'->' WHILE<TransitionBody> TransitionBody ::= '{' [<Guard><Assign>] '}' Guard ::= 'guard' <Rel><expression1>';' Assign ::= 'assign' <statement_list>';' States ::= (',' <StateDecl>) StateDecl ::= END_WHILE Transition ::= WHILE'->' ID <TransitionBody>(ID = END_WHILE) TransitionBody ::= '{' [<Guard>] '}' Guard ::= 'guard' <' !' expression1>';' </pre>

Now we represent the transformation of the SCL code example of figure 3 presented in the second chapter, to a file .xta and also to the corresponding automata:

Table 3 – From a simple example in SCL to the .xta file to the TA

SCL program	.xta input file	UPPAAL model
variable declaration (variable_name, variable_type) VAR_INPUT n : INT; END_VAR VAR_OUTPUT ADD : INT; SQUARE : INT; END_VAR VAR x:INT; END_VAR	// Place global declarations here. int n; int add; int square; int x;	// Place global declarations here. int n; int add; int square; int x;
----- variable name : variable type var: type_var $\forall i=1,...,m$	type_var var; $\forall i=1,...,m$	
n:=5; x:=0; ADD:=0; SQUARE:=0;	// Place global declarations here. const int n=5; int x=0; int add=0; int square=0;	the variables initialization will be associated to the edge out of the start node
----- variable name := value; var: val $\forall i=1,...,m$	type_var var = val; $\forall i=1,...,m$	
BEGIN	process Template(){ state start; init start;	start 

<p>while loop</p> <pre> WHILE $x < n$ DO ADD := ADD + x; x := x + 1; END_WHILE; </pre>	<p>State</p> <pre> while, do, end_while; trans trans start → while { }, while → do { guard $x < n$; }, do → while { assign add = add + x, x = x + 1; }, while → end_while { guard $x \geq n$; }; </pre>	
<pre> WHILE $expr_1$ DO $expr_2$ END_WHILE </pre>	<p>state{while, do, end_while};</p> <p>trans{</p> $\Sigma \text{while}_{\{prev\}i} \rightarrow \text{while}\{ \}$, while → do { guard $expr_1$; }, do → while { assign $expr_2$; }, while → end_while { ! $expr_1$ } ;	
<pre> IF SUM ≤ 181 THEN SQUARE := ADD * ADD; ELSE SQUARE := 32767; END_IF; </pre>	<p>State</p> <pre> then, else, end_if; trans end_while → then { guard add < 181; assign square = add * add; }, end_while → else { guard add ≥ 181; assign square = 32767; } then → end_if{} else → end_if{}; </pre>	
<pre> IF cond₁ THEN $expr_1$ ELSE $expr_2$ END_IF </pre>	<p>state{ then, else, end_if };</p> <p>trans{</p> $\Sigma \text{if}_{\{prev\}i} \rightarrow \text{then}\{ \text{guard } cond_1; \text{ assign } expr_1; \}$, $\Sigma \text{if}_{\{prev\}i} \rightarrow \text{else}\{ \text{guard } ! cond_1; \text{ assign } expr_2; \}$ then → end_if{} else → end_if{};	



6 Experimental Results

In this final chapter we present the whole picture of the project. Starting from a PLC program written in the Structure Control Language, we transform it into a code written in the DSL language using Xtext. Then we create the .xta file, with the help of Acceleo, to automatically create the Timed Automata. Finally we verify some properties in order to check the robustness of our system with the UPPAAL tool.

The example that we consider is inspired by a video published in YouTube [25]. It consists of an automated line for boxing of bolts and nuts, managed by a PLC. We have chosen this example because it is interactive and input dependent, and so we want to prove the robustness of this model either with correct inputs but also with unexpected inputs.

The following image illustrates the physical ambient:



Figure 7 – Physical ambient

The Programmable Logic Controller allows a user to insert a key in order that the machine starts working. Until the key is not turned on, the machine cannot start. Then the user can select if he wants only nuts, only bolts or both nuts and bolts. If the choice is only nuts or only bolts then the box that is needed is a box with a single partition; instead if a user wants both bolts and nuts then the box has to be

bipartitioned, in order to separate the nuts from the bolts. The user has also to select the number of pieces that he desires. If he has selected only bolts he can choose a number of bolts between 3 and 30. Instead, if he has chosen only nuts the number should be between 3 and 60. Note that the maximum allowed number is quite bigger than the maximum number of bolts because the nuts have a smaller dimension with respect to the dimension of bolts and so there can be more pieces inside a box. If the choice was bolts and nuts then the number of bolts and nuts should be equal and can be between 3 and 20.

A box is put on the main conveyor by another machine that coordinates with the main machine, for example by using another automaton that is synchronized with the automaton of this machine. The system checks if the box is open or closed. In the second case, it is discarded. In addition, the box is discarded also if it is not correct for the choice selected by the user, so if a person wants only bolts or nuts, the desired box is the simplest one with no partitions, but if the machine receives a bipartitioned box (that one for the choice of both bolts and nuts together) then this box is discarded, and the process starts again remembering the choice already done by the user. The same happens in the opposite case, if the choice was both bolts and nuts and the box is that one with no partition then it is discarded because the desired box is that one bipartitioned.

To recognise if there is the correct box and to see if the box is opened there is a camera that checks the conditions of the box and sends an input value to the PLC, and so the PLC is able to decide if the box can be used or has to be discarded.

Once we have the correct box the main conveyor stops when the box is in the corrected position to be filled with nuts or bolts or both. There are some sensors that permit to understand when the main conveyor has to stop. So for example if the box has to be filled with only nuts then before the slide, where the pieces (nuts or bolts) will fall, there is a sensor that detects when there is the box in front of it and sends a message to the PLC saying that the main conveyor has to stop. This sensor (for nuts) is activated only when the choice is nuts or both nuts and bolts. Instead, if the choice is bolts the main conveyor will stop a little bit after, when another sensor (that one for bolts, that is placed after the slide) detects the box.

Once the box is stopped in the correct position the conveyor of nuts or the one of bolts is started until the correct number of pieces falls into the box.

If the choice was of both bolts and nuts then the main conveyor stops when the box is in correspondence of the first sensor before the slide, the conveyor of nuts is activated and the pieces fall into the box, then the conveyor of nuts is stopped and the main conveyor is started but move only a little bit, because the sensor for bolts, placed after the slide, stops it again in order that the conveyor of bolts start moving and the bolts fall into the other part of the box.

Once the box is filled with the correct number of pieces the main conveyor restarts and the box arrived in a point where there is another sensor that captures the presence of a box in front of it and the box is closed. The main conveyor starts again, the box arrives ahead of another sensor and there is a “mechanical arm” that pushed the box away.

Our system checks also if in the two smaller conveyors, that have the bolts and nuts, there are still pieces or not; if there are no more pieces the system stops until the addition of new pieces. So, the system is synchronized with another machine that fill the conveyor with pieces when they are finished.

The PLC program written in the SCL language corresponding to the system described above is:

```
FUNCTION TEST : INT
VAR_INPUT
    key : INT;
    choice : ARRAY [0 .. 2] OF INT;
    box : ARRAY [1 .. 2] OF INT;
    openBox : ARRAY [0 .. 1] OF INT;
    numberBolts : ARRAY [3 .. 30] OF INT;
    numberNuts : ARRAY [3 .. 60] OF INT;
    numberPieces2 : ARRAY [3 .. 20] OF INT;
END_VAR
VAR_OUTPUT
    counterBolts : INT;
    counterNuts : INT;
    close : BOOL;
    positionPush : BOOL;
END_VAR
VAR
    mainConveyor : BOOL;
    conveyorB : BOOL;
    conveyorN : BOOL;
    piecesB : INT;
    piecesN : INT;
    rechargeB : BOOL;
    rechargeN : BOOL;
    position : BOOL;
```

```

push : BOOL;
discard : BOOL;
i: INT;
j: INT;
k: INT;
END_VAR
WHILE TRUE DO
  IF key=0 THEN
    mainConveyor:=FALSE;
  ELSEIF key=1 THEN
    mainConveyor:=TRUE;
    IF choice=0 THEN
      IF box=2 OR openBox=0 THEN
        discard:= TRUE;
      ELSEIF box=1 AND openBox=1 THEN
        i:=0;
        mainConveyor:= FALSE;
        IF piecesB=0 THEN
          rechargeB:=TRUE;
        ELSE
          WHILE i<numberBolts DO
            conveyorB:=TRUE;
            counterBolts:=counterBolts + 1;
            i:=i + 1;
          END_WHILE;
        END_IF;
        conveyorB:=FALSE;
        mainConveyor:=TRUE;
      END_IF;
    ELSEIF choice=1 THEN
      IF box=2 OR openBox=0 THEN
        discard:= TRUE;
      ELSEIF box=1 AND openBox=1 THEN
        j:=0;
        mainConveyor:= FALSE;
        IF piecesN=0 THEN
          rechargeN:=TRUE;
        ELSE
          WHILE j<numberNuts DO
            conveyorN:=TRUE;
            counterNuts:= counterNuts + 1;
            j:=j + 1;
          END_WHILE;
        END_IF;
        conveyorN:=FALSE;
        mainConveyor:=TRUE;
      END_IF;
    ELSEIF choice=2 THEN
      IF box=1 OR openBox=0 THEN
        discard:= TRUE;
      ELSEIF box=2 AND openBox=1 THEN
        k:=0;
        numberBolts:= numberPieces2;
        numberNuts:= numberBolts;
        mainConveyor:= FALSE;
        IF piecesN=0 THEN
          rechargeN:=TRUE;
        ELSE
          WHILE k<numberNuts DO
            conveyorN:=TRUE;
            counterNuts:=counterNuts + 1;
            k:=k + 1;
          END_WHILE;
        END_IF;
        conveyorN:=FALSE;

```

```

        mainConveyor:=TRUE;
        k:=0;
        mainConveyor:=FALSE;
        IF piecesB=0 THEN
            rechargeB:=TRUE;
        ELSE
            WHILE k<numberBolts DO
                conveyorB:=TRUE;
                counterBolts:=counterBolts + 1;
                k:=k + 1;
            END_WHILE;
        END_IF;
        conveyorB:=FALSE;
        mainConveyor:=TRUE;
    END_IF;
END_IF;
IF position=TRUE THEN
    mainConveyor:=FALSE;
    close:=TRUE;
END_IF;
close:=FALSE;
mainConveyor:=TRUE;
IF positionPush=TRUE THEN
    push:=TRUE;
    counterNuts:=0;
    counterBolts:=0;
END_IF;
END_IF;
END_WHILE;
END_FUNCTION

```

The corresponding .xta file is:

```

// place global declarations here.
int key ;
int [0,1,2] choice;
int [1,2] box;
int [0,1] openbox;
int [3 ..30] numberbolts;
int [3 .. 60] numbernuts;
int [3 .. 20] numberpieces2;
bool mainconveyor;
bool conveyorb;
bool conveyorn;
int piecesb;
int piecesn;
bool rechargeb;
bool rechargen;
bool position;
bool push;
bool discard;
int i;
int j;
int k;
int counterbolts;
int counternuts;
bool close;
bool positionpush;
process test(){
state
    s_0,
    s_1,
    s_2,

```


s_3,
s_4,
s_5,
s_6,
s_7,
s_8,
s_9,
s_10,
s_11,
s_12,
s_13,
s_14,
s_15,
s_16,
s_17,
s_18,
s_19,
s_20,
s_21,
s_22,
s_23,
s_24,
s_25,
s_26,
s_27,
s_28,
s_29,
s_30,
s_31,
s_32,
s_33,
s_34,
s_35,
s_36,
s_37,
s_38,
s_39,
s_40,
s_41,
s_42,
s_43,
s_44,
s_45,
s_46,
s_47,
s_48,
s_49,
s_50,
s_51,
s_52,
s_53,
s_54,
s_55,
s_56,
s_57,
s_58,
s_59,
s_60,
s_61,
s_62,
s_63,
s_64,
s_65,
s_66,
s_67,

```

s_68,
s_69,
s_70,
s_71,
s_72,
s_73,
s_74,
s_75,
s_76,
s_77,
s_78,
s_79,
s_80,
s_81,
s_82,
s_83;

init
s_0;

trans

s_0 -> s_1    {guard TRUE;},
s_1 -> s_2    {guard key=0;},
s_2 -> s_3    {assign mainConveyor:=FALSE;},
s_1 -> s_4    {guard !key=0;},
s_4 -> s_5    {guard key=1;},
s_5 -> s_6    {assign mainConveyor:=TRUE;},
s_6 -> s_7    {guard choice=0;},
s_7 -> s_8    {guard box=2 OR openBox=0;},
s_8 -> s_9    {assign discard:= TRUE;},
s_7 -> s_10   {guard !box=2 OR openBox=0;},
s_10 -> s_11  {guard box=1 AND openBox=1;},
s_11 -> s_12  {assign i:=0;},
s_12 -> s_13  {assign mainConveyor:= FALSE;},
s_13 -> s_14  {guard piecesB=0;},
s_14 -> s_15  {assign rechargeB:=TRUE;},
s_15 -> s_16  {assign conveyorB:=TRUE;},
s_16 -> s_17  {assign counterBolts:=counterBolts + 1;},
s_17 -> s_18  {assign i:=i + 1;},
s_18 -> s_15  {},
s_15 -> s_19  {guard !();},
s_19 -> s_20  {},
s_20 -> s_21  {assign conveyorB:=FALSE;},
s_21 -> s_22  {assign mainConveyor:=TRUE;},
s_9 -> s_23   {},
s_22 -> s_23  {},
s_6 -> s_24   {guard !choice=0;},
s_24 -> s_25  {guard choice=1;},
s_25 -> s_26  {guard box=2 OR openBox=0;},
s_26 -> s_27  {assign discard:= TRUE;},
s_25 -> s_28  {guard !box=2 OR openBox=0;},
s_28 -> s_29  {guard box=1 AND openBox=1;},
s_29 -> s_30  {assign j:=0;},
s_30 -> s_31  {assign mainConveyor:= FALSE;},
s_31 -> s_32  {guard piecesN=0;},
s_32 -> s_33  {assign rechargeN:=TRUE;},
s_33 -> s_34  {assign conveyorN:=TRUE;},
s_34 -> s_35  {assign counterNuts:=counterNuts + 1;},
s_35 -> s_36  {assign j:=j + 1;},
s_36 -> s_33  {},
s_33 -> s_37  {guard !();},
s_37 -> s_38  {},
s_38 -> s_39  {assign conveyorN:=FALSE;},
s_39 -> s_40  {assign mainConveyor:=TRUE;},
s_27 -> s_41  {},
s_40 -> s_41  {},

```

```

s_24 -> s_42      {guard !choice=1;},
s_42 -> s_43      {guard choice=2;},
s_43 -> s_44      {guard box=1 OR openBox=0;},
s_44 -> s_45      {assign discard:= TRUE;},
s_43 -> s_46      {guard !box=1 OR openBox=0;},
s_46 -> s_47      {guard box=2 AND openBox=1;},
s_47 -> s_48      {assign k:=0;;},
s_48 -> s_49      {assign numberBolts:= numberPieces2;},
s_49 -> s_50      {assign numberNuts:= numberBolts;},
s_50 -> s_51      {assign mainConveyor:= FALSE;},
s_51 -> s_52      {guard piecesN=0;},
s_52 -> s_53      {assign rechargeN:=TRUE;},
s_53 -> s_54      {assign conveyorN:=TRUE;},
s_54 -> s_55      {assign counterNuts:=counterNuts + 1;},
s_55 -> s_56      {assign k:=k + 1;},
s_56 -> s_53      {},
s_53 -> s_57      {guard !();},
s_57 -> s_58      {},
s_58 -> s_59      {assign conveyorN:=FALSE;},
s_59 -> s_60      {assign mainConveyor:=TRUE;},
s_60 -> s_61      {assign k:=0;},
s_61 -> s_62      {assign mainConveyor:=FALSE;},
s_62 -> s_63      {assign rechargeB:=TRUE;},
s_63 -> s_64      {assign conveyorB:=TRUE;},
s_64 -> s_65      {assign counterNuts:=counterNuts + 1;},
s_65 -> s_66      {assign k:=k + 1;},
s_66 -> s_63      {},
s_63 -> s_67      {guard !();},
s_67 -> s_68      {},
s_68 -> s_69      {assign conveyorB:=FALSE;},
s_69 -> s_70      {assign mainConveyor:=TRUE;},
s_45 -> s_71      {},
s_70 -> s_71      {},
s_23 -> s_72      {},
s_41 -> s_72      {},
s_71 -> s_72      {},
s_72 -> s_73      {assign mainConveyor:=FALSE;},
s_73 -> s_74      {assign close:=TRUE;},
s_74 -> s_75      {},
s_75 -> s_76      {assign close:=FALSE;},
s_76 -> s_77      {assign mainConveyor:=TRUE;},
s_77 -> s_78      {assign push:=TRUE;},
s_78 -> s_79      {assign counterNuts:=0;},
s_79 -> s_80      {assign counterBolts:=0;},
s_80 -> s_81      {},
s_3 -> s_82      {},
s_81 -> s_82      {},
s_82 -> s_0      {},
s_0 -> s_83      {guard !(TRUE);};
}
// Place template instantiations here.
Process = test();
// List one or more processes to be composed into a system.
system Process;

```

From the previous .xta file, the following timed automata is created:

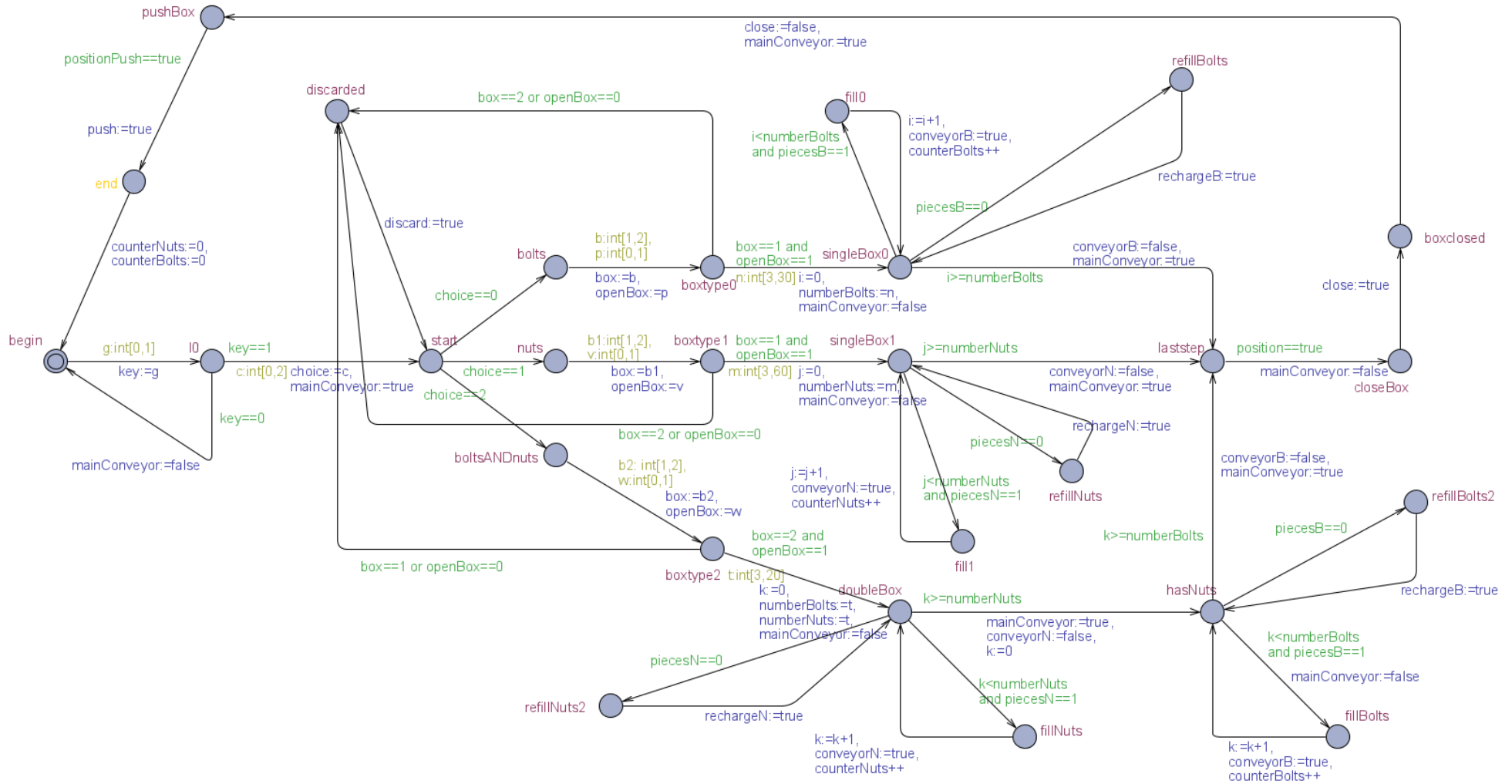


Figure 8 – Timed Automata in UPPAAL

Now that we have the timed automata in UPPAAL we can check some robustness properties. But we find out that as the system is very big then the verifier of UPPAAL is not able to check the true properties because it gives the following error:

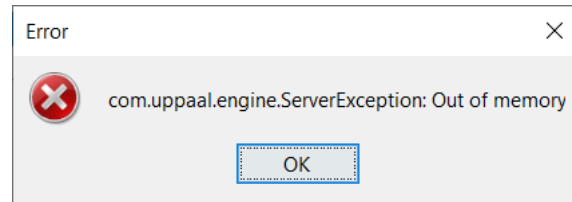


Figure 9 – Error: out of memory

By the usage of the performance monitor we were able to see that there is a limit in the UPPAAL tool that allows the usage of memory until 2 GB, after using two gigabytes, it will give the out of memory error, also if there is still available memory. We were able to understand that there is this limitation by run the verification of a true property with the initial automaton in UPPAAL and check, in the performance monitor, how much memory is used by that process, and we have seen that after the usage of 2 GB, UPPAAL stops and gives the out of memory error.

In the following image we can see that the system gave the out of memory error when the memory used was almost of 2 GB (1.992.696 KB):

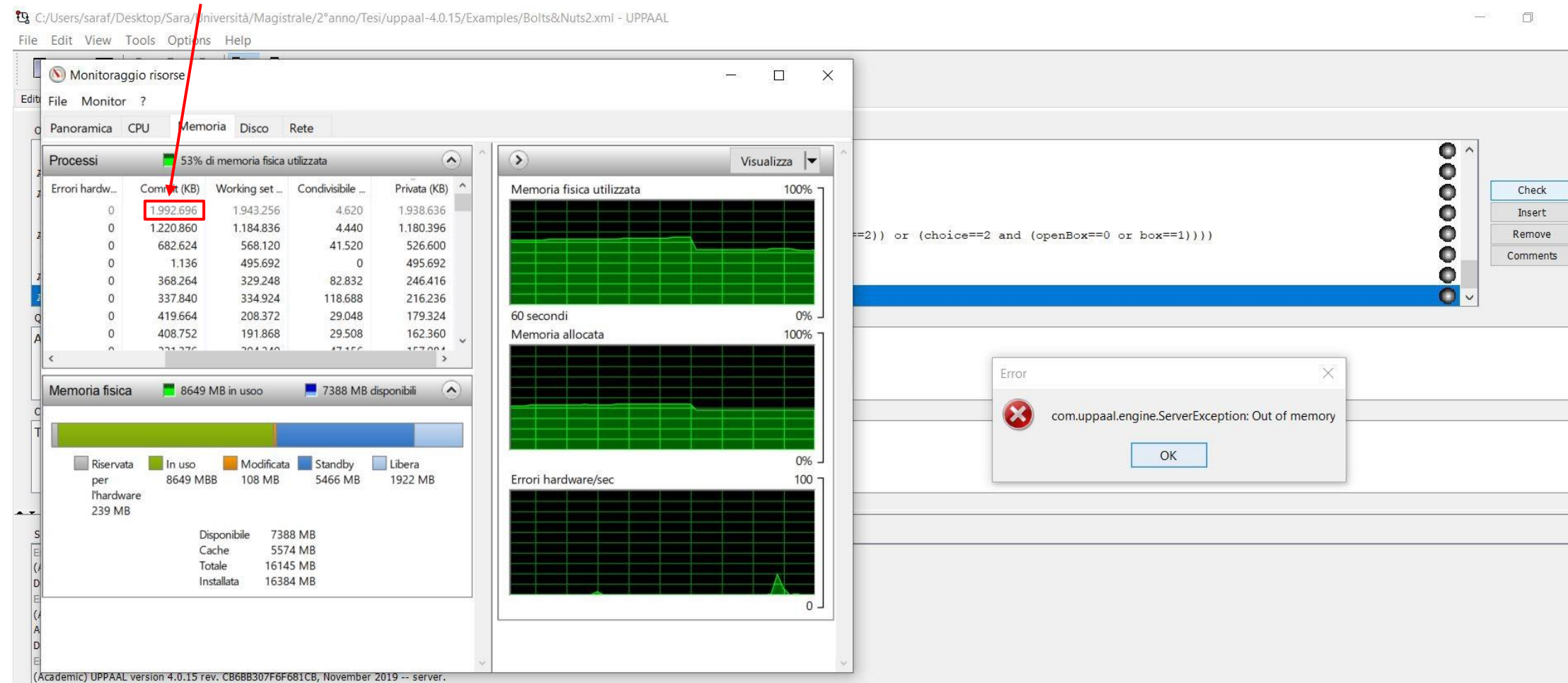


Figure 10 – Memory usage

With the model represented in figure 8 we are able to verify only properties that do not hold:

- $A[]$ deadlock: NOT SATISFIED
this property is not satisfied, in fact in this system we never have deadlock, the system is always available to work.
- $A[]$ Process.start imply key==0: NOT SATISFIED
if we are in state *start* then the key cannot be turned off, because in order to start working the machine needs the key to allow a user to use it. This property is telling us that if we are in the state *start* then the key is equal to 0, it means that there is not the key, but this is not possible and so this property is not satisfied, because only if there is the key then we can reach the node *start*.
- $A[]$ Process.bolts imply choice==1: NOT SATISFIED
 $A[]$ Process.bolts imply choice==2: NOT SATISFIED
if we are in the node *bolts* then it means that the choice made by the user was bolts and cannot be equal to 1 (the user selected nuts) or equal to 2 (both bolts and nuts).
- $A[]$ Process.nuts imply choice==0: NOT SATISFIED
 $A[]$ Process.nuts imply choice==2: NOT SATISFIED
similar to the previous two properties, if we reach the state *nuts*, it means that the choice done by the user was nuts (choice==1) and then cannot be choice==0 or choice==2.
In a similar way we define these two properties:
- $A[]$ Process.boltsANDnuts imply choice==1: NOT SATISFIED
 $A[]$ Process.boltsANDnuts imply choice==0: NOT SATISFIED
it means that the node *boltsANDnuts* is reached only when the choice is equal to 2 (user wants both nuts and bolts), and so the choice cannot be equal to 1=only nuts or 0=only bolts.
- $A[]$ (Process.laststep and choice==0) imply (counterBolts!=numberBolts): NOT SATISFIED
 $A[]$ (Process.laststep and choice==1) imply (counterNuts!=numberNuts): NOT SATISFIED

$A[]$ (Process.laststep and choice==2) imply (counterNuts!=numberNuts and counterBolts!=numberBolts): NOT SATISFIED

once we reach the state *laststep*, if the choice was equal to 0 then the preselected number of bolts by the user has to be equal to the counterBolts that counts the number of bolts that are inserted into the box, this is done in order to assure that the correct number of pieces is inserted in the box. In a similar way we check the same thing for the case in which the user's choice was only nuts, or both nuts and bolts.

As we say that with the initial complex big Timed Automata in UPPAAL we are not able to demonstrate the properties that are satisfied then we consider smaller models that are subsets of the initial whole system.

The first submodel that we use is:

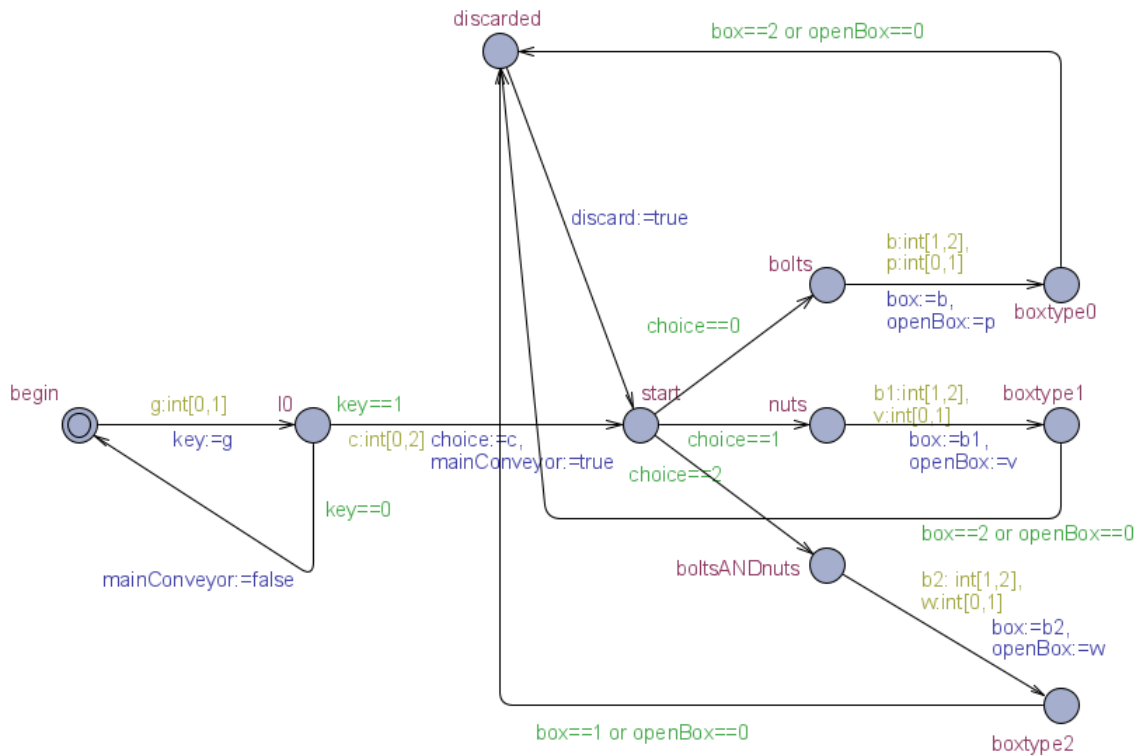


Figure 11 – First part of the automaton

Some properties that we want to check to verify the robustness of our system can be verified over the smaller models; these properties are:

- $A[]$ Process.start imply key==1: SATISFIED

the physical system can be used only if the key is inserted, so only a person with the key can turn on the machine. If in the automaton the state *start* is reached it means that the key was inserted. If at a certain moment someone removes the key, the machine continues working until it finishes the current process. Once it has finished, the screen shows an alert stating that in order to start, the key is needed.

- $A[\] \text{ Process.bolts} \text{ imply } \text{choice}==0: \text{ SATISFIED}$

the state *bolts* is reached only if the choice was 0, then only if the choice was bolts. This property ensures that once a user does a choice, so given a specific input (in this case bolts desired), the system will give always the correct output (the box will contain bolts).

Obviously, we can check the same property when the choice is nuts:

- $A[\] \text{ Process.nuts} \text{ imply } \text{choice}==1: \text{ SATISFIED}$

it means that if I am in the node *nuts*, the choice was equal to 1 (=nuts).

And we can check the same properties also when the choice is both nuts and bolts:

- $A[\] \text{ Process.boltsANDnuts} \text{ imply } \text{choice}==2: \text{ SATISFIED}$

if we reach the state *boltsANDnuts* it means that the user selection was both bolts and nuts.

- $A[\] (\text{Process.discarded} \text{ imply } (((\text{choice}==0 \text{ or } \text{choice}==1) \text{ and } (\text{openBox}==0 \text{ or } \text{box}==2)) \text{ or } (\text{choice}==2 \text{ and } (\text{openBox}==0 \text{ or } \text{box}==1))))): \text{ SATISFIED}$

when the system reaches node *discarded*, it means that or the box was closed or with respect to the choice the type of the box (with no partitions or bipartitioned) was not correct. The property in fact, said that if the choice is 0 or 1 (bolts or nuts) then we need the box of type 1 (with no partition) but if the box is of the other type (bipartitioned $\rightarrow \text{box}==2$), then the box is discarded; if the choice was of both bolts and nuts ($\text{choice}==2$) then the type of box needed is the box bipartitioned ($\text{box}==2$), but if there is a box with no partition ($\text{box}==1$) then the box is discarded. Whatever the choice (0,1 or 2), if a box arrives that is closed ($\text{openBox}==0$), then it is discarded.

Considering instead this other submodel, for checking properties on the second part of the whole initial system:

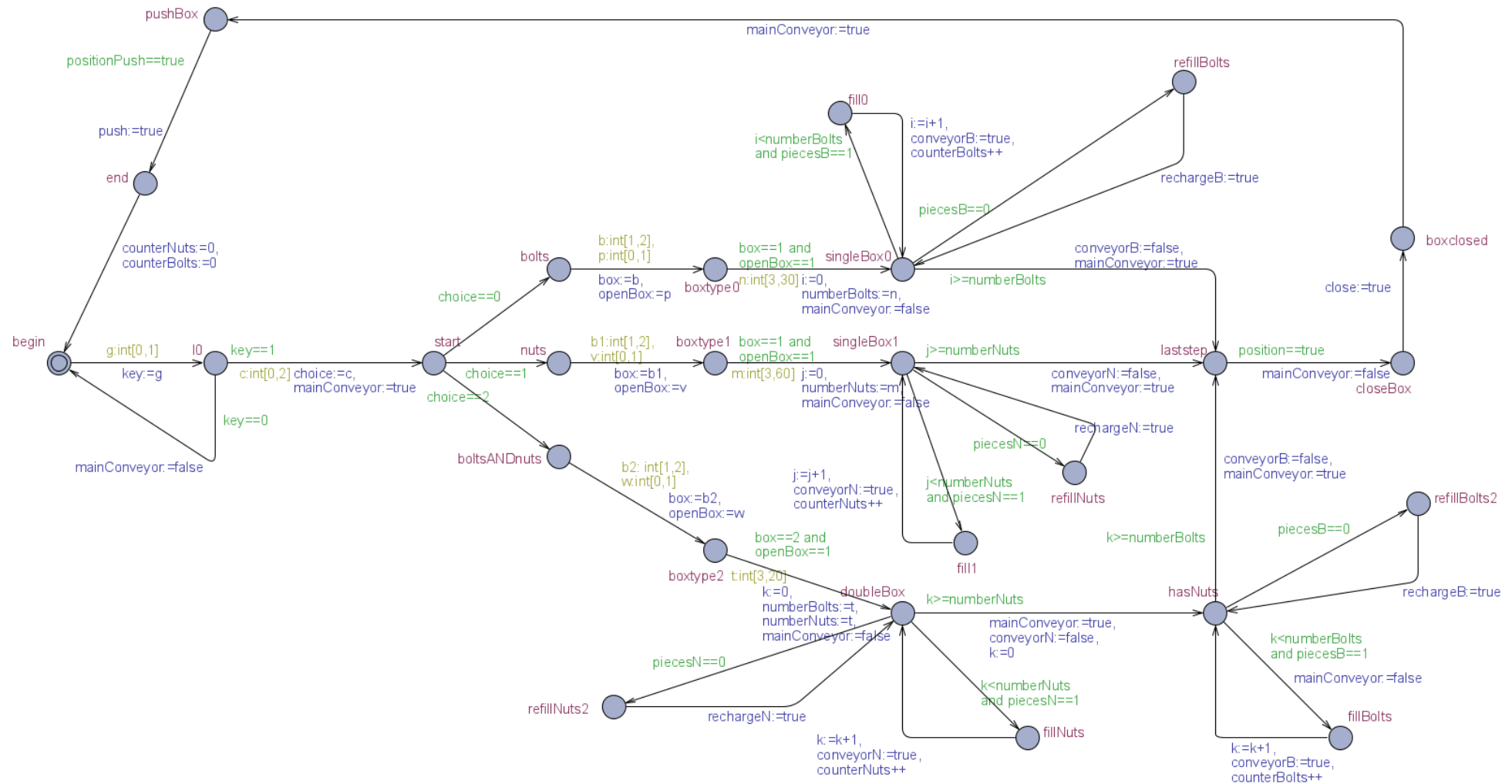


Figure 12 – Second part of the automaton

The difference with the initial model consists in the removal of the part where the box is discarded if it is closed or of the erroneous type (with no partition or bipartitioned).

Other properties that we checked over this second submodel to verify the robustness of our system are:

- $A[]$ (Process.singleBox0 or Process.singleBox1) imply (openBox==1 and box==1): SATISFIED

if I am in the node *singleBox0* or in the node *singleBox1* it means that the choice was only bolts or only nuts respectively, and we reach these states only if the box is of the correct type. In this case, we need the box with no partition and also the box needs to be open, because if the box is closed, then it is discarded and there is a “restart” where the choice made by the user remain in memory.

- $A[]$ Process.doubleBox imply (openBox==1 and box==2): SATISFIED

If the choice is both bolts and nuts, then if the box is that one bipartitioned and it is open then we reach the state *doubleBox*.

- $A[]$ (Process.laststep and choice==0) imply (counterBolts==numberBolts): SATISFIED

if we are in the state *laststep* and the choice was bolts, then the counterBolts (that is a counter used to count how many pieces are inserted in the box) should be equal to the desired number of bolts selected before by the user (numberBolts).

The same we have in the case in which the choice is nuts, then the number of nuts selected by the user should be equal to the counterNuts that count how many nuts are inserted in the box:

$A[]$ (Process.laststep and choice==1) imply (counterNuts==numberNuts): SATISFIED

- $A[]$ (Process.laststep and choice==2) imply (counterNuts==numberNuts and counterBolts==numberBolts and numberBolts==numberNuts): SATISFIED

if the choice was of both nuts and bolts, once we arrive in the state *laststep* the number of nuts desired by the user should be equal to the counterNuts,

the same for the desired number of bolts should be equal to the pieces of bolts inside the box (*counterBolts*); also we can demonstrate that as the number of bolts selected from the user is the same of the number of nuts, then in the box at the end we will have the same number of pieces of bolts and nuts.

- $A[] \text{ Process.singleBox0} \text{ imply } (\text{numberBolts} \geq 3 \text{ and } \text{numberBolts} \leq 30)$: SATISFIED

in the node *singleBox0* the number of bolts selected by the user can be between 3 and 30; another way to express the same property is:

$A[] \text{ Process.singleBox0} \text{ imply } (!(\text{numberBolts} < 3) \text{ and } !(\text{numberBolts} > 30))$

- $A[] \text{ Process.singleBox1} \text{ imply } (!(\text{numberNuts} < 3) \text{ and } !(\text{numberNuts} > 60))$: SATISFIED

the number of nuts instead would be between 3 and 60, we can have more pieces than bolts because the nuts have a smaller dimension and then in the box there can be more pieces of nuts.

- $A[] \text{ Process.doubleBox} \text{ imply } (!(\text{numberNuts} < 3) \text{ and } !(\text{numberNuts} > 20) \text{ and } !(\text{numberBolts} < 3) \text{ and } !(\text{numberBolts} > 20))$: SATISFIED

if we are in the state *doubleBox* then the both the number of bolts and the number of nuts are between 3 and 20.

- $A[] \text{ Process.doubleBox} \text{ imply } (\text{choice} == 2 \text{ and } \text{numberBolts} == \text{numberNuts})$: SATISFIED

when we reach the node *doubleBox* it means that the user selection was of both nuts and bolts and then the number of bolts should be equal to the number of nuts.

- $A[] (\text{Process.refillBolts} \text{ or } \text{Process.refillBolts2}) \text{ imply } \text{rechargeB} == \text{true}$: SATISFIED

when we reach the node *refillBolts* or *refillBolts2* it means that the pieces of bolts in the smaller conveyor for bolts are finished, then we need to recharge the conveyor with new pieces, once there are again some pieces then the process continues.

- $A[] (\text{Process.refillNuts} \text{ or } \text{Process.refillNuts2}) \text{ imply } \text{rechargeN} == \text{true}$: SATISFIED

the same as the previous property, once there are no more nuts then we reach the node *refillNuts* or *refillNuts2* and we need to refill the conveyor for nuts with new pieces of nuts.

- A[] Process.boxclosed imply close==true: SATISFIED
if we are in the state *boxclosed* then it means that the box has already been closed.
- A[] Process.end imply push==true: SATISFIED
when we are in the node *end* it means that a new box has been fill with bolts or nuts or both, and then is pushed in order to be separated from the discarded boxes.

Our model will always get to a state where the box is discarded or contains bolts and nuts, so the system never stops, because once the box is discarded the process “restart” and the model already have the choice of the user of bolts or nuts or both in the memory.

We can get stuck when we remain in the loops that recharge the conveyors of nuts and bolts (from node *singleBox0* to node *refillBolts*, or from node *singleBox1* to node *refillNuts*, or from node *doubleBox* to node *refillNuts2*, or from node *hasNuts* to node *refillBolts2*), because if no one recharges the conveyors with new pieces then the machine stops until it has new pieces to fill the boxes. Also in this case, we can say that our model is robust, because if there are no more pieces we want that the system stops until there are more pieces, because if the machine go on anyway when there no more pieces, at the end in the box there will not be the correct number of nuts or bolts.

We get stuck also in the loop where we control if there is the key to turn on the machine or not, but the system is anyway robust because we do not want the system to start working until there is no key turned on.

Then we can conclude that by verifying the previous properties we are able to say that our system is robust, because either when it receives correct inputs (given a choice there is the correct box opened) the machine works correctly, and also when it receives erroneous inputs (such as a closed box or given a user choice there is not the box of the corrected type: with no partition or bipartitioned), the machine does not stop working but simply discard the box and restart the

process keeping in memory the user's choice. We get stuck in our model, only in the case that the pieces are finished and no one recharge the machine, but we can assume that if there are no more pieces it is correct that the process stops and restart when new pieces are inserted by another machine; we can conclude that our system works always correctly and so it is a robust system.

Conclusions

Programmable Logic Controllers are increasingly used in Industrial Control Systems in order to automate the processes. As they are user interactive and input dependent, they are subject to cyber security attacks. Therefore, we want to obtain a system that is always reliable, and so we need to create a robust system.

Our approach starts from a PLC program written in the SCL code applying some transformation rules we wanted to get an automaton. We did it by transforming the SCL code into a Domain Specific Language with Xtext, we create the .xta file with the help of Acceleo in Eclipse, the .xta file automatically creates the Timed Automata. Finally, we check the robustness of our system by using UPPAAL, where we check if some properties were satisfied or not, over our automaton.

The main limitation of this approach is the fact that model checking suffers the state explosion problem. For industrial software which is simple enough to be represented by a limited number of states, then this approach might be useful; whereas with complex system, we have to either use other techniques or to provide huge amount of computational power. In our case, we have solved the problem by representing false properties with the initial whole timed automata, and the true properties have been checked by using two smaller automata that are submodels of our main system.

We have chosen the bolts and nuts example because it is user interactive and input dependent, and so we wanted to prove the robustness of this model either with correct inputs but also with unexpected inputs; we have seen that also with unacceptable inputs our model does not stop working; we were able to guarantee that anyway the behaviour of the PLC program is not inconsistent and so we can conclude that our system is robust.

Future work is focused towards applying our approach to more complex examples, also more user interactive examples; overcoming the state explosion problem by using other techniques or dividing a complex big timed automata in smaller automata and synchronized them.

References

1. Wen Chinn Yew, "PLC Device Security - Tailoring needs", SANS Institute Information Security Reading Room, February 15, 2017.
2. R. Johnson, "Survey of scada security challenges and potential attack vectors", in Proceedings of International Conference for Internet Technology and Secured Transactions (ICITST), IEEE, London, UK, Nov. 2010.
3. H. Wardak, S. Zhioua and A. Almulhem, "PLC access control: a security analysis", 2016 World Congress on Industrial Control Systems Security (WCICSS), London, 2016, pp. 1-6. doi: 10.1109/WCICSS.2016.7882935.
4. Yong Wang, Jinyong Liu, Can Yang, Lin Zhou, Shuangfei Li, Zhaoyan Xu, "Access Control Attacks on PLC Vulnerabilities", Journal of Computer and Communications, 6, 311 - 325, 2018.
5. Z. Basnight, J. Butts, J. L. Jr., and T. Dube, "Firmware modification attacks on programmable logic controllers", International Journal of Critical Infrastructure Protection, vol. 6, no. 2, pp. 76 - 84, 2013.
6. L. Garcia and S. A. Zonouz, "Hey, My Malware Knows Physics! Attacking PLCs with Physical Model Aware Rootkit", in Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS '17), 2017.
7. A. Abbasi and M. Hashemi, "Ghost in the PLC: Designing an Undetectable Programmable Logic Controller Rootkit via Pin Control Attack", in Black Hat Europe '16, November 2016, pp. 1 - 35.
8. L. Cojocar, K. Razavi, and H. Bos, "Off-the-Shelf Embedded Devices as Platforms for Security Research", in Proceedings of the 10th European Workshop on Systems Security (EuroSec'17), April 2017, pp. 1:1 - 1:6.
9. Shengqi Yang, Liang Chih Cheng, Mooi Choo Chuah, "Detecting Payload Attacks on Programmable Logic Controllers (PLCs)", IEEE Conference on Communications and Network Security (CNS), pp. 1-9, 2018. DOI:10.1109/cns.2018.8433146.
10. J. O. Malchow, D. Marzin, J. Klick, R. Kovacs, and V. Roth, "PLC Guard: A Practical Defense against Attacks on Cyber-Physical Systems", in 2015 IEEE Conference on Communications and Network Security (CNS), September 2015, pp. 326 - 334.

11. H. Janicke, A. Nicholson, S. Webber, and A. Cau, "Runtime-Monitoring for Industrial Control Systems", *Electronics*, vol. 4, no. 4, pp. 995 - 1017, December 2015.
12. S. E. McLaughlin, S. A. Zonouz, D. J. Pohly, and P. D. McDaniel, "A Trusted Safety Verifier for Process Controller Code", in *Proceedings of the 2014 Network and Distributed System Security Symposium (NDSS '14)*, 2014.
13. S. Zonouz, J. Rrushi, and S. McLaughlin, "Detecting Industrial Control Malware Using Automated PLC Code Analytics", *IEEE Security Privacy*, vol. 12, no. 6, pp. 40 - 47, November 2014.
14. Jean-Claude Fernandez, Laurent Mounier, and Cyril Pachon, "A model-based approach for robustness testing".
15. Gerd Behrmann, Alexandre David, and Kim G Larsen, "A tutorial on uppaal", In *Formal methods for the design of real-time systems*, pp. 200 - 236. Springer, 2004.
16. Doaa Soliman, Kleanthis Thramboulidis, Georg Frey, "Transformation of Function Block Diagrams to UPPAAL timed automata", *Annual Reviews in Control* 36 (2012), 338 - 345.
17. K. Thramboulidis, D. Soliman, G. Frey, "Towards an automated verification process for industrial safety applications". In *IEEE 7th International conference on Automation Science and Engineering (IEEE CASE 2011)*. August 24 - 27, Trieste, Italy.
18. G. Canet, S. Couffin, J. -. Lesage, A. Petit and P. Schnoebelen, "Towards the automatic verification of PLC programs written in Instruction List", *Smc 2000 conference proceedings. 2000 IEEE international conference on systems, man and cybernetics. 'cybernetics evolving to systems, humans, organizations, and their complex interactions' (cat. no.0, Nashville, TN, 2000, pp. 2449 - 2454 vol.4. doi: 10.1109/ICSMC.2000.884359)*.
19. O. Rossi, O. de Smet, S. Lampérière-Couffin, J.-J. Lesage, H. Papini, and H. Guennec. "Formal verification: a tool to improve the safety of control systems". In *4th Symposium on Fault Detection, Supervision and Safety for Technical Processes (IFAC Safeprocess 2000)*, Budapest, Hungary, 2000. to appear.

20. Siemens, "SIMATIC Programming with STEP7", - Manual, 2010, A5E02789666-01.
21. Siemens, "SIMATIC Structured Control Language (SCL) for S7-300/S7-400 Programming Manual", 6ES7811-1CA02-8BA0.
22. G. Behrmann, A. David, K. G. Larsen, "A Tutorial on Uppaal", Aalborg University, 2004.
23. 61131-3. "Programmable controllers - Part 3: Programming languages. Norma Internacional, IEC (International Electrotechnical Commission)", Reference number IEC 61131-3:2003(E), 2003.
24. P. Ferrara, A. Cortesi, F. Spoto, "From CIL to Java bytecode: Semantics-based translation for static analysis leveraging". Science of Computer Programming. 102392. 10.1016/j.scico.2020.102392, 2020.
25. Riccardo Gaboardi, "Progetto maturità 2018 elettrotecnica, linea automatizzata per l'inscatolamento gestita dal PLC", available: <https://www.youtube.com/watch?v=T-svOlnpYuU>
26. "List of model checking tools", available: https://en.wikipedia.org/wiki/List_of_model_checking_tools
27. "UPPAAL", available: <http://www.uppaal.org/>
28. Kally Anton, "Siemens Intro to Structure Control Language (SCL) in TIA Portal with S7-1200/1500", SCL - Professional Control Corporation, 2019, available: <https://www.pccweb.com/wp-content/uploads/2019/11/S17-SCL.pdf>
29. Siemens, "Simatic SCL Totally Integrated Automation (TIA) Portal", 2013, available: https://s4458523b90cc6aef.jimcontent.com/download/version/1491242329/module/6897624456/name/SCL_TIA_PORTAL_%201500.pdf
30. D. Darvas, B. Fernández Adiego, E. Blanco Viñuela, "PLCverif: a tool to verify PLC programs based on model checking techniques", CERN, Geneva, Switzerland.