



Universidad de Oviedo

Department of Electrical Engineering, Electronics, Computers, and  
Systems

PhD Programme in Process Control, Industrial Electronics and  
Electric Engineering

PhD Thesis:

## **Bringing Automated Formal Verification to PLC Program Development**

A Thesis submitted by Borja Fernández Adiego for the  
degree of Doctor of Philosophy in the University of  
Oviedo.

Supervised by:

Dr. Víctor Manuel González Suárez

Dr. Enrique Blanco Viñuela





*A mis abuelos Alberto y Fernando.*

*“Yes, there are two paths you can go by,  
but in the long run, there’s still time  
to change the road you’re on.”  
Led Zeppelin, Stairway to Heaven (1971)*



# Summary

Automation is the field of engineering that deals with the development of control systems for operating systems such as industrial processes, railways, machinery or aircraft without human intervention. In most of the cases, a failure in these control systems can cause a disaster in terms of economic losses, environmental damages or human losses.

For that reason, providing safe, reliable and robust control systems is a first priority goal for control engineers. Ideally, control engineers should be able to guarantee that both software and hardware fulfill the design requirements. This is an enormous challenge in which industry and academia have been working and making progresses in the last decades.

This thesis focuses on one particular type of control systems that operates industrial processes, the PLC (Programmable Logic Controller) - based control systems. Moreover it targets one of the main challenges for these systems, guaranteeing that PLC programs are compliant with their specifications.

Traditionally in industry, PLC programs are checked using testing techniques. Testing consists in checking the requirements on the real system. Although these testing techniques have achieved good results in different kind of systems, they have some well-known drawbacks such as the difficulty to check safety and liveness properties (e.g. ensuring a forbidden output value combination should never occur).

This thesis proposes an alternative for checking PLC programs. A methodology based on formal verification techniques, which can complement the testing techniques to guarantee that a PLC program is compliant with the specifications.

Formal verification is a technique meant to prove the correctness of a system by using formal methods. One of the most popular formal verification techniques is model checking, which consists in checking a

formalized requirement in a formal model of the system. Comparing model checking with testing, model checking explores all the possible combinations of the state space in the formal model to guarantee that the formal requirement is satisfied.

Formal verification and in particular model checking appears to be a very appropriate technique for this goal. However, the industrial automation community has not adopted yet this approach to verify PLC code, even if some standards, like the IEC 61508 (2010), highly recommend the use of formal methods for Safety Instrumented Systems. This is due to following challenges for control engineers: (1) the difficulty of building formal models representing real-life PLC programs, (2) the difficulty of using specification formalisms to express the requirements and finally, (3) when creating formal models out of real-life software, the number of combinations can be huge and model checking tools may not be able to handle the state space, thus cannot evaluate the given requirement.

This research deals with these three main challenges and tries to fill the gap between the industrial automation and the formal verification communities.

The thesis proposes a general methodology for applying automated formal verification to PLC programs and any complexity related to formal methods is hidden from control engineers.

In this methodology, formal models are built automatically out of the PLC programs. The model transformations are divided in two parts: PLC programs, from the IEC 61131 (2013) standard, are translated to an Intermediate Model (IM), which is the central piece of this methodology. The IM is then transformed to the input modeling languages of different verification tools (e.g. nuXmv, UPPAAL or BIP). This modeling strategy simplifies the model transformations and makes the integration of new verification tools easier.

Regarding the requirements formalization, this methodology provides a solution that allows control engineers to express the requirements in a simple and natural language based on patterns with well-defined semantics. Then, these requirements are translated to temporal logic formalisms as they are the most common formalisms used by the verification tools.

Regarding the state space explosion problem, this methodology provides a set of reduction and abstraction techniques that are applied

to the IM. These techniques are a fundamental part of the methodology, as they make the verification of real-life PLC programs, which usually have huge state spaces, possible.

The methodology has been applied to real-life PLC programs developed at CERN. These experimental results have demonstrated the usability of this methodology by control engineers with no experience in formal methods.



# Resumen

Automatización industrial es el campo de la ingeniería que se dedica al desarrollo de sistemas de control para operar sistemas como procesos industriales, trenes, maquinaria o aviones sin intervención humana. En la mayoría de los casos, un fallo en estos sistemas de control puede provocar un desastre en términos de pérdidas económicas, daños medioambientales o pérdidas humanas.

Por esa razón, proporcionar sistemas de control seguros, fiables y robustos es un objetivo de primera prioridad para los ingenieros de control. Idealmente, ingenieros de control deberían de ser capaces de garantizar que tanto el hardware como el software satisfacen los requisitos del diseño. Esto es un gran reto en el cual industria y academia han estado trabajando y haciendo progresos en las últimas décadas.

Esta tesis se centra en un tipo particular de sistemas de control que opera procesos industriales, los sistemas de control basados en PLCs (Programmable Logic Controller) y tiene como meta uno de los principales retos para estos sistemas, garantizar los programas del PLC respetan las especificaciones de diseño.

Tradicionalmente en industria, los programas de PLCs son comprobados utilizando técnicas de testeo. Testear un sistema consiste en comprobar los requisitos del diseño en el sistema real. Aunque estas técnicas han logrado buenos resultados en diferentes tipos de sistemas, tiene algunas limitaciones bien conocidas, como son la dificultad de testear propiedades de seguridad o “liveness” (por ejemplo asegurar que un valor prohibido de una variable de salida nunca ocurra en el sistema).

Esta tesis propone una alternativa para chequear programas de los PLCs. Una metodología basada en técnicas de verificación formal, las cuales pueden complementar las técnicas de testeo para garantizar que el programa del PLC respeta sus especificaciones.

Verificación formal es una técnica que tiene por objetivo probar que un sistema está correctamente diseñado o implementado utilizando métodos formales. Una de las técnicas de verificación formal más populares es “model checking”, la cual consiste en comprobar que un requisito formalizado es respetado en un modelo formal del sistema real. Comparando model checking con las técnicas de testeo, model checking explora todas las posibles combinaciones en el modelo formal para garantizar que el modelo respeta el requisito formalizado.

Técnicas de verificación formal y en particular model checking, parecen ser técnicas muy apropiadas para el objetivo de esta tesis. Sin embargo, la comunidad de automatización industrial no ha adoptado aún estas técnicas para verificar el código de los PLCs, incluso si estándares como el IEC 61508 (2010) recomiendan el uso de métodos formales para sistemas de seguridad. Esto es debido a tres factores fundamentales: (1) la dificultad de construir los modelos formales que representan programas de PLC, (2) la dificultad de usar los formalismos para especificar los requisitos del sistemas y finalmente, (3) cuando se crea un modelo formal de un software real, el número de combinaciones posibles a explorar puede ser enorme y las herramientas que implementan model checking pueden no ser capaces de explorar el espacio de estados, por lo tanto no pueden evaluar los requisitos dados.

Este trabajo de investigación se centra en estos tres aspectos e intenta cubrir la grieta que existe en las comunidades científicas de automatización industrial y verificación formal.

La tesis propone una metodología para aplicar automáticamente (sin intervención humana) técnicas de verificación formal a programas PLC y toda complejidad relacionada con métodos formales está oculta para los ingenieros de control.

En esta metodología, modelos formales son generados a partir de los programas PLC. La transformación de modelos está dividida en dos partes: los programas PLC, definidos en el estándar IEC 61131 (2013), son traducidos a un modelo intermedio, referenciado como IM en todo el documento por sus siglas en Inglés (Intermediate model). Este modelo intermedio es la pieza central de esta metodología. Más tarde, el IM es transformado en los modelos necesarios para las diferentes herramientas de verificación (por ejemplo, nuXmv, UPPAAL o BIP). Esta estrategia de modelado simplifica las transformaciones entre mo-

delos y hace mucho más sencilla la integración de nuevas herramientas de verificación.

En cuanto la formalización de los requisitos que el programa debe cumplir, esta metodología proporciona una solución que permite a los ingenieros de control expresar los requisitos con un lenguaje natural y sencillo basado en el uso de patrones con una semántica bien definida. Más tarde, estos requisitos son traducidos a lógica temporal, ya es que el formalismo más utilizado por las herramientas de verificación formal.

En cuanto al problema de la explosión del espacio de estados, esta metodología proporciona un conjunto de técnicas de reducción y de abstracción de modelos, las cuales son aplicadas al IM. Estas técnicas son una pieza fundamental de la metodología, ya que hace posible la verificación formal de programas PLC que controlan sistemas industriales reales, los cuales normalmente tiene un espacio de estados enorme.

La metodología ha sido empleada en programas PLC reales desarrollados en el CERN. Los resultados experimentales han demostrado la utilidad de esta metodología, usada por ingenieros de control sin experiencia en métodos formales.



# Contents

<b>Summary</b>	<b>v</b>
<b>Resumen</b>	<b>ix</b>
<b>Acknowledgements</b>	<b>xvii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Context . . . . .	3
1.2 Motivation and objectives . . . . .	5
1.3 Contributions of this thesis . . . . .	6
1.4 Publications linked to this thesis . . . . .	8
1.5 Document structure . . . . .	10
<b>2 Background and related work</b>	<b>13</b>
2.1 Introduction . . . . .	13
2.2 PLC-based control systems . . . . .	14
2.2.1 Control system classification . . . . .	16
2.2.2 Standards . . . . .	17
2.2.3 Frameworks . . . . .	19
2.2.4 PLC hardware . . . . .	23
2.2.5 PLC software . . . . .	25
2.3 Formal methods and formal verification . . . . .	32
2.3.1 Formal methods . . . . .	32
2.3.2 Formal verification . . . . .	35
2.3.3 Model checking . . . . .	37
2.3.4 Verification tools . . . . .	44
2.4 Related work . . . . .	45
2.4.1 Testing-based techniques . . . . .	46
2.4.2 Formal verification based techniques . . . . .	47

2.5	Summary of the chapter . . . . .	56
<b>3</b>	<b>Approach</b>	<b>59</b>
3.1	Introduction . . . . .	59
3.2	General overview of the approach . . . . .	62
3.3	Intermediate model . . . . .	70
3.3.1	Intermediate model syntax . . . . .	70
3.3.2	Intermediate model semantics . . . . .	71
3.4	Formal specifications . . . . .	73
3.4.1	Patterns . . . . .	75
3.5	PLC hardware modeling . . . . .	79
3.5.1	PLC inputs . . . . .	80
3.5.2	Safety and standard PLCs . . . . .	82
3.5.3	Interrupts and restarts in PLCs . . . . .	82
3.6	PLC code – IM transformation . . . . .	83
3.6.1	General PLC – IM transformation . . . . .	84
3.6.2	ST – IM transformation . . . . .	89
3.6.3	SFC – IM transformation . . . . .	93
3.7	Reduction techniques . . . . .	94
3.7.1	Cone of influence . . . . .	98
3.7.2	Rule-based reductions . . . . .	104
3.7.3	Mode selection . . . . .	107
3.7.4	Iterative variable abstraction . . . . .	108
3.8	IM– verification tools transformation . . . . .	130
3.8.1	IM–nuXmv transformation . . . . .	130
3.8.2	IM–UPPAAL transformation . . . . .	137
3.8.3	IM–BIP transformation . . . . .	148
3.9	Modeling timing aspects of PLCs . . . . .	160
3.9.1	Realistic approach . . . . .	161
3.9.2	Abstract approach . . . . .	169
3.9.3	Refinement between the two approaches . . . . .	172
3.10	Process modeling . . . . .	174
3.11	Verification and counterexample analysis . . . . .	177
3.11.1	Counterexample analysis . . . . .	178
3.12	Methodology CASE tool . . . . .	181
3.13	Summary of the chapter . . . . .	182

<b>4</b>	<b>Case studies and measurements</b>	<b>185</b>
4.1	Introduction . . . . .	185
4.2	UNICOS framework . . . . .	186
4.3	UNICOS baseline object case study . . . . .	191
4.3.1	Object description and specification . . . . .	191
4.3.2	Experimental results regarding model generation	192
4.3.3	Experimental results regarding verification of complex properties . . . . .	199
4.3.4	Experimental results regarding verification of time properties . . . . .	201
4.3.5	Counterexample analysis . . . . .	205
4.4	Full UNICOS application case study . . . . .	208
4.4.1	Process description and specification . . . . .	208
4.4.2	Experimental results regarding verification of safety properties . . . . .	209
4.5	Summary of the chapter . . . . .	217
<b>5</b>	<b>Evaluation and analysis</b>	<b>219</b>
5.1	Introduction . . . . .	219
5.2	Evaluation of requirements formalization . . . . .	220
5.3	Evaluation of PLC hardware and process modeling . . . . .	222
5.4	Evaluation of the PLC program – IM transformation . . . . .	224
5.4.1	Evaluation of the time-related transformation rules . . . . .	226
5.5	Evaluation of the reduction techniques . . . . .	227
5.5.1	Property preserving reduction techniques . . . . .	228
5.5.2	State space . . . . .	228
5.5.3	Verification run-time . . . . .	229
5.5.4	Variable abstraction . . . . .	231
5.6	Evaluation of the IM – verification tools transformation	233
5.7	Evaluation of the verification results . . . . .	234
5.8	Correctness of our approach . . . . .	235
5.9	Summary of the chapter . . . . .	236
<b>6</b>	<b>Conclusions and future work</b>	<b>239</b>
6.1	Discussion . . . . .	239
6.2	Contributions . . . . .	241
6.2.1	Methodology . . . . .	241

6.2.2	IM syntax and semantics . . . . .	241
6.2.3	<i>PLC code - IM</i> transformation rules . . . . .	242
6.2.4	<i>IM- input verification tools</i> transformation rules	242
6.2.5	Reduction techniques . . . . .	242
6.2.6	Modeling the timing aspects of PLCs . . . . .	243
6.2.7	PLC behavior analysis . . . . .	243
6.2.8	Applicability on CERN control systems . . . . .	244
6.3	Future work . . . . .	245
<b>7</b>	<b>Conclusiones y trabajo futuro</b>	<b>249</b>
7.1	Discusión . . . . .	249
7.2	Contribuciones . . . . .	251
7.2.1	Metodología . . . . .	251
7.2.2	Sintaxis y semántica del IM . . . . .	252
7.2.3	Reglas de transformación <i>código PLC - IM</i> . . . . .	252
7.2.4	Reglas de transformación <i>IM- modelo para las herramientas de verificación</i> . . . . .	252
7.2.5	Técnicas de reducción . . . . .	253
7.2.6	Modelado de los aspectos temporales de los PLCs	253
7.2.7	Análisis del funcionamiento de un PLC . . . . .	254
7.2.8	Aplicación en sistemas de control desarrollados en el CERN . . . . .	254
7.3	Trabajo futuro . . . . .	256
<b>A</b>	<b>PLC programs</b>	<b>259</b>
<b>B</b>	<b>nuXmv models</b>	<b>277</b>
	<b>List of Acronyms</b>	<b>325</b>
	<b>List de Figures</b>	<b>329</b>
	<b>List of Tables</b>	<b>333</b>
	<b>List of Listings</b>	<b>335</b>
	<b>Bibliography</b>	<b>337</b>

# Acknowledgements

This part of the thesis is certainly the most important one for me, as it allows me to thank and express my appreciation to some of the people who have helped me during the years of my PhD candidature.

I would like to start with my supervisors Enrique Blanco Vinuela and Víctor Manuel González Suárez, who gave me the opportunity to work with them during all these years. I am extremely grateful for their wise advices on the scientific and technological aspects of this research.

To all my colleagues from the EN-ICE group at CERN and specially to Philippe Gayet, who gave me the opportunity to be part of this group.

To my colleague Daniel Darvas, who worked with me in this project for the past one and a half years, for his high quality work and the constructive discussions. I am very pleased that Daniel will continue and complement this research during his PhD candidature.

To my colleague Jean-Charles Tournier for his support on the initial idea of this thesis and for the interesting brainstorming and discussions during the last three years.

To my colleague Luis Gómez Palacín, for his help and good advices during the last two years at CERN.

To my colleague Iván Prieto Barreiro, from whom I have learned a lot about different programming languages and with whom I had the pleasure to work closely during my first two years at CERN.

To all my colleagues from the PLC section at CERN, from whom I have learned a lot during all these years and for the great moments I have shared with them in coffee breaks, dinners, etc.

To Efthymia Laderi from CERN and Gemma Hernández Redondo from the Oviedo University for their patience and help on adminis-

trative matters.

To all the members of the RiSD laboratory directed by Prof. Joseph Sifakis at the EPFL University in Lausanne (Switzerland), especially to Simon Bliudze from whom I have learned a lot about formal methods and formal verification.

To András Vörös and Tamás Bartha from the BUTE University in Budapest (Hungary), for the interesting discussions about formal verification applied to safety critical systems.

To Jan Olaf from the RMIT University in Melbourne (Australia), for the interesting discussions about formal semantics and formal verification applied to PLC programs.

Thanks to these five years at CERN, I have met a lot of people from all over the world who are today an important part of my life. Some of these people are Borja, Guillermo, Piotr, David, Pablo, Miguel, Marc, Edu, Cris, Alberto and my friends from the basketball team.

I cannot forget my friends Kostis and Benja with whom I have shared so many great trips and moments.

Me gustaría agradecer a mi profesor y amigo Pepe, por sus consejos para mis estudios universitarios y mi carrera profesional.

A toda mi familia, por su apoyo, cariño y toda la confianza que siempre depositaron en mí. Desde los más jóvenes, Jimena, Jorge, María, Pablo y Alberto, pasando por mi hermano Adrián, hasta mis abuelos Cuca, Fernando, Mari y Alberto y mi bisabuela Abi.

A mis amigos y compañeros de universidad, entre ellos Samu, Edu, Ablanado y Manu, con los que he compartido grandes momentos.

A mis amigos de La Bañeza, David, Edu, Javi, Ruben, Saúl e Iván, por todos los momentos compartidos, por las risas juntos y porque cualquier problema o mal rato se olvida cuando paso tiempo con ellos.

A Jenny, por su cariño y paciencia conmigo, por todo su apoyo y ayuda en momentos que no fueron muy fáciles para mí.

A mis padres, a los cuales les debo todo. Cualquier persona que les hubiera tenido como padres hubiera conseguido lo mismo o mucho más de lo que yo he conseguido. Mi mayor motivación en mi carrera profesional es hacerles sentir orgullosos, para agradecerles todo su cariño y apoyo incondicional.

# Chapter 1

## Introduction

In the nineteenth century, the industrial revolution provoked very significant economic and social changes in the human society. From this event until now, the development of new machines and technologies have had an enormous impact on our lives. Industrial automation was created to free humans from tedious monitoring tasks in industrial processes. Since 1910, when Henry Ford automated his automobile assembly plant, industrial automation has been evolving and becoming an essential element in the development of any industry. The transformation of automation technologies, from the first electro-pneumatic devices to the complex programmable controllers existing today, shows the evolution of this industry, led by a high demand of complex and flexible control devices from other industries.

New progresses in control theory and in control technologies, such as distributed systems, have transformed industrial automation into a relevant area of research with an countless number of practical applications in any aspect of our lives.

“Overall, with more than two centuries’ development and evolution, industrial control and automation technologies have so advanced that they benefit us in all aspects of our life and in all kinds of production systems; they are closely integrated with computer hardware and software, network devices and communication technologies; they are faithfully based on modern results in the mathematical and physical sciences.” (Zhang (2010)).

Nowadays, one of the main needs in industrial automation is the ability to design safe, reliable and robust control systems which are

compliant with their specifications. Model-based design and verification of controllers (hardware and software) are some of the research trends in automation nowadays. Software failures in control systems can provoke enormous damages to human beings, to the environment and cause an enormous economic breakdown. One of many unfortunate examples is the software error which provoked the accident of Ariane 5 space rocket from the European Space Agency. The Ariane 5 software was previously tested.

The formal verification community provides solutions to guarantee that a system is compliant with the system specifications, using strong mathematical bases.

This thesis fills the gap between these two communities, industrial automation and formal verification, to make progress in the development of safer, more reliable and more robust control systems, by guaranteeing that the control software is compliant with the specifications. This thesis focuses on PLC (Programmable Logic Controller) – based control systems, as PLCs are the most common control devices used in industry.

It is important to mention that in the automation industry the so-called “Safety Instrumented Systems” (SIS), defined by the IEC 61511 (2003), are systems designed to protect people, environment and industrial installations. With the evolution of technologies these systems are mostly based on programmable controllers, such as PLCs. This thesis concentrates on the software of PLC control systems independently of its purpose. Obviously guaranteeing that the PLC program of a SIS is compliant with the specification is essential and highly recommended by the standards (for example, the IEC 61508 (2010) standard) but this is also desirable in any PLC control system that monitors and regulates an industrial installation. This is because a software bug can provoke significant damages or the cost of fixing this bug is bigger than the cost of using formal methods. Moreover, nowadays industry demands high availability of industrial control systems to maximize the uptime of the controlled process, even if a bug in the code would not provoke a damage to the installation, it could stop the installation which would cause economic losses.

This chapter is divided in the following sections:

1. Section 1.1 presents the general context of this thesis.

2. Section 1.2 gives an overview of the motivation for this research.
3. Section 1.3 enumerates the contributions of this thesis.
4. Section 1.4 lists the scientific publications that are the result of this research.
5. Finally, Section 1.5 introduces the structure of this thesis.

## 1.1 Context

This PhD has been developed under the doctoral student program at CERN (European Organization for Nuclear Research) in collaboration with the University of Oviedo (Spain).

CERN is the biggest particle physics laboratory in the world, located at the border between France and Switzerland. CERN has four main goals:

1. Push forward the frontiers of knowledge in particle physics.
2. Develop new technologies for accelerators and detectors. Many of these techniques can also be applied to other industries. The transfer of knowledge and technology is also an important goal of CERN.
3. Train the scientists and the engineers of tomorrow.
4. Unite people from different countries and cultures.

CERN has a particle accelerator complex to perform the experiments and some of these particle accelerators are: PS (Proton Synchrotron), SPS (Super Proton Synchrotron), LINACs (LINEar ACcelerators) and LHC (Large Hadron Collider) (See Fig. 1.1). The LHC is the biggest particle accelerator in the world with 27 km of circumference. The goal of this machine is to recreate the conditions that existed immediately after the big bang. To do that, the LHC accelerates two high-energy particle beams (e.g. protons) at close to the speed of light and makes them collide. These collisions that happen at a very high energy level (14 TeV), provide a very valuable insight of the particles by observing their outcome: energies, trajectories and eventually, the creation of

new particles. Some of them are well-known by the physicists but some others are just part of theories and their existence has never been proved. The LHC aims at proving or disproving these theories, which should help in the understanding of the universe. To detect these new particles the LHC has four main particle detectors (ATLAS, CMS, ALICE and LHCb).

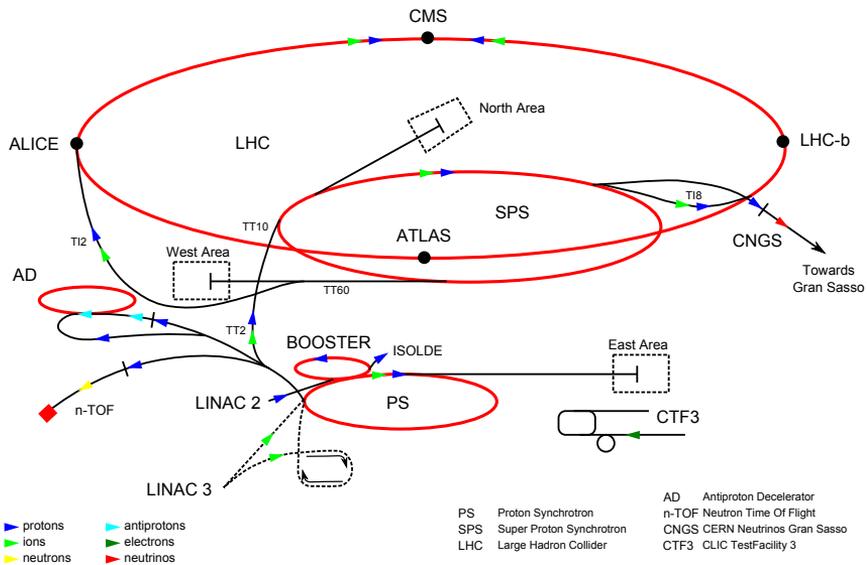


Figure 1.1: CERN accelerator complex from <http://en.wikipedia.org/wiki/CERN>

To produce these high energy collisions, it requires some auxiliary industrial processes (e.g. cryogenics and vacuum systems) to provide the optimized conditions for the particle accelerators and detectors. Moreover, these industrial processes have to be operated automatically, therefore industrial control systems are needed. This research was done at CERN in the ICE (Industrial Controls & Engineering) group inside the EN (Engineering) department, which develops solutions and provides support in the domain of large and medium scale industrial control systems as well as the accelerator systems.

At the Oviedo University, the thesis is supported by the ISA (System Engineering and Automation) area inside the DIEECS (Electric, Electronics, Computers and Systems Engineering) department. ISA focuses its research on the different areas of the automation field.

## 1.2 Motivation and objectives

Nowadays any industrial installation has to be controlled and operated automatically, looking for these three main objectives:

1. Increasing the quality and efficiency of the process.
2. Enforcing the safety by reducing human interaction with the industrial process.
3. Reducing the costs by optimizing the energy consumption of the installation.

It is obvious that having a control system that contains flaws may cause damages and economic losses to the installation itself, but it also can affect the environment and people. Having a safe, robust and bug-free control system is a common “desire” of designers and developers involved in the project. However, producing a control system that is fully compliant with the project specifications is a very challenging task. The automation industry lacks of modern software engineering best practices to guarantee it and this research is focused on this topic.

Some standards address this problem, specially standards focused on functional safety, like the IEC 61508 (2010). This standard proposes some guidelines for the development of safety systems and their verification (more details about the standard can be found in Chapter 2).

Certainly, the most challenging task is to guarantee that the control system software is compliant with the specification. In industry, many techniques have been applied to achieve this goal, like manual and automated testing or applying simulation. However these techniques have several drawbacks, like the difficulty of checking liveness properties (e.g. “after a manual request from the operator, the valve  $v$  will eventually be closed”) or the difficulty of checking all the possible combinations of the state space in a safety property (e.g. “if valve  $v$  is closed, then valve  $w$  can never be closed at the same time”).

The IEC 61508 standard recommends the use of formal methods to guarantee that the software is compliant with the specifications, specially for systems with a high Safety Integrity Level (SIL)<sup>1</sup>, see

---

<sup>1</sup>SIL is an integrity characteristic to measure the risk-reduction provided by a safety function in a safety instrumented system.

Table 1.1. This table appears in “Part 3: Software requirements (Table A.2, page 37)” of the IEC 61508 (2010) standard.

Table 1.1: Software design and development: software architecture design (see Table 7.4.3 from the IEC 61508 (2010) standard)

<b>Technique/Measure</b>	<b>Ref</b>	<b>SIL1</b>	<b>SIL2</b>	<b>SIL3</b>	<b>SIL4</b>
7b Semi-formal methods	Table B.7	R	R	HR	HR
7c Formal methods	C.2.4	–	R	R	HR
8 Computer-aided specification tools	B.2.4	R	R	HR	HR

NR: Not Recommended.

R: Recommended.

HR: Highly Recommended.

Programmable Logic Controllers are the most widely used control devices in industry for automation purposes. At CERN, many industrial installation such as the LHC cryogenic system, cooling & ventilation systems, LHC vacuum systems, etc. are controlled by PLCs (e.g. Willeman et al. (2011)). These systems are developed using the so-called UNICOS framework described in Blanco Viñuela et al. (2011).

The fundamental goal of this thesis is to provide a methodology which guaranties that any PLC program is compliant with the project specifications. This methodology is based on applying automated formal verification to PLC programs, hiding the complexity from the control engineers.

### 1.3 Contributions of this thesis

To the best of the author’s knowledge, this thesis presents several contributions on the field of formal verification of PLC programs, as follows:

1. The internal behavior (informal semantics) of a PLC is described in this thesis. This description is focused on Siemens PLCs as it

is one of the two PLC suppliers at CERN. This analysis, which is presented in Chapter 2, is essential to justify the decisions and the modeling strategy.

2. This thesis presents a general methodology for applying automated formal verification to PLC programs. The methodology is based on an Intermediate Model (IM) and PLC programs written in any PLC language from the IEC 61131 (2013) standard can be expressed in this IM. This IM can be transformed to the input modeling languages of many verification tools. In Chapter 3, the methodology is presented in Section 3.2.
3. The syntax and formal semantics of the IM are described. In Chapter 3, the IM is presented in Section 3.3.
4. As the ST and SFC languages are the most relevant languages in CERN PLC control systems, the formal transformation rules from ST and SFC to the IM are described. The rules are presented in Chapter 3, Section 3.6.
5. The methodology is supported by a CASE tool, which is able to generate formal models from ST and SFC PLC programming languages passing through the IM. Currently the tool generates models for the following verification tools: nuXmv Cavada et al. (2014), UPPAAL Amnell et al. (2001) and BIP Basu et al. (2011). The rules are presented in Chapter 3, Section 3.8.
6. The generated models of real-life PLC programs have usually a huge state space, which cannot be verified by any of the verification tools. Reduction and abstraction techniques are applied to the intermediate model making verification of these programs feasible (including novel and existing algorithms adapted to this methodology, achieving better results than existing solutions).
7. Modeling time and real time systems increases the complexity of the model and the state space. The strategy of modeling the timing aspects of PLC programs is presented. These particular rules are presented in Chapter 3, Section 3.9.

8. The methodology has been applied to real-life systems at CERN, showing the feasibility and applicability of the methodology. The experimental results are presented in Chapter 4.

## 1.4 Publications linked to this thesis

The following articles were published during the development of this thesis:

1. Fernández Adiego, B., Blanco Viñuela, E., and Merezhin, A. (2013a). Testing & verification of PLC code for process control. In *Proc. of 13th ICALEPCS*. This paper describes two different approaches to verify PLC programs: testing and formal verification.
2. Fernández Adiego, B., Blanco Viñuela, E., Tournier, J.-C., González Suárez, V. M., and Bliudze, S. (2013b). Model-based automated testing of critical PLC programs. In *11th IEEE Int. Conf. on Industrial Informatics*, pages 722–727. This paper describes a first approach of modeling UNICOS control systems in order to apply automated model-based testing.
3. Darvas, D., Fernández Adiego, B., and Blanco Viñuela, E. (2013). Transforming PLC programs into formal models for verification purposes. Internal Note CERN-ACC-NOTE-2013-0040, CERN. This technical report describes the first version of some of the transformation rules from PLC code to the NuSMV modeling language.
4. Fernández Adiego, B., Darvas, D., Tournier, J.-C., Blanco Viñuela, E., and González Suárez, V. M. (2014c). Bringing automated model checking to PLC program development – A CERN case study. In *Proc. of the 12th IFAC-IEEE International Workshop on Discrete Event Systems*. This paper presents a case study of applying the proposed methodology to a PLC control system developed at CERN.
5. Darvas, D., Fernández Adiego, B., Vörös, A., Bartha, T., Blanco Viñuela, E., and González Suárez, V. M. (2014). Formal verification of complex properties on PLC programs. In

Ábrahám, E. and Palamidessi, C., editors, *Formal Techniques for Distributed Objects, Components, and Systems*, volume 8461 of *Lecture Notes in Computer Science*, pages 284–299. Springer. This paper describes the property preserving reduction techniques applied in the methodology.

6. Fernández Adiego, B., Darvas, D., Blanco Viñuela, E., Tournier, J.-C., González Suárez, V. M., and Blech, J. O. (2014a). Modelling and formal verification of timing aspects in large PLC programs. In *Proc. of IFAC World Congress*. This paper describes the modeling strategy of the timing aspect of PLCs.
7. Fernández Adiego, B., Darvas, D., Tournier, J.-C., Blanco Viñuela, E., Blech, J. O., and González Suárez, V. M. (2014b). Automated generation of formal models from ST control programs for verification purposes. Internal Note CERN-ACC-NOTE-2014-0037, CERN. This paper describes the transformation rules from ST PLC code into the modeling languages of verification tools through an intermediate model.

Before focusing my research on the formal verification of PLC programs, I studied the most relevant control system frameworks which have been applied mainly in scientific installations. In addition, the real-life PLC programs, presented in experimental results of this thesis, have been developed at CERN using the UNICOS framework. Three papers, in which I have participated, have been published about the UNICOS framework:

1. Fernández Adiego, B., Blanco Viñuela, E., and Barreiro, P. (2011). UNICOS CPC6: Automated code generation for process control applications. In *Proc. of 12th ICALEPCS*. This paper describes the architecture of the automatic generation tool for UNICOS control systems.
2. Copy, B., Blanco Viñuela, E., Fernández Adiego, B., Nogueira Fernandes, R., and Barreiro, P. (2011). Model oriented application generation for industrial control systems. In *Proc. of 12th ICALEPCS*. This paper presents the UNICOS metamodel used as definition of the UNICOS library meant to represent control system instrumentation as UNICOS objects.

3. Blanco Viñuela, E., Merezhin, A., Bradu, B., Fernández Adiego, B., Willeman, D., Rochez, J., Beckers, J., Ortola Vidal, J., Durand, P., and Izquierdo Rosas, S. (2011). UNICOS evolution: CPC version 6. In *Proc. of 12th ICALEPCS*. This paper describes the new features of the CPC package from the UNICOS framework.

## 1.5 Document structure

This thesis is organized as follows:

**Chapter 1:** this first chapter describes the context of this thesis, presents the motivation and objectives and gives a general overview of the contributions. It also lists the publications produced out of this research.

**Chapter 2:** this chapter gives an overview of the two different “worlds” linked to this research, industrial automation and formal verification. Initially, it introduces the chapter explaining the link between both fields. Then, it describes the main features related to PLC-based control systems needed to explain the proposed modeling strategy. Lately, an overview of main concepts of formal verification is introduced. Finally, the related work of formal verification applied to PLC-based control systems is presented.

**Chapter 3:** this chapter presents the core of this thesis. The proposed methodology is introduced and lately the details of the different steps of the methodology are presented: formalization of requirements, system modeling, reduction techniques, verification strategy and analysis of verification results.

**Chapter 4:** this chapter presents the experimental results obtained by applying the methodology on real-life CERN control systems. These systems are developed using the UNICOS framework. This framework is introduced in this chapter and the two selected systems are presented.

**Chapter 5:** this chapter presents an evaluation and analysis of the obtained results.

**Chapter 6:** this chapter concludes the thesis, summarizing the contributions and results and presenting the future work of this research.

**Appendix A:** this appendix presents one of the real-life PLC programs, which have been verified applying the proposed methodology

**Appendix B:** this appendix presents some of the generated nuXmv models out of the real-life PLC programs.

**Lists** The thesis finishes with the list of acronyms, figures, tables and bibliography.



# Chapter 2

## Background and related work

### 2.1 Introduction

This chapter gives an overview of the relevant concepts of the two fields of study involved in this research: industrial automation and formal verification. The automation concepts are either fundamental to understand the proposed methodology for applying formal verification to PLC programs (e.g. description of hardware and software in PLCs) or relevant in the design of reliable control systems (e.g. standards or frameworks).

Automation, in general, is the use of control systems for operating different equipments such as processes in factories, aircraft, machinery, etc. by reducing human intervention. Many different devices are used as control systems depending on the equipment to be controlled (e.g. digital or analog devices) and the needs of control (e.g. speed, PID regulation). Some of the most common devices are: FPGAs (Field-Programmable Gate Array), industrial PCs and PLCs. PLC is the most popular control device for industrial processes such as cooling and ventilation, oil or chemical processes. PLCs are appropriate devices for processes with both digital and analog instrumentation (e.g. digital and analog valves, temperature sensors, digital pumps, etc.) and with strong requirements in reliability under harsh environment. Section 2.2 describes the main characteristics and concepts of PLC-based control systems that have been taken into account in this

research.

Formal verification is a technique meant to prove the correctness of a system by using formal methods of mathematics. Formal verification and formal methods are also introduced in this chapter. In the last two decades, these techniques have been applied successfully in many different industries such as aircraft, space shuttle, and railway systems. The related work section presents a brief overview of the applicability of these techniques in other areas. However, despite PLCs are the most common control device for safety and for standard control systems, formal verification is not really applied in this industry yet. Section 2.4 presents a detailed overview of the different techniques designed to check PLC programs and the differences with this thesis approach.

## 2.2 PLC-based control systems

The first PLC was born in 1968, when *GM Hydramatic* asked for an alternative to replace all the hard-wired relay systems by a programmable alternative. The company *Bedford Associates* came with the winning proposal and the first PLC was created. This first prototype was called *084*. *Bedford Associates* started a new company dedicated to develop, manufacture, sell, and service this new product: *Modicon*, which is currently property of *Schneider Electric*s (Schneider Electric's automation website (2014)). One of the person who worked on that project was Dick Morley, who is considered to be the “father” of the PLC.

The PLC is the most widely-used programmable electronic device designed for controlling industrial processes. Even if other kind of controllers, such as industrial PCs, are more powerful, sophisticated and offer more alternatives in terms of programming capacities, PLCs remain the most popular control device due to its reliability and robustness in industrial environments.

A PLC mainly consists of a processing unit and peripheries to connect with sensors and actuators of the process or with other electronic devices.

This thesis is based on *Siemens* PLCs (Siemens automation website (2014)), as they are widely used at CERN. Although some minor differences exist with the models produced by other manufacturers,

they all have a common PLC architecture described in the IEC 61131 (2013) standard.

Industrial control systems are traditionally divided in three main layers:

1. Supervision: this layer provides the interface with the process operator. The tool is usually called SCADA (Supervisory Control and Data Acquisition). It is in charge of showing the process variables to the operator, storing their values in databases and giving access to the operator to send manual orders to the control layer.
2. Control: this layer is composed of the control devices (e.g. PLCs) which contain the logic to automatize the process.
3. Field: this layer is in contact with the process to be controlled. It is composed of sensors, which take the information from the process, and actuators, which execute the orders given by the control layer.

Note that modern control systems include more parts (layers or sub-layers) to this traditional architecture. For example MES (Manufacturing Execution System) or ERP (Enterprise Resource Planning). Fig. 2.1 shows a common representation of the control system layers.

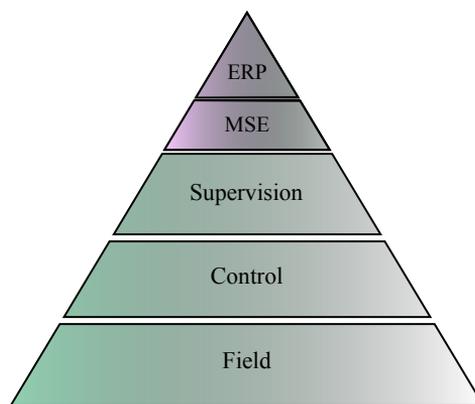


Figure 2.1: Control system layers

### 2.2.1 Control system classification

Independently of the use of PLCs or other control device, industrial control systems can have many different classifications based on the characteristics of the control layer: type of control applied to the process, controllers architecture, its purpose, etc.

Regarding the *control applied to the process*, an industrial control system can be classified as:

- Open-loop control system: it is a control system that only uses the current state of the input values and the program of the controller. It does not use feedback to guarantee if the output has the intended value due to the input request.
- Closed-loop control system: it is a control system that uses the current state of the input values, the program of the controller and some information from the output of the system that is returned back to the input and computed by the control program.

Regarding the *controllers architecture*, an industrial control system can be classified as:

- Centralized control system: it is a control system managed by a single controller, which contains the whole control logic.
- Distributed control system (DCS): it is a control system which is managed by more than one controller. In these systems the logic is split into different controllers. The controllers use field buses for the communication between them. The IEC 61499 (2013) standard proposes a methodology to design distributed systems based on function blocks (this concept is introduced in Section 2.2.5).

Please note that very often in industry the term DCS may refer to a distributed system in terms of the supervision layer, where different servers store the information from the industrial process in a database (usually a distributed database). Both architectures can be combined in the same control system.

Both centralized and distributed control systems can have a decentralized periphery, which uses field buses to communicate with the

controllers. Some of the most common field buses are Modbus (Modbus website (2014)), Profibus and Profinet (Profibus - Profinet website (2014)).

Regarding the *purpose*, an industrial control system can be classified as:

- Standard control system: it is a control system dedicated to control and monitor the industrial process.
- Safety control system: it is a control system dedicated to protect people, environment and installations. They are defined by the IEC 61508 (2010) standard as SISs and their goal is to provide *Functional Safety*, which increases the reliability of the global system.

### 2.2.2 Standards

When developing industrial control systems, the developer should follow a set of rules to guarantee certain common requirements, such as safety, maintainability or quality of control code. These rules, presented as guidelines, are given by the standards. There are several international organizations in charge of producing standards. The two more relevant organizations for the standardization in the automation discipline are:

- IEC (International Electrotechnical Commission): it is the world's leading organization that prepares and publishes international standards for all electrical, electronic and related technologies (International Electrotechnical Commission website (2014)).
- ISA (International Society of Automation): is a leading, global, nonprofit organization that prepares and publishes international standards for automation (International Society of Automation website (2014)).

Other organizations, like ISO (International Organization for Standardization website (2014)) or CENELEC (European Committee for Electrotechnical Standardization website (2014)) in Europe, are also relevant in the automation community.

Below a summary of the most relevant standards in the development or industrial control systems is presented:

1. IEC 61131 (2013) standard: it is a general standard for programmable controllers. It is divided into eight parts and it describes the main features of programmable controllers, such as equipment requirements, programming languages, guidelines for the application and implementations of these languages, etc. It is important to emphasize that PLC programming languages from different vendors have been developed or adapted, following the recommendations of this standard, although there are some minor differences between them. Five languages are defined by this standard, three graphical languages and two textual languages: Ladder diagram (LD), Function block diagram (FBD), Sequential function chart (SFC), Structured text (ST) and Instruction list (IL).
2. IEC 61499 (2013) standard: this standard extends the concept of function block (FB) defined in the IEC 61131 standard. The extension has the goal of improving this concept using object-oriented features. It also describes a methodology to design the control software based on these FBs. Fig. 2.2 shows the representation of the proposed FB by this standard. This FB is composed of the ECC (Execution Control Chart) and the control algorithms. The execution of these algorithms is triggered by the ECC, which is an event-driven state machine similar to the well-known Harel Statecharts.
3. ISA 88 (2010) standard: this standard focuses on the batch process control. It provides a methodology to design control systems for this particular case of processes, with the goal of optimizing the production in a flexible way. It is divided into four parts and it describes the applied models and terminology, the data structures, gives some recipe models and provides examples of use.
4. IEC 61512 (2009) standard: as the previous standard, it focuses on the batch process control. It is an extended version of the ISA 88. It proposes the creation of a hierarchy of modules to model and control the batch processes.
5. IEC 61508 (2010) standard: it is the standard focused on the *Functional Safety* of Electrical/Electronic/Programmable Elec-

tronic Safety-related Systems (E/E/PE, or E/E/PES). The goal of this standard is to become the reference standard in terms of functional safety to all kind of industries. It provides guidelines for all the needed steps for the development of a SIS. These steps correspond to the so-called *Safety Life Cycle*, from risk analysis to the decommissioning of the SIS.

6. IEC 61511 (2003) standard: it is a standard about Functional Safety as well, but focused on the industrial process. It provides the definition of safety concepts such as SIS (Safety Instrumented System), SIF (Safety Instrumented Function) and SIL (Safety Integrity Level). SIL is used to categorize the level of safety of a system and it ranges from 1 (for the lowest safety level in this category) to 4 (for the highest safety level in this category).
7. IEC 62061 (2012) standard: it is a specific implementation of the IEC 61508 standard for machinery as its title reveals: “Safety of machinery: Functional safety of electrical, electronic and programmable electronic control systems”.

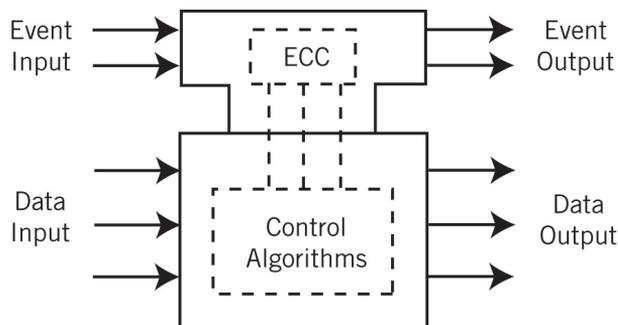


Figure 2.2: IEC 61499 FB representation

### 2.2.3 Frameworks

On the top of the standards, several companies and organizations have developed some “frameworks”, with the goal of homogenizing and standardizing the control systems even more. Many control system frameworks can be found in the literature, mostly the ones developed

by research organizations (e.g. CERN, ESRF, etc.) and universities as private companies usually do not publish this knowledge. These frameworks standardize many different aspects of the control systems such as communication protocols, data storage, control code, SCADA visualization, etc. Some of the advantages for using these frameworks are:

- It reduces the development time, as they usually provide some tools dedicated to solve repetitive tasks, for example code generation tools.
- It improves and simplifies the maintenance of the control systems due to the standardization.
- It reduces the chances of having software errors when the framework provides a solution to standardize the control code, due to the reusability of software blocks (e.g. FBs).

As this thesis has been developed at CERN, some of the most relevant control system frameworks applied to big scientific installations are presented in this section. These frameworks are: EPICS, TANGO, FESA and UNICOS.

### 2.2.3.1 EPICS

EPICS (Experimental Physics and Industrial Controls) is a control system framework co-developed by the “Accelerator technology” control group at Los Alamos (Accelerator and Operations Technology website (2014)) and by the “Advanced Photon Source” control group at the Argonne National Laboratory (Advanced Photon Source website (2014)) in USA. This framework was born to provide a solution for control and supervision in big scientific installations, as the control needs of these installation were not available in industry. EPICS provides a standard solution for the communication between the control and supervision layers in distributed systems. Fig. 2.3 shows the EPICS architecture, which is based on a *Client – Server* model. The communication is based on the “Channel Access” protocol. The first documentation about its architecture can be found in Dalesio et al. (1991). It also provides a set of tools for the control systems developers. They are called the EPICS subsystems (alarm manager, display

manager, etc.) and the *OPI* (Operator Interface) consists in workstations where the EPICS subsystems are executed. The *IOC* (Input-Output Controllers) supports real time databases. The hardware is based on VME/VXI systems using VxWorks or RTEMS operating systems.

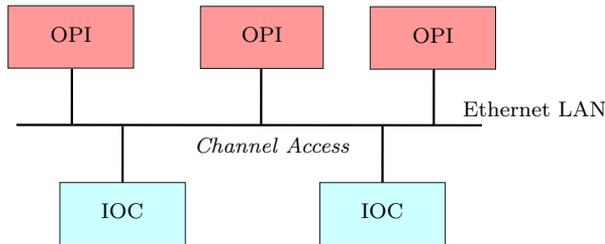


Figure 2.3: EPICS architecture

In addition EPICS uses a distributed database providing local control for each IOC. It provides data acquisition, data conversion, alarm detection and control closed loops. It provides a timing system for the events synchronization along the network. As it is shown in Fig. 2.4, different control devices can be connected to the IOCs, e.g. PLCs, FPGAs, etc.

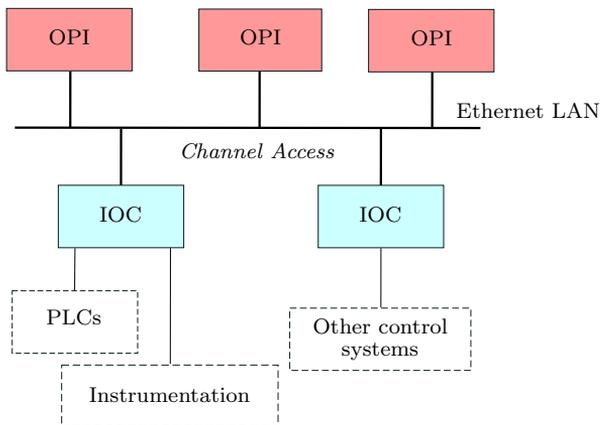


Figure 2.4: EPICS control system example

EPICS is used in more than 100 different projects all over the world, mainly in big scientific installations (e.g. DESY website

(2014)). More information about EPICS can be found on the EPICS website (2014).

### 2.2.3.2 TANGO

TANGO is the European alternative to EPICS. It was developed initially by the control group of ESRF (ESRF website (2014)), and currently more research centers collaborate on the development (e.g. ALBA website (2014)). Nowadays these institutes constitute the TANGO consortium (TANGO website (2014)), which provides a standard solution for the communication between the different elements of the system as it is shown in Fig. 2.5. It was designed for distributed control systems and it uses the object oriented paradigm based on CORBA (CORBA website (2014)). It provides a set of tools for the development of “device servers”. These elements provide access to all the elements of the control system. Thanks to the consortium, it is possible to develop “devices servers” and “clients” in different languages, like C++, Java or Python. In addition, TANGO provides several APIs (Application Programming Interface) for the development of these elements. Hardware can range from simple digital input/outputs up to sophisticated detector systems or entire plant control systems. More information about TANGO can be found in Chaize et al. (1999).

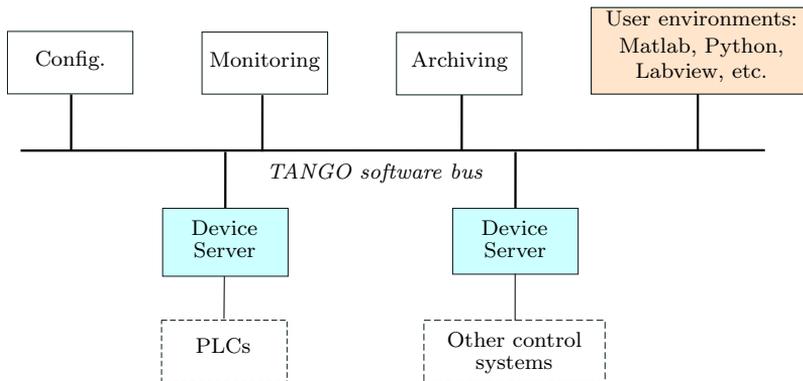


Figure 2.5: TANGO architecture

### 2.2.3.3 FESA

FESA (Front-End Software Architecture) is an object oriented framework, developed at CERN by the BE/CO group CERN BE/CO group website (2014). It generates automatically code for the control layer, which is based on industrial PCs called FECs (Front End Computers). It provides a set of tools for design, development and test of real time software. At the SCADA level, Java applications have been developed to monitor the information from the FECs. At the control level, the so-called FESA classes represent real devices from the instrumentation of the control system. It also provides a *Timing system* for the synchronization along the network. The LHC control system at CERN is mainly based on FESA. Other scientific institutes started to use FESA, for example, GSI ( GSI website (2014)). More information about FESA can be found in Peryt and Martín Marquez (2009).

### 2.2.3.4 UNICOS

UNICOS (UNified Industrial COntrol System) is a industrial control framework developed at CERN by the EN/ICE group (CERN EN/ICE group website (2014)). This framework provides a methodology and a set of tools to develop industrial control systems. The experiments presented in Chapter 4 use PLC programs from the UNICOS library and PLC programs developed using this framework. Therefore, this framework is described in detail in Section 4.2.

Control system frameworks are important in this research as they contribute to produce more reliable control systems. However, none of the described frameworks include the use of formal methods in their development process yet. More general information about control system frameworks can be found in Lopez (2006).

## 2.2.4 PLC hardware

This section describes the main characteristics of the PLC hardware. The concepts presented in this section are fundamental to understand the modeling strategy of the PLC execution platform.

### 2.2.4.1 Execution schema

The main particularity of the PLC is its execution scheme, the so-called *scan cycle*. It consists of three main steps:

1. Reading the inputs from periphery to the *Input Image Memory* (IIM).
2. Executing the user program that reads and modifies the *Output Image Memory* (OIM) contents.
3. Writing the values to the output periphery.

The PLC reads the information coming from the process (sensors and actuators) and stores it either in the IIM or the OIM. The IIM values are frozen during the execution of the user program.

The execution of the scan cycle can be interrupted if an event (e.g. timer, hardware event, hardware error) triggers the execution of an interrupt handler. The interrupts are preemptive and they are assigned to priority classes in compilation time that will determine their priority.

### 2.2.4.2 PLC memory

The “data memory” of a PLC is divided into different areas depending of its purpose. The memory can be divided into two main parts (Siemens (1998a); Siemens AG (2010)).

1. One part of the memory is globally accessible and allocated statically. This part stores the image of the input and output values, and the internal computation results.
2. The rest of the memory is only accessible locally and is allocated dynamically. The so-called *L Stack* is in this part and it stores the temporary data of functions, separately for each priority class.

### 2.2.4.3 PLC interrupts and restarts

PLC interrupts are described by the IEC 61131 standard. In Siemens PLCs, interrupts trigger the call of Organization Blocks (OBs), which

are the interfaces between the operating system and the PLC program. They have different priorities and therefore a higher priority OB can interrupt any lower priority OB. Restarts are a special case of interrupts and they trigger a specific OB as well.

There are different kind of interrupts in Siemens PLCs, some of the main examples are:

- Time-of-day interrupts (OB10..17). These interrupts are triggered at a specific time on each day or month.
- Cyclic interrupts (OB30..38). These interrupts are triggered cyclically with a defined cyclic time.
- Hardware interrupts (OB40..47). These interrupts are triggered by hardware events (e.g. errors, diagnostics, etc.).
- Startup (OB100..102). These interrupts are triggered when a PLC restart occurs.

## 2.2.5 PLC software

The third part of the IEC 61131 (2013) standard describes the different software resources, which are necessary to build the user program in PLCs. However, there are some minor differences between the implementations provided by the PLC vendors. The following paragraphs describes the PLC resources given by Siemens PLCs.

### 2.2.5.1 PLC blocks

In Siemens PLCs, several kinds of program blocks are defined for various purposes Siemens AG (2010).

- A *function* (FC) is a piece of executable code with input, output and temporary variables. The variables are stored on the L Stack and they are deleted after the execution of the function.
- An *organization block* (OB) is a special function that can be only called by the system. These are the entry points of the user code. The main program and the interrupt handlers are implemented as OBs.

- A *data block* (DB) is a collection of static variables that can be accessed globally in the program. These variables are stored permanently. The data block does not contain any executable code.
- A *function block* (FB) is a piece of executable code with input, output, static and temporary variables. Static variables are stored in *instance data blocks* and these variables can be accessed globally, even before or after the execution of the FB. The temporary variables are stored on the L Stack, similarly to the FC's variables.

### 2.2.5.2 Programming

There are five PLC languages defined by the IEC 61131 (2013) standard: *ST*, *SFC*, *Ladder*, *FBD* and *IL*. The PLC programmer can choose one or several of these languages, depending on the characteristics of the application, to build the PLC code.

ST is the most used PLC programming language at CERN. However SFC and IL are also used:

- ST is a textual high-level language that is syntactically similar to Pascal.
- The SFC language is a graphical programming language based on steps and transitions. It is useful when a part of the PLC program can be conveniently represented as a finite-state machine (FSM).
- The IL language is a low level language that is syntactically similar to assembly.
- The FBD language is a graphical language based on logic gates.
- The Ladder language is a graphical language based on electric circuit diagrams of relay logic hardware.

All these languages are compiled to a common byte code called MC7 and this is the code transferred to the PLC. The MC7 instructions are assumed to be atomic and they cannot be interrupted. A

single ST or SFC statement can correspond to several MC7 instructions.

Listing 2.1 shows an example ST code. This example code defines a function block (FB100) with three variables ( $a$ ,  $b$ ,  $c$ ). There is an instance data block (DB1) defined for FB100. In the organization block OB1, the FB100 is called using the instance data block DB1 with input parameter  $a=false$ . Then the  $c$  variable of this instance is assigned to the output Q1.0.

```

1 FUNCTION_BLOCK FB100
2   VAR_INPUT
3     a : BOOL;
4   END_VAR
5   VAR_TEMP
6     b : BOOL;
7   END_VAR
8   VAR
9     c : BOOL;
10  END_VAR
11  BEGIN
12    b := NOT a;
13    c := b;
14  END_FUNCTION_BLOCK
15
16 DATA_BLOCK DB1 FB100
17  BEGIN
18  END_DATA_BLOCK
19
20 ORGANIZATION_BLOCK OB1
21  VAR_TEMP
22    info : ARRAY[0..19] OF BYTE; // reserved
23  END_VAR
24  BEGIN
25    FB100.DB1(a := FALSE);
26    Q1.0 := DB1.c;
27  END_ORGANIZATION_BLOCK

```

Listing 2.1: Example of ST code

Listing 2.2 shows an example SFC code. It defines a FSM with three states (*Stop*, *Fill*, *Run*) with three possible transitions. The conditions of the transitions are Boolean input variables (*StopCond*, *FillCond*, *RunCond*) and their value is assigned outside of the FSM-representing function block. The graphical representations can be seen in Fig. 2.6 and Fig. 2.7 (this last representation corresponds with screenshot of the SIMATIC tool of Siemens).

```

1 FUNCTION_BLOCK FB101
2   VAR_INPUT

```

```

3      FillCond : BOOL := FALSE;
4      RunCond  : BOOL := FALSE;
5      StopCond : BOOL := FALSE;
6  END_VAR
7
8  INITIAL_STEP Stop: END_STEP
9  STEP Fill: END_STEP
10 STEP Run: END_STEP
11
12 TRANSITION S_F
13     FROM Stop TO Fill CONDITION := FillCond
14 END_TRANSITION
15
16 TRANSITION F_R
17     FROM Fill TO Run CONDITION := RunCond
18 END_TRANSITION
19
20 TRANSITION R_S
21     FROM Run TO Stop CONDITION := StopCond
22 END_TRANSITION
23 END_FUNCTION_BLOCK

```

Listing 2.2: Example of the textual representation of SFC code

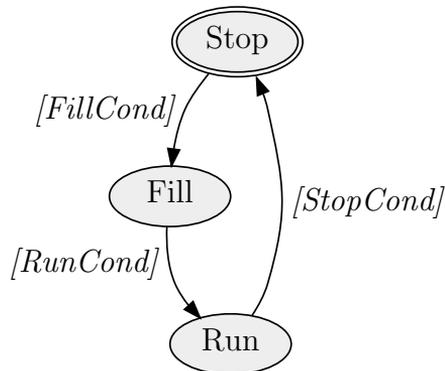


Figure 2.6: Example SFC

### 2.2.5.3 Timing behavior of PLCs

PLC control systems can perform timing operations and these operations are very common when controlling industrial processes. Un-

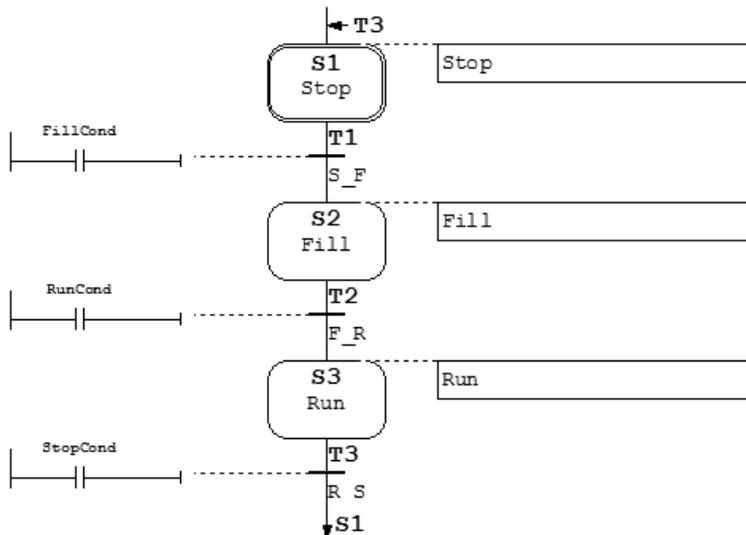


Figure 2.7: Example SFC (Screenshot from SIMATIC tool by Siemens)

Understanding the behavior of these operations is essential to justify the proposed modeling strategy.

In standard PLCs, the cycle time is not fixed, but there is an upper limit surveyed by a *watchdog* module. If the PLC cycle time gets longer than this upper limit, e.g. due to an infinite loop in the PLC program, the PLC executes a special part of the program responsible for handling timing errors. By contrast, safety PLCs have a fixed cycle time in order to avoid dangerous situations.

Timing operations, such as timers defined by IEC 61131 (2013), can be considered as function blocks that delay a signal or produce a pulse.

Different types of timers can be found in PLCs. The most common timers are TON (Timer On-delay), TOFF (Timer Off-delay) and TP (Pulse Timer) timers. Fig. 2.8 shows the three timer diagrams. All three timers have the same input and output variables. Two input variables: *IN* and *PT*. *IN* is a Boolean input signal and *PT* is the delay time. And two output variables: *Q* and *ET*. *Q* is the Boolean output variable and *ET* is the elapsed time.

**TON:** in this timer, the value of  $Q$  will be true after the predefined delay ( $PT$ ) when  $IN$  performs a rising edge, and it will be false if  $IN$  is false. The value of  $ET$  is increased until  $PT$ , starting when a rising edge occurred on  $IN$ .

**TOFF:** in this case, the value of  $Q$  will be true if  $IN$  is true and it will be false after the predefined delay ( $PT$ ) when  $IN$  performs a falling edge. The value of  $ET$  is increased until  $PT$ , starting when a falling edge occurred on  $IN$ .

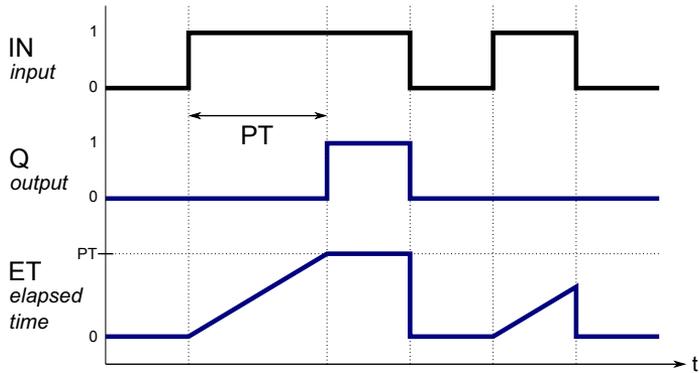
**TP:** in this case, the value of  $Q$  will be true if  $IN$  is true and it will be false after the predefined delay ( $PT$ ), independently of the value of  $IN$ . The value of  $ET$  is increased until  $PT$ , starting when a rising edge occurred on  $IN$ .

PLC timers use a specific data type for timing operations called *TIME*. The IEC 61131 (2013) standard defines this data type as a finite variable and states the following:

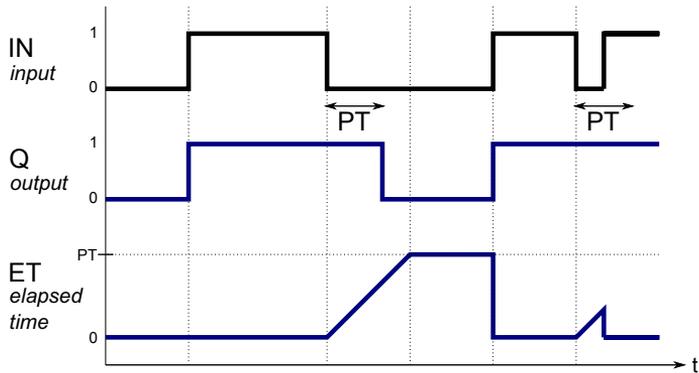
“The range of values and precision of representation in these data types is implementation-dependent.”

Representing time by a finite variable leads to a non-monotonic time representation as the variable can overflow (c.f. the upper part of 2.9). For example, in Siemens S7 PLCs, the *TIME* data type is defined as a signed 32-bit integer with a resolution of 1 ms (see Siemens (1998b)), having an upper limit of approximately +24 days and a lower limit of -24 days. However, in Schneider and Beckhoff PLCs, the *TIME* data type is an unsigned 32-bit integer with a resolution of 1 ms. In this thesis, the signed time interpretation as defined in Siemens PLCs is considered.

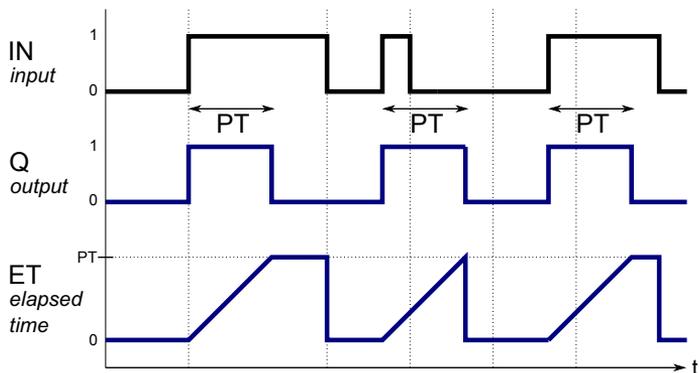
The following section presents some fundamental concepts about formal methods and formal verification.



(a) TON time diagram



(b) TOFF time diagram



(c) TP time diagram

Figure 2.8: Timer diagrams

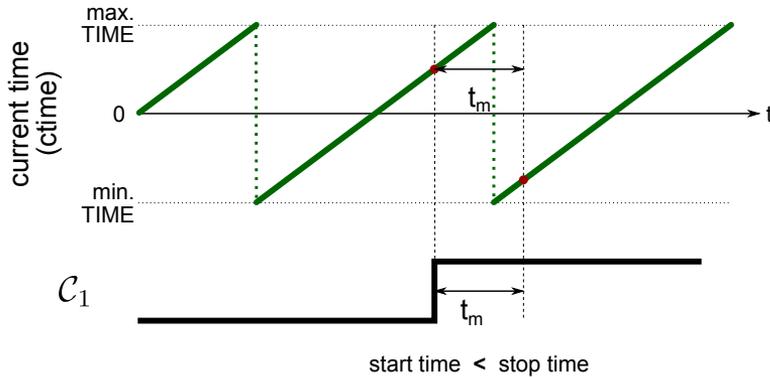


Figure 2.9: Consequences of finite time representation

## 2.3 Formal methods and formal verification

The selected strategy to guarantee that PLC programs are compliant with the specification implies to apply formal verification to these programs. This requires the knowledge and understanding of formal methods and formal verification. This section presents a small review of these two fields of research related to this project.

### 2.3.1 Formal methods

The term “formal methods” has been defined by many authors. According to NASA (1977), formal methods can be defined as follows:

“The term Formal Methods refers to the use of techniques from logic and discrete mathematics in the specification, design and construction of computer systems and software. The word formal derives from formal logic and means pertaining to the structural relationship (i.e., form) between elements. Formal logic refers to methods of reasoning that are valid by virtue of their form and independent of their content.”

In the last decades, formal methods have proved its importance in the design and implementation of complex and critical systems. The main goal of formal methods is to minimize human interpretation in

the different steps of critical system development, by providing languages with well-defined semantics.

“That is, reduce the acceptability of an argument to a calculation that can, in principle, be checked mechanically thereby replacing the inherent subjectivity of the review process with a repeatable exercise.” (from NASA (1977)).

Formal methods can be applied in different levels of the development process of a system. For example, they can be used to produce unambiguous system requirements or to facilitate communication between different steps of the development (i.e. design, implementation and review process). They can be classified by the level of formalization or by the scope of use. Each level of formalization corresponds to a particular scope of formal methods use, as it is represented in Table 2.1 from NASA (1977).

Table 2.1: The Range of Formal Methods Options Summarized in Terms of (a) Levels of Formalization and (b) Scope of Formal Methods Use.

<b>Levels of Formalization</b>	<b>Scope of FM Use</b>
1. Mathematical concepts and notation, informal analysis (if any), no mechanization	Life cycle phases: all/selected
2. Formalized specification languages, some mechanized support	System components: all/selected
3. Formal specification languages, comprehensive environment, including automated proof checker/theorem prover	System functionality: full/selected

Therefore formal methods can be applied in different steps of a system development. Here we enumerate some of the most common ones:

- Specification and modeling: the use of an unambiguous language describing a system.

- Execution/Simulation: formal models are create to simulate them and analyze the behavior of the real system.
- Formal verification: formalized properties are checked against a formal model.
- Refinement: code refinement is the process of proving that two pieces of code behave identically, where one version is more abstract. It can be achieved by using formal methods.
- Test generation: formal models can be used to automatically generate relevant test cases for the real system.

The benefits of using formal methods in the development of critical systems can be summarized in two essential aspects:

1. Discipline: formal methods provide rigor to the design and development processes. A formal proof of correctness can be achieved by using formal methods.
2. Precision: natural languages are ambiguous and open to interpretations. Formal methods provide specification languages with formal semantics that can be used to describe a system or functional requirement with precision.

However, in some industries like industrial automation, formal methods are not widely used. These are the main reasons:

1. Cost: using formal methods is more expensive than traditional alternatives in engineering. The initial cost of using formal methods in the design and development of a system is much higher than other approaches.
2. Limits of computational models: formal models of real-life systems can be too large to be handled by a simulator or model checker.
3. Usability: using formal methods implies a learning process of the formalism and how to use it. This is sometimes a barrier in industry which makes engineers to look for other alternatives.

Traditionally, formal methods are considered highly abstract, extremely rigorous, and very expensive techniques, ignoring some of the theoretical advances that provide solutions for these drawbacks in the last years. However, many success stories can be referenced to prove the applicability and benefits of these techniques in industry.

Under the umbrella of formal methods, a huge range of techniques and languages can be included, for example: Boolean algebra, finite automata, Petri Nets, temporal logic, etc. Some of these concepts are briefly described in the following paragraphs. When using formal methods for verification purposes, a fact has to be taken into consideration: the more expressive the formalism, the less the proof (that a property holds) is amenable to be automatized.

More general information about the formal methods can be found in Clarke and Wing (1996), Virtual Library On Formal Methods (2014), Formal Methods Europe website (2014), NASA Formal Methods Symposium website (2014) and NASA Langley Formal Methods website (2014) as some relevant web-pages.

### **2.3.2 Formal verification**

The term verification refers the act of checking requirements on a system as it is shown in Fig. 2.10. In this figure, the requirement specification represents “what the engineer wants” and the implementation “what the engineer gets”. The transformation from the specification (the idea) into the implementation (the real systems) is done by the design process. The verification process consists in checking that the real system “behaves” as it is described in the specifications. The term verification includes both testing (the specifications are checked on the real system) and formal verification.

Formal verification is the act of checking the correctness of a system with respect to a formal property or specification by using formal methods. Formal verification techniques explore all the possible combinations of the state space to prove that the formal requirement is satisfied.

There are two main families of formal verification techniques:

1. Axiomatic verification.
2. Algorithmic verification.

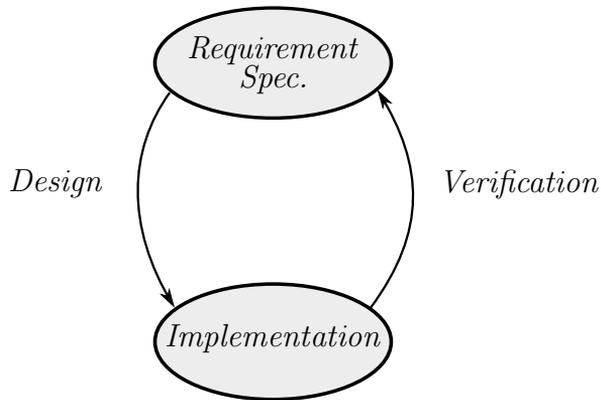


Figure 2.10: Verification schema

**Axiomatic verification:** it consists in a semiautomatic verification technique based on reasoning about the functional correctness of a structured sequential program, by tracing its state changes from an initial condition  $P$  to a final condition  $R$  according to a set of self-evident rules (i.e., axioms).

Theorem provers (tools which apply axiomatic verification) are driven by skilled researchers or engineers, and the full automation of the verification process using these techniques is very challenging and in some cases cannot be achieved.

Formally it can be defined as:  $\{P\}S\{R\}$ , where  $P$  is the precondition,  $R$  the postcondition and  $S$  is the program. The correctness of  $S$  is achieved when  $P$  holds before the execution of  $S$ ,  $S$  terminates and  $R$  will hold afterward.

**Algorithmic verification:** it consists in the use of semi-algorithms to check that a global model, which represents the system, meets the given formal requirements. Techniques like model checking or static analysis are included in this category. The main benefit of these techniques is that they are oriented to be fully automatized, avoiding human interaction. The most well-known drawback of these techniques is the state explosion problem of real-life models, but it can be overcome by using appropriate abstraction techniques.

In industry, algorithmic verification is the most popular formal verification technique because it can be automatized. For that reason,

algorithmic verification is the selected verification technique in this thesis. In particular, model checking has been applied to verify PLC programs. The following sections give an overview of this technique.

### 2.3.3 Model checking

Model checking is an algorithmic verification, which was first described by Edmund Clarke and Allen Emerson in Clarke and Emerson (1982), and also in parallel by Joseph Sifakis in Queille and Sifakis (1982). Due to their contributions in this area of research, they received the *Turing award* in 2007.

Model checking is an automatic verification technique for finite state systems. Given a global model of the system and a formal property (requirement), the model checking algorithm checks exhaustively that the model meets the property. Three steps are required to perform model checking:

1. Requirement formalization.
2. System modeling.
3. Model checking algorithms execution (model checker).

#### 2.3.3.1 Requirements formalization

Requirements engineering is one of the major research challenges nowadays. The goal of producing a complete and unambiguous specification of a system by engineers is still an utopia in many industries. Requirements engineering can be defined as follows (from Laplante (2013)):

“Requirement engineering is the branch of engineering concerned with the real-world goals for, functions of, and constraints on, systems. It is also concerned with the relationship of these factors to precise specifications of system behavior and to their evolution over time and across families of related systems.”

In order to produce precise and unambiguous requirements, the use of formal methods is required. In model checking techniques, temporal logic is the most common formalism to express the requirements or

properties. Temporal Logic, in general, can be defined as follows (from Bérard et al. (2001)):

“Temporal logic is a form of logic specifically tailored for statements and reasoning which involve the notion of order in time.”

With temporal logic, engineers and researchers can specify ongoing behaviors, rather than input/output relations. There are multiple temporal logic formalisms. The two more extended formalisms are: LTL (Linear Temporal Logic) and CTL (Computation Tree Logic). Other temporal logic formalism are: PLTL (Propositional Linear Temporal Logic) or ACTL (subset of CTL only using the path quantifier  $A$ ).

**LTL:** properties can be expressed as an infinite sequence of states where each state has a unique successor. LTL can use the following temporal operators:  $G$  (always),  $F$  (future),  $X$  (next),  $U$  (until). Fig. 2.11 represents the meaning of these LTL operators. The violet shading represents the state space where the atomic proposition  $p$  holds.

**CTL:** properties can be expressed as a combinations of path quantifiers and linear-time operators. The possible path quantifiers are:  $A$  (for every path) and  $E$  (there exists a path). The liner-time operators are:  $Xp$  ( $p$  holds true next time),  $Fp$  ( $p$  holds true sometime in the future),  $Gp$  ( $p$  holds true globally in the future) and  $pUq$  ( $p$  holds true until  $q$  holds true), where  $p$  and  $q$  are atomic propositions. Fig. 2.12 represents the meaning of these CTL properties. The violet shading represents the state space where  $p$  holds.

Some of the properties that can be expressed using CTL and LTL are:

- Reachability Properties: it states that a particular situation can be reached. CTL is more suitable than LTL for expressing reachability properties (i.e.  $EFp$ ).
- Safety Properties: it expresses that, under certain conditions, an specific event should never occurs. Both CTL and LTL can easily express safety properties (i.e.  $AG\neg p$  for CTL and  $G\neg p$  for LTL).

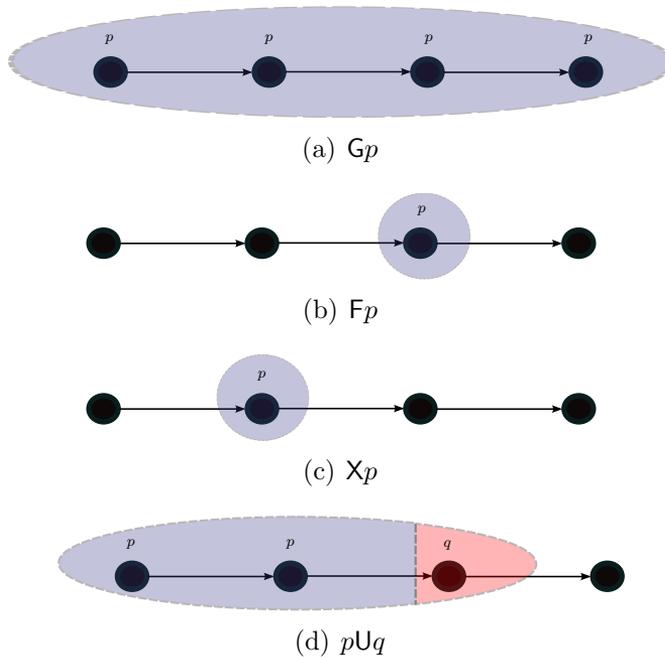


Figure 2.11: LTL operators representation

- Liveness Properties: it states that, under certain conditions, some event will ultimately occur. The  $F$  operator is the most appropriate for this kind of properties ( $AG(p \rightarrow AFq)$  for CTL and  $G(p \rightarrow Fq)$  for LTL).
- Deadlock-freeness: it states that the system can never be in a situation in which it cannot progress. CTL is more suitable than LTL for expressing reachability properties (i.e.  $AGEXtrue$ ).

More information about the applicability of temporal logic can be found in Bérard et al. (2001).

### 2.3.3.2 System Modeling

Once the properties to be verified are identified and formally described, the construction of the formal model is the second step. As the formal model consists of an abstraction of the real system, it is important to express those properties that are relevant for verification.

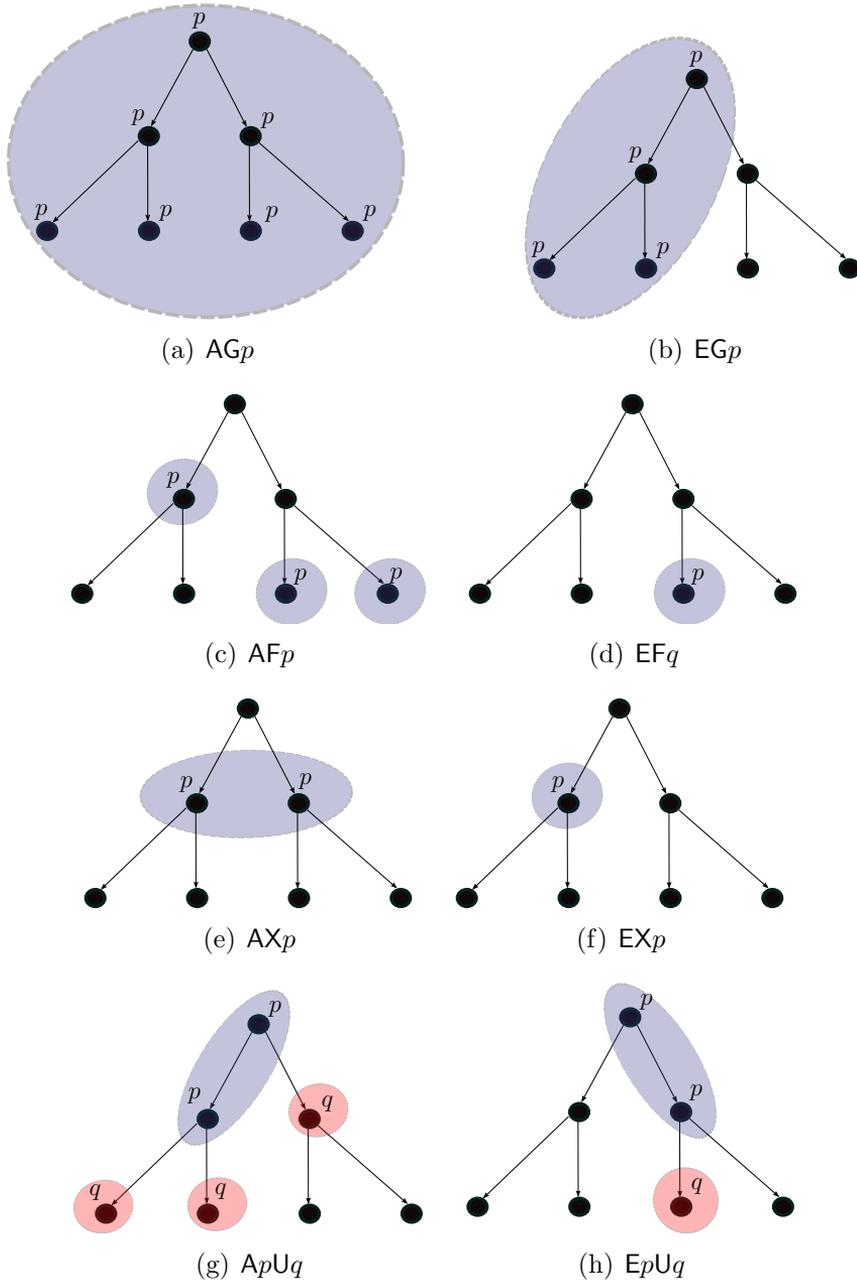


Figure 2.12: CTL operators representation

The first model checking algorithms use an explicit representation of the Kripke structure. Kripke structure consists in a simple abstract machine (a mathematical object) to model a computing machine. It is a graph whose nodes represent the reachable states of the system and whose edges represent state transitions. A labeling function maps each node to a set of properties that hold in the corresponding state. This can be formally defined as follows (from Clarke et al. (1999)):

**Definition 1 (Kripke structure)** *Let  $AP$  be a set of atomic propositions. A Kripke structure is a 4-tuple  $M = (S, S_0, R, L)$ , where  $S$  is a finite set of states,  $S_0 \subseteq S$  is a finite set of initial states,  $R \subseteq S \times S$  is the transition relation that must be total, that is, for every state  $s \in S$  such that  $R(s, s')$  and  $L : S \rightarrow 2^{AP}$  is a function that labels each state with the set of atomic propositions true in that state.*

Nowadays in practice, there are many different formalisms used by model checkers, for example: automata, timed automata, Petri net, etc. More details about different modeling formalism used by model checkers can be found in Bérard et al. (2001).

### 2.3.3.3 Formal definition of model checking

Once the requirements are formalized and the formal model of the system is built, the model checking algorithm can be applied. A general model checking definition can be expressed as follows (from Clarke (2008)):

**Definition 2 (Model checking problem)** *Let  $M$  be a Kripke structure (i. e., state-transition graph). Let  $f$  be a formula of temporal logic (i.e., the specification or property to be checked). Find all states  $s$  of  $M$  such that  $M, s \models f$ .*

This is a so-called global model checking approach.

### 2.3.3.4 Model checking approaches

Formal models representing real-life systems usually have a huge state space. The state space exploration performed by model checking algorithms may find the limits. For that reason, researchers have been working in improving the model checking algorithms and providing

some alternatives, in the last decades. The first proposed algorithms for model checking were called *explicit model checking*, as they store and handle the states individually. In the following paragraphs, two of the most popular model checking approaches are briefly introduced: *symbolic model checking* and *bounded model checking*.

**Symbolic model checking:** Instead of enumerating reachable states one at a time, in 1987 Ken McMillan implemented a version of the CTL model checking algorithm using a symbolic representation of the state space based on BDDs (Binary Decision Diagrams). More information about symbolic model checking can be found in McMillan (1993).

BDDs are data structures where the state space can be encoded. This symbolic representation allowed to verify much larger systems (formal model of  $10^{20}$  states and beyond, see Bruch et al. (1992)).

After the first definition of symbolic model checking, many new algorithms have been proposed. Some of them are based on decision diagrams, for example using MDD (Multivalued Decision Diagrams). In addition, other kind of symbolic model checking is based in SMT (Satisfiability Modulo Theories) solvers, consisting in SAT-based algorithms.

**Bounded model checking:** Traditional model checking algorithms consists in the exploration of the full state space. Sometimes this is not needed, or not even possible due to the huge size of some models. Bounded model checking allows to check properties on a part of the state space.

This technique is appropriate for reachability properties (i.e.  $EFp$  in CTL). If the evaluation of the property or formula  $p$  is false, no answer can be provided, the algorithm will explore a bigger part of the state space until the evaluation is true or the full state space is explored. It is also good for safety (invariant) properties. If a counterexample is found in a part of the state space is a valid counterexample for the whole model (and the model checking can be finished). Some of the advances in this technique can be found in Biere et al. (2003) and Vörös et al. (2011).

Many other techniques meant to optimize the verification process

can be mentioned here. For example, statistical model checking (Legay et al. (2010)), compositional verification (Besalem et al. (2010)), etc. They all propose alternatives and improvements to deal with the state space explosion problem.

In addition to the different model checking algorithms, reduction and abstraction techniques can be applied to the formal models to face the problem of the state space explosion (e.g. cone of influence or predicate abstraction). A review of these techniques for model checking can be found in Radek (2006).

### **2.3.3.5 Advantages and disadvantages of model checking**

Model checking techniques can be found in many different industries, however still some myths about formal methods in general provoke that engineers look for other alternatives (Hall (1990) and Bowen and Hinchey (1995)).

Summarizing the main advantages of applying model checking techniques are:

- It is an automatic verification method.
- Model checking explores all the possible combinations of the state space model to guarantee that a property holds.
- When a property does not hold on the model, a counterexample is produced by the model checker, which contains relevant information to identify the source of the problem.
- Comparing with testing techniques, model checking can check safety or liveness properties, e.g. ensuring that a forbidden output value combination should never occur. This is one of the main limitations of testing.

The main disadvantages of formal verification techniques are:

1. State space explosion.
2. Complexity of building formal models.
3. Difficulty of using property specification formalism like temporal logic.

### 2.3.4 Verification tools

All the concepts and algorithms mentioned before are implemented in verification tools, which allow engineers to automatize the verification process or guide them at least. Some of the most popular verification tools are:

- NuSMV: it performs symbolic model checking for CTL and LTL formulae on networks of automata extended with variables. The new version of NuSMV (2014) is called nuXmv. More information in NuSMV website (2014) and Cavada et al. (2014).
- UPPAAL: it supports timed automata based models and a subset of CTL for property specification. Internally models are represented as CDDs (Clock Difference Diagrams). More information in UPPAAL website (2014) and Amnell et al. (2001).
- BIP: BIP is a component-based framework for rigorous system design aiming at correctness-by-construction for essential properties of the designed system. A BIP model consists of three layers: Behavior, Interaction and Priority. The BIP framework provides C code generation from BIP models and its verification tool, called DFinder, is specialized in deadlock detection. More information in BIP website (2014) and Basu et al. (2011).
- SPIN: it performs symbolic model checking on LTL formulas. The modeling language is called PROMELA and it supports embedded C code as part of model specification. More information in SPIN website (2014).
- KRONOS: it performs symbolic model checking on TCTL (extension of the temporal logic CTL that allows quantitative temporal reasoning over dense time). Its modeling language is based on timed automaton. More information in KRONOS website (2014).
- COQ: it is a theorem prover which implements a program specification and mathematical higher-level language called “Gallina”. More information in Coq website (2014).

- PVS: it is a mechanized environment for formal specification and verification. It includes an interactive theorem prover and a symbolic model checker. More information in PVS website (2014).

## 2.4 Related work

This section presents the related work to formal verification of PLC control systems. As it was mentioned in Chapter 1, the goal of this thesis is to provide a general solution to guarantee that PLC programs are compliant with the specifications, independently of the purpose of the PLC-based control systems.

In addition to safety standards like IEC 61508 (2010) for Safety Instrumented Systems, where guaranteeing the correctness of the control code is a critical task, the automation industry has adopted some solutions trying to minimize the number of errors in PLC programs.

Siemens is one of the most important companies in the development of Safety Instrumented Systems based on PLCs. This company provides the *Safety Matrix* and the *Distributed Safety* products, which are software packages integrated in the SIMATIC and TIA portal programming environments for PLCs.

There is an assumption that states that the logic of a Safety Instrumented System program is much simpler than a standard PLC program. The Safety matrix package provides a tool and a methodology that reduces the configuration, testing and maintenance time by merging the traditionally separate steps of creating a cause and effect matrix diagram, and configuring the safety system. It provides a cause and effect matrix diagram where the PLC programmer can describe the safety functions and the PLC code is automatically generated from this matrix. Only very simple Safety PLC programs can be created. The “philosophy” here is to reduce the number of potential error in the code by restricting the PLC program variability to the maximum.

A similar approach is provided by the Distributed safety package, where a library of software blocks (usually FBs), approved by the *TÜV* (the certification provider TÜV Rheinland website (2014)), which can be used by the programmer to build the safety PLC programs. The goal is to reduce the amount of code produced directly by the PLC

programmer and allow him to reuse these objects.

In addition to these packages, the Siemens programming environment of SIMATIC and TIA portal for safety programs is restricted compared with standard PLC programs which follow the IEC 61131 (2013) standard. For example, only Ladder and FBD languages are allowed, the data types are also restricted (e.g. the data type `REAL` is not allowed), etc.

All these restrictions in the development of PLC programs from the PLC vendors do not exclude the need of applying testing or other mechanism to check the code against the specifications.

The use of frameworks like UNICOS and the standardization or code generation of safety critical software is essential to reduce the number of potential bugs in the PLC (e.g. Coupat et al. (2014)).

In addition to that, many other tools and methodologies can be applied in the design of safety systems. These tools are usually not specific for PLCs (e.g. LDRA, Exidia, ISOGRAPH, RAM commander, etc.). These tools contribute indirectly to guarantee the safety in industrial control systems.

The two main “families” or techniques to check control software against the specifications are *testing* and *formal verification*. This section presents the related work divided into these two groups. Then, more details about algorithmic formal verification related work is presented as this is the solution adopted in this thesis.

### 2.4.1 Testing-based techniques

Testing is a verification technique that implies the check of certain properties or test cases in the real system. In the automation industry, manual and automated testing are the most popular verification techniques. Following the industrial automation standard: ISA 62381 (2010), process control applications testing is done in two main stages: Factory Acceptance Test (FAT) and Site Acceptance Test (SAT). FAT is conducted to check if the requirements of a specification are met. SAT checks if the system meets the requirements and these tests are performed in the real plant.

In academia, some authors propose solutions for checking PLC programs based on testing techniques. For example, in Kramer (2001) a testing approach based on simulation is proposed. In Jee et al.

(2005), the authors propose a control and data flow testing coverage criteria to the flow graph in order to generate test cases. The same authors in Jee et al. (2010) provide a test coverage criteria for FBD programs.

Testing activities are very useful and needed techniques in the verification of industrial control systems. However they have some limitations, testing is unable to check effectively safety and liveness properties. This is main reason for us, to adopt a formal verification approach in the verification of PLC programs.

It is important to remark that, testing and formal verification techniques complement each other and for specific parts of a system testing can be more effective than formal verification.

Our paper Fernández Adiego et al. (2013a), briefly describes some of the applied testing techniques for PLC programs at CERN and how they can complement the formal verification approach presented in this thesis.

## 2.4.2 Formal verification based techniques

The following sections present the related work to this project. Before focusing on related work to formal verification of PLC programs, a brief overview of formal methods applied in different industries is presented here. The goal is to show how extended are formal methods in industries with critical systems.

### 2.4.2.1 Formal verification in industry

The use of formal methods and formal verification in different industries with critical systems began more than 20 years ago and has many success stories. Nowadays more and more industries integrate formal methods in the design, development and verification of their systems. When talking about the formal verification technique employed, the big majority applied algorithmic verification techniques (e.g. model checking or static analysis).

In some industries with safety critical systems, the use of formal methods has become a common practice. Some of these industries are: aircraft, space, robotic systems, transportation, etc. Here some examples are presented:

- Verification and validation of a NASA flight software: static analysis, runtime analysis and model checking were compared to traditional testing with respect to their ability to find seeded errors in a prototype of the Mars Rover controller (Brat et al. (2004)).
- Air traffic control systems: formal methods were applied to the CCF (Central Control Facility) display information system at the London Air Traffic Control Centre (Hall and Isaac (1992)).
- Train and subway industries: in the subway of Paris, the B method was employed in the specification of new subway line (Behm et al. (1999)). Model checking was applied to the European train control system (Faber and Meyer (2006)). SAT-based model checking has been applied to an interlock railway system (James and Roggenbach (2010)).
- There are many examples in the automation industry but mainly restricted to the robotic field. One example can be found in Abdellatif et al. (2012).
- In nuclear plants, some examples of using formal methods are: Fukumoto et al. (1998), IAEA (1999), Yoo et al. (2005)
- Finally, one of the most advanced industries in the use of formal verification and formal methods is the aircraft industry. An example of the verification of the flight control system can be found in Meenakshi et al. (2007).

All these examples and many more prove the applicability of formal methods in real-life systems. Although different industries have different challenges to solve, these examples inspired this research project of providing a solution for the industrial automation systems.

The application of formal methods for PLCs has been studied in previous works for many years. In Frey and Litz (2000), one of the first studies about the applicability of formal methods and formal verification strategies to PLC programs is presented. The rest of the Section is divided into two parts, corresponding with the two groups of formal verification strategies: axiomatic formal verification and algorithmic formal verification applied to PLC programs.

### 2.4.2.2 Axiomatic formal verification applied to PLC programs

Not many studies or developments are found in the literature when looking for axiomatic verification applied to PLC programs. The reason, as it was mentioned earlier in this chapter (Section 2.3), is that axiomatic verification is difficult to automatize and its success highly depends on the engineer skills.

Some examples of axiomatic verification applied to PLC programs are presented in the following paragraphs.

In Blech and Ould Biha (2011), a proposal of formal semantics of two of the PLC languages (IL and SFC) is presented for verification purposes using the Coq theorem prover.

The paper Xiaoa et al. (2012) presents a modeling strategy of data types, statements and the denotational semantics of PLC program with the “Gallina” language. In addition, the Coq theorem prover is used to prove the correctness of the programs. A similar approach can be found in Chen et al. (2010).

In Mader et al. (2001), the paper reports on the systematic design and validation of a PLC control program for the batch plant. The formal proof of correctness was obtained using the PVS theorem prover and the SPIN model checker. Both strategies are presented and the limitations are discussed.

In Völker and Krämer (1999), the authors suggest to complement testing techniques with compositional theorem proving for FB-based industrial control systems.

In Sadolewski (2011), the Coq theorem prover is applied to PLC programs. The property specification expressiveness is reduced to simple safety assertions.

These approaches cannot be applied to our systems and needs, as one of the main requirements of this project was to hide any complexity related to formal methods to the control engineers. In addition, none of these approaches deals with PLC programs with a significant size, therefore they are not applicable to CERN control systems.

### 2.4.2.3 Algorithmic formal verification applied to PLC based programs

Model checking is the most popular algorithmic formal verification technique suggested by the literature for PLC programs. However static analysis solutions for PLC programs have been also proposed by some authors. Static analysis and model checking techniques can complement each other.

For example, the Arcade.PLC tool (ARCADE.PLC website (2014); Biallas et al. (2014); Stattelmann et al. (2014)) from the RWTH Aachen University and the PLC Checker tools (PLC Checker website (2014)) from the Itris Automation Square company provide static analysis solutions for PLC programs.

The following paragraphs present an exhaustive analysis of related publications where model checking was the adopted verification technique for PLC programs. The analysis presents the main differences with the adopted approach of this thesis, regarding the following aspects: the PLC languages, the requirement specification, the modeling and verification constrains, the abstraction and reduction techniques and the modeling strategy of the timing aspects in PLC programs.

Many studies and proposals can be found in the literature of applying model checking to PLCs programs: Bartha et al. (2012); Bauer et al. (2004); Biallas et al. (2010); Blech et al. (2011); Campos et al. (2008); Canet et al. (2000a); Flake et al. (2004); Gourcuff et al. (2008); Huuch (2003); Lange et al. (2013); Mader and Wupper (1999); Mokadem et al. (2010); Pavlović and Ehrich (2010); Perin and Faure (2013); Sarmiento et al. (2008); Smet et al. (2000); Soliman and Frey (2011); Soliman et al. (2012); Sülflow and Drechsler (2008); Yoo et al. (2008).

However, none of the described methods can be applied directly to the control systems developed at CERN, for the reasons discussed hereafter.

**Regarding the PLC languages:** in the literature, most of the the modeling strategies target a single PLC language.

Formal verification has been mainly applied to IL (e.g. Canet et al. (2000a); Mader and Wupper (1999)) and SFC (e.g. Bauer et al. (2004); Sarmiento et al. (2008)).

A few works target the verification of FBD programs (e.g. Bartha et al. (2012)) and Ladder (e.g. Bender et al. (2008))

Some work targets specifically the verification of ST programs (e.g. Gourcuff et al. (2006, 2008)).

Finally a very few approaches can handle multiple PLC languages (e.g. Biallas et al. (2010); Gourcuff et al. (2008); Sadolewski (2011)).

The approach proposed in this document differs from previous works as the modeling strategy has been designed to provide a general methodology where any PLC language can be included. For that, an *automata* based formalism has been adopted as an intermediate step between the PLC code and modeling languages of the verification tools. This formalism is appropriate to model all the features of any PLC language.

In addition, the automatic translation from the PLC languages into the modeling language of the verification tools is split into two parts, making the methodology more flexible as adding a new language does not imply a completely new modeling approach.

**Regarding the requirement specification:** very few authors targeted the problem of requirement specification in the application of formal verification of PLC programs, even if this issue is one of the main obstacles of these approaches to be deployed in industry.

Authors like, Campos et al. (2008) and Campos and Machado (2013) propose an approach to hide the complexity of using complex specification formalism like temporal logic. This approach is based on patterns that can be easily used by the engineers and they are automatically translated to temporal logic formulas.

The solution provided in this thesis is also based on patterns. A set of patterns with a well-defined semantics is provided to engineers. Once the requirement is created with these patterns, the corresponding temporal logic formula is generated automatically. The main difference with the previous approaches is the set of patterns. Previous works did not cover all the specification needs we had for our systems. Our patterns are based on the experience of the control engineers at CERN.

These approaches based on patterns are useful to find bugs in the PLC programs, although it does not guarantee to provide a complete and coherent formal specification.

Other authors propose solutions based on UML techniques (e.g. Flake et al. (2004); Remenska et al. (2014)) to help engineers to build the temporal logic formulas. Although there are not specific for PLC control systems, the ideas can be adapted to this field.

An overview of the different approaches in requirements engineering can be found in Laplante (2013).

**Regarding the modeling and verification constraints:** In certain cases, the modeling and verification strategies are applicable to a reduced set of PLC programs due to some restrictions.

Some authors apply formal verification, but only for small examples, without discussing the reduction of the models that is unavoidable for verifying industrial-sized programs. For example, Bartha et al. (2012); Canet et al. (2000a); Mokadem et al. (2010); Pavlović and Ehrich (2010); Perin and Faure (2013); Sarmiento et al. (2008); Soliman and Frey (2011); Soliman et al. (2012); Sülflow and Drechsler (2008). Only a few authors deal with this important issue (e.g. Biallas et al. (2010); Gourcuff et al. (2008); Lange et al. (2013)).

Some of the studies only target the modeling of PLCs without providing a verification solution. For example, Blech et al. (2011); Mader and Wupper (1999).

Many papers do not address the automatic generation of the model from the PLC program, or just explain the high-level principles. For example, Bartha et al. (2012); Bauer et al. (2004); Biallas et al. (2010); Canet et al. (2000a); Mokadem et al. (2010); Pavlović and Ehrich (2010); Perin and Faure (2013); Sarmiento et al. (2008); Smet et al. (2000); Soliman and Frey (2011); Yoo et al. (2008).

In some cases, the modeling strategies restrict the expressiveness of the property specifications. In Sadolewski (2011), the proposed approach restricts the requirements to assertions that provide smaller expressiveness than LTL or CTL (in this work, axiomatic formal verification was applied). In other cases, the abstraction techniques imply constraints in the expressiveness of properties (e.g. Biallas et al. (2010); Lange et al. (2013))

In addition, most of the approaches are attached to a specific verification tool. In the literature, the most popular verification tools for PLC programs are NuSMV and UPPAAL.

Table 2.2 summarizes some of the constraints of the previous work.

Table 2.2: Related work

<b>Reference</b>	<b>Input lang.</b>	<b>Verifier</b>	<b>Req. language</b>
Bauer et al. (2004)	SFC	Cadence SMV	CTL
Sarmiento et al. (2008)	SFC	UPPAAL	CTL subset
Bauer et al. (2004)	timed SFC	UPPAAL	CTL subset
Blech et al. (2011)	SFC, FBD	BIP	—
Bartha et al. (2012)	FBD	PetriDotNet	CTL
Canet et al. (2000b)	IL	Cadence SMV	LTL
Mader and Wupper (1999) <sup>1</sup>	IL	UPPAAL	CTL subset
Gourcuff et al. (2006) <sup>2</sup>	ST	NuSMV	—

<sup>1</sup> Only Boolean variables are permitted.

<sup>2</sup> The following limitations apply: only Boolean variables, no iteration statements. The method can be applied for LD and IL too, but it is not presented in Gourcuff et al. (2006).

The main differences of this thesis with previous works, are the following ones:

- We propose a fully automated methodology from the property specification to the counterexample analysis.
- We do not target a single verification tool, which allow us to select the appropriate tool for different verification cases.
- We verify “complex” properties expressed with temporal logic and we provide two kind of reduction technique for our models: property preserving reduction techniques and the variable abstraction technique. The first group does not bring any constraint to the expressiveness of the property specification but the second group is restricted to safety properties. The second group is only applied when the property preserving techniques are not powerful enough for a specific property and PLC program.

**Regarding the abstraction and reduction techniques:** As it was mentioned before, only a few authors targeted the issue of reduction techniques for PLC program verification.

In Gourcuff et al. (2008), the authors address the problem of state explosion of formal models derived from IL (Instruction List) PLC programs by using an algorithm based on data and interpretation abstraction. Their algorithm has some limitations, e.g. only Boolean variables can be used.

The paper Biallas et al. (2010) applies CEGAR (counterexample-guided abstraction refinement, proposed in Clarke et al. (2000)) to models of PLC programs written in IL, but they do not present large case studies that would show the scalability of the methods. The limitations of the approach are the same as for other CEGAR approaches: it can handle only ACTL properties.

In Biallas et al. (2013), the authors apply predicate abstraction to PLC programs. As well as the previous paper, the author focus on safety properties.

In Lange et al. (2013), bounded model checking is applied to PLC programs. The authors introduce powerful reduction methods applied to IL code. Reduction techniques such as constant folding, slicing, and forward expression propagation are employed to optimize the models

for bounded model checking. This approach restricts the requirement language to simple safety properties.

Table 2.3 summarizes the related work of abstraction and reduction techniques applied to formal models from PLC programs.

Table 2.3: Related work: Abstraction techniques

Reference	Abstraction techniques	PLC language	Constraints
Gourcuff et al. (2008)	data and interpretation abstraction	IL	only Boolean var.
Lange et al. (2013)	constant folding, slicing, and forward expression propagation	IL	only Safety properties
Biallas et al. (2010)	CEGAR	IL	only Safety properties
Biallas et al. (2013)	Predicate abstraction	ST	only Safety properties

These papers provide very interesting ideas and contributions to improve the performance of PLC programs verification. Our approach differs with these works in the following aspects:

- Our reductions are independent of the verification tools as they are applied in a higher level of our methodology, the IM.
- In addition, as mentioned before, we provide two kind of reduction techniques for our models: property preserving reduction techniques and the variable abstraction technique. The first group does not bring any constraint to the expressiveness of the property specification but the second group is restricted to safety properties.

**Regarding the modeling strategy for the timing aspects of PLC programs:** finally, most of the described works do not model

time in PLCs. Only a few works have been found targeting time and timers in PLCs, i.e. Mader and Wupper (1999); Mokadem et al. (2010); Perin and Faure (2013); Wang et al. (2013).

Indeed, Mader and Wupper (1999) or Perin and Faure (2013) propose an approach for modeling PLC timers using timed automaton models, but they do not present verification results. As time is considered as a linear and monotonic function, the generated models would have a huge state space, making verification not feasible if this approach would be applied to large systems, as the systems developed at CERN.

Similarly, Mokadem et al. (2010) presents a case study where a global model for a timed multitask PLC program is created for verification purposes. This approach is similar to the one proposed by Mader and Wupper (1999) but verification is performed with UPPAAL using clocks and therefore with monotonic time representation.

In Wang et al. (2013), several aspects of PLC control systems including timers are modeled, using the component-based BIP framework. In this case, they assume fixed PLC cycle length which is a big constraint, and timer models are not precise enough compare to real PLC timers. In addition, verification results are not presented.

Table 2.4 summarizes the related work of modeling strategies for the timing aspects of PLC programs.

This thesis proposes two different approaches to model time and timers in PLCs:

- The first one includes a realistic approach in which time is modeled as a finite variable as it is implemented in PLCs.
- The alternative is a very abstract model of timer where time is not modeled in order to deal with the state space explosion problem.

All the details are presented in Section 3.9.

## 2.5 Summary of the chapter

This chapter presented the theoretical background and the related work needed to understand the contributions of this thesis. From the automation community, the main concepts of PLC-based control

Table 2.4: Related work: Time modeling strategy

<b>Reference</b>	<b>Formalism</b>	<b>Ver. tool</b>	<b>Time model</b>
Mader and Wupper (1999)	Timed automaton	–	monotonic representation
Perin and Faure (2013)	Timed automaton	–	monotonic representation
Mokadem et al. (2010)	Timed automaton	UPPAAL	monotonic representation
Wang et al. (2013)	BIP language	–	monotonic representation

systems are described, including classification, standards and basic concepts about the hardware and software of these systems. From the formal methods community, some definitions of formal methods and formal verification are included. The focus of this section is on the model checking technique. Finally, the related work of this thesis is presented, including a discussion of the most relevant contributions on this topic of research. The main differences of previous works with this thesis are also described.



# Chapter 3

## Approach

### 3.1 Introduction

This chapter presents the proposed approach and main contribution of this thesis. It consists in a methodology that allows to apply automated formal verification to PLC programs. The two main goals of this research are:

1. Finding bugs in the PLC code, increasing the quality of the software.
2. Hiding the complexity of applying formal methods, from control engineers and designers.

Nowadays, development of PLC programs in industry is a very traditional procedure in most of the cases as it is represented in Fig. 3.1. An informal specification is the starting point, where the control engineer and the process engineer describe the required logic to control the process in a natural language. Then the PLC programmer (often the same person as the control engineer) implements the PLC code according to the specifications. In the best case, the PLC programmer uses some standards in the development process (e.g. IEC 61499 (2013)) or frameworks (e.g. UNICOS framework from Blanco Viñuela et al. (2011) for CERN control systems). Finally manual testing is performed on the PLC before the commissioning of the system. For safety systems the procedure is more strict, as all the steps defined in the so-called “Safety Life Cycle” have to be performed. The “Safety Life Cycle” is defined in IEC 61508 (2010).

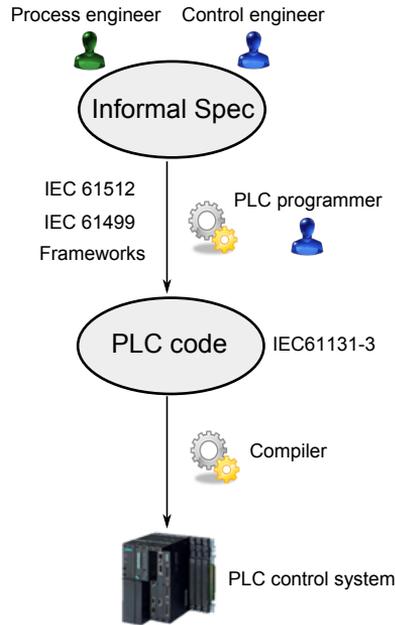


Figure 3.1: Traditional PLC program development

The methodology proposed here is meant to be integrated in this traditional development process or in any variant of this schema. It has the goal of not modifying the PLC program development process of any company or developer. Therefore existing PLC programs can also be verified.

This methodology creates automatically formal models out of PLC programs and is meant to be general and able to transform the languages defined in IEC 61131 (2013) standard into different modeling languages used by formal verification tools. This transformation is based on an Intermediate Model (IM), which is the central point of the methodology.

The content of this chapter is divided according to the different steps of the methodology.

- Section 3.2 presents a general overview of the approach.
- Section 3.3 describes the syntax and semantics of the IM.
- Section 3.4 presents the procedure for formalizing the informal

requirements from developers and designers.

- Section 3.5 presents the “execution platform” knowledge included in the formal models, extracted from the PLC hardware.
- Section 3.6 presents the transformation rules from the PLC programs into the IM. Currently the methodology includes ST and SFC languages
- Section 3.7 presents the reduction techniques on the formal models that make possible formal verification.
- Section 3.8 presents the transformation rules from the IM into the modeling languages used by the formal verification tools. Currently the methodology includes nuXmv, UPPAAL and BIP modeling languages.

Two extra sections have been created to describe in detail two particular subjects in the modeling strategy:

- Section 3.9 presents the transformation rules related to time and timers. Due to the complexity of these transformation rules, they are presented as an independent section.
- Section 3.10 presents the adopted solution for including process information in the formal models.

The chapter concludes with the last step of the methodology and a brief overview of the implementation of the methodology in a CASE (Computer-Aided Software Engineering) tool.

- Section 3.11 presents the final step of the methodology, how the verification procedure is applied and how the information extracted from the verification tools is used to help the PLC programmers to find the source of the problem.
- Section 3.12 describes describes the basis of the CASE tool, which implements the methodology.

## 3.2 General overview of the approach

The methodology presented in this chapter allows control engineers and designers to apply formal verification to PLC programs without any expertise in the field of formal methods or formal verification. The methodology is designed to be general so any PLC language can be included in it. Moreover, its design allows to extend the methodology with several formal verification tools. To achieve that, it relies on an *intermediate model* (IM) which is the central piece of the methodology. This IM is an automata-based formalism extended with data (the formal definition of the IM syntax and semantics are defined in Section 3.3). PLC code is automatically translated to the IM, including the PLC execution platform knowledge. In a second step, the IM is translated to several input models for different verification tools. It also provides a solution for formalization of the requirement specification. The methodology is represented in Fig. 3.2.

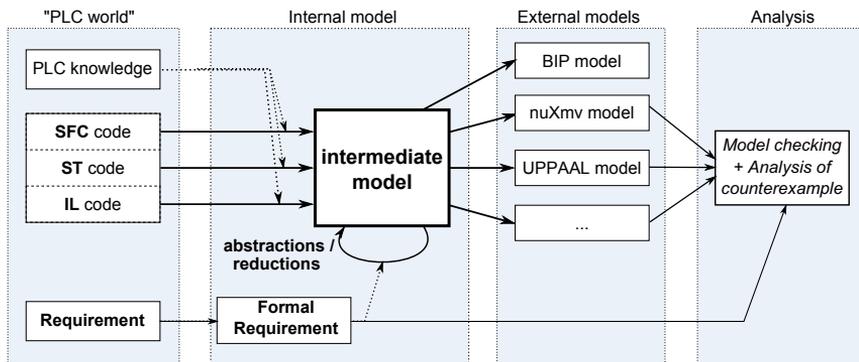


Figure 3.2: Methodology overview

The figure is divided in four blocks, suggesting the reader the different steps of the methodology.

- The first block is called “PLC world”. It shows the source of information needed to build the formal models and the formal requirements. In this block, the PLC programs, the informal requirements and the execution platform knowledge are included.

- The second block is called “Internal model”. It is composed of the IM, the formalized requirements to be verified and the reduction techniques to simplify the model.
- The third block is called “External models”. In this block, the IM is translated into the input modeling language of the selected verification tools.
- The fourth and final block is called “Analysis”. It is where the verification is performed and the verification results are analyzed and presented to the PLC developers in order to find the source of the problem if any.

From the user point of view, i.e. the control system developer and designer, they only need to feed the methodology with the requirements to be fulfilled and the PLC programs. As a result, an analysis report is obtained, extracting the information from the verification results given by the verification tools. This report tells the user if the models meet the requirements, and if not, it provides useful information to find the source of the problem. All the intermediate transformations are hidden and fully automatized by the CASE tool.

The internal steps of the methodology are the following:

1. *Formalization of the requirements*: usually the requirement specifications are expressed in a non-formal language by the control system developers and designers. Temporal logic formalisms are the most popular formalism used by verification tools for property specification. However, they are complex and control systems developers and designers are not familiar with them. Section 3.4 describes this first step, hiding the complexity of using temporal logic for the requirement specification.
2. *PLC code – IM transformation*: This step consists in the transformation of the PLC code into the IM. The PLC code is parsed, building an Abstract Syntax Tree (AST), which represents the abstract syntax of the PLC code. This AST is then transformed to a Control Flow Graph (CFG) which represents the semantics of the code as an automata-based IM. The transformation process uses the abstract model of the PLC hardware to be able to represent the *PLC scan cycle*. The transformation rules from

the PLC code into the IM are presented in Section 3.6. The execution platform information used to build the abstract model of the PLC hardware is presented in Section 3.5.

3. *Reduction and abstraction techniques applied to the IM:* when modeling real-life PLC programs, the state space of the generated model is huge and in most cases formal verification cannot be applied. Section 3.7 presents the reduction techniques proposed in this methodology to make formal verification possible.
4. *Transformation IM– modeling languages of the verification tools:* once the IM is reduced, it is translated into the input modeling language of specific verification tools. Section 3.8 shows the IM translation into the selected verification tools. Currently NuSMV/nuXmv (from this point referenced as nuXmv), UPPAAL and BIP are included.
5. *Verification and analysis of the results:* when the verification tool finds discrepancies between the formal model and the formal specification, a counterexample is produced. This information is analyzed, reduced and presented in a report to the developers and designers, providing useful information in order to find the source of the problem. Section 3.11 presents this analysis of the information provided by the verification tools.

A CASE tool based on EMF (Eclipse Modeling Framework) that supports this methodology has been developed. It provides a graphical interface for the PLC program developer.

This brief description of the methodology may have risen some obvious questions to the reader. This section is dedicated to answer them in order to introduce and justify our decisions.

**Why model checking?** Among different formal verification techniques, model checking was the most appropriate for our purposes. The main reasons are: it is possible to automatize and it is possible to hide the complexity from PLC developers as it was mentioned before. About the specific model checking technique, the methodology is meant to be general and open to any tool with different model checking techniques. For example, symbolic model checking is applied as

nuXmv is integrated and by using BIP framework, compositional verification (Bensalem et al. (2011)) and statistical model checking (Basu et al. (2012)) can be applied.

**Why the models are created out of the PLC code?** When the methodology design started, two possibilities were considered: (1) The first one was to build the models from a high-level specification, then verify these models and finally generate the PLC code (See Fig. 3.3(a)). This is the so-called “correctness by construction” approach, and many authors and even companies promote this approach (e.g the SCADE tool-set from Esterel technologies<sup>1</sup> or the BIP framework from the Verimag laboratory<sup>2</sup>). This approach has multiple and obvious benefits in the design of safety critical systems. For example, at the specification level, many constraints can be introduced and the variability is reduced, therefore the state space of the models is smaller. It is actually the ideal approach for any software design. But it also implies to impose the developers and designers to use a specific formalism for specification, however, there is not a standard nor a single solution adopted so far by the automation community. This approach would imply to change the development process of PLC programs and verification of existing PLC control systems would not be possible. (2) The second possibility considered, which was the selected approach, was to build the models out of the PLC code (See Fig. 3.3(b)). This approach has two main drawbacks: the complexity of the transformation rules from the PLC code into the formal models and the state space size of the generated models, as the PLC developer can write PLC programs without any constraint (just the ones imposed by the PLC language). However, we have strong arguments for using the second approach. As it was mentioned before, nowadays there is no unified formalism to specify industrial control systems, including of course PLC control systems. Requirements for software engineering is a topic addressed by some authors (e.g. Laplante (2013)), even some general standards give some guidelines about software specification (e.g. IEEE 830 (1998)) but in industry, almost every company that develops control systems has a different way to specify. However, the PLC languages are unified by the IEC 61131 (2013) standard, there are only small differences in

---

<sup>1</sup><http://www.esterel-technologies.com/products/scade-suite/>

<sup>2</sup><http://www-verimag.imag.fr/Rigorous-Design-of-Component-Based.html>

the languages implemented by the different PLC vendors, and every PLC developer uses them. In addition, a requirement for this project was to provide a solution for verifying and finding bugs in existing systems. These systems have been developed using completely different specification techniques. The methodology is meant to be general and it can be used by any PLC developer. Similar approaches were adopted also for many different authors (See Section 2.4).

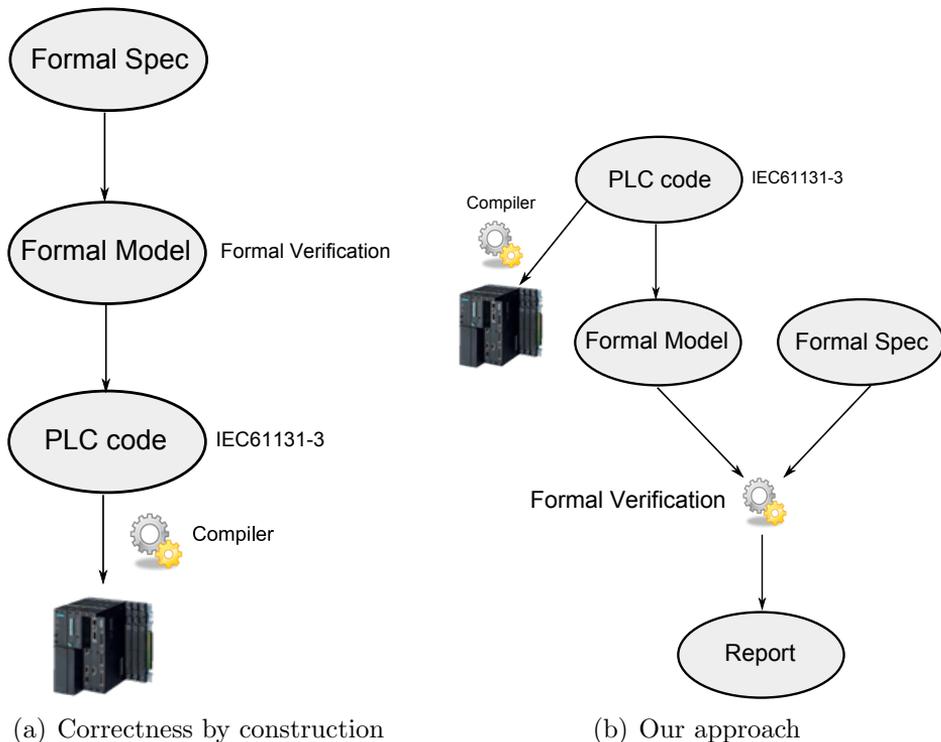


Figure 3.3: Two main approaches for the verification and design of PLC programs.

In addition, as it was mentioned before, this approach is meant to be integrated in the existing, traditional process of PLC programs development. Fig. 3.4 shows the proposed methodology integrated in the PLC program development process. In the left part of the figure, the “traditional” PLC program development is shown. The PLC code is produced from an informal specification. In the best cases, the developer uses some standards (e.g. IEC 61499 (2013)) or frameworks (e.g. UNICOS framework Blanco Viñuela et al. (2011) for CERN control

systems) and testing is performed before or during the commissioning phase of the project. The new methodology creates formal models out of the PLC code and the formal specification is created by using patterns. A model checker is used to check the requirements on the model and when a bug is found, the counterexample can be used to generate a “Verification report” with the useful information to help the developers to find the source of the problem. Moreover, a “PLC demonstrator” can be automatically generated and integrated in the PLC program for proving that the bug found by the model checker exists in the real system.

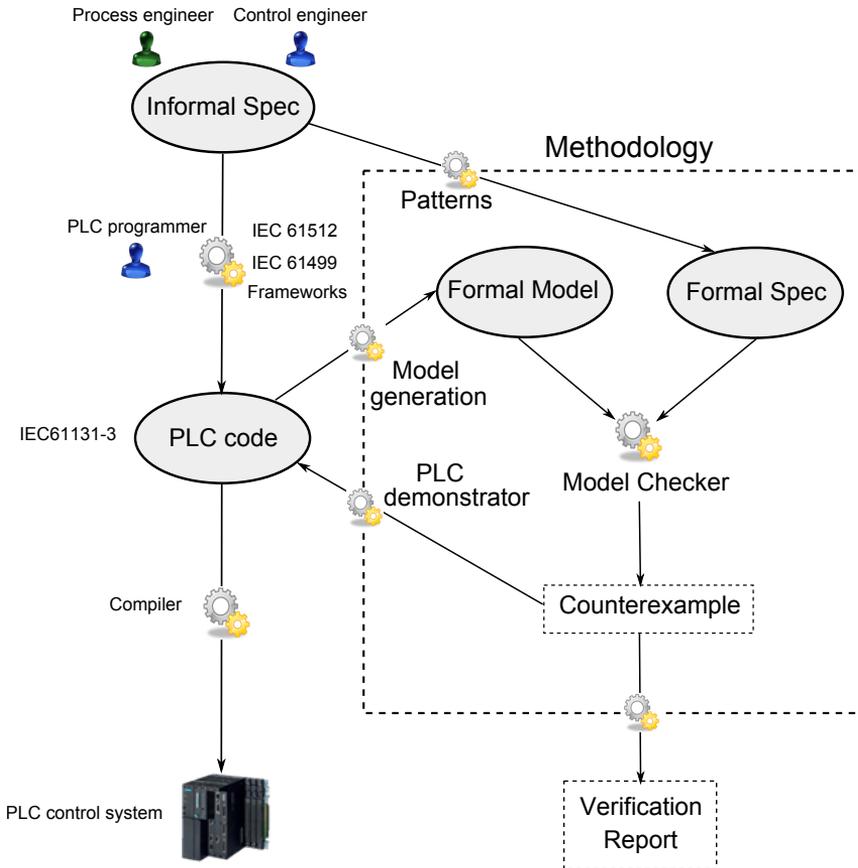


Figure 3.4: Integration of the proposed methodology in the PLC program development process.

**Why introducing an intermediate model?** There are multiple reasons to include an intermediate step in our approach, including:

- It implies the transformation from PLC code into the input languages of the verification tools. The transformation rules are split into two parts: “AST to CFG” (Abstract Syntax Tree to Control Flow Graph) transformation from the PLC code into the IM and “CFG to CFG” (Control Flow Graph to Control Flow Graph) transformation from the IM into the different model checker inputs. The “CFG to CFG” transformations are much simpler than the first ones, it consists in transforming the IM in another formalism with a different syntax. By decoupling the transformations in two different steps it allows to clearly separate the two roles of the transformation by making them independent one from another (separation of concerns design approach).
- This approach allows us to easily add new model checkers, if their input languages are close to an automata-based formalism.
- Abstraction techniques can be applied to the IM, therefore the performance of the verification can be increased for all the model checkers included in the tool chain.

**Why automata-based intermediate model?** Automata-based formalism is a simple formalism but strong enough to model all the features of a PLC control system. In addition, many verification tools use modeling languages close to this idea, e.g. nuXmv (Cavada et al. (2014)), UPPAAL (Amnell et al. (2001)) and the BIP framework (Basu et al. (2011)). Therefore the transformation rules between the IM and the input of nuXmv, UPPAAL, DFinder (from the BIP framework) or any other similar model checkers are simple to implement and the methodology, as well as the tool, can be easily extended.

**Why not timed automata as intermediate model?** When modeling PLC programs, one of the main problems we had to face is how to model the timing aspects of these systems, i.e. modeling time and timers. As a first thought, we considered to use timed automata for this purpose, using for example the UPPAAL formalism. However PLC timers use a specific data type defined by the IEC 61131 (2013)

standard called *TIME*. This data type is a finite variable and its representation is a non-monotonic time representation as the variable can overflow. Each PLC brand has its own definition of the *TIME* data type. For example, Simenes S7 PLCs represents *TIME* as a signed 32-bit integer with a precision of 1 ms. Modeling time with clocks, for example using the UPPAAL formalism, will imply a monotonic representation of the *TIME* data type, which it was not desirable for our approach. For that reason, we decided to use an automata-based formalism which is closer to more verification tools formalism and the *TIME* data is modeled as a finite variable. All the details about modeling timing aspects of PLCs are described in Section 3.9.

**Why more than one verification tool is needed?** The existing verification tools provide different advantages and disadvantages in terms of performance, simulation facilities and properties specification. The formal verification community is advancing fast and new algorithms are providing better verification performance. Our goal is not to develop a new verification tool, therefore we wanted to compare them according to this three features and provide the PLC developers the best alternative for our models. For instance, up to now, nuXmv provides better results in terms of verification performance for our current models. nuXmv also supports the full LTL and CTL for the specification properties but it lacks good simulation facilities; UPPAAL provides very good simulation facilities but it only supports a subset of CTL; BIP provides a language for modeling component-based systems, code generation and simulation facilities. From BIP, model-based automated testing can be applied, some results applied to PLC control systems can be found in Fernández Adiego et al. (2013b) and its verification tool for compositional verification (called DFinder) only supports deadlock and safety properties. The formal verification community is continuously proposing new and improved verification algorithms and tools, which can be easily included in the proposed methodology. This strategy makes the methodology independent of a single verification tool.

**How to avoid state space explosion in large PLC program models?** Automata-based models of PLC programs, as any software model for verification purposes, usually face the problem of huge state

space. Therefore abstraction and reduction techniques are needed to be able to apply formal verification to PLC programs. In this methodology these techniques are applied to the IM and all the specific model formats can benefit from these techniques. Section 3.7 presents the proposed reduction and abstraction techniques.

### 3.3 Intermediate model

This section presents the definition of syntax and semantics of the IM.

#### 3.3.1 Intermediate model syntax

The IM is based on an automata network model, consisting of synchronized automata.

**Definition 3 (Network of automata)** *A network of automata is a tuple  $N = (A, I)$ , where  $A$  is a finite set of automata,  $I$  is a finite set of synchronizations.*

**Definition 4 (Interactive automaton)** *An automaton is a structure  $a = (L, T, l_0, V_a, \underline{Val}_0) \in A$ , where  $L = \{l_0, l_1, \dots\}$  is a finite set of locations,  $T$  is a finite set of guarded transitions,  $l_0 \in L$  is the initial location of the automaton,  $V_a = \{v_1, \dots, v_m\}$  is a finite set of variables, and  $\underline{Val}_0 = (Val_{1,0}, \dots, Val_{m,0})$  is the initial value of the variables.*

*Let  $\hat{V}$  be the set of all variables in the network of automata  $N$ , i.e.  $\hat{V} = \bigcup_{a \in A} V_a$ . ( $\forall a, b \in A : V_a \cap V_b = \emptyset$ )*

**Definition 5 (State)** *A state of an interactive automaton is a pair  $LV_a = (l, \underline{Val})$ , where  $l \in L$  is the current location and  $\underline{Val}$  is the vector of current values of each variable  $v \in V$  (in a fixed order).*

**Definition 6 (Interactive transition)** *A transition is a tuple  $t = (l, g, amt, i, l')$ , where  $l \in L$  is the source location,  $g$  is a logical expression on variables of  $\hat{V}$  that is the guard,  $amt$  is the memory change (variable assignment),  $i \in I \cup \{NONE\}$  is a synchronization attached to the transition, and  $l' \in L$  is the target location.*

**Definition 7 (Synchronization)** *A synchronization is a pair  $i = (t, t')$ , where  $t \in T$  and  $t' \in T'$  are two synchronized transitions in different automata. The variable assignments attached to the transitions  $t$  and  $t'$  should not use the same variables.*

Fig. 3.5 shows an example of the IM built from PLC code. In this example, the reader can observe all the elements, which have just been defined in this section. Please note that initial and final locations (i.e.  $l_0$  and  $l_n$ ) are represented as *init* and *end* respectively on the figures to make them more clear. The green box represents a part the user PLC program logic. The red box represents a function call to the function *FC1* (represented by the automaton *FC1*). All the details about the transformation from the PLC program into the IM will be explained in the following Sections.

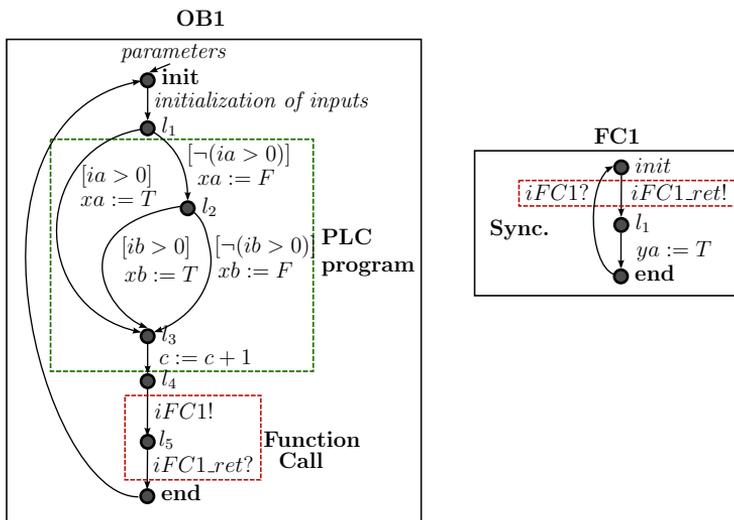


Figure 3.5: Example of IM

### 3.3.2 Intermediate model semantics

The behavior of this automata-based formalism can be easily explained informally as follows: when an automaton is in location  $l$  and a transition  $t$  goes from  $l$  to  $l'$ , it is enabled if its guard  $g$  is satisfied and it has no synchronizations ( $i = NONE$ ). If this transition  $t$  occurs (fires), the location of the automaton will be  $l'$  and the variable assignments defined for  $t$  will be executed. The transitions joined by a synchronization can only fire together and only if both are enabled (synchronous composition).

In the next paragraphs, the previous informally introduced semantics is presented in a formal way. For that purpose, the product-

automaton of our IM (with regards to the synchronizations) is represented as a state transition system (STS).

**Definition 8 (State transition system)** *A (finite) state transition system is a  $(S_{STS}, s_0, \mathcal{N})$  tuple, where  $S_{STS}$  is a finite set of states,  $s_0 \subseteq S_{STS}$  is a set of initial states,  $\mathcal{N} \subseteq S_{STS} \times S_{STS}$  is a relation over  $S_{STS}$  which describes the possible state changes i.e. transitions. In the following, a transition from state  $s$  to  $s'$  will be marked as  $s \rightarrow s'$  to improve readability.*

**Definition 9 (Semantics of interactive automata network)**

*Let  $N = (A, I)$  be an interactive automata network (where  $A = \{a_1, \dots, a_n\}$ ). The semantics of this automata network can be defined as a state transition system  $(S_{STS}, s_0, \mathcal{N})$ , where  $S_{STS}$  is the set of states  $LV_{a_1} \times \dots \times LV_{a_n}$ , and  $s_0 = (l_{0,1}, \underline{Val}_{0,1}, \dots, l_{0,n}, \underline{Val}_{0,n})$  is the initial state. The set of transitions  $\mathcal{N} \subseteq S_{STS} \times S_{STS}$  is constructed in the following way:*

- For every transition  $t = (l, g, amt, int, l') \in T$  in an automaton  $a_i$ , where  $int = NONE$  (defined as  $i$  in the IM syntax), for every  $(\underline{Val}, \underline{Val}^*) \in amt$ : if  $g(\underline{Val}) = true$ , then add:

$$(lv_1, \dots, lv_{i-1}, (l, \underline{Val}), lv_{i+1}, \dots, lv_n) \rightarrow$$

$$(lv_1, \dots, lv_{i-1}, (l', \underline{Val}^*), lv_{i+1}, \dots, lv_n)$$

$(\forall lv_j \in LV_{A_j}, \text{ where } j \neq i).$

- For every transition  $t = (l, g, amt, int, k) \in T$  in automaton  $a_i$ , where  $int = (t, t')$  (defined as  $i$  in the IM syntax) and  $t' = (l', g', amt', int, k')$  in automaton  $a_j$ , for every  $(\underline{Val}, \underline{Val}^*) \in amt$ , for every  $(\underline{Val}', \underline{Val}^{*'}) \in amt'$ , for every  $(\underline{Val}^*, \underline{Val}^{**}) \in iamt$ , and for every  $(\underline{Val}^{*'}, \underline{Val}^{**'}) \in iamt'$ : if  $g(\underline{Val}) = true$  and  $g'(\underline{Val}') = true$ , then add:

$$(lv_1, \dots, lv_{i-1}, (l, \underline{Val}), \dots, (l', \underline{Val}'), lv_{j+1}, \dots) \rightarrow$$

$$(lv_1, \dots, lv_{i-1}, (k, \underline{Val}^{**}), \dots, (k', \underline{Val}^{*'}), lv_{j+1}, \dots)$$

$(\forall lv_x \in LV_{A_x}, \text{ where } x \neq i \text{ and } x \neq j).$

An example of this approach to describe the automata semantics as STS is shown in Fig. 3.6. In this example an automata network, which contains two automata is represented by a STS. The automaton  $A$  contains one variable  $a$ , three states, three transitions and one interaction  $i$  with the automaton  $B$ . The automaton  $B$  contains one variable  $b$ , one state, one transition and one interaction  $i$  with the automaton  $A$ . The corresponding STS contains six states in which the location of each automaton and the values of the variables  $a$  and  $b$  are located. For example the first state of the STS is  $(init, false, p_0, false)$ .

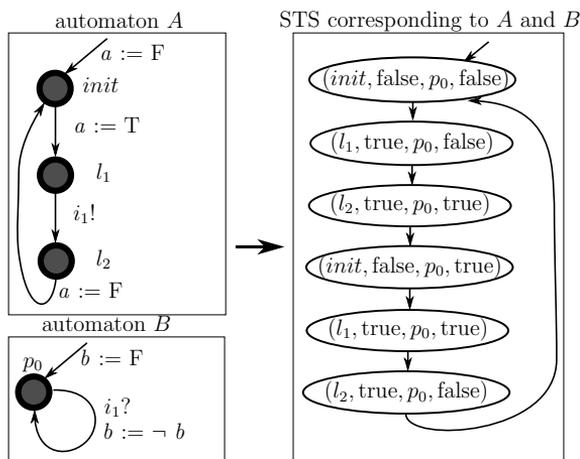


Figure 3.6: Example of representing an interactive automata network as a STS

### 3.4 Formal specifications

The first step of the methodology is to formalize the control system specifications. Specification is usually carried out by two actors in collaboration: the process engineer, who has the knowledge about the industrial process to be controlled, and the control engineer, who can design the control strategy to fulfill the requirements established by the process engineer. Traditionally in any field of engineering, specification is a major problem. Requirements specification are usually incomplete, ambiguous and even contradictory. Finding a solution to this problem is very challenging, as an expressive formal specification

language is needed and it has to be easy to understand and natural for process and control engineers.

In industrial control systems development, this problem remains unsolved. In addition, control and process engineers are not familiar with complex formal specification languages used by the verification tools, e.g. LTL or CTL. Previous works have presented some solutions for the formalization of requirements in automation, e.g. Campos et al. (2008). However, nowadays there is not a unified standard for control system specification.

In this methodology, we provide a similar approach than the one provided in Campos et al. (2008) for the formalization of the requirements specification, which is based on patterns. Patterns are simple templates with well-defined semantics, where the control and process engineer can easily write the specifications using a simple and natural language. Then these pseudo-formalized requirements can be automatically translated to the specification formalism used by the verification tools (typically temporal logic). Obviously, this solution does not solve the specification challenge for industrial control systems, but it provides a simple solution to formalize the requirements so the methodology can be applied. Comparing the patterns included in this methodology with those presented in Campos et al. (2008), more complex patterns were needed. Discussions with our PLC program developers at CERN allowed us to provide the list of patterns that cover all the current needs for verification of their PLC programs. Completeness is guaranteed this way.

According to our experience, control and process engineers usually provide ambiguous requirements, even for simple examples. For example:

“If  $\mathcal{A}$  is true, then  $\mathcal{B}$  has to be true.”

This requirement is obviously ambiguous. It is needed to specify in which moment of the PLC execution this property has to hold. In most of the cases, PLC developers want to verify that a property holds at the end of the PLC cycle. This is the critical moment as the output variables are assigned to the real peripheries at this point. Assuming this, the unambiguous requirement is:

“If  $\mathcal{A}$  is true at the end of the PLC cycle, then  $\mathcal{B}$  is always true at the end of the same PLC cycle.”

This requirement is formal enough to be expressed as a CTL formula. One possible formalization is the following:

$$\text{AG}((EoC \wedge \mathcal{A}) \rightarrow \mathcal{B})$$

In these formalizations,  $EoC$  means “end of PLC cycle” and is true only at the end of PLC cycles. Another typical example of an ambiguous requirement:

“If there is a rising edge on  $\mathcal{A}$  and  $\mathcal{B}$  is true, then  $\mathcal{C}$  is true.”

This pattern is ambiguous, as there are multiple possible interpretations of “there is a rising edge”. This has to be reformalized in order to have an unambiguous meaning:

“If in Cycle  $N$ :  $\mathcal{A}$  is false and  $\mathcal{B}$  is true, and  
in Cycle  $N + 1$ :  $\mathcal{A}$  is true and  $\mathcal{B}$  is true,  
then  $\mathcal{C}$  is always true (in Cycle  $N + 1$ ).”

As in previous example, this requirement is formal enough to be expressed as an LTL formula. One possible formalization is the following:

$$\begin{aligned} & \mathbf{G} \left( (EoC \wedge \neg \mathcal{A} \wedge \mathcal{B} \wedge \mathbf{X}(\neg EoC \cup (EoC \wedge \mathcal{A} \wedge \mathcal{B}))) \right. \\ & \left. \rightarrow \mathbf{X}(\neg EoC \cup (EoC \wedge \mathcal{C})) \right) \end{aligned}$$

In the formal models produced by this methodology,  $EoC$  is a symbol that represents the end of the PLC cycle. The details about this mechanism will be described in Section 3.5.

### 3.4.1 Patterns

The list of patterns that have been currently implemented in the methodology, can be classified in four families: Patterns for safety properties involving one PLC cycle, Patterns for safety properties involving several PLC cycles, Patterns for liveness properties and Patterns for time-related properties. Here both LTL and CTL are used for formalization.

### 3.4.1.1 Patterns for safety properties involving one PLC cycle

**Pattern TL1 (General truth under condition)** If .....[2] is true (at the end of the PLC cycle), .....[1] is always true (at the end of the PLC cycle).

or

.....[1] is always true (at the end of the PLC cycle), if ..... [2] is true (at the end of the PLC cycle).

The corresponding temporal logic formula is:

$$\text{AG}\left((EoC \wedge [2]) \rightarrow [1]\right)$$

**Pattern TL2 (Impossibility under condition)** If ..... [2] is true (at the end of the PLC cycle), ..... [1] is impossible (at the end of the PLC cycle).

or

..... [1] is impossible (at the end of the PLC cycle), if ..... [2] is true (at the end of the PLC cycle).

The corresponding temporal logic formula is:

$$\text{AG}\left((EoC \wedge [2]) \rightarrow \neg[1]\right)$$

**Pattern TL3 (General truth)** .....[1] is always true (at the end of every PLC cycle).

The corresponding temporal logic formula is:

$$\text{AG}\left(EoC \rightarrow [1]\right)$$

**Pattern TL4 (State change during a cycle)** If ..... [2] is true (at the beginning of the PLC cycle), ..... [1] is always true (at the end of the same PLC cycle).

The corresponding temporal logic formula is:

$$\text{AG}\left((SoC \wedge [2]) \rightarrow \text{A}(\neg EoC \cup [1])\right)$$

where *SoC* means “Start of PLC cycle” and is true only at the beginning of PLC cycles.

### 3.4.1.2 Patterns for safety properties involving several PLC cycles

**Pattern TL5 (General truth on rising edge)** If ..... [1] has a rising edge (at the end of the PLC cycle), ..... [2] is always true (at the end of the PLC cycle).

A rising edge on [1] means that [1] is false at the end of cycle N and true at the end of cycle N+1. The corresponding temporal logic formula is:

$$\begin{aligned} &G\left((EoC \wedge \neg[1] \wedge X(\neg EoC \cup (EoC \wedge [1])))\right) \\ &\rightarrow X(\neg EoC \cup (EoC \wedge [2])) \end{aligned}$$

**Pattern TL6 (General truth on rising edge under condition)** If ..... [1] has a rising edge and ..... [3] is true (at the end of the PLC cycle), then ..... [2] is always true (at the end of the PLC cycle).

The corresponding temporal logic formula is:

$$\begin{aligned} &G\left((EoC \wedge \neg[1] \wedge X(\neg EoC \cup (EoC \wedge [1] \wedge [3])))\right) \\ &\rightarrow X(\neg EoC \cup (EoC \wedge [2])) \end{aligned}$$

**Pattern TL7 (State change between cycles)** If ..... [1] is true at the end of cycle N and ..... [2] is true at the end of cycle N+1, then ..... [3] is always true at the end of cycle N+1.

The corresponding temporal logic formula is:

$$\begin{aligned} &G\left((EoC \wedge [1] \wedge X(\neg EoC \cup (EoC \wedge [2])))\right) \\ &\rightarrow X(\neg EoC \cup (EoC \wedge [3])) \end{aligned}$$

**Pattern TL8 (State change between cycles (emulating sequence))**

If ..... [1] is true at the end of cycle N, then ..... [2] is true at the end of cycle N, and also if ..... [3] is true at the end of cycle N+1, then ..... [4] is always true at the end of cycle N+1.

The corresponding temporal logic formula is:

$$\begin{aligned} &G\left((EoC \wedge [1] \wedge X(\neg EoC \cup (EoC \wedge [3])))\right) \\ &\rightarrow [2] \wedge X(\neg EoC \cup (EoC \wedge [4])) \end{aligned}$$

### 3.4.1.3 Patterns for liveness properties

**Pattern TL9 (Liveness)** If ..... [1] is true (at the end of the PLC cycle), then ..... [2] will definitely be true sometime (at the end of a PLC cycle).

The corresponding temporal logic formula is:

$$\text{AG}\left(\text{EoC} \wedge [1] \rightarrow \text{AF}(\text{EoC} \wedge [2])\right)$$

### 3.4.1.4 Patterns for time-related properties

When verifying time-related properties with explicit time in it, a “monitor” or observer automata is added to the model, as CTL and LTL do not provide the required expressiveness. In this case, a monitor consists in an automaton added to the global model of the PLC program with the same behavior as a TON timer but independent of the rest of the program logic. With this monitor the formal property is simplified to a safety as it is shown in the Pattern TL10. The output of this monitor is compared to the variable or group of variables which is part of the property to be verified (referenced as [3] in the pattern) and affected by the timer and by the program logic. All the details about time and timers modeling and verification of time-related properties can be found in Section 3.9.

**Pattern TL10 (TON-like property)** If there is a rising edge on ..... [1] and it remains true then after ---- [2] seconds, ..... [3] will be true (at the end of the PLC cycle).

TON monitor on [1] with PT=[2] and the corresponding temporal logic formula is:

$$\text{AG}\left(\left(\text{EoC} \wedge \text{MonitorQ}\right) \rightarrow [3]\right)$$

**Pattern TL11 ((11) TP-like property)** If there is a rising edge on ..... [1], then for the next ---- [2] seconds, ..... [3] will be true (at the end of the PLC cycles).

TP monitor on [1] with PT=[2] and the corresponding temporal logic formula is:

$$\text{AG}\left(\left(\text{EoC} \wedge \text{MonitorQ}\right) \rightarrow [3]\right)$$

## 3.5 PLC hardware modeling

In order to apply formal verification to PLC software, the PLC execution platform has to be modeled. In Fig. 3.2, the first block of the methodology called “PLC world” contains the “PLC knowledge” needed to build the IM. A simple modeling approach is proposed in this methodology for the PLC hardware.

*Assumption 1.* Currently, only centralized PLC control systems consisting in one single PLC are considered. Modeling distributed systems will require to find an appropriate level of abstraction to minimize the state explosion problem.

A simple model is proposed, consisting in modeling the scan cycle: input reading, execution of the logic, and writing the variables in the outputs. This knowledge provides the skeleton of the complete model.

*Assumption 2.* Other hardware devices, such as input and output cards, communication interfaces, field buses or any kind of communication with the SCADA, are not modeled. This implies that failures coming from the PLC hardware will not be detected by using this approach (however it is not the goal of the methodology).

The PLC hardware information, which gives the skeleton of the models, is shown in the example of Fig. 3.7. This example shows the four main characteristics introduced in the models from the PLC hardware:

- The cyclic execution of the IM, representing the *PLC scan cycle*.
- The initialization of the variables of the IM in the first location of the model (*init*). In this location, the example shows a special case of input variables: the *parameters*. Parameters are input variables that are constant during the execution, as they are hard-coded into the PLC application.
- In the first transition of the model, from *init* to  $l_1$ , random values are assigned to all the *input variables* of the system, representing the first step of the scan cycle: reading the input values from the periphery and writing this values in the Input Image Memory.
- The rest of the model represents the execution of PLC code and the final location of the model *end* (i.e. *EoC*). This final location represents the moment when the values are written from the

Output Image Memory to the output periphery. This location will be used in the properties to be verified as it was described in Section 3.4.

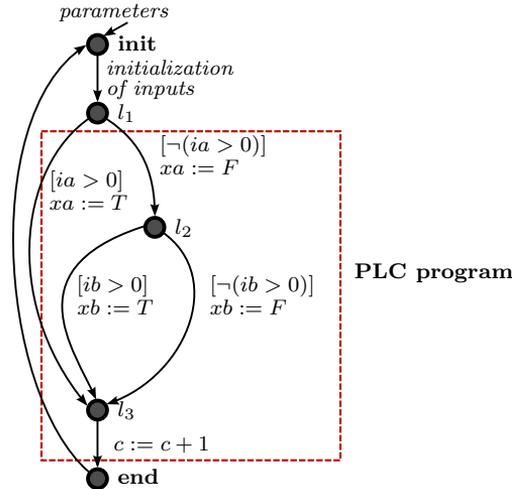


Figure 3.7: Example of IM

It is obvious that identifying automatically the input variables from the system is critical for the methodology. Even if it may look a simple task it is quite complex. When a real-life PLC program is modeled, different kind of input can be found. The next paragraph describes the different kind of inputs, how to identify them and how they are modeled in this methodology.

### 3.5.1 PLC inputs

In a centralized or decentralized control system (i.e. one single PLC), four kind of inputs can be found:

1. The input variables connected to the process through a local input card: these variables are mapped in the IIM and can be easily identified. For example in Siemens PLCs, the I0.0 is a variable from the IIM connected to a digital sensor. Its value can be modified at the beginning of every PLC cycle.

2. The input variables connected to the process through a decentralized periphery: when using field buses like Profibus or Profinet the values of these inputs can be mapped in the IIM, but sometimes they are mapped in different variables: for example in Siemens PLCs they can be mapped in a *DataBlock (DB)* variable or in a *Memory (M)* variable. In this case, its value can also be modified at the beginning of every PLC cycle.
3. The input variables coming from the SCADA: these variables are the orders coming from the the operator of the system. These orders are not mapped in the IIM, for example in Siemens PLCs they can be mapped in a *DataBlock (DB)* variable. Their value can be modified at any moment in every PLC cycle, but it depends on the implementation of the communication protocol between the PLC and the SCADA.
4. Parameters: they can be considered as a special case of input variables. A parameter has a constant initial value and this value does not change in the whole execution of the PLC program.

The main problem now is how to identify these four input variables and how to model them. In general, any input variable coming from the SCADA or from the process can be identified as no value is assigned to these variables in the PLC program. Parameters are assigned once with a constant value and never modified again. However, based in our experience, it may happen that in real-life PLC programs other variables, not only input variables are never assigned. So if this happens, the generated models will be obviously wrong. For that reason, before identifying the input variables of the system, we recommend to perform a static analysis and ensure that this situation will not happen.

As it was mentioned before, any input variable coming from the process is modeled assigning them a random value in the first transition of the IM. In the case of input variables coming from the SCADA, these values can be modified in the PLC program at any time of the PLC program execution, as it depends of the communication protocol between the SCADA and PLC. As we do not consider concurrency problems, any input variable coming from the SCADA is modeled assigning them a random value in the first transition of the IM as well.

Parameters are initialized in the initial location of the model and its value remains constant along the execution of the model.

### 3.5.2 Safety and standard PLCs

For the methodology, the difference in terms of modeling strategy between safety and standard PLCs has been analyzed. If time-related properties are not required, there is no difference from the modeling point of view. But if required, a small difference in the model is produced. Modeling the timing aspects of PLCs is described in the Section 3.9. In a safety PLC, the PLC cycle time is fixed and constant due to the safety constrains, however in a standard PLC it is variable. As it is described in Section 3.9, when a time related property is required, a 16-bits variable modeling time is created. This variable will be incremented with the PLC cycle time at the end of the PLC cycle. Therefore, models from Safety PLCs will increment this variable with a constant value of time and standard PLCs will increment the time variable with a random value between some bounds.

### 3.5.3 Interrupts and restarts in PLCs

As it was described in Section 2.2, PLC programs are composed of the main program and it may contain interrupts and restarts (which are a particular case of interrupts), for example, OB35 in Siemens PLCs. These interrupts can occur at any time. The main program is interrupted and the routine associated to the interrupt is executed. When the execution of this routine finishes, the main program continues the execution from the same point where it was interrupted. This behavior of the PLC scheduler is not modeled in the methodology, therefore currently concurrency problems cannot be detected. For that reason, concurrency problems should be checked before, by applying static analysis techniques and checking that the main program, the interrupts and restarts routines do not modify the same variables of the PLC programs.

Once the program does not contain these kind of problems, model checking can be applied. The PLC program, the interrupts and restarts routines are then modeled. The following modeling strategy is applied:

1. For each interrupt or restart routine, an automaton will be created.
2. The model of the PLC scheduler consists in the main program being executed at every cycle. At the end of the main program automaton, two extra transitions and one location are created per interrupt or restart routine. These transition contain a synchronization  $i$ , to synchronize the main program automaton with the interrupt or restart automaton.
3. These interactive transitions are non-deterministically executed, simulating a random execution of the PLC interrupt.

Fig. 3.8 shows a model example of a Siemens PLC program . The original PLC program is composed of the main program (OB1) and a cyclic interrupt (OB35). The corresponding model contains two automata, modeling the behavior of both routines. In addition, two Interactive transition are added in order to synchronize OB1 with OB35. But these transitions will be executed depending on a random value assigned to the variable  $ic$ .

## 3.6 PLC code – IM transformation

Once the specification is formalized and the skeleton of the models is defined, the PLC code is automatically translated to the IM. This section describes the most relevant transformation rules from PLC code to the IM. As it was mentioned in Section 2.2, five languages are defined by the IEC 61131 (2013). Here the transformation from ST and SFC languages are presented, as they are the two most relevant and used PLC languages at CERN. These transformation rules can be extended for the rest of PLC languages defined by the standard. It has to be noticed that although not all of the PLC vendors follow the IEC 61131-3 standard 100%, sometimes there are only a few syntactic differences, it does not imply any difference for the modeling strategy. Along this Section, when examples of PLC code are shown to illustrate the transformation rules, they correspond to Siemens PLC code as it is the main brand used at CERN. The section is divided in the following subsections:

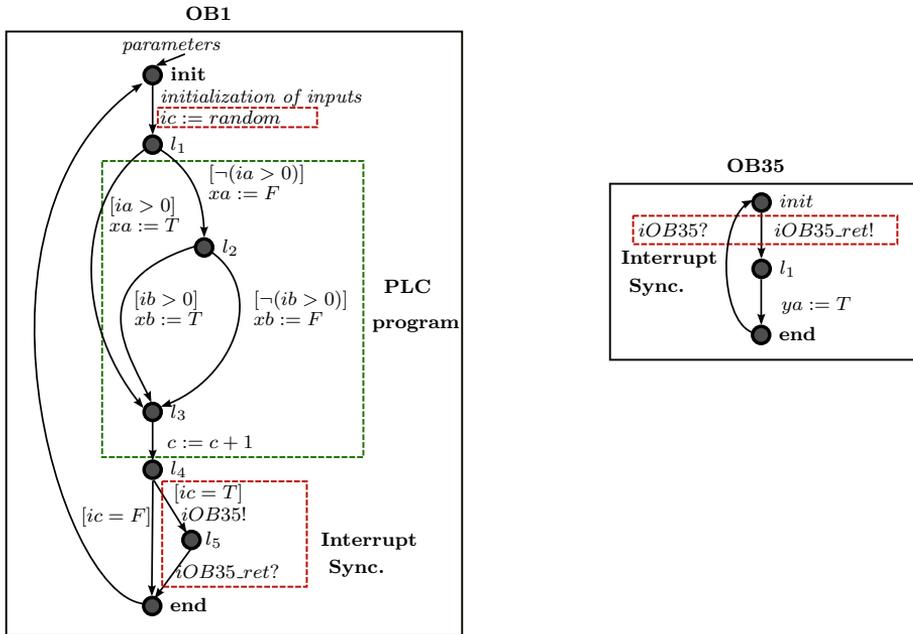


Figure 3.8: Example of IM with one PLC interrupt

- General transformation rules from PLC code into IM (Section 3.6.1).
- Transformation rules from ST code into IM (Section 3.6.2).
- Transformation rules from SFC code into IM (Section 3.6.3).

Before the rules, the data type mapping between the PLC code and the different modeling languages has to be introduced. Table 3.1 presents the mapping between the most common data types used in PLC, the data types used in IM and the data types used in the verification tools (nuXmv, UPPAAL and BIP).

### 3.6.1 General PLC – IM transformation

The transformation rules are presented hierarchically from high level to low level rules. Actually the first rule, which is included here, is the link with PLC hardware modeling strategy.

Table 3.1: Data type mapping

PLC	IM		nuXmv	BIP2	UPPAAL
bool	Boolean		Boolean	char	bool
int	signed Integer (16 bit)		signed word[16]	int	int
uint	unsigned Integer (16 bit)		unsigned word[16]	int	int[0,65535]
dint	signed Integer (32 bit)		signed word[32]	int	int <sup>1</sup>
udint	unsigned Integer (32 bit)		unsigned word[32]	int	int
real	signed Real (32 bit)		signed word[32]	int	int
char	unsigned Char (8 bit)		unsigned word[8]	char	int[0,255]
time	signed Time (32 bit)		signed word[32]	int	int
date	signed Date (16 bit)		signed word[16]	int	int
byte	8 x Boolean				
word	16 x Boolean				
dword	32 x Boolean				

<sup>1</sup> The int in UPPAAL is 16 bit long, thus some information is missing for this particular mapping. (The default range of int in UPPAAL is int[-32768, 32767]).

**Rule PLC1 (Multiple concurrent code blocks)** As it was mentioned in Section 3.5, PLC programs are composed of the main program (i.e. OB1 in Siemens PLCs), which is executed cyclically, and the interrupt handlers.

*Assumption 3.* Interrupting blocks and the interrupted blocks should use disjoint set of variables. This is a reasonable assumption, since it can be validated by existing static analysis techniques.

According to our experience, different OBs usually use different variables. Furthermore, high level of concurrency is rare in PLC programs.

Having this assumption, instead of modeling the interrupts in a preemptive manner, we model them with non-preemptive semantics: the model of the PLC scheduler consists in the main program being executed at every cycle, whereas one or several interrupts can be executed non-deterministically at the end of the PLC cycle. An example of this rule was presented in Fig. 3.8.

**Rule PLC2 (FC)** This rule translates functions into IM. An OB can be considered as a special FC that is invoked by the operating system, thus this rule also applies to OBs.

*Assumption 4.* Recursion is not allowed, i.e. no FC or FB can directly or indirectly generate a call to itself. This assumption is consistent with the IEC 61131 (2013) standard. However, Siemens PLCs allows the use of recursion with some restrictions even if it is not recommended. Recursion can be statically detected by building the call graph of a program and checking whether it contains cycles. Thus we can assume that variables of a function are stored at most once on each L Stack stack.

For each function  $Func$ , we create an automaton  $A_{Func}$ . The locations, transitions and initial location of this automaton are generated using the rules presented below. For each variable defined in  $Func$  we create a corresponding variable in  $A_{Func}$ . If the return type of the function is different from `void`, a special output variable called  $RET\_VAL$  is also added to the automaton.  $A_{Func}$  contains at least the initial location  $init$ , the final location  $end$  and the transition  $t_{end}$  from  $end$  to  $init$ .

Fig. 3.9 shows an example of the transformation of a FC into the IM. This rule is general for any PLC language.

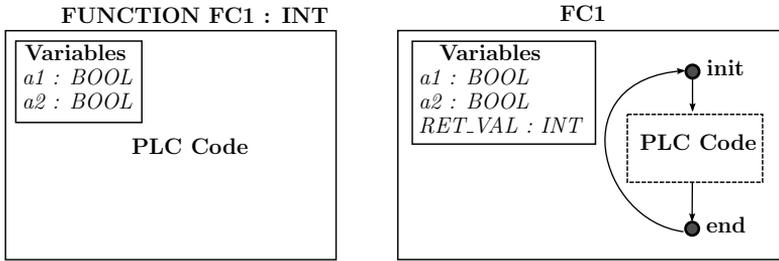


Figure 3.9: FC – IM transformation

**Rule PLC3 (FB instance)** This rule translates FB instances into IM. Assumption 4 also applies here.

For each instance  $inst$  of each function block  $FBlock$ , we create an automaton  $A_{FBlock,inst}$ . The locations, transitions and initial location of this automaton are generated using the rules presented below. For each variable in  $FBlock$  we create a corresponding variable in all the corresponding  $A_{FBlock,inst}$  automata. Each automaton contains at least the initial location  $init$ , the final location  $end$  and the transition  $t_{end}$  from  $end$  to  $init$ .

Fig. 3.10 shows an example of the transformation of a FB instance into the IM. This rule is general for any PLC language.

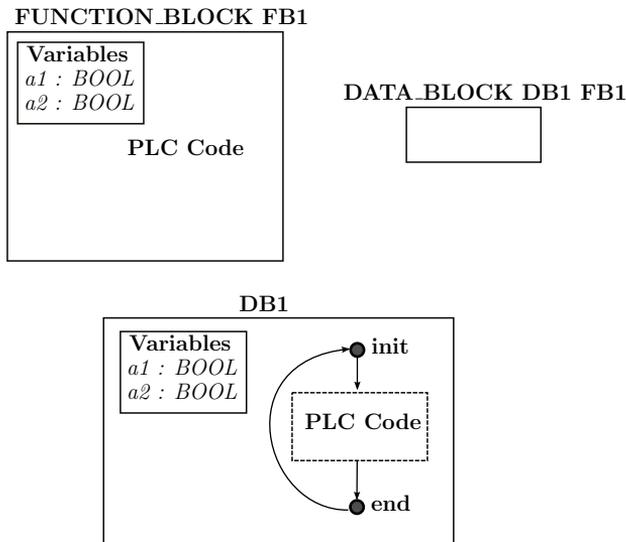


Figure 3.10: FB instance – IM transformation

**Rule PLC4 (Variables)** This rule maps program variables to variables in the IM model.

*Assumption 5.* All variables, except system inputs, that do not have uniquely defined initial values on the PLC platform (e.g. temporary variables, output variables of FCs) are written before they are read. This means that we do not have to model such variables as non-deterministic variables in the IM model, which allows us to limit the state space growth of the generated model.

For each variable  $v$  in the program block, there is exactly one corresponding variable  $F_V(v)$  in the corresponding automaton. If the variable represents a system input (i.e. variables representing signals coming from the field), it is assigned non-deterministically at the beginning of each PLC cycle. How to identify input variables is explained in the Section 3.5.

**Rule PLC5 (FC or FB call)** This rule translates FC or FB calls into the IM.

*Assumption 6.* All the input variables are assigned in the caller, and all the output variables are assigned in the callee in order to avoid the accessing of uninitialized variables that could contain unpredictable values. This means that we do not have to model them as non-deterministic variables and the state space will not be incremented drastically.

For every *function (block) call*  $\langle [r :=]Func(p_1 := Expr_1, p_2 := Expr_2, \dots) \rangle$  in a code block represented by automaton  $A_{caller}$ , we add the following elements. (*Func* can be a function or an instance of a function block, represented by automaton  $A_{callee}$ . If *Func* is a function block or a void function, the “ $r :=$ ” part is omitted.)

- A new location  $l_{wait}$  is added to  $A_{caller}$ . It represents the state when the caller block is waiting for the end of the function call. (For every function call, we add a separate  $l_{wait}$  location.)
- A transition  $t_1$  is added to  $A_{caller}$ , which has no guard and goes from the corresponding location marked as  $F_S(stmt)$  to  $l_{wait}$ . It assigns the function call parameters to the corresponding variables in  $A_{callee}$ . (It assigns  $Expr_1$  to  $F_V(p_1)$ , etc.)
- A transition  $t_2$  is added to  $A_{caller}$ , which has no guard and goes from  $l_{wait}$  to the location end of the corresponding automaton

$F_S(n(stmt))$ . It assigns  $RET\_VAL$  of the callee to the corresponding variable (variable  $F_V(r)$ ) in  $A_{caller}$ , if  $RET\_VAL$  exists. It also assigns the corresponding values to the output variables.

- A synchronization  $i_1$  is added to the automata network, connecting transition  $t_1$  with the first transition of  $A_{callee}$ .
- A synchronization  $i_2$  is added to the automata network, connecting the  $end \rightarrow init$  transition of  $A_{callee}$  with transition  $t_2$ .

Fig. 3.11 shows an example of the transformation of a FB instance into the IM. This rule is general for any PLC language.

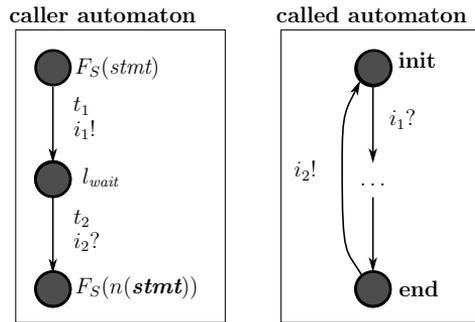


Figure 3.11: FC call – IM transformation

**Rule PLC6 (Building blocks)** PLC code often uses functions or function blocks provided by the system. The source code of these blocks is not available, it has to be created manually. One widely used group of these system blocks are timers. Modeling the timing aspects of timers is a challenging task and it highly increases the state space of the models. Section 3.9 describes the details of corresponding models for the different PLC timers.

### 3.6.2 ST – IM transformation

This section is dedicated to the specific transformation rules from the ST language into the IM.

**Rule ST1 (ST statement)** A statement is the smallest standalone element of an ST program. It can contain other components (e.g.

expressions). There are different kinds of statements like conditional branches, loops, variable assignments. In this section we define the representation of a single code block consisting of these statements in our IM.

For each statement  $stmt$ , let  $n(stmt)$  be the next statement after  $stmt$ . As we model cyclic PLC programs, the last statement in the ST code is followed by the first one. Furthermore, for a statement list  $sl$  let  $first(sl)$  be the first statement of the list. The Assumption 3 is also applied here. For each ST *statement*  $stmt$  in the program block, there is at least one corresponding location marked as  $F_S(stmt)$  in automaton  $A$ .

**Rule ST2 (Variable assignment)** This rule translates ST variable assignments into the IM.

*Assumption 7.* For each “variable access” the variable to be accessed can be determined in transformation time. This means that pointers are not supported. However, we support various single and compound variables (arrays and user defined structures). Typically, this is not a restriction as the usage of pointers is not recommended in PLC programs.

For each *variable assignment*  $stmt = \langle v_i := Expr \rangle$ , we add a transition to automaton  $A$  which goes from  $F_S(stmt)$  to  $F_S(n(stmt))$  which has no guard and no synchronization, and it assigns  $Expr$  to the variable  $F_V(v_i)$  and does not modify the other variables. (Formally:  $t = (F_S(stmt), TRUE, \langle F_V(v_i) := Expr \rangle, NONE, F_S(n(stmt)))$ .)

Listing 3.1 shows an example of ST code with a simple variable assignment and Fig. 3.12 shows the corresponding automaton fragment.

```

1 FUNCTION FC150
2   VAR_INPUT
3     ...
4   END_VAR
5   VAR
6     v1 : BOOL;
7   END_VAR
8   BEGIN
9     v1 := Expr;
10  END_FUNCTION_BLOCK

```

Listing 3.1: Variable assignment in ST code

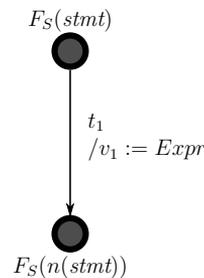


Figure 3.12: Corresponding automaton fragment for variable assignment

**Rule ST3 (Conditional statement)** For each *conditional statement*  $stmt = \langle IF\ c\ THEN\ sl_1\ ELSE\ sl_2\ END\_IF \rangle$ , we add two transitions ( $t_1$  and  $t_2$ ) to automaton  $A$ :

- The transition  $t_1$  goes from  $F_S(stmt)$  to  $F_S(first(sl_1))$ , it has no assignments and no synchronizations, and it has a guard  $c$ . (Formally:  $t_1 = (F_S(stmt), c, a_{identity}, NONE, F_S(first(sl_1)))$ )
- The transition  $t_2$  goes from  $F_S(stmt)$  to  $F_S(first(sl_2))$ , it has no assignments and no synchronizations, and it has a guard  $NOT\ c$ . (Formally:  $t_2 = (F_S(stmt), NOT\ c, a_{identity}, NONE, F_S(first(sl_2)))$ )

Listing 3.2 shows an example of ST code with a simple conditional statement and Fig. 3.13 shows the corresponding automaton fragment. The statement lists  $sl_1$  and  $sl_2$  are modeled according to the rule 1.

```

1 FUNCTION FB_A
2   VAR_TEMP
3     c : BOOL;
4   END_VAR
5 BEGIN
6   IF c THEN
7     sl1;
8   ELSE
9     sl2;
10  END_IF;
11 END_FUNCTION

```

Listing 3.2: Conditional statement in ST code

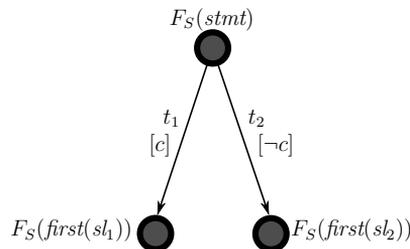


Figure 3.13: Corresponding automaton fragment for conditional statement

Listing 3.3 shows a small example of Siemens ST program, composed of the main program (OB1), a function block (FB.B) and an instance of this function block. The corresponding IM is shown in Fig. 3.14.

```

1 ORGANIZATION_BLOCK OB1
2   VAR_TEMP
3     a2 : BOOL;
4     a3 : BOOL;
5   END_VAR
6   BEGIN

```

```

7   IF a2 THEN
8       FB_B.DB1(b1:=NOT a2);
9   END_IF;
10  a2 := NOT I0.0;
11  a3 := DB1.b2;
12 END_FUNCTION_BLOCK
13
14 FUNCTION_BLOCK FB_B
15   VAR_INPUT
16       b1 : BOOL;
17   END_VAR
18   VAR_OUTPUT
19       b2 : BOOL;
20   END_VAR
21   BEGIN
22       b2 := NOT b1;
23 END_FUNCTION_BLOCK
24
25 DATA_BLOCK DB1 FB_B
26 BEGIN
27 END_DATA_BLOCK

```

Listing 3.3: Example of ST code

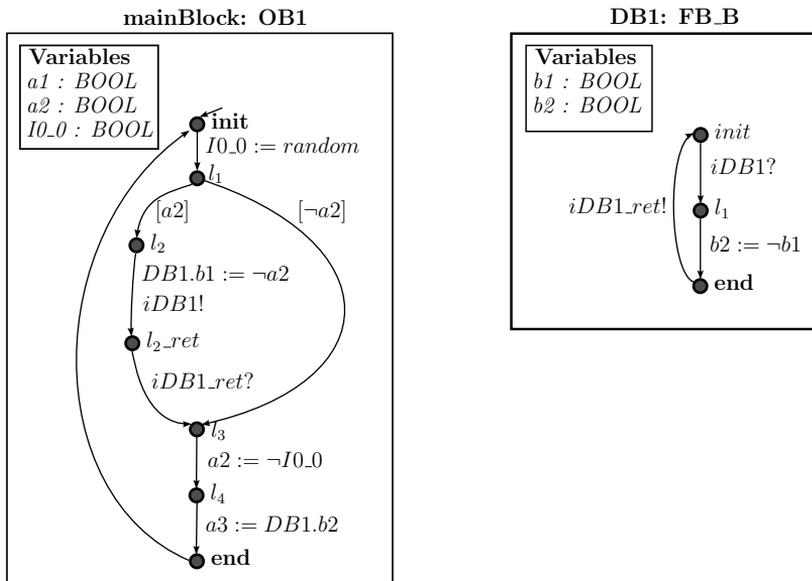


Figure 3.14: Example of transformation from a FC call into the IM

### 3.6.3 SFC – IM transformation

This section is dedicated to a high-level overview of the specific SFC–IM transformation rules.

*Assumption 8.* There is no parallel branch in the FSM, thus at most one step can be active at the same time. Also steps do not contain any actions. This allows to simplify the discussion.

Each call of a FB implemented in SFC will initiate one transition in the FSM (or zero, if there is no allowed transitions) under our assumptions, multiple transitions cannot be fired by one call.

**Rule SFC1 (SFC block)** This rule adds to the IM the needed extra information for SFC blocks.

Each automaton  $A$  representing an SFC is extended with a transition  $t = (l_1, g, amt_{identity}, NONE, end)$ , where  $g$  is true iff no other  $l_1 \rightarrow end$  transitions are enabled. The transition does not change the values of the variables. It represents that if no SFC transitions are allowed, the current state of the FSM will not be modified.

**Rule SFC2 (SFC steps)** For each step “STEP  $stepName$ : END\_STEP” we create a Boolean variable  $stepName.x$  in  $A$ . These variables represent the current state of the FSM.

**Rule SFC3 (SFC initial step)** For the initial step “INITIAL\_STEP  $initStepName$ : END\_STEP” besides of creating a Boolean variable  $initStepName.x$  in  $A$ , we add also a location  $l_1$  and a transition  $t_1$  from  $init$  to  $l_1$  without any condition.

We also add a new transition  $t = (l_1, g, amt, NONE, end)$ . The guard  $g$  is  $stepName_1 = false \wedge stepName_2 = false \wedge \dots$ . The assignment is  $amt = \langle initStepName.x := true \rangle$ . It means if no steps are active, then the initial step should be active.

**Rule SFC4 (SFC transitions)**

*Assumption 9.* Based on our experiments with the SFC editor of Siemens we assume that transitions are defined in ascending order of priority.

For each transition “TRANSITION  $tName$  FROM  $step1$  TO  $step2$  CONDITION :=  $C$  END\_TRANSITION” we create a transition  $t = (l_1, g, amt, NONE, end)$  in  $A$ . The guard  $g$  of  $t$  is a Boolean expression that is only true if (1)  $C$  is true, (2)  $step1.x$  is true, (3) all the guards

of automaton transitions representing higher priority SFC transitions are false. The assignment is  $amt = \langle step1.x := false; step2.x := true \rangle$ .

An example automaton representing the SFC shown in Listing 3.4. Its graphical form (Screenshot from SIMATIC tool by Siemens) in Fig. 3.15. Finally, Fig. 3.16 shows the corresponding IM for this example.

```

1 FUNCTION_BLOCK FB101
2   VAR_INPUT
3     FillCond : BOOL := FALSE;
4     RunCond  : BOOL := FALSE;
5     StopCond : BOOL := FALSE;
6   END_VAR
7
8   INITIAL_STEP Stop: END_STEP
9   STEP Fill: END_STEP
10  STEP Run: END_STEP
11
12  TRANSITION S_F
13    FROM Stop TO Fill CONDITION := FillCond
14  END_TRANSITION
15
16  TRANSITION F_R
17    FROM Fill TO Run CONDITION := RunCond
18  END_TRANSITION
19
20  TRANSITION R_S
21    FROM Run TO Stop CONDITION := StopCond
22  END_TRANSITION
23 END_FUNCTION_BLOCK

```

Listing 3.4: Example of the textual representation of SFC code

## 3.7 Reduction techniques

Once the IM is built and the requirements are formalized, the next step of the methodology is to apply automatic reduction and abstraction techniques to reduce the state space of the IM. For industrial size PLC programs, this step is extremely important as formal verification will not be possible without these reduction techniques.

Some verification tools provide some automatic reduction and abstraction techniques. If these tools are integrated in the methodology,

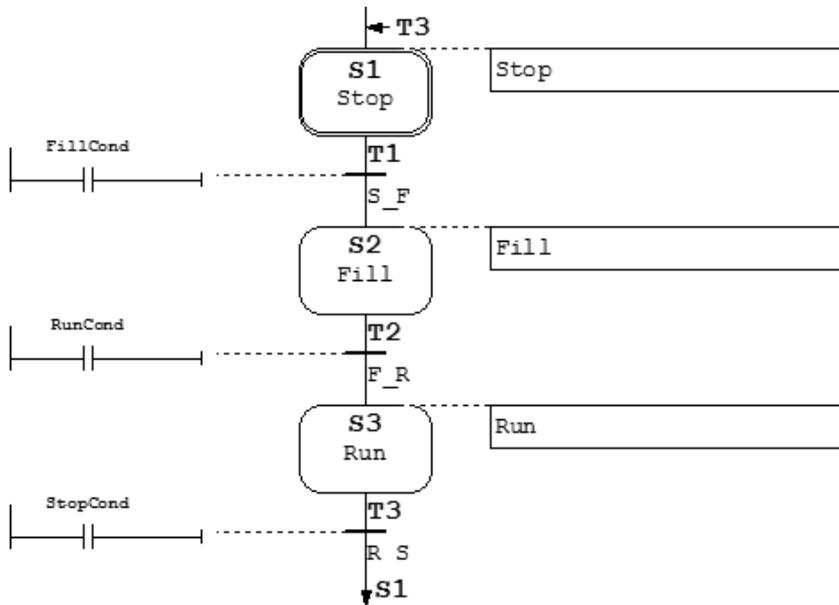


Figure 3.15: Example SFC (Screenshot from SIMATIC tool by Siemens)

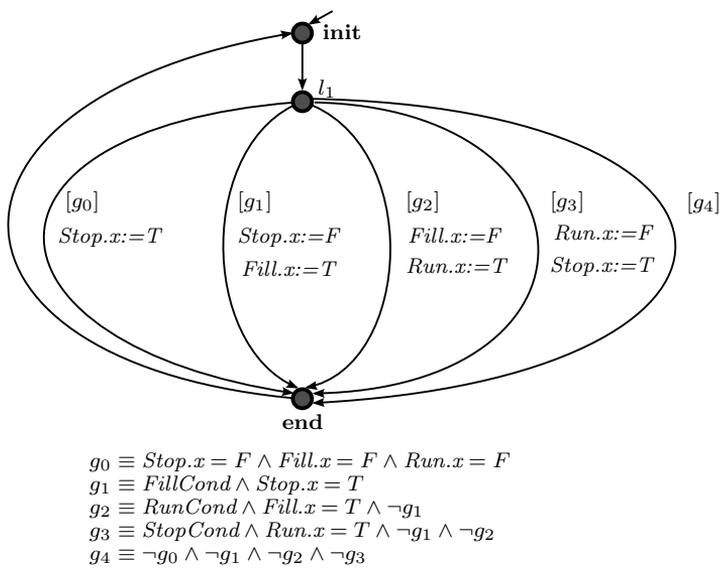


Figure 3.16: IM representation of the SFC example

the methodology will benefit from the reduction techniques. Some authors have proposed reduction techniques applied to a specific input model of a verification tool. In some cases, these reductions cannot be applied to other formalism, because these reduction use the specificities of the modeling language (e.g. Gourcuff et al. (2008) with NuSMV models built from PLC programs).

In this methodology, these reductions are applied to the IM as it is meant to be independent from the verification tools and all the verification tools that are integrated in the methodology can benefit from these techniques. Most of these reduction techniques are general for any CFG, not just for the CFGs of PLC programs. Both alternatives (apply reduction to the IM or the input models of the verification tools) are valid in our approach as it can be seen in Fig. 3.17.

As it was mentioned in Section 3.4, the requirements to be verified are often complex, containing multiple temporal logic operators and involving a large number of variables, which limits the set of possible reduction techniques.

In this methodology, the reduction and abstraction techniques that have been implemented can be classified in two groups:

1. Property preserving reduction techniques: These algorithms aim at preserving only those behaviors of the system that are relevant from the specification point of view. By using property preserving techniques, the meaning of the model is not modified with regard to our assumptions, thus these reductions do not induce any false results that would make the verification much more difficult.
2. Non-preserving abstractions, where the resulting model “loose” some information regarding the original model.

The correctness of the transformation when applying the techniques from the first group is easier to prove, as the reduced model and the original are equivalent for a specific property. However, in some particular cases non-preserving abstractions are applied in this methodology, for example, Section 3.9 presents the modeling strategy for timers. In this case, non-preserving abstractions were applied.

Before describing the reduction techniques, it has to be noticed that in a PLC, the requirements usually are checked only at the beginning and at the end of the PLC cycles, but not in the intermediate

states, as the transient values are not written to the physical outputs of the PLC. As a consequence, the order of the variable assignments is not important, as long as the result at the end of the PLC cycle will be the same. Thus, we know that the requirements are only to be checked at specific locations of the CFG. As it was mentioned in Section 3.6.2, concurrency problems are not considered (assumption 3, described in the previous section).

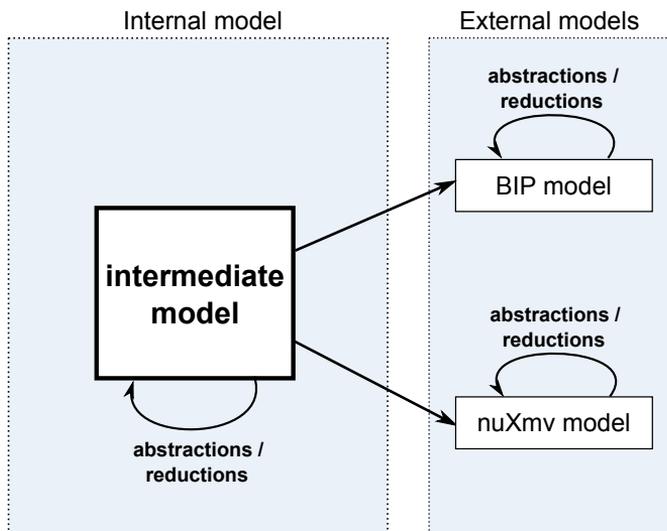


Figure 3.17: Reduction techniques applied to the IM and to the verification tool models

This section presents the four reduction and abstraction techniques implemented in this methodology. The first three techniques are property preserving reduction techniques and the last one is not:

- First, a new heuristic *cone of influence reduction* algorithm adapted for models representing CFGs is presented in Section 3.7.1.
- Secondly, a heuristic *rule-based reduction* techniques to support the cone of influence algorithm is presented in Section 3.7.2.
- Then, a *mode selection* method which allows the developer to fine-tune the verification by setting the operational mode to be verified in Section 3.7.3.

- Finally, a method called *variable abstraction* is described in Section 3.7.4. This method allows to verify large PLC programs, when the previous reduction are insufficient but only verification of safety properties is possible.

The first three reduction techniques have been published in our paper Darvas et al. (2014).

### 3.7.1 Cone of influence

*Cone of influence* (COI), described in Clarke et al. (1999), is one of the most powerful property preserving reduction techniques. It consists in identifying which part of the model is relevant for the evaluation of the given requirement. The unnecessary parts of the model can be removed without affecting the result.

#### 3.7.1.1 Motivation for applying COI at the IM level

Some model checkers, such as NuSMV has a built-in COI implementation that can reduce the verification time drastically. Some experiments applied with our models with the NuSMV model checker, show a bad verification performance even when the COI algorithm could theoretically reduce the model size to trivial. By analyzing the NuSMV’s COI implementation, we found out that this reduction technique could be much more powerful if it was applied to IM directly, where the structure of the CFG is known, before it is transformed to a general state-transition system. In addition, the rest of the verification tools can benefit from this reduction technique if it is applied to the IM.

In higher level models there is usually more knowledge about the modeled system, therefore the reductions are generally more efficient if they are applied to a higher abstraction level. In this case, the COI algorithm can benefit from the structure information present in the intermediate model, but missing from the generated NuSMV model. Therefore, a new heuristic cone of influence algorithm has been developed, which outperforms the COI method built in NuSMV for our models and highly improves the verification performance of our models. To explain the idea behind our approach, first an overview of the COI implementation of NuSMV is given. Then, the new heuristic is introduced and compared with the NuSMV’s COI.

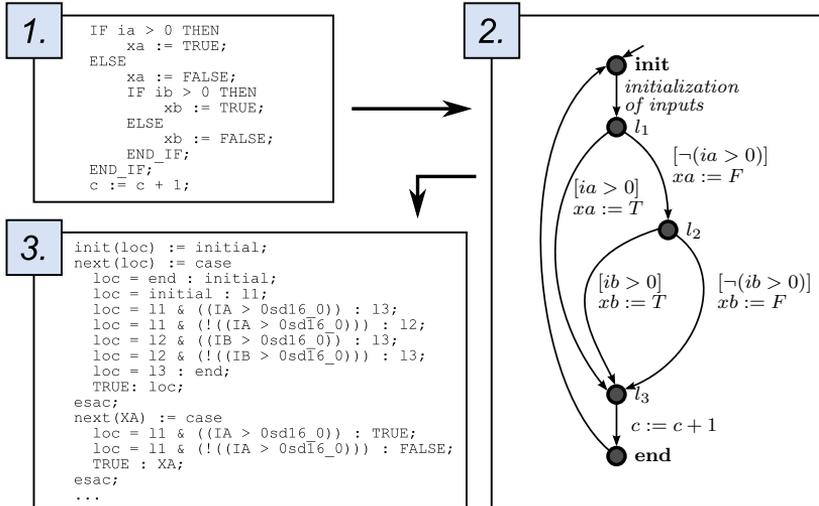


Figure 3.18: Example for control flow graph representation of an ST code in NuSMV

The input language of NuSMV represents a Mealy machine, whose states comprise the set of possible valuations of every defined state variable Cavada et al. (2014, 2011). In order to create a NuSMV model corresponding to a control flow graph, a natural way of modeling is to express every variable with a dedicated state variable, and to represent the current location in the CFG with an additional state variable (called *loc* in the methodology). This variable *loc* encodes the structure of the CFG along with the guard expressions on its transitions. All the details about the transformation from IM to nuXmv models are described in Section 3.8. The example on Fig. 3.18 shows an extract from a simple ST code, the corresponding CFG and its NuSMV representation.

The COI algorithm of NuSMV has no public documentation. However, as this tool is released under the LGPL license, it is possible to analyze its source code.

At the creation of the NuSMV's internal COI model, the dependencies of each variable are computed (function `coiInit`). A variable *v* depends on every variable *v'* used for the computation of *v* (e.g. if the next state of *v* is defined with a case block in the NuSMV model, all the variables and values will be in the set of dependencies.) Then, a transitive closure is computed for all the variables occurring in the

formula to be evaluated. If a variable  $x$  is necessary for some reason, all the variables presented in the assignment block of  $x$  will be necessary too. Therefore, it is trivial that according to the NuSMV implementation, the variable  $loc$  is necessary for a variable  $v$ , if it is assigned at least once in the CFG, because this assignment is guarded by a location of the CFG. As  $loc$  is necessary, all the variables in its next-state relation definition will be necessary too, which means all variables taking place in guards are necessary. Thus, none of the variables used in guards can be eliminated by the COI of NuSMV.

### 3.7.1.2 COI algorithm for the IM

It has been observed that it would be possible to reduce the models by removing conditional branches that do not affect the variables in the requirement. In the example shown in Fig. 3.18, none of the conditional branches is necessary if only variable  $c$  is used in the requirement, therefore the variables  $ia$  and  $ib$  could be eliminated as well.

Guards can affect the control flow, therefore it is not possible to eliminate all the guards. Here the heuristic is proposed, which provides good results when applying it to the IM. The idea is that there are states (locations) in the CFG that are “unconditional”, i.e. all possible executions go through them.

This COI variant consists of three steps (formally: see Algorithm 1):

1. Identification of unconditional states.
2. Identification of variables influencing the evaluation of the given requirement.
3. Elimination of non-necessary variables.

In the following part of this section, these three steps are introduced.

**Identification of Unconditional States.** The first step of our COI reduction is to identify the set  $\mathcal{L}$  of unconditional states. To ensure the correctness of the algorithm, it is necessary that  $\mathcal{L}$  does not include conditional states.

To identify the unconditional states, we defined a measure for the states. A *trace* is a list of states  $(init, l_1, l_2, \dots, l_n, end)$ , where *init*

---

**Algorithm 1:** ConeOfInfluence

---

**input** :  $\mathcal{M}$  : network of automata,  $Q$ : requirement**output**: true, iff COI modified the model $\mathcal{L} \leftarrow \text{UnconditionalStates}(\mathcal{M});$  $D \leftarrow \text{NecessaryVariables}(\mathcal{M}, \mathcal{L}, Q);$ Removal of all variables in  $V \setminus D$ , with their guards and assignments;**return**  $V \setminus D \neq \emptyset;$ 

---

is the initial state of the automaton, *end* is the end state of the automaton, and there is a transition between each  $(l_i, l_{i+1})$  state pairs. Let  $F(l)$  be the fraction of possible traces<sup>3</sup> going through a state  $l$ . It is known that all the traces go trough the initial state *init*, therefore  $F(\text{init}) = 1$ . For the rest of the states,  $F$  can be calculated based on the incoming transitions.

Let  $I_T(l) \in 2^T$  depict the set of incoming transitions to state  $l$ , and mark the set of outgoing transitions from state  $l$  as  $O_T(l) \in 2^T$ . Let  $I_L(t) \in L$  be the source state of transition  $t$ . With this notation, the formal definition of  $F(l)$  is the following:

$$F(l) := \begin{cases} 1 & \text{if } l = \text{init}, \\ \sum_{t \in I_T(l)} \frac{F(I_L(t))}{|O_T(I_L(t))|} & \text{otherwise} \end{cases} . \quad (3.1)$$

After calculating  $F$  for each state, it is easy to compute the set of unconditional states:  $\mathcal{L} = \{l \in L : F(l) = 1\}$ .

Notice, that the intermediate model can contain loops (beside the main loop representing the PLC cycle) due to loops in the input model. In this case, we handle all states in the loop as conditional states (without computing  $F$  for them).

**Identification of Necessary Variables.** The goal of the second step is to collect automatically all the variables necessary to evaluate the given requirement (see Algorithm 2). Let  $D$  be the set of necessary variables. It is trivial that every variable in the requirement should be

---

<sup>3</sup>Here we do not consider cycles inside the CFG beside of the cycle corresponding to the main cycle.

in  $D$ . After that, for each variable assignments that modify a variable in  $D$ , the variables in the assignment will also be added to the set  $D$ . Furthermore, the guard dependencies should also be added to  $D$ , i.e. all the variables that can affect the execution of the analyzed variable assignment. If the set of  $D$  grew, all the assignments should be checked again.

In the following, we define a function  $\mathcal{A}_T: T \rightarrow 2^V$ , which gives all the variables that can affect the execution of variable assignments on transition  $t$ .

First, a supporting function  $\mathcal{A}_L: L \rightarrow 2^V$  is defined which gives all the variables necessary to determine if state  $l$  will be active or not during an execution. This function can benefit from the previously explored unconditional states  $\mathcal{L}$ , as it is known that no variable is necessary to decide if they can be activated during an execution. Thus, for every state  $l \in L$ , the function  $\mathcal{A}_L(l)$  is the following:

$$\mathcal{A}_L(l) := \begin{cases} \emptyset & \text{if } l \in \mathcal{L} \\ \bigcup_{t \in I_T(l)} \mathcal{A}_T(t) & \text{if } l \notin \mathcal{L} \end{cases} . \quad (3.2)$$

It means that for the unconditional states, the set of affecting variables is empty, as a consequence of their definition. If a state is not unconditional, the set of variables affecting that this state is active or not is the set of variables affecting the firing of all its incoming transitions.

For every transition  $t \in T$ , the function  $\mathcal{A}_T(t)$  is the following:

$$\mathcal{A}_T(t) := \mathcal{A}_L L(I_L(t)) \cup \bigcup_{t' \in O_T(I_L(t))} \{\text{variables in guard of } t'\}. \quad (3.3)$$

It means that the firing of transition  $t$  is affected by the variables that influences its source state ( $I_L(t)$ ) and by the variables in the guard of  $t$ . Furthermore, it is also influenced by the variables used in the guard of transitions that can be in conflict with  $t$ , i.e. the guard variables of  $t' \in O_T(I_L(t))$ .

**Elimination of Non-necessary Variables.** In the second step, the set  $D$  of necessary variables is determined. In the last step, all the variables not in  $D$  should be deleted. Also, to ensure the syntactical

---

**Algorithm 2:** NecessaryVariables

---

**input** :  $\mathcal{M}$  : network of automata,  $\mathcal{L}$  : set of unconditional states,  
 $Q$ : requirement  
**output**:  $D$  : set of necessary variables

$D \leftarrow \{\text{variables in } Q\};$   
**repeat**  
    **foreach** variable assignment  $\langle v := Expr \rangle$  on transition  $t$  **do**  
        **if**  $v \in D$  **then**  
            // adding assignment and guard dependencies  
             $D \leftarrow D \cup \{\text{variables in } Expr\} \cup \mathcal{A}_T(t);$   
**until**  $D$  is not changed;  
**return**  $D;$

---

correctness, all the guards and variable assignments containing variables  $v \notin D$  should be deleted. They do not affect the evaluation of the requirement, otherwise they should be in set  $D$ .

**Difference between our COI and the COI of NuSMV.** The main difference between our COI and the COI implementation of NuSMV is in the handling of conditional branches. If there is a conditional branch in the CFG, but the variables in the requirement are not affected by this choice, there is no need for the variables in its guard. This difference can be observed in Fig. 3.19, which shows the CFG introduced in Fig. 3.18, after applying the different COIs, and assuming that only variable  $c$  is used in the requirement (e.g.  $EFc < 0$ ). The COI of NuSMV can identify that the variables  $xa$  and  $xb$  are not used to compute  $c$ , therefore they will be removed. But the variables  $ia$  and  $ib$  are used in guards, thus they are kept (see Fig. 3.19(a)).

Our COI algorithm can detect that because  $xa$  and  $xb$  are deleted, the  $ia$  and  $ib$  variables can be deleted too, because those guards do not affect  $c$ , i.e. no matter which computation path is executed between locations  $l_1$  and  $l_3$ , the assignment of variable  $c$  will be the same (see Fig. 3.19(b)). The resulting IM after applying our COI algorithm is shown in Fig. 3.19(c).

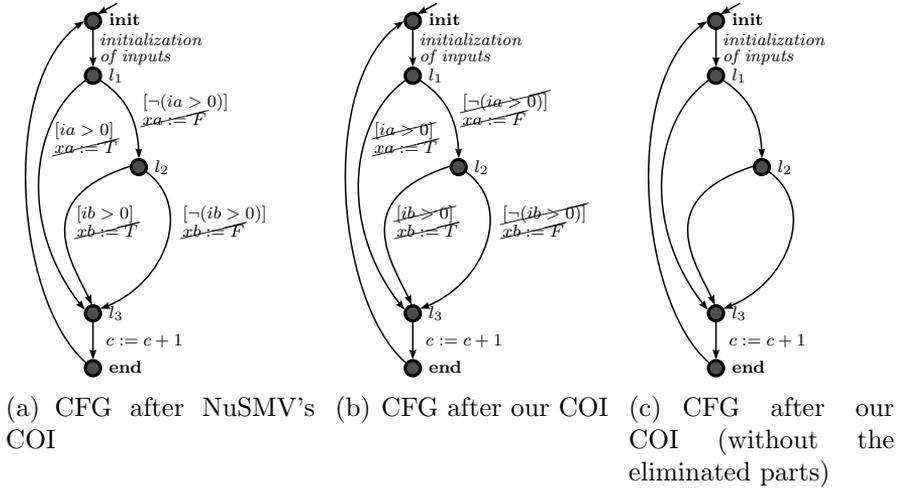


Figure 3.19: Example comparing our and the NuSMV's cone of influence algorithm from Darvas et al. (2014)

### 3.7.2 Rule-based reductions

The COI applied to the IM only eliminates variables, the connected variable assignments and guards, without modifying the structure of the CFG. Therefore, the resulting model of the COI often contains empty transitions, unnecessary states, etc. For these situations, rule-based reductions have been implemented that make the model smaller and easier to verify.

This kind of CFG reductions are not new, they are used for example in numerous compilers to simplify the machine code and to improve the performance. More details can be found in Cooper and Torczon (2012).

The rule-based reductions applied to the IM can be classified as follows:

- **Model simplifications.** The aim of these reductions is to simplify the intermediate model without reducing its potential state space (thus without eliminating states or variables). This group includes elimination of unnecessary variable assignments or transitions, and simplification of logical expressions. These reductions do not reduce the model themselves, we created them

to support the other reduction techniques. For example, if the model contains an empty conditional branch (due to for example another reduction), it can be removed without changing the meaning of the model. This rule is illustrated in Fig. 3.20(a).

- **Model reductions.** These methods can reduce the potential state space by eliminating states and variables. This group includes heuristics to merge transitions or states, to eliminate variables that have always the same constant value, etc. For example, if a transition has no guard, no synchronization, and no variable assignment (which can be a side effect of the COI reduction), then the two states on the end of the transition can be merged and the transition can be deleted. This rule is illustrated in Fig. 3.20(b).
- **Domain-specific reductions.** These reductions are benefited by the PLC domain knowledge using the assumption introduced at the beginning of Section 3.7, that variables are only checked at the end of the PLC cycle. Two main domain-specific reductions are developed:
  - *Transition merging.* Two consecutive transitions can be merged if they represent a sequence in the CFG and their variable assignments are not in conflict, i.e. their order does not affect each other's value. Formally,  $v_a := E_a$  and  $v_b := E_b$  are not in conflict if  $v_a \notin \text{Vars}(E_b)$ ,  $v_b \notin \text{Vars}(E_a)$ , and  $v_a \neq v_b$ , where  $\text{Vars}(E)$  means all variables used in expression  $E$ . This can easily be extended to two set of variable assignments, thus it can be reapplied to merged transitions. Fig. 3.20(c) illustrates this reduction rule.
  - *Variable merging.* Two variables can be merged if they always have the same value at the end of the PLC cycles. While creating two variables for the same purpose can be considered as a bad programming practice, it is a common pattern in the source code of our systems. This feature is used to improve code readability by assigning a meaningful name to a bit of an input array (for example `PFailSafePosition` instead of `ParReg01[8]`).

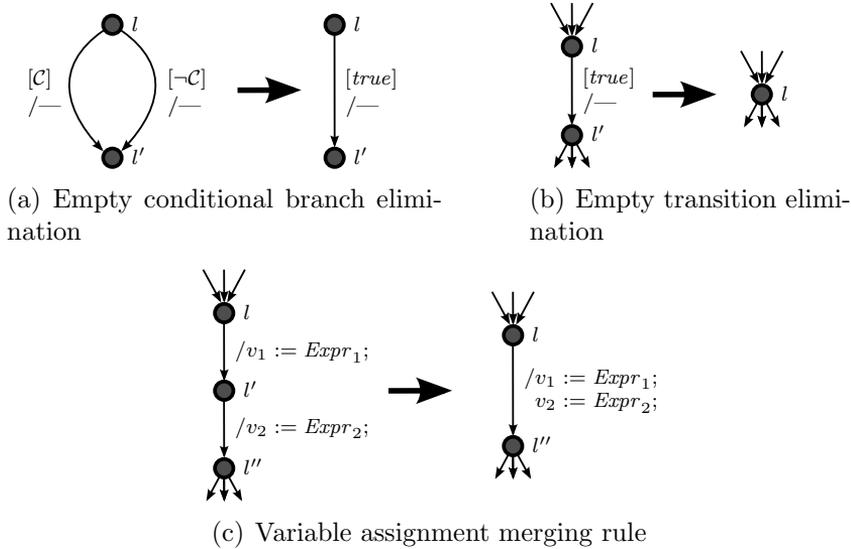


Figure 3.20: Example reduction rules from Darvas et al. (2014)

It has to be noted that not all the previously mentioned reduction techniques are property preserving in themselves. For instance, if two variables  $v$  and  $w$  are merged, thus  $w$  will be deleted, the properties containing  $w$  are not possible to evaluate. For this reason, we generate a mapping throughout the reductions that contains all the variable substitutions. Based on this mapping, the necessary aliases can be generated for the NuSMV model. For example, an alias  $w$  can be added to the model having always the same value as  $v$ , as it is known that they always contain the same value at the end of the PLC cycles. Aliases do not modify the size of the state space or the complexity of the model checking.

As discussed at the beginning of Section 3.7.2, the COI algorithm can enable the rule-based reductions. Therefore, after applying the COI algorithm, all the reduction techniques introduced in this section will be applied, which can enable other reductions or other possible variable removal for the COI algorithm. Therefore we implemented the reductions in an iterative manner. First, the COI is executed. Then, all the possible reductions are applied. If the COI or one of the rule-based reductions was able to reduce the model, all the reductions are executed again. This iterative workflow is described formally in

Algorithm 3. (The function  $\text{Reduce}(r, \mathcal{M})$  applies reduction  $r$  on the model  $\mathcal{M}$  and returns true, iff the reduction modified the model.)

A simple example is shown here illustrating our reduction workflow. If our COI is applied to the Fig. 3.18 CFG example, then Fig. 3.19(c) CFG will be obtained. If the reductions presented on Fig. 3.20(a) and Fig. 3.20(b) are applied on it, the result will be the simple Fig. 3.21 CFG.

---

**Algorithm 3: Reductions**

---

```

input :  $\mathcal{M}$  : model,  $Q$ :
           requirement

bool changed;
repeat
  | changed  $\leftarrow$ 
  |  $\text{ConeOfInfluence}(\mathcal{M}, Q)$ ;
  | foreach
  |  $r \in \{\text{rule-based reductions}\}$  do
  | | changed  $\leftarrow$ 
  | |  $\text{Reduce}(r, \mathcal{M}) \vee \text{changed}$ ;
until changed = false;

```

---

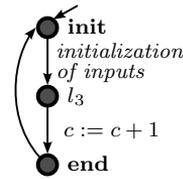


Figure 3.21: Example CFG after our COI and the reductions from Darvas et al. (2014)

### 3.7.3 Mode selection

Good practices and standards in PLC program development (e.g. IEC 61499 (2013)) suggest to have a library of Function Blocks, which are the basic building modules of the control systems. These modules can be reused in different PLC programs. For example, at CERN, the UNICOS framework provides a library of objects where the elements from the instrumentation can be represented. In the PLC code, these objects are Function Blocks and they are generic so can be adapted for the specific uses by setting some parameters. Parameters are input variables that are constant during the execution, as they are hard-coded into the PLC application.

A significant part of our real requirements have assumptions on the parameters, like:

“if parameter  $p_1$  is true, then it is always true that ...”.

For these requirements, general models can be used and the parameter configuration can be included in the temporal logic formula, or by adding invariants to the model. However, better performance can be achieved if the configuration is encoded in the model itself by replacing the parameter with its constant value. This way, the developers can select the operational mode of the object on which the verification should be performed, i.e. they can fine-tune the verification to their needs. This method is applied only once, before all the other reductions.

The advantages of this method are the following:

- There is no need to create a variable for the fixed parameter as it is substituted with a constant.
- The rule-based reductions can simplify the expressions. For example, if there is a guard  $[p_1 \wedge p_2 \wedge p_3]$  in the CFG and  $p_1$  is fixed to false, the guard will be always false and can be replaced by a constant false.
- The simplification of the expressions can help the cone of influence reduction to remove the unnecessary variables, e.g. if  $p_2$  and  $p_3$  are only used in the guard  $[p_1 \wedge p_2 \wedge p_3]$ , and  $p_1$  is fixed to false, the variables  $p_2$  and  $p_3$  can be removed. Using the COI in NuSMV or if the fixed parameter is expressed through invariants, this reduction cannot be done.

Now the complete reduction workflow can be summarized. After the transformation of the source code into the intermediate model, the given parameters are fixed to the required values. Then the cone of influence reduction and the rule-based reductions are performed iteratively (as seen in Fig. 3.22). It is important to note, that since every reduction rule deletes something from the model upon firing, they cannot “reduce” the model infinite times, thus the iteration cycle will stop in finite time.

### 3.7.4 Iterative variable abstraction

The effectiveness of the previous reduction techniques highly depends on the property specification. In some cases, these reductions can be very effective but in some others they are not sufficient for the

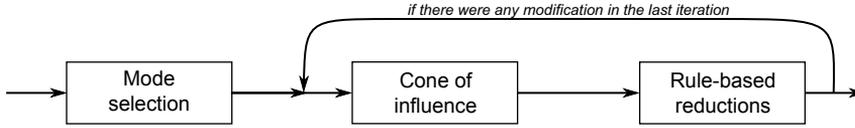


Figure 3.22: Overview of the reduction workflow

verification tools, for example when the reduced model contains still a lot of variables or the data types of these variables are not just Boolean. In these cases, the state space of the reduced model is still too large to be verified by any tool, for example in most of the verification cases of complete UNICOS PLC programs (Chapter 4 presents one of these programs).

For that reason, a new method called *variable abstraction* is included in the methodology. Using this technique, abstract models are created by replacing some of the variables by non-deterministic values, transforming these variables in “input variables”, which were described in Section 3.5. As these new input variables do not depend on any other variables, the COI algorithm will be able to eliminate more variables from the model automatically.

The resulting model, after this abstraction, is an over-approximation of the original model, which contains less variables than the original model and it represents a bigger range of possible behaviors of a system. This means that the abstract model has some states (combination of variable values) that do not belong to the original model and therefore to the real PLC program.

Before describing the method, a few basic concepts are discussed. This technique is currently limited to the following safety properties:  $\text{AG}(\alpha \rightarrow \beta)$ . This is because the requirements for the verification of complete UNICOS PLC program are currently safety properties. By restricting the possibilities in the property specification, a more aggressive abstraction technique can be applied and we increase the chances of getting a verification result from the model checker. This is the reason of including this restriction.

In this technique, abstract models are created using the variable dependency graph of the original model (which corresponds with the PLC program).

Consider an abstract model  $AM'_n$  and a safety property  $p$  ( $\text{AG}(\alpha \rightarrow \beta)$ ). When applying model checking, if  $p$  holds on  $AM'_n$ , it implies that

$p$  also holds in the original model  $OM'$  as a bigger range of possible behaviors is explored. If  $p$  does not hold on  $AM'_n$ , a counterexample  $c$  is generated. To determine if  $p$  holds on  $OM'$ , it is needed to analyze if  $c$  is a real or a spurious counterexample. If it is a spurious counterexample,  $AM'_n$  needs to be refined.

Fig. 3.23 represents this idea. The violet shading represents all the possible models where  $p$  holds. In other words, it represents all the possible behaviors where the property  $p$  is respected. Note that  $AM'_n$  is an over-approximation of  $OM'$ .  $AM'_n$  has a state space size smaller than  $OM'$  (as it contains less variables) but it represents a bigger range of possible behaviors.

In Fig. 3.23(a),  $p$  holds in  $AM'_n$  and therefore it holds in  $OM'$ . In Fig. 3.23(b) and Fig. 3.23(c),  $p$  does not hold in  $AM'_n$  and it is not possible to determine if  $p$  holds on  $OM'$  without analyzing the counterexample.

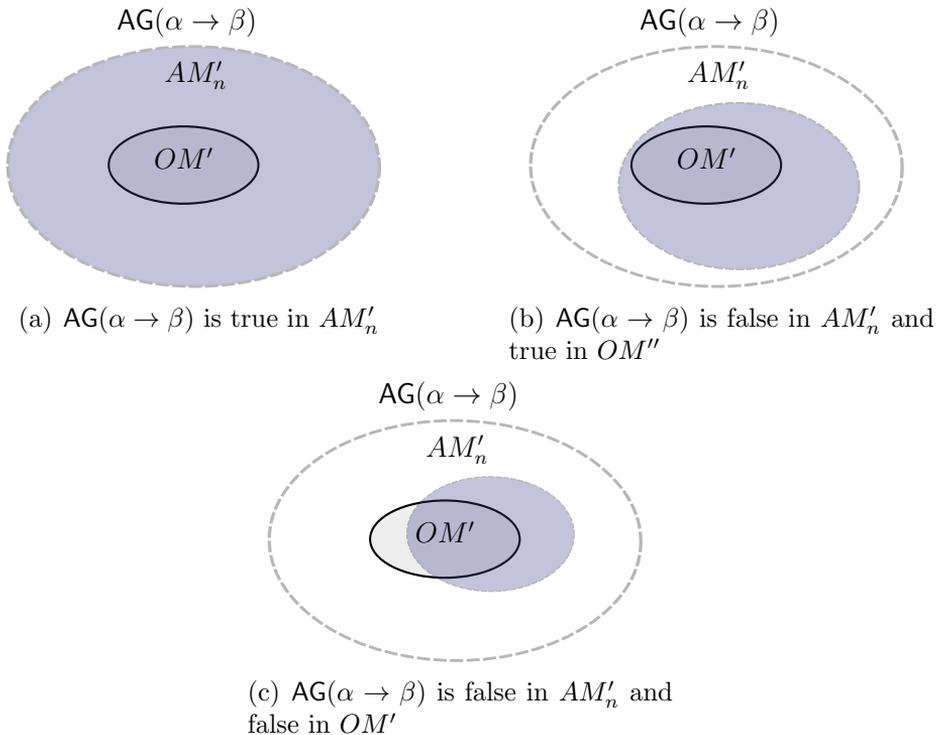


Figure 3.23: Verification cases for  $AG(\alpha \rightarrow \beta)$  in abstract models

In order to determine if this counterexample is real or spurious,

one possibility is to transform this counterexample in a reachability property ( $\text{EF}(\gamma \& \theta)$ ). Regarding the verification of a reachability property  $r$  in an abstract model  $AM''_n$ , the logic is different. If  $r$  does not hold in  $AM''_n$ , then  $r$  does not hold in the original model ( $OM''$ ). But if  $r$  holds on  $AM''_n$ , we cannot determine if  $r$  holds on  $OM''$  or not.

Fig. 3.24 describes this idea. In this case, the black dot represent the state where the property  $r$  holds.

In Fig. 3.24(a),  $r$  does not hold on  $AM''_n$  and therefore it does not hold in  $OM''$ . In Fig. 3.24(b) and Fig. 3.24(c),  $r$  holds in  $AM''_n$  and it is not possible to determine if  $r$  holds on  $OM''$ .

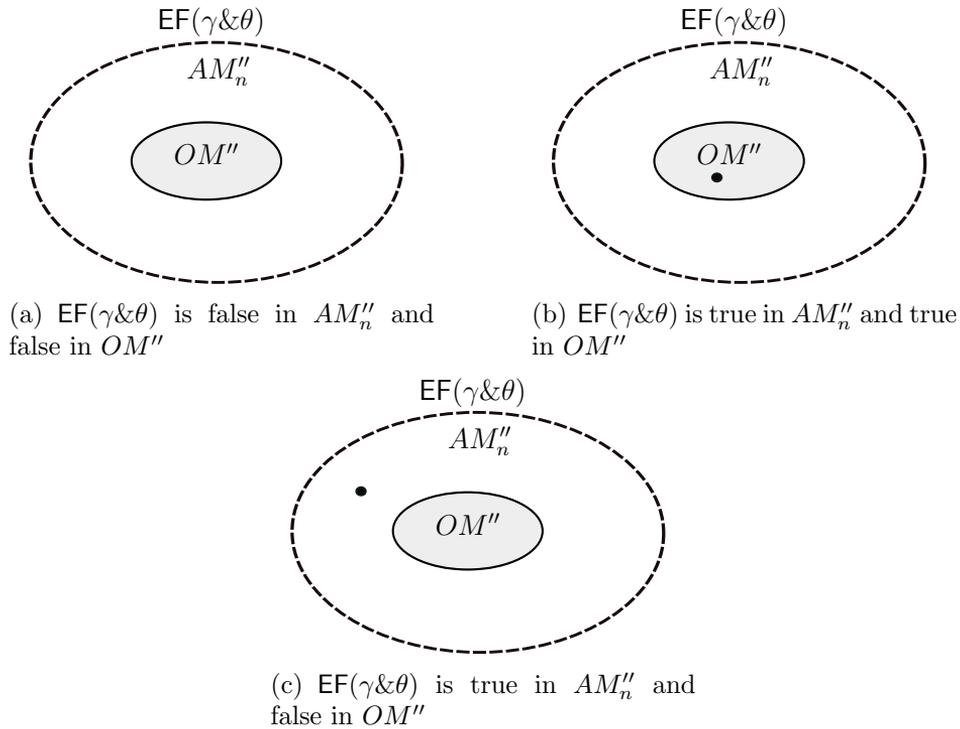


Figure 3.24: Verification cases for  $\text{EF}(\gamma \& \theta)$  in abstract models

These concepts are applied in the variable abstraction technique.

The variable dependency graph is provided by the COI algorithm (defined in 3.7.1) and it gives all the necessary variables  $D$  for a given requirement. Fig. 3.25 shows an example of the variable dependency graph in a model from a real-life PLC program. The gray variables are part of the requirement, the red edges represent assignment depen-

dencies and the blue edges represent conditional dependencies. Two different variables connected by an edge are in a distance equals to 1. The concept of distance between variables is used to build the abstract models.

### 3.7.4.1 General description of the variable abstraction technique

This technique is represented in Fig. 3.26 and as well as the previous reduction techniques, it is applied to the IM.

The basic idea of the algorithm is the following: The safety property ( $p$ ) is verified in an abstract model ( $AM'_n$ ), where  $n$  corresponds to the number of iterations of the algorithm. If  $p$  holds on  $AM'_n$ , the algorithm is over and we can conclude that  $p$  holds on the original model ( $OM'$ ). If  $p$  does not hold on  $AM'_n$ , a counterexample ( $c$ ) is provided by the model checker but we need to determine if  $c$  is a spurious counterexample due to the abstraction or it is real. To do so, two steps are performed: first a new safety property ( $q$ ) is checked on  $AM'_n$  with the goal of obtaining more information about the state space of  $AM'_n$  where  $p$  holds. Then, if no extra information is provided,  $c$  is transformed into a reachability property ( $r$ ) to prove or discard that  $c$  is real. If  $c$  is real, it means that  $p$  does not hold on  $OM'$ . If  $c$  is spurious, an invariant is added to  $AM'_n$  or a new abstraction of  $AM'_n$  (a more refined model) is created if the number of potential invariants ( $m$ ) is not worthwhile to be analyzed. Then the process is repeated.

The algorithm can be divided then in five steps (see Fig. 3.27), regarding the five decision points of the flow diagram (the initial one is not considered, as it is the input to this algorithm: a time out is obtained when verifying  $p$  on the original model  $OM'$ ):

1. Checking the original safety property ( $p$ ) on an abstract model ( $AM'_n$ ).
2. Checking the safety property ( $q$ ) on the same abstract model ( $AM'_n$ ).
3. Checking the reachability property ( $r$ ) on the original model ( $OM''$ ).

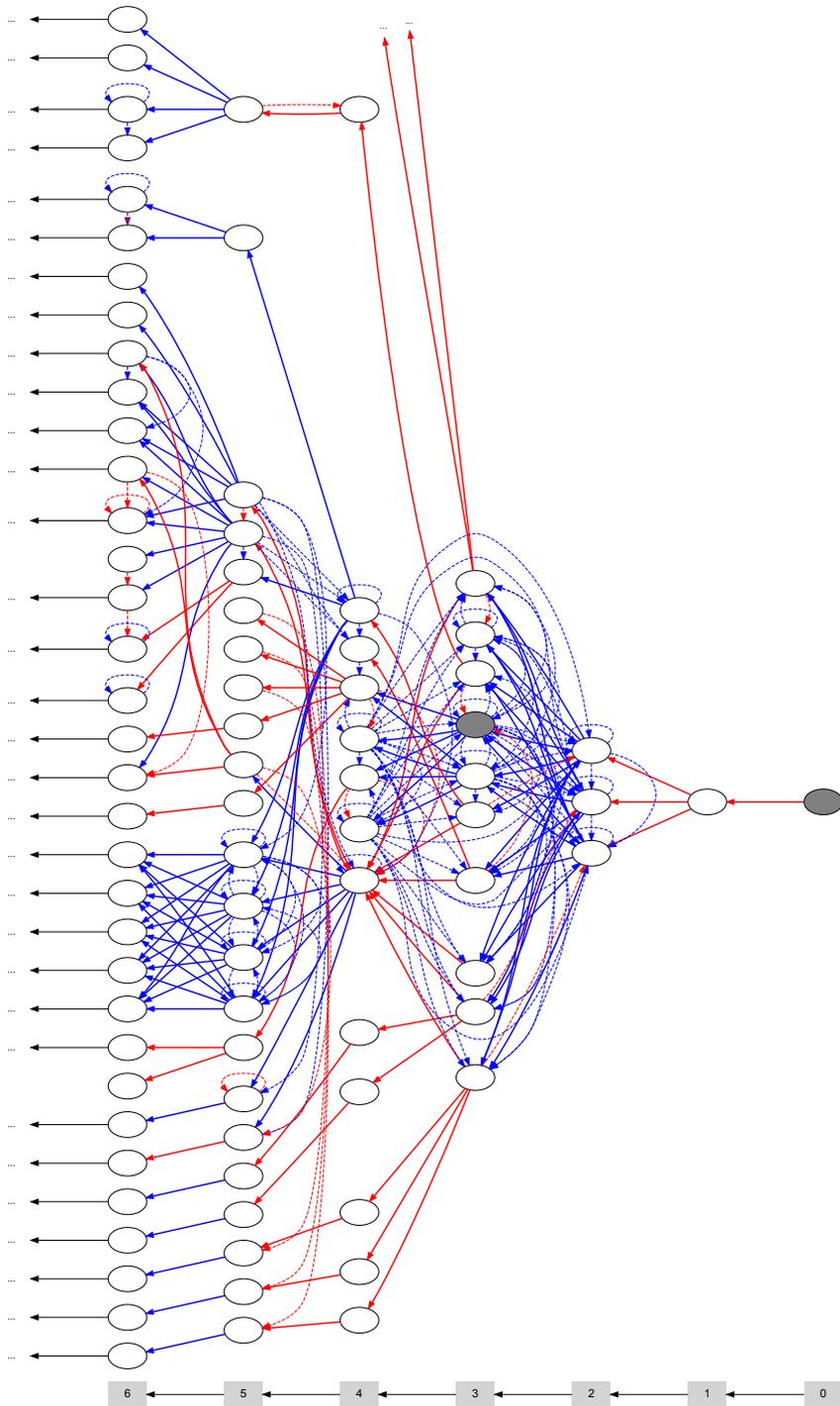


Figure 3.25: Variable dependency graph example

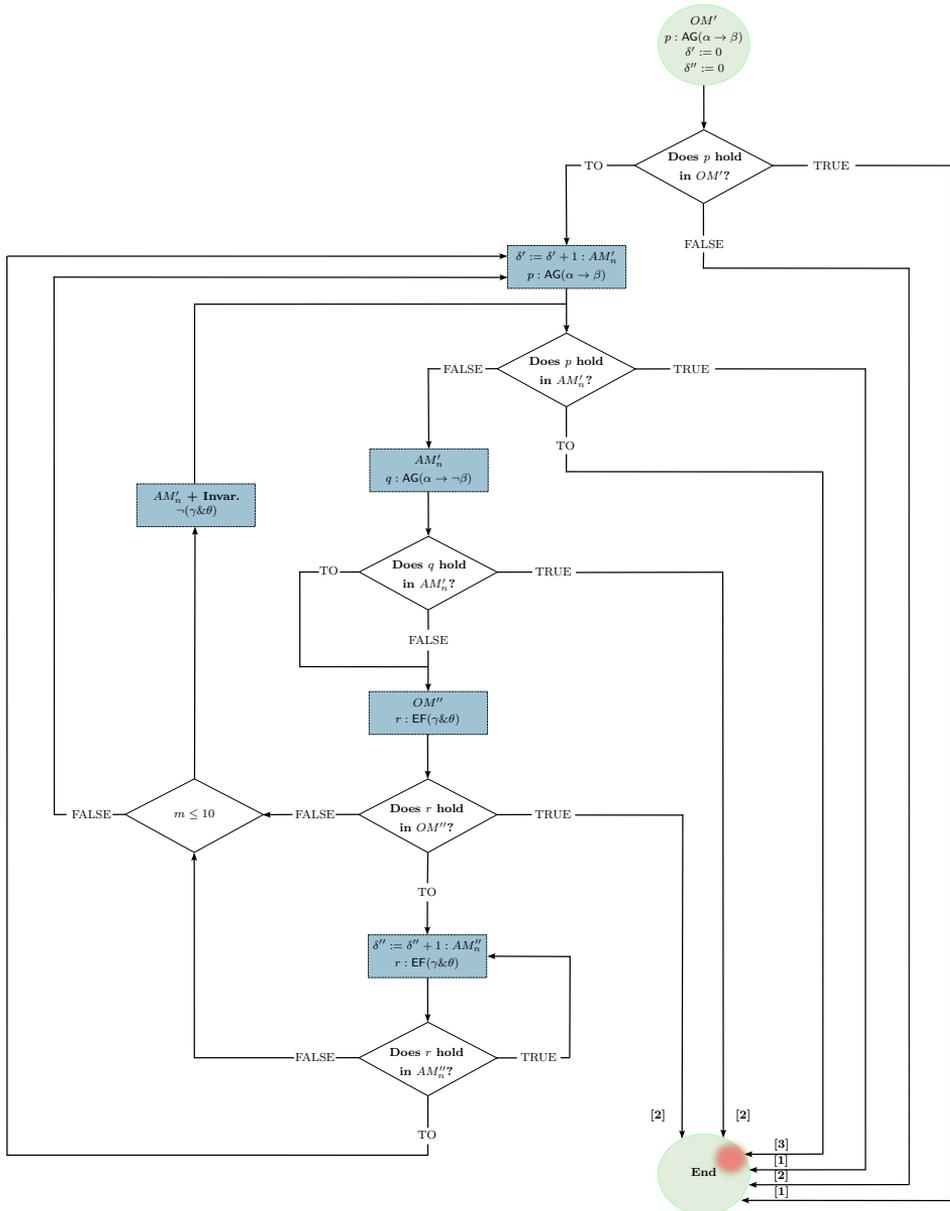


Figure 3.26: Variable Abstraction Flow Diagram. Verification results: [1]  $p$  holds on  $OM'$ ; [2]  $p$  does not hold on  $OM'$ ; [3] no answer.

4. Checking the reachability property ( $r$ ) on an abstract model ( $AM''_n$ ).
5. Extracting an invariant from the counterexample obtained in step 2 if  $m$  is  $\leq 10$ .

The following paragraphs describe the details of the five steps of this technique.

### 3.7.4.2 Checking the original safety property ( $p$ ) on an abstract model ( $AM'_n$ )

The algorithm starts by checking the safety property  $p$  (i.e.  $\text{AG}(\alpha \rightarrow \beta)$ ) on the original model  $OM'$ .  $OM'$  is the result of applying the property preserving reductions (described earlier in this section) to the generated model from the PLC code. In addition, two parameters ( $\delta'$  and  $\delta''$ ) are initialized to 0.

$\delta'$  represents the distance between variables in the variable dependency graph for  $OM'$ .  $\delta''$  represents the distance between variables in the variable dependency graph for  $OM''$ .

$OM'$  is the resulting model of applying the property preserving reduction techniques for the original safety property  $p$ .  $OM''$  is the resulting model of applying the property preserving reduction techniques for the reachability property  $r$ .

Abstract models ( $AM'_n$  and  $AM''_n$ ) are created automatically using  $\delta'$  and  $\delta''$  respectively.

$TO$  (Time Out) is a parameter set at the beginning of this technique, which represents the maximum amount of time given to the model checker to provide an answer (true or false) for the verification.

In the first iteration, if the model checker cannot provide an answer when verifying  $p$  on  $OM'$  and the  $TO$  is reached, an abstract model is automatically generated using  $\delta' = 1$ , i.e.  $AM'_1$ . Then the model checker verifies if  $p$  holds on  $AM'_1$ . Three possible answer can be provided by the model checker:

1. *True*:  $p$  holds on  $AM'_n$ , therefore the algorithm is over and we can conclude that  $p$  holds on  $OM'$  (as it was described in Fig. 3.23).
2. *TO*: the algorithm is over and we cannot prove whether  $p$  holds or not on  $OM'$  (this situation should never happen on the first

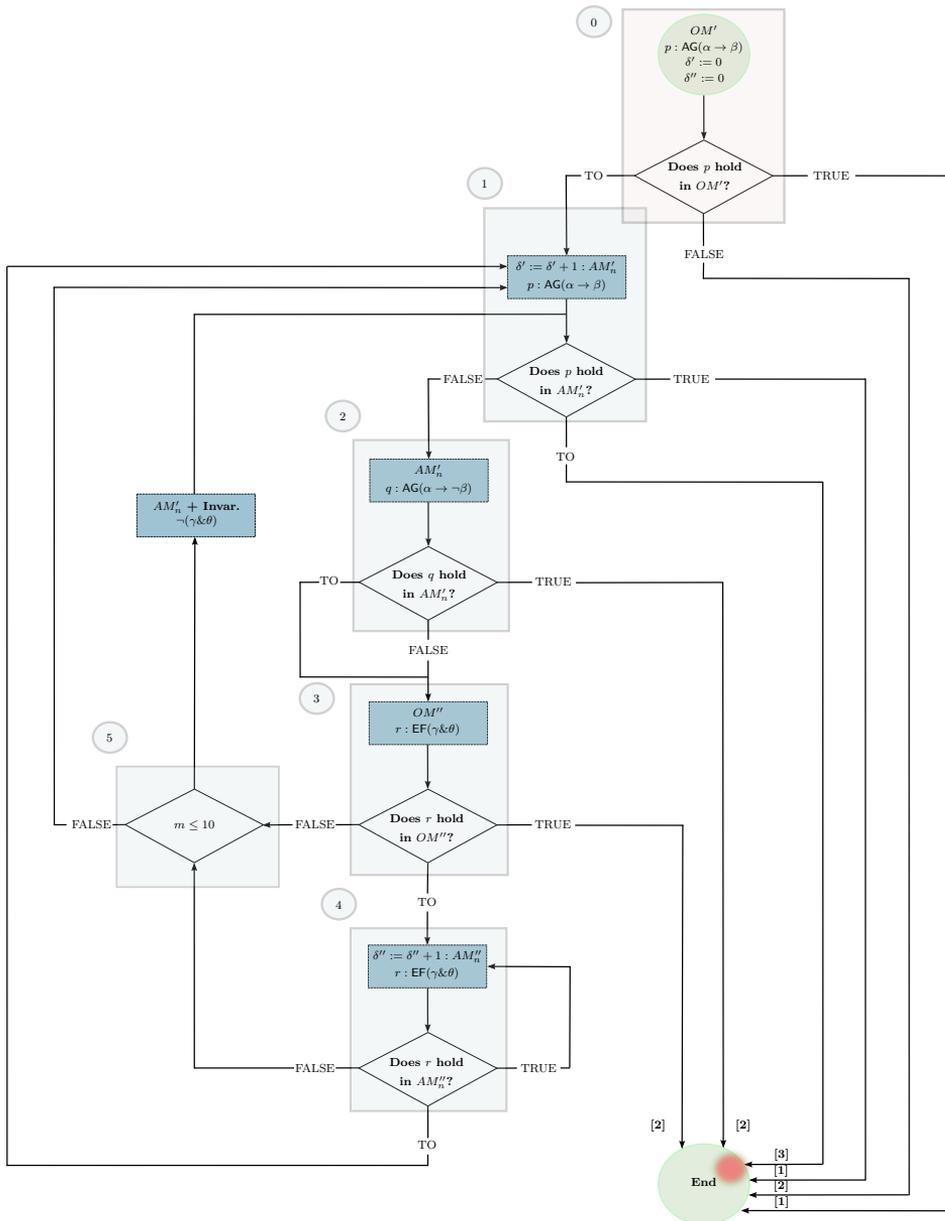


Figure 3.27: Variable Abstraction Flow Diagram. Verification results: [1]  $p$  holds on  $OM'$ ; [2]  $p$  does not hold on  $OM'$ ; [3] no answer.

iterations of the algorithm, e.g.  $AM'_1$  and  $AM'_2$ , as they usually have a much smaller state space size when compared to  $OM'$ ). This is the limitation of the methodology, it is not possible to determine if the generated counterexamples from previous abstractions ( $AM_{n-1}, AM_{n-2}, \dots, AM_1$ ) were real or spurious.

3. *False*:  $p$  does not hold on  $AM'_n$ , and the algorithm moves to the step 2 in order to prove that the provided counterexample ( $c$ ) is real or spurious (as it was described in Fig. 3.23).

The abstract models are built applying the following logic: Having  $OM'$  and  $p$  ( $\text{AG}(\alpha \rightarrow \beta)$ ), a set of variables from  $OM'$  are converted to “input variables”. The idea here is that all variables with a distance ( $d$ ) equals to  $\delta'$  from the variable  $\beta$  are transformed to “input variables”.  $\alpha$  is also transformed to an “input variable”, if the distance of  $\alpha$  with respect of  $\beta$  is  $d \geq \delta'$ .

The initial value of  $\delta'$  is 0 and incremented to 1 in the first iteration ( $n = 1$ ). This means that in the first iteration the resulting abstract model ( $AM'_1$ ) is the most abstract model possible. In the following iterations, depending on the decisions of the steps 3 and 5,  $\delta'$  is increased ( $\delta' := \delta' + 1$ ) or is not modified.

The algorithm 4 summarizes the proposed heuristic.

---

**Algorithm 4:** ModelAbstraction
 

---

**input** :  $D'_g$ : dependency graph of the  $OM'$ ,  $p$ : requirement,  $\delta'$ : distance

**output**:  $AM'_n$ : abstract model

$\mathcal{V} \leftarrow \text{AbstractedVariables}$ ;

Identify the variables to be abstracted  $V$ :  $d = \delta'$  from  $\beta$ .  $\alpha$  is also included in  $V$  (if  $\alpha$  is a distance  $d \geq \delta'$ );

Removal of all the assignments of the abstracted variables  $V$ ;

Replace all the abstracted variables  $V$ , by non-deterministic values;

**return**  $AM'_n$ ;

---

### 3.7.4.3 Checking the safety property ( $q$ ) on the same abstract model ( $AM'_n$ )

This step is included in the algorithm to extract more information about the verification of  $p$  on  $AM'_n$  before analyzing the counterexample  $c$ . The idea is to check the property  $q$ :  $\mathbf{AG}(\alpha \rightarrow \neg\beta)$ , on the same abstract model ( $AM'_n$ ) as  $p$ .  $q$  only differs from  $p$  on the negation of  $\beta$ .

Verifying the property  $q$  on  $AM'_n$  has a similar complexity as verifying  $p$  on  $AM'_n$ , because the model is the same and the property has the same temporal operator ( $\mathbf{AG}$ ) and the same variables. Therefore a  $TO$  should never happen in this step if it did not happen with  $p$  in the previous step. However this possibility is also considered on the algorithm. Therefore three possible answers can be provided by the model checker:

1. *True*:  $q$  holds on  $AM'_n$  and the algorithm is over. We can conclude that  $p$  does not hold on  $OM'$  and the provided counterexample in the previous step ( $c$ ) is real.
2. *False*:  $q$  does not hold on  $AM'_n$  and the algorithm moves to the step 3 in order to prove that the provided counterexample ( $c$ ) is real or spurious by transforming  $c$  in a reachability problem.
3. *TO*: no further information is provided and the algorithm moves to the step 3 in order to prove that the provided counterexample ( $c$ ) is real or spurious by transforming  $c$  in a reachability problem.

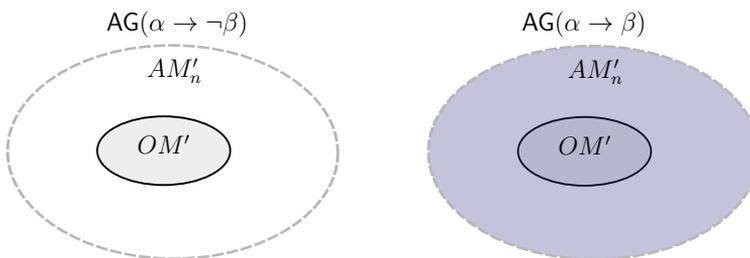
Fig. 3.28 describes graphically the different verification cases that may occur in this step. Again, the violet shading represents all the possible models or state spaces where the properties  $q$  or  $p$  hold.

In 3.28(a),  $q$  holds in  $AM'_n$  and therefore  $p$  does not hold on  $AM'_n$  and we can conclude that  $p$  does not hold on  $OM'$ . The counterexample  $c$ , provided in step 1, is real.

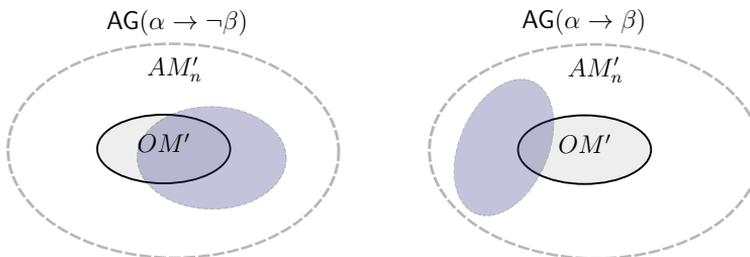
In 3.28(b), 3.28(c) and 3.28(d)  $q$  does not hold in  $AM'_n$  and it is not possible to determine if  $p$  holds on  $AM'_n$  and  $OM'$  without analyzing the counterexample.



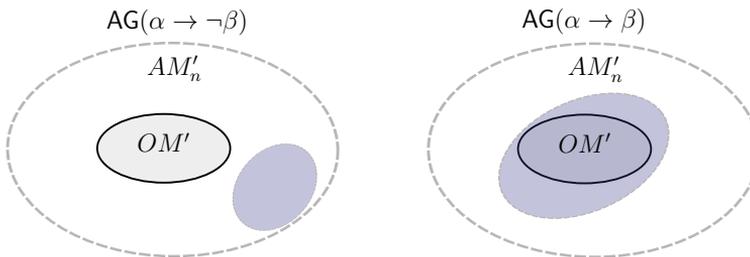
(a)  $\text{AG}(\alpha \rightarrow \neg\beta)$  is true in  $AM'_n$ , therefore  $\text{AG}(\alpha \rightarrow \beta)$  is false in  $AM'_n$  and  $OM'$



(b)  $\text{AG}(\alpha \rightarrow \neg\beta)$  is false in  $AM'_n$  and  $\text{AG}(\alpha \rightarrow \beta)$  is true in  $AM'_n$  and  $OM'$



(c)  $\text{AG}(\alpha \rightarrow \neg\beta)$  is false in  $AM'_n$  and  $\text{AG}(\alpha \rightarrow \beta)$  is false in  $AM'_n$  and  $OM'$



(d)  $\text{AG}(\alpha \rightarrow \neg\beta)$  is false in  $AM'_n$  and  $\text{AG}(\alpha \rightarrow \beta)$  is false in  $AM'_n$  but true in  $OM'$

Figure 3.28: Relationship between the verification cases for  $\text{AG}(\alpha \rightarrow \neg\beta)$  and  $\text{AG}(\alpha \rightarrow \beta)$  in abstract models

#### 3.7.4.4 Checking the reachability property ( $r$ ) on the original model ( $OM''$ )

The third step of the algorithm consists in extracting a reachability property  $r$  from the counterexample  $c$ . In addition, using the property preserving reduction techniques with this property, a new “original model”  $OM''$  is generated. Note that  $OM''$  and  $OM'$  may be different as the properties  $p$  and  $r$  contain different variables and therefore the reduction techniques, such as, COI will eliminate different variables from the model.

Initially  $r$  is verified on  $OM''$  and three possible answers can be provided by the model checker:

1. *True*:  $r$  holds on  $OM''$ , which means that  $c$  is real and therefore  $p$  does not hold on  $OM'$ .
2. *False*:  $r$  does not hold on  $OM''$ , which means that  $c$  is spurious and the algorithm moves to the step 5.
3. *TO*: an answer cannot be provided and the algorithm moves to the step 4.

It has to be noticed that the counterexample  $c$  can be complex (it contains the variable values in any location of the IM). However, as it was mentioned in Section 3.5, in PLC programs the relevant values are at the end of the PLC cycle. This assumption reduces the counterexample, as we need the value of the variables at the location *end* of the IM.

For the reachability property, it is only needed the “input variables”, as they are the ones that add more possible behaviors to the  $AM$ . For example, if a generated counterexample contains the “input variables”  $\gamma$  and  $\zeta$ , only these variables are part of the reachability property, i.e.  $EF(\gamma \& \zeta)$ .

In addition, even more complex counterexamples can be produced: counterexamples containing variable values from different PLC cycles. In this situation, several reachability property can be extracted. For example, if the counterexample has three cycles, three reachability properties checking the values of the variables at the end of each PLC cycle and two reachability properties checking the transitions between

the cycles could be extracted. If only one of the properties are not satisfied, this means the counterexample is spurious.

Potentially in step 5 of the algorithm, spurious counterexample can be added as invariants to  $AM'_n$  and the process is repeated. However, to the best of the author's knowledge, most of the model checkers only allow to add invariants without temporal operators (e.g. NuSMV), therefore the two reachability properties checking the transitions between the cycles cannot be extracted.

The strategy in this situation is that only one reachability property  $r$  is extracted, corresponding to the values of the transformed "input variables" at the end of the first PLC cycle. If when verifying  $r$  on  $OM''$  the result is false and in step 5 an invariant is added, in the next iterations of the algorithm new counterexamples will be generated and the same strategy is applied.

#### 3.7.4.5 Checking the reachability property ( $r$ ) on an abstract model ( $AM''_n$ )

When a  $TO$  is reached in the previous step, the property  $r$  is verified in an abstract model  $AM''_n$ . The strategy of producing abstract models here is the same as in the step 1 (algorithm 4), but now the inputs are  $D''_g$  (dependency graph of the  $OM''$ ),  $r$  and  $\delta''$  and the output  $AM''_n$ .

In the first iteration,  $AM''_1$  is automatically generated using a  $\delta'' = 1$ . Three possible answers can be provided by the model checker:

1. *False*:  $r$  does not hold on  $AM''_n$  and therefore it does not hold on  $OM''$  (as it was described in Fig. 3.24), which means that  $c$  is spurious and the algorithm moves to the step 5.
2. *True*:  $r$  holds on  $AM''_n$  but we cannot determine if it holds on  $OM''$  (as it was described in Fig. 3.24). In this case, a new abstract model  $AM''_n$  is created by incrementing  $\delta''$  ( $\delta'' = \delta'' + 1$ ), and the process is repeated.
3. *TO*: an answer cannot be provided and the algorithm moves to the step 1 and a new iteration of the algorithm is executed with a new abstract model  $AM''_n$  ( $\delta' := \delta' + 1$ ).

### 3.7.4.6 Extracting an invariant from the counterexample obtained in step 2 if $m$ is $\leq 10$

When the previous steps have determined that the counterexample  $c$  is spurious, this step takes the decision of adding a new invariant to the same abstract model  $AM'_n$  or moving to a new abstraction  $AM'_{n+1}$  by incrementing  $\delta'$ .

The strategy in this step is based on the relationship between the abstract models and the maximum number of possible invariants for all the abstract models  $(AM'_1, AM'_2, \dots, AM'_n)$ , where  $AM'_n$  correspond to the original model  $OM'$ .

Fig. 3.29 shows four examples corresponding with three different PLC programs of the relationships between the abstract models and the maximum number of invariants to be extracted if *variable abstraction* is applied. In all of the PLC programs we have verified, we found the same pattern as it can be observed in this figure. In most of them, for  $\delta' \leq 2$ , the number of invariants is reasonable to be extracted but from  $\delta' > 2$  is more optimal to move to a new abstract model as it contains more information and the computational problem is likely smaller.

The mapping between the abstract models and the maximum number of possible invariants is automatically generated when the variable dependency graph is created. Therefore, the strategy is the following: for each  $AM'_n$  with a maximum number of possible invariants  $m \leq 10$  (in our experiments it usually corresponds with  $\delta' \leq 2$ ) and the reachability property verification result is false, an invariant is added to the same  $AM'_n$ . If  $m > 10$ , then no invariant is added and the algorithm moves to a new iteration with a new  $AM'_n + 1$  by incrementing  $\delta'$  ( $\delta' := \delta' + 1$ ).

For example, if the reachability property is  $\text{EF}(\gamma \& \zeta)$ , the extracted invariant is the following:

$$\text{INVAR: } \neg(\gamma \& \zeta)$$

### 3.7.4.7 Example of applicability

To help the understanding of this method, the following paragraphs present the verification a very simple PLC program using this technique (obviously the model of this PLC program does not need ab-

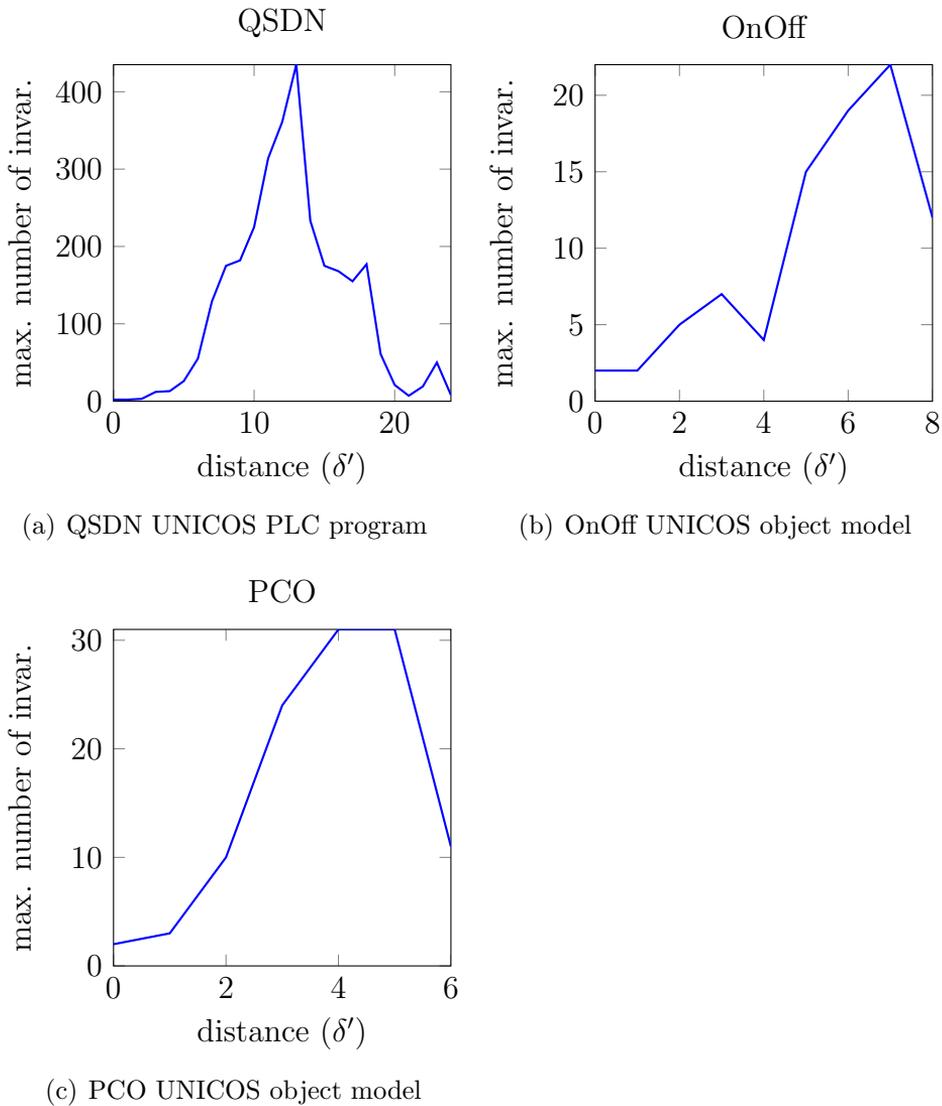


Figure 3.29: Relationship between the abstract models and the maximum number of possible invariants for the variable abstraction technique

straction to be verified, but it is useful to understand the different steps of this technique). The PLC code of this example is shown in Listing 3.5. It consists of a simple function with six Boolean variables and it uses two input variables from the IIM. The requirement to be checked is the following:

“If  $b$  is true at the end of the PLC cycle, then  $a$  is always true at the end of the same PLC cycle.”

The formalized requirement  $p$  using the patterns is:

$$\text{AG}\left(\left(EoC \wedge b\right) \rightarrow a\right)$$

$EoC$  is the variable which is TRUE when the active location of the IM is *end* and FALSE when is any other location.

```

1 FUNCTION Test1
2 VAR
3     a : BOOL;
4     b : BOOL;
5
6     w : BOOL;
7     x : BOOL;
8     y : BOOL;
9     z : BOOL;
10
11 END_VAR
12 BEGIN
13     z := I0.0 AND I0.1;
14     b := z;
15     w := I0.1 AND NOT I0.0;
16     y := z AND NOT w;
17     x := b AND I0.0;
18     a := x AND y;
19 END_FUNCTION

```

Listing 3.5: Example of ST code

The corresponding IM for this PLC program is represented in Fig. 3.30.

**Step 1:** the variable dependency graph for this PLC program is represented in Fig. 3.31.

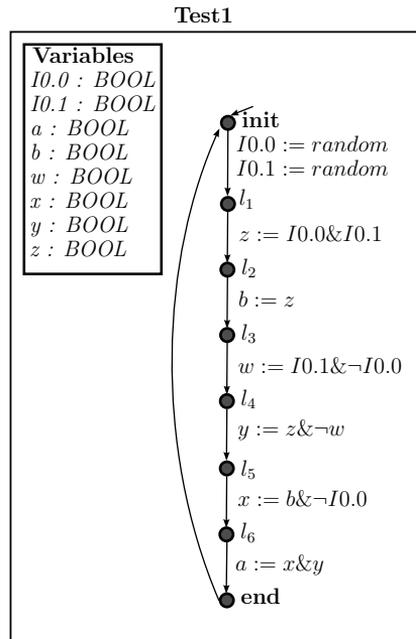


Figure 3.30: Example of IM

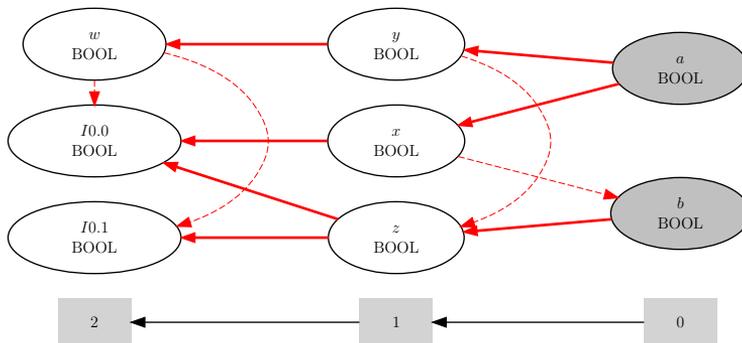


Figure 3.31: Variable dependency graph of the original model corresponding to the PLC program example

Following this algorithm, the resulting variable dependency graph for the proposed PLC program example is represented in Fig. 3.32.

In the first iteration  $AM'_1$  is generated using  $\delta := 1$ . The variables  $x$  and  $y$  are at distance 1 from  $a$ , and therefore they are converted to input variables. The variable  $b$  is not at distance 1 from  $b$ , therefore it is also converted to an input variable.

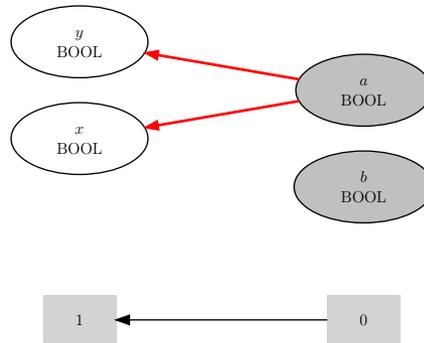


Figure 3.32: Variable dependency graph of the abstract model corresponding to the PLC program example

Due to this abstraction based on the variable dependency graph, the IM of the new  $AM$  is presented in Fig. 3.33. It can be observed that the variables  $w$ ,  $z$ ,  $I0.0$  and  $I0.1$  are eliminated and the variables converted to input variables are  $y$  and  $z$  and  $a$ .

The verification result for  $p$  is false, and the generated counterexample  $c$  is shown in Table 3.2.

Table 3.2: Counterexample for  $p$  on  $AM'_1$

Variable	End of Cycle1
a	FALSE
b	TRUE
x	FALSE
y	FALSE

**Step 2:** the safety property  $q$  is the following:

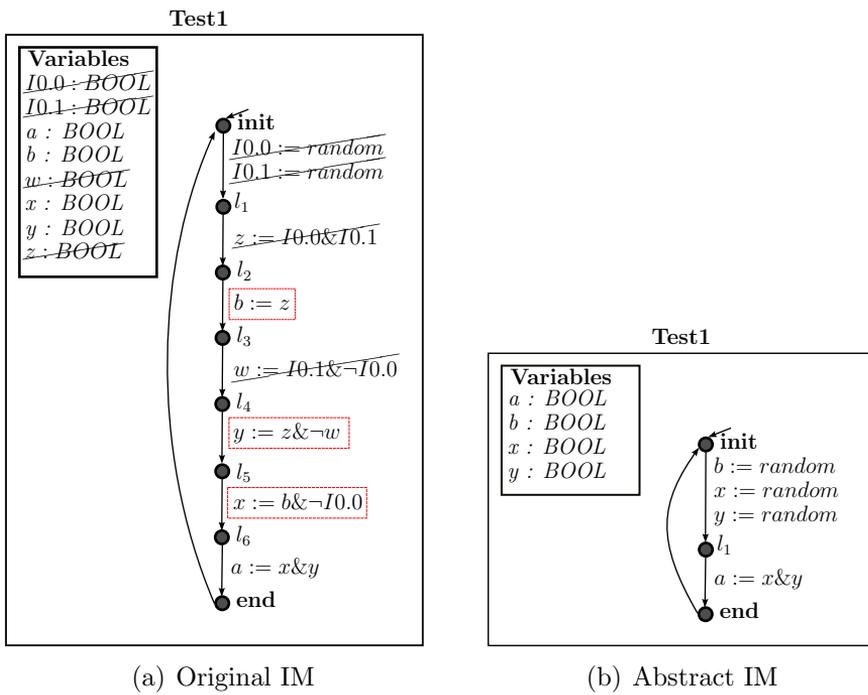


Figure 3.33: Abstraction strategy applied to the example

$$\text{AG}\left(\left(EoC \wedge \mathbf{b}\right) \rightarrow \neg \mathbf{a}\right)$$

$EoC$  is the variable which is TRUE when the location of the IM is *end* and FALSE when is a any other location.

In this case, the verification result is false again and we cannot determine yet if  $c$  is real or spurious. Therefore the algorithm moves to the step 3.

**Step 3:** the reachability property  $r$  generated from  $c$  is:

$$\text{EF}\left(EoC \wedge \mathbf{b} \wedge \neg \mathbf{x} \wedge \neg \mathbf{y}\right)$$

When verifying  $r$  in the new  $OM''$ , the result is false, which means that  $c$  is spurious and the algorithm moves to the step 5.

**Step 5:** as the number of potential invariants for  $AM'_1$  is 8, an invariant is added to the same abstract model ( $AM'_1$ ). The extracted invariant is the following:

$$\text{INVAR: } \neg(\mathbf{b} \wedge \neg \mathbf{x} \wedge \neg \mathbf{y})$$

**Step 1:**  $p$  is verified again on  $AM'_1$  but containing the new invariant. The verification result is false and the new counterexample  $c$  is shown in Table 3.3.

Table 3.3: Counterexample for  $p$  on  $AM'_1 + 1$  invariant

Variable	End of Cycle1
a	FALSE
b	TRUE
x	TRUE
y	FALSE

**Step 2:** the safety property  $q$  is verified again on  $AM'_1 + \text{invariant}$ . In this case, the verification result is false again and we cannot determine yet if  $c$  is real or spurious. Therefore the algorithm moves to the step 3.

**Step 3:** the reachability property  $r$  generated from  $c$  is:

$$\text{EF} \left( EoC \wedge \mathbf{b} \wedge \mathbf{x} \wedge \neg \mathbf{y} \right)$$

When verifying  $r$  in the new  $OM''$ , the result is false, which means that  $c$  is spurious and the algorithm moves to the step 5.

**Step 5:** as the number of potential invariants for  $AM'_1$  is 8, a second invariant is added to the same abstract model ( $AM'_1$ ). The extracted invariant is the following:

$$\text{INVAR: } \neg(\mathbf{b} \wedge \mathbf{x} \wedge \neg \mathbf{y})$$

**Step 1:**  $p$  is verified again on  $AM'_1$  but also containing the second invariant. The verification result is false and the new counterexample  $c$  is shown in Table 3.4.

Table 3.4: Counterexample for  $p$  on  $AM'_1 + 2$  invariants

Variable	End of Cycle1
a	FALSE
b	TRUE
x	FALSE
y	TRUE

**Step 2:** the safety property  $q$  is verified again on  $AM'_1 + 2$  invariants. In this case, the verification result is false again and we cannot determine yet if  $c$  is real or spurious. Therefore the algorithm moves to the step 3.

**Step 3:** the reachability property  $r$  generated from  $c$  is:

$$\text{EF}\left(EoC \wedge \mathbf{b} \wedge \neg \mathbf{x} \wedge \mathbf{y}\right)$$

When verifying  $r$  in the new  $OM''$ , the result is false, which means that  $c$  is spurious and the algorithm moves to the step 5.

**Step 5:** as the number of potential invariants for  $AM'_1$  is 8, a third invariant is added to the same abstract model ( $AM'_1$ ). The extracted invariant is the following:

$$\text{INVAR: } \neg(\mathbf{b} \wedge \neg \mathbf{x} \wedge \mathbf{y})$$

**Step 1:**  $p$  is verified again on  $AM'_1$  but also containing the third invariant. The verification result is true, the algorithm is over and we can conclude that  $p$  holds on  $OM'$ .

## 3.8 IM– verification tools transformation

This section describes the transformation from the IM to the modeling languages used as inputs by the verification tools. Currently, three tools have been integrated in the methodology: nuXmv, UPPAAL and BIP. This section is divided in 3 parts explaining the rules for each tool:

- Transformation rules from IM code to nuXmv.
- Transformation rules from IM code to UPPAAL.
- Transformation rules from IM code to BIP.

### 3.8.1 IM–nuXmv transformation

The following paragraphs contain a brief introduction of the nuXmv modeling language and the transformation rules to produce nuXmv code from the IM.

### 3.8.1.1 nuXmv models

NuSMV is a symbolic model checker developed by FBK-IRST, Carnegie Mellon University, University of Genova and University of Trento<sup>4</sup>. In 2014, a new version of this verification tool called nuXmv has been released.

The input language of nuXmv describes the input model as a Finite State Machine (FSM) that range from completely synchronous to completely asynchronous. It uses BDD-based and SAT-based techniques as model checking techniques. It provides some heuristics to handle the state explosion problem and for the analysis of specification it supports CTL and LTL formalisms.

The definition of a model consists of variable declarations and the transition rules between states. It provides the following set of finite data types: boolean, scalars and fixed arrays. Static data types can also be constructed. There are two possible definition syntaxes for the transition rules: with the TRANS keyword, (from\_state; to\_state) pairs can be declared, with the ASSIGN keyword, the next states of each variable can be declared. If no next state is defined for a variable, then it will non-deterministically get a value from its range.

Consider a small example of a FMS with two states: *ready* and *busy*. When the FSM is in *ready* and there is a *request*, it will move to *busy*. If there is not request, it stays in *ready*. If the FSM is in the state *busy*, it can move to *ready* or stay in *busy* non-deterministically.

Fig. 3.34 shows the graphical representation of the FSM and Listing 3.6 shows the corresponding nuXmv code. In NuSMV, The FSM is represented by a MODULE. two variables are defined: *request* and *state*. *request* is a boolean variable and *state* can have to values: *ready* and *busy*. In the ASSIGN section, the value in the *state* is set. As the next value of the *request* variable is not defined, it will have a non-deterministic value for each state.

---

<sup>4</sup><http://nusmv.fbk.eu/>

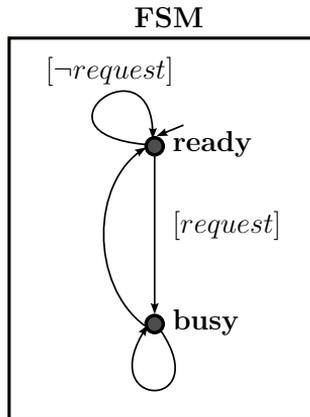


Figure 3.34: Graphical representation as a FSM

```

1 MODULE main
2 VAR
3   request : boolean;
4   state : {ready, busy};
5 ASSIGN
6   init(state) := ready;
7   next(state) := case
8     state = ready & request =
9       TRUE : busy;
10    state = ready & request =
11      FALSE : ready;
12    TRUE : {ready, busy};
13   esac;
  
```

Listing 3.6: NuSMV code example

All the details about the modeling language of nuXmv can be found in Cavada et al. (2011). Note that there exists small differences between the NuSMV and nuXmv input language, for example two new data types are introduced: `real` and `integer`. However, our models remains the same for both model checkers.

### 3.8.1.2 Rules

These are the most relevant IM– nuXmv transformation rules, implemented in the methodology:

**Rule nuXmv1** For the *automata network*  $N$ , a new module *main* is created. This will contain all the automaton instances. It contains also a variable *interaction*, whose domain is the set of possible synchronizations in the automata network ( $I = \{i_1, i_2, \dots\}$ ) and the value *NONE*. This variable and the *main* module are passed as parameters to every module instance to be able to access them. The corresponding nuXmv code fragment can be seen on Listing 3.7.

```

1 MODULE main
2 VAR
3   interaction : {NONE, i1, i2, ...};
  
```

Listing 3.7: Representation of automata network  $a$  in nuXmv

**Rule nuXmv2** For every *automaton*  $a$ , a new module  $a$  is created in the nuXmv model with one single instance  $inst\_a$ . This module will contain a variable  $loc$  which stores the current location of the automaton. The domain of this variable is the set of possible locations in automaton  $a$  (i.e.,  $L = \{l_0, l_1, \dots\}$ ). The default value of the variable  $loc$  will be the initial location  $l_0$ . The corresponding nuXmv code fragment can be seen on Listing 3.8.

```

1 MODULE a(main, interaction)
2   VAR
3     loc : {l0, l1, l2, ...};
4     ...
5   ASSIGN
6     init(loc) := l0;
7     ...
8   MODULE main
9     VAR
10    inst_a : a(self, interaction);

```

Listing 3.8: Representation of automaton  $a$  in nuXmv

**Rule nuXmv3** For every variable  $v$  in automaton  $a$ , a new variable  $v$  is created in the nuXmv model  $a$ . Each variable will have a next-value assignment statement in the assignment block of the module (See Listing 3.9). If a variable is not modified by an assignment explicitly, it will keep its value.

```

1 next(v) := case
2   ...
3   TRUE : v;
4 esac;

```

Listing 3.9: Next-value assignment statement

**Rule nuXmv4** For every *transition*  $t = (l_1, g, amt, i, l_2)$  in automaton  $a$ , a new assignment rule is added for the corresponding  $loc$  variable. It will express that if the current location is  $l_1$ , the next location will be  $l_2$ , if guard  $g$  is true and interaction  $i$  is enabled. If there is no interaction or guard connected to transition  $t$ , the corresponding condition can be omitted. This transformation can be seen on Fig. 3.35 and Listing 3.10.

If the transition  $t$  contains *variable assignment*  $v := Expr$  for variable  $v$ , the next-value assignment corresponding to  $v$  is extended too, as it can be seen on Fig. 3.36 and Listing 3.11.

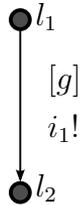


Figure 3.35: Automaton fragment for a transition

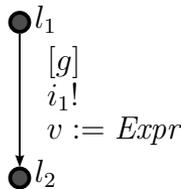


Figure 3.36: Automaton fragment for a variable assignment

```

1 next(loc) := case
2   ...
3   loc = l1 & g & interaction
4     = i1 : l2;
5   esac;

```

Listing 3.10: Corresponding nuXmv code

```

1 next(v) := case
2   ...
3   loc = l1 & g &
4     interaction = i1 :
5     Expr;
6   TRUE : v;
7   esac;

```

Listing 3.11: Corresponding nuXmv code

**Rule nuXmv5** For each *synchronization*  $i_1$  connecting transition  $t_1$  which goes from location  $l_1$  to  $l_2$  in automaton  $a$ , and transition  $t_2$  which goes from location  $l_1$  to  $l_2$  in automaton  $b$ , an invariant is added to the model, as it can be seen on Fig. 3.37 and Listing 3.12.

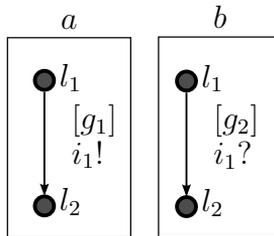


Figure 3.37: Automaton fragment for an interaction

```

1 INVAR (interaction = i1 <->
2   (inst_a.loc = l1 &
3   inst_b.loc = l1 & g1 &
4   g2));

```

Listing 3.12: Corresponding nuXmv code

Listing 3.13 shows an example of a small ST code, the corresponding IM is shown in Fig. 3.38 and the nuXmv code in Listing 3.14. The

ST code contains a single function block which implements a counter without any function calls. The corresponding IM has a single automaton with no interactions. The nuXmv model generated from IM contains two modules: the module *main* for the network, and the module *counter* for the single automaton.

```

1 FUNCTION_BLOCK counter
2   VAR_INPUT
3     enabled : BOOL;
4     reset : BOOL;
5   END_VAR
6   VAR_OUTPUT
7     cntr : INT := 0;
8   END_VAR
9   BEGIN
10    IF enabled THEN
11      cntr := cntr+1;
12    END_IF;
13    IF reset THEN
14      cntr := 0;
15    END_IF;
16  END_FUNCTION_BLOCK

```

Listing 3.13: PLC ST code

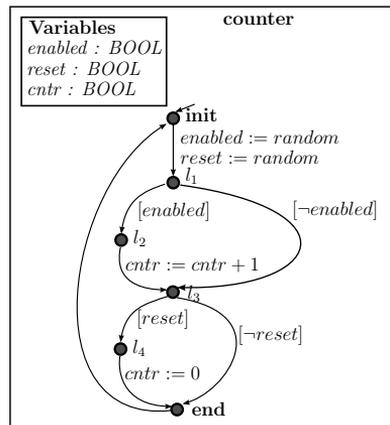


Figure 3.38: Corresponding IM

```

1 MODULE COUNTER(interaction, main)
2 VAR
3   loc : {10, end, 11, 12, 13, 14};
4   ENABLED : boolean;
5   RESET : boolean;
6   CNTR : signed word[16];
7 ASSIGN
8   init(loc) := 10;
9   next(loc) := case
10    loc = end : 10;
11    loc = 10 : 11;
12    loc = 11 & (ENABLED) : 12;
13    loc = 11 & (!(ENABLED)) : 13;
14    loc = 12 : 13;
15    loc = 13 & (RESET) : 14;
16    loc = 13 & (!(RESET)) : end;
17    loc = 14 : end;
18   TRUE: loc;
19 esac;
20 next(ENABLED) := case
21   loc = 10 : {TRUE, FALSE};
22   TRUE : ENABLED;
23 esac;
24 next(RESET) := case
25   loc = 10 : {TRUE, FALSE};
26   TRUE : RESET;
27 esac;
28 init(CNTR) := 0sd16_0;

```

```

29 next(CNTR) := case
30   loc = 12 : (CNTR + Osd16_1);
31   loc = 14 : Osd16_0;
32   TRUE : CNTR;
33 esac;
34 MODULE main
35 VAR
36   interaction : {NONE};
37   inst_counter : COUNTER(interaction, self);

```

Listing 3.14: nuXmv code

Another example of nuXmv Code, which is generated automatically from PLC code that contains two blocks is shown in Listing 3.15. The original PLC code contains the main program (OB1) and an instance of FB.

```

1 MODULE module_MAINBLOCK(interaction, main)
2 VAR
3   loc : {10, end, 11, 12, 13, 14, 12_ret};
4   A1 : boolean;
5   A2 : boolean;
6   A3 : boolean;
7 ASSIGN
8   init(loc) := 10;
9   next(loc) := case
10    loc = end : 10;
11    loc = 10 : 11;
12    loc = 11 & (A2) : 12;
13    loc = 11 & (!(A2)) : 13;
14    loc = 12 & ((interaction = interactionFB_B1)) : 12_ret;
15    loc = 12_ret & ((interaction = interactionFB_B_ret2)) : 13;
16    loc = 13 : 14;
17    loc = 14 : end;
18    TRUE: loc;
19  esac;
20  next(A1) := case
21    loc = 10 : {TRUE, FALSE};
22    TRUE : A1;
23  esac;
24  init(A2) := FALSE;
25  next(A2) := case
26    loc = 13 : !(A2);
27    TRUE : A2;
28  esac;
29  init(A3) := FALSE;
30  next(A3) := case
31    loc = 14 : DB1.B2;
32    TRUE : A3;
33  esac;
34
35 MODULE module_DB1(interaction, main)
36 VAR
37   loc : {10, end, 11};
38   B1 : boolean;
39   B2 : boolean;

```

```

40 ASSIGN
41   init(loc) := 10;
42   next(loc) := case
43     loc = end & ((interaction = interactionFB_B_ret2)) : 10;
44     loc = 10 & ((interaction = interactionFB_B1)) : 11;
45     loc = 11 : end;
46   TRUE: loc;
47   esac;
48   init(B1) := FALSE;
49   next(B1) := case
50     main.MAINBLOCK.loc = 12 & ((interaction = interactionFB_B1)) : !(MAINBLOCK.
51       A2);
52     TRUE : B1;
53     esac;
54     init(B2) := FALSE;
55     next(B2) := case
56       loc = 11 : !(B1);
57       TRUE : B2;
58     esac;
59 MODULE main
60   VAR
61     interaction : {NONE , interactionFB_B1, interactionFB_B_ret2};
62     MAINBLOCK : module_MAINBLOCK(interaction, self);
63     DB1 : module_DB1(interaction, self);
64   INVAR
65     ((
66       (interaction = interactionFB_B1) <->
67         (MAINBLOCK.loc = 12 &
68           DB1.loc = 10)
69     ) & (
70       (interaction = interactionFB_B_ret2) <->
71         (DB1.loc = end &
72           MAINBLOCK.loc = 12_ret)
73     ));
74   DEFINE
75     PLC_START := (MAINBLOCK.loc = 11);
76     PLC_END := (MAINBLOCK.loc = end);

```

Listing 3.15: nuXmv model example

## 3.8.2 IM-UPPAAL transformation

The following paragraphs contain a brief introduction of the UPPAAL modeling language and the transformation rules to produce UPPAAL code from the IM.

### 3.8.2.1 UPPAAL models

UPPAAL is a toolbox for verification of real-time systems developed by Uppsala University and Aalborg University. It provides an integrated

tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata, extended with data types.

The input language of UPPAAL is based on the theory of timed automata, including extra features like bounded integer variables and urgency. The internal representation of the models is based on CDDs (Clock Difference Diagrams). For the property specification it supports the TCTL (timed computation tree logic) formalism.

A timed automaton is a FSM extended with clock variables. All the clocks progress synchronously. In UPPAAL, a system is modeled as a network of several such timed automata in parallel.

Consider a similar small example, as the one presented for nuXmv, consisting in a FMS with two states: *ready* and *busy*. When the FSM is in *ready* and there is a *request*, it will move to *busy*. If there is not request, it stays in *ready*. If the FSM is in the state *busy*, it will remain there for five units of time (representing seconds) and then it will move to *ready*.

Fig. 3.39 shows the graphical representation of the FSM and the Listing 3.16 its representation in UPPAAL. In UPPAAL, the FSM is represented by a *template* inside the *network of automata (nta)*. In the *declaration* section, two elements are defined: a Boolean variable *request* and a clock *x*.

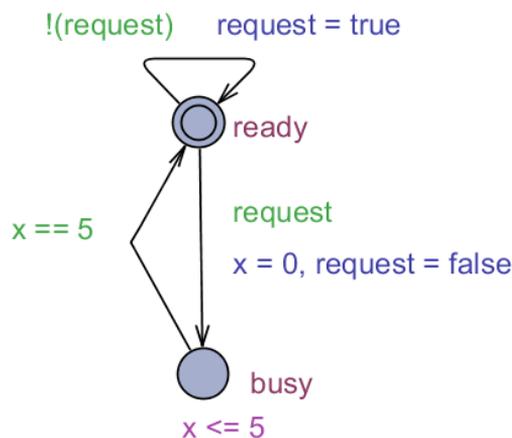


Figure 3.39: Screenshot of the UPPAAL model

```

1 <?xml version="1.0" encoding="utf-8"?><!DOCTYPE nta PUBLIC '-//Uppaal Team
//DTD Flat System 1.1//EN' 'http://www.it.uu.se/research/group/darts/
uppaal/flat-1_1.dtd'>
2 <nta>
3   <declaration>
4     bool request;
5     clock x;
6   </declaration>
7   <template><name>test</name>
8     <location id="id0"><name x="56" y="72">ready</name></location>
9     <location id="id1"><name>busy</name><label kind="invariant">x &lt;= 5</
    label></location>
10    <init ref="id0"/>
11    <transition><source ref="id1"/><target ref="id0"/>
12      <label kind="guard">x == 5</label>
13    </transition>
14    <transition><source ref="id0"/><target ref="id0"/>
15      <label kind="guard">! (request)</label>
16      <label kind="assignment">request = true</label>
17    </transition>
18    <transition><source ref="id0"/><target ref="id1"/>
19      <label kind="guard">request</label>
20      <label kind="assignment">x = 0, request = false</label>
21    </transition>
22  </template>
23  <system>
24    mainBlock = test();
25    system mainBlock;
26  </system>
27 </nta>

```

Listing 3.16: UPPAAL code example

All details about UPPAAL can be found in Amnell et al. (2001) and Behrmann et al. (2004).

### 3.8.2.2 Rules

These are the most relevant IM– UPPAAL transformation rules, implemented in the methodology:

**Rule UPPAAL1** For the *automata network*  $N$ , a new network of timed automata (*nta*) is created. This *nta* is composed of the following sections: the *template* section, which contains the automaton definitions; the *declaration* section, which contains all the variables of the model (accessible from any part of the model) and the channels (*chan*) for the synchronizations; finally the section *system*, which contains the instances of all the templates. An example of an UPPAAL code fragment can be seen on Listing 3.17.

```

1 <?xml version="1.0" encoding="utf-8"?><!DOCTYPE nta PUBLIC '
  -//Uppaal Team//DTD Flat
2 System 1.1//EN' 'http://www.it.uu.se/research/group/darts/
  uppaal/flat-1_1.dtd'>
3 <nta>
4   <declaration>
5     ...
6   </declaration>
7   <template>
8     ...
9   </template>
10  ...
11  <system>
12    mainBlock = TEMPLATE_mainBlock();
13    DB1 = TEMPLATE_DB1();
14    system mainBlock, a;
15  </system>
16 </nta>

```

Listing 3.17: Representation of automata network in UPPAAL

**Rule UPPAAL2** For every *automaton*  $a$ , a template  $a$  is created in the UPPAAL model with one single instance  $inst\_a$  in the system section. For each location in automaton  $a$  (i.e.,  $L = \{l_0, l_1, \dots\}$ ), a *location* element is created in this template. The default *location* will be the initial location  $l_0$ . An example of an UPPAAL code fragment can be seen on Listing 3.18.

```

1 <template>
2   <name>a</name>
3   <location id="l0"><name>initial</name></location>
4   <location id="l2"><name>end</name></location>
5   <location id="l1"><name>l1</name></location>
6   <init ref="l0" />
7   ...
8 </template>
9   ...
10 <system>
11   inst_a = a();
12 </system>

```

Listing 3.18: Representation of automaton  $a$  in UPPAAL

**Rule UPPAAL3** For every variable  $v$  in automaton  $a$ , a new variable  $v$  is created in the UPPAAL model in the *declaration* section as this variable can be accessed globally at the IM model. An example of an UPPAAL code fragment can be seen on Listing 3.19.

```

1 <declaration>
2   bool v;
3 </declaration>

```

Listing 3.19: Variables of automaton  $a$  in UPPAAL

**Rule UPPAAL4** For every *transition*  $t = (l_1, g, amt, i, l_2)$  in automaton  $a$ , a new *transition* element is added to the corresponding *template*. This transition is composed of the following elements: *source*, representing the initial location of the transition; *target*, representing the final location of the transition; *label kind="guard"*, representing the guard  $g$  if exists; *label kind="assignment"*, representing the assignment  $amt$  if exists; *label kind="synchronization"*, representing the synchronization  $i$  if exists. Fig. 3.40 shows a basic transition in IM, and Listing 3.40 shows its representation in UPPAAL. The variable assignment is shown in Fig. 3.41 and the corresponding UPPAAL code in Listing 3.21.

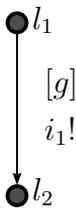


Figure 3.40: Automaton fragment for a transition

```

1 <transition>
2   <source ref="l1"/>
3   <target ref="l2"/>
4   <label kind="guard">g</
5     label>
6   <label kind="
7     synchronization">i1
8     !</label>
9 </transition>

```

Listing 3.20: Corresponding UPPAAL code

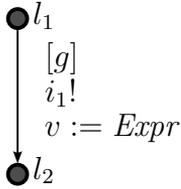


Figure 3.41: Automaton fragment for a variable assignment

```

1 <transition>
2   <source ref="l1"/>
3   <target ref="l2"/>
4   <label kind="guard">g</
5     label>
6   <label kind="
7     synchronization">i1
      !</label>
      <label kind="assignment">
        v=Expr</label>
      </transition>

```

Listing 3.21: Corresponding UPPAAL code

**Rule UPPAAL5** For each *synchronization*  $i_1$  connecting transition  $t_1$  which goes from location  $l_1$  to  $l_2$  in automaton  $a$  and transition  $t_2$  which goes from location  $l_1$  to  $l_2$  in automaton  $b$ , a *label kind="synchronization"* is added. The symbol ! and ? indicate which transition is the caller transition and which is the callee respectively. The corresponding IM is shown on Fig. 3.42 and the UPPAAL code in Listing 3.42.

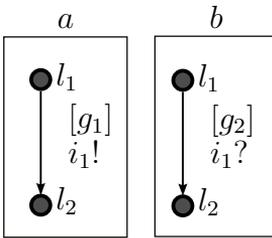


Figure 3.42:  
Automaton fragments  
for a synchronization

```

1 // for automaton a
2 ...
3 <transition>
4   <source ref="a_l1"/>
5   <target ref="a_l2"/>
6     <label kind="guard">g1
7       </label>
8     <label kind="synchronization
9       ">i1!</label>
10  </transition>
11 ...
12 // for automaton b
13 ...
14 <transition>
15   <source ref="b_l1"/>
16   <target ref="b_l2"/>
17   <label kind="guard">g2</
18     label>
19   <label kind="synchronization
20     ">i1?</label>
21 </transition>

```

Listing 3.22: Corresponding  
UPPAAL code

Listing 3.23 shows an example of a small ST code and Fig. 3.43 its corresponding IM. The corresponding UPPAAL code is shown in Listing 3.24. The ST code contains a single function block that implements a counter without any function calls. The corresponding IM have a single automaton with no interactions. The UPPAAL model generated from IM contains two modules: the module *main* for the network, and the module *counter* for the single automaton.

```

1 FUNCTION_BLOCK counter
2   VAR_INPUT
3     enabled : BOOL;
4     reset : BOOL;
5   END_VAR
6   VAR_OUTPUT
7     cntr : INT := 0;
8   END_VAR
9   BEGIN
10    IF enabled THEN
11      cntr := cntr+1;
12    END_IF;
13    IF reset THEN
14      cntr := 0;
15    END_IF;
16  END_FUNCTION_BLOCK

```

Listing 3.23: PLC ST code

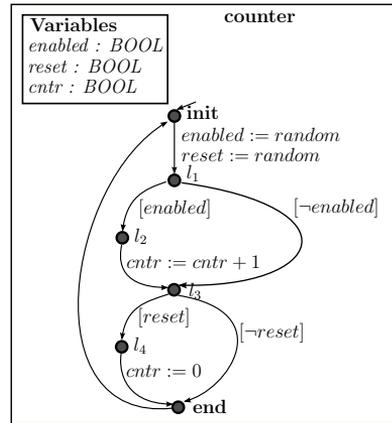


Figure 3.43: Corresponding IM

```

1 <?xml version="1.0" encoding="utf-8"?><!DOCTYPE nta PUBLIC "-//Uppaal Team
  //DTD Flat
2 System 1.1//EN" 'http://www.it.uu.se/research/group/darts/uppaal/flat-1_1.
  dtd'>
3 <nta>
4 <declaration>
5 bool enabled;
6 bool reset;
7 bool cntr;
8 </declaration>
9 <template>
10 <name>COUNTER</name>
11 <location id="init"><name>init</name></location>
12 <location id="end"><name>end</name></location>
13 <location id="l1"><name>l1</name></location>
14 <location id="l2"><name>l2</name></location>
15 <location id="l3"><name>l3</name></location>
16 <location id="l4"><name>l4</name></location>
17 <init ref="init" />
18 <transition>
19 <source ref="init"/>
20 <target ref="l1"/>
21 <label kind="assignment">
22   enabled = random_1
23 </label>
24 <label kind="select">
25   random_1 : int[0,1]
26 </label>
27 <label kind="assignment">
28   reset = random_2
29 </label>
30 <label kind="select">
31   random_2 : int[0,1]
32 </label>
33 </transition>
34 <transition>

```

```

35 <source ref="l1"/>
36 <target ref="l2"/>
37 <label kind="guard">
38   enabled
39 </label>
40 </transition>
41 <transition>
42   <source ref="l2"/>
43   <target ref="l3"/>
44   <label kind="assignment">
45     cntr = cntr + 1
46   </label>
47 </transition>
48 <transition>
49   <source ref="l1"/>
50   <target ref="l3"/>
51   <label kind="guard">
52     !(enabled)
53   </label>
54 </transition>
55 <transition>
56   <source ref="l3"/>
57   <target ref="l4"/>
58   <label kind="guard">
59     reset
60   </label>
61 </transition>
62 <transition>
63   <source ref="l4"/>
64   <target ref="end"/>
65   <label kind="assignment">
66     cntr = 0
67   </label>
68 </transition>
69 <transition>
70   <source ref="l3"/>
71   <target ref="end"/>
72   <label kind="guard">
73     !(reset)
74   </label>
75 </transition>
76   <transition>
77     <source ref="end"/>
78     <target ref="init"/>
79 </transition>
80 </template>
81 <system>
82   mainBlock = COUNTER();
83 </system>
84 </nta>

```

Listing 3.24: Corresponding UPPAAL model

Another example of UPPAAL code, generated automatically from PLC code with two blocks is shown in Listing 3.25. The original PLC code contains the main program (OB1) and an instance of FB.

```

1 <?xml version="1.0" encoding="utf-8"?><!DOCTYPE nta PUBLIC "-//Uppaal Team
  //DTD Flat
2 System 1.1//EN" 'http://www.it.uu.se/research/group/darts/uppaal/flat-1.1.
  dtd'>
3 <nta>
4 <declaration>
5 chan interactionFB_B1;
6 chan interactionFB_B_ret2;
7 // Global variables
8 // Local variables
9 // For automaton mainBlock
10 bool mainBlock_A1;
11 bool mainBlock_A2 = FALSE;
12 bool mainBlock_A3 = FALSE;
13 // For automaton DB1
14 bool DB1_B1 = FALSE;
15 bool DB1_B2 = FALSE;
16 </declaration>
17 <template>
18   <name>TEMPLATE_mainBlock</name>
19   <location id="MAINBLOCK_init"><name>initial</name></location>
20   <location id="MAINBLOCK_end"><name>end</name></location>
21   <location id="MAINBLOCK_l1"><name>l1</name></location>
22   <location id="MAINBLOCK_l2"><name>l2</name></location>
23   <location id="MAINBLOCK_l3"><name>l3</name></location>
24   <location id="MAINBLOCK_l4"><name>l4</name></location>
25   <location id="MAINBLOCK_l2_ret"><name>l2_ret</name></location>
26   <init ref="MAINBLOCK_init" />
27   <transition>
28     <source ref="MAINBLOCK_end"/>
29     <target ref="MAINBLOCK_init"/>
30   </transition>
31   <transition>
32     <source ref="MAINBLOCK_init"/>
33     <target ref="MAINBLOCK_l1"/>
34     <label kind="assignment">
35       mainBlock_A1 = random_1
36     </label>
37     <label kind="select">
38       random_1 : int[0,1]
39     </label>
40   </transition>
41   <transition>
42     <source ref="MAINBLOCK_l1"/>
43     <target ref="MAINBLOCK_l2"/>
44     <label kind="guard">mainBlock_A2</label>
45   </transition>
46   <transition>
47     <source ref="MAINBLOCK_l1"/>
48     <target ref="MAINBLOCK_l3"/>
49     <label kind="guard">! (mainBlock_A2)</label>
50   </transition>
51   <transition>
52     <source ref="MAINBLOCK_l2"/>
53     <target ref="MAINBLOCK_l2_ret"/>
54     <label kind="assignment">
55       DB1_B1 = !(mainBlock_A2)

```

```

56 </label>
57 <label kind="synchronization">interactionFB_B1!</label>
58 </transition>
59 <transition>
60 <source ref="MAINBLOCK_12_ret"/>
61 <target ref="MAINBLOCK_13"/>
62 <label kind="synchronization">interactionFB_B_ret2?</label>
63 </transition>
64 <transition>
65 <source ref="MAINBLOCK_13"/>
66 <target ref="MAINBLOCK_14"/>
67 <label kind="assignment">
68   mainBlock_A2 = !(mainBlock_A2)
69 </label>
70 </transition>
71 <transition>
72 <source ref="MAINBLOCK_14"/>
73 <target ref="MAINBLOCK_end"/>
74 <label kind="assignment">
75   mainBlock_A3 = DB1_B2
76 </label>
77 </transition>
78 </template>
79 <template>
80 <name>TEMPLATE_DB1</name>
81 <location id="DB1_init"><name>initial</name></location>
82 <location id="DB1_end"><name>end</name></location>
83 <location id="DB1_l1"><name>s4</name></location>
84 <init ref="DB1_init" />
85 <transition>
86 <source ref="DB1_end"/>
87 <target ref="DB1_init"/>
88 <label kind="synchronization">interactionFB_B_ret2!</label>
89 </transition>
90 <transition>
91 <source ref="DB1_init"/>
92 <target ref="DB1_l1"/>
93 <label kind="synchronization">interactionFB_B1?</label>
94 </transition>
95 <transition>
96 <source ref="DB1_l1"/>
97 <target ref="DB1_end"/>
98 <label kind="assignment">
99   DB1_B2 = !(DB1_B1)
100 </label>
101 </transition>
102 </template>
103 <system>
104   mainBlock = TEMPLATE_mainBlock();
105   DB1 = TEMPLATE_DB1();
106   system mainBlock, DB1;
107 </system>
108 </nta>

```

Listing 3.25: Synchronization of two automaton in UPPAAL

### 3.8.3 IM–BIP transformation

The following paragraphs contain a brief introduction of the BIP modeling language and the transformation rules to produce BIP code from the IM.

#### 3.8.3.1 BIP models

BIP is a component-based framework for rigorous system design aiming at correctness-by-construction for essential properties of the designed system. To this end, the implementation of the system is obtained by automatic code generation preceded, where necessary, by a series of transformations from a high-level model formally analyzed to validate the required properties.

A BIP model consists of the following three layers:

1. The *Behavior* of the atomic components, modeled by LTS (Labeled Transition Systems) extended with data. Each transition is labeled by a port name. Furthermore transitions can be guarded by boolean conditions and trigger updates of the local data.
2. The *Interaction model* is a set of interactions, i.e. sets of ports, which defines allowed synchronizations between ports of different atomic components.
3. The *Priority model* is a strict partial order on interactions, which allows to restrict non-determinism when multiple allowed interactions are possible simultaneously.

Figure 3.44 shows a system composed of two atomic components and a binary connector, which imposes a strong synchronization between their respective transitions *begin* and *work*. Thus, the left component can only “begin work” when the right component is in the *On* mode. The three transitions *end*, *on* and *off* can only happen individually, i.e. without any synchronization. Furthermore, the priority  $on < end$  imposes that, when both transitions are possible (in the state (Work, Off)) the transition *end* will be chosen (if the system has received an instruction to switch off while working, the *on* instructions

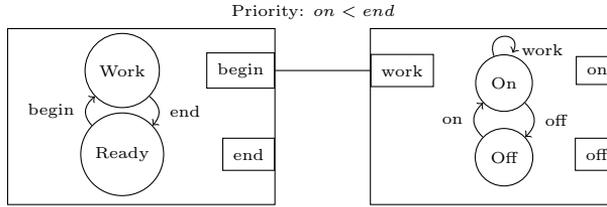


Figure 3.44: BIP model with two components

should be suppressed until the current job is done). Detailed presentation of the BIP model and semantics can be found in Bliudze and Sifakis (2007).

### 3.8.3.2 Rules

Before starting with the definition of the transformation rules from IM to BIP, it is necessary to mention an important difference. BIP language does not support global variables. Therefore this requires a modeling strategy for the situation when one automaton is accessing (reading or writing) variables from a different automaton or global variables at the IM level. These are the most relevant IM- BIP transformation rules, implemented in the methodology:

**Rule BIP1** For the *automata network*  $N$ , a new package *PLCProgram* is created. This will contain all the automaton definitions (atomic component or *atom*). The package contains the definition of the needed *ports* and *connectors*. Table 3.1 shows the data type mapping between the IM and BIP. This mapping imposes the definition of the required ports and connectors. The following port types are created: (1) the *voidPort* port type is used by the connector for the synchronization between *atoms* and for triggering the transitions of the automaton. Basically *voidPort* is used for every transition that does not contain any assignment *amt* with variables from a different automaton or global variables (data transfer); (2) the *intDataPort* and *charDataPort* ports are used when an assignment in the automaton *a* contains variables from a different automaton or global variable (see rules BIP3 and BIP6), according with the corresponding data type (see Table 3.1).

The following connector types are created: (1) the *SynchroAtoms*

connector is used for the synchronization between atoms and it uses the *voidPort*; (2) the *Singleton* connector is used for triggering single transitions with no synchronization and it also uses the *voidPort*; (3) the *IntDataTransferAtoms* and the *CharDataTransferAtoms* are used when an assignment in the automaton *a* contains variables from a different automaton or global variable (see rules BIP3 and BIP6), according with the corresponding data type (see Table 3.1). They use the *intDataPort* and *charDataPort* port types.

As global variables are not supported in BIP but they are at the IM, two special atoms called *intGlobalVariable* and *charGlobalVariable* are created. They contain one single place *INIT* and two ports: *p1* for reading the value of the variable and *p2* for writing this value. In addition a *compound* will be created containing all the atom and connector instances. The corresponding BIP code fragment can be seen on Fig. 3.26.

```

1 package PLCProgram
2   port type voidPort()
3   port type intDataPort(int data)
4   port type charDataPort(char data)
5   ...
6   atom type intGlobalVariable()
7     data int Value
8     export port intDataPort p1(Value)
9     export port intDataPort p2(Value)
10    place l0
11    initial to l0
12    on p1 from l0 to l0 do{}
13    on p2 from l0 to l0 do{}
14  end
15
16  atom type charGlobalVariable()
17    data int Value
18    export port charDataPort p1(Value)
19    export port charDataPort p2(Value)
20    place l0
21    initial to l0
22    on p1 from l0 to l0 do{}
23    on p2 from l0 to l0 do{}
24  end
25
26  connector type SynchroAtoms(voidPort a, voidPort b)
27    define a b

```

```

28  end
29
30  connector type IntDataTransferAtoms(intDataPort a,
31      intDataPort b)
32      define a b
33      on a b down {b.data = a.data;}
34  end
35
36  connector type CharDataTransferAtoms(charDataPort a,
37      charDataPort b)
38      define a b
39      on a b down {b.data = a.data;}
40  end
41
42  connector type Singleton(voidPort a)
43      define a
44  end
45
46  compound type PLCProgram()
47  ...

```

Listing 3.26: Representation of automata network in BIP

**Rule BIP2** For every *automaton*  $a$ , a new atom  $a$  is created in the BIP model with one single instance or *component inst.a*. The number of places is the set of possible locations in the automaton  $a$  (i.e.,  $L = \{l_0, l_1, \dots\}$ ). However if a transition  $t$  in the automaton contains a variable assignment  $amt$  that contains variables from a different automaton, extra places, ports and connectors have to be created (See rule BIP6). The default *place* will be the initial location  $l_0$ . An example of the the corresponding BIP code fragment can be seen on Listing 3.27.

```

1  /* DEFINITION OF THE AUTOMATON */
2  atom type a()
3
4  /* VARIABLES */
5  data int B1
6  data int B2
7
8  /* PORTS */
9  export port VoidPort p1()
10 export port VoidPort p2()

```

```

11 export port VoidPort p3()
12 export port intDataPort t1(B1)
13 export port intDataPort t2(B2)
14
15 /* STATES */
16 place 10, 11, 12, 13, 14
17
18 /* INITIAL STATE*/
19 initial to 10 do {
20     /* VARIABLE INITIALIZATIONS */
21     B1 = 0; // Default value
22     B2 = 0; // Default value
23 }
24
25 /* TRANSITIONS */
26 on p2 from 10 to 11
27 on t1 from 11 to 12
28 on p3 from 12 to 13 do {B2 = !((B1 == 1));}
29 on t2 from 13 to 14
30 on p1 from 14 to 10
31 end
32 ...
33 component a inst_a(param1)

```

Listing 3.27: Representation of automaton  $a$  in BIP

**Rule BIP3** For every variable  $v$  in an automaton  $a$ , a local variable  $v$  is created in the BIP atom  $a$ . If a variable  $v1$  is used in the automaton  $a$  but it belongs to another automaton, an extra variable  $v1$  is created in the atom  $a$ . In addition, for every variable  $v1$  from an automaton  $a$  that is accessed (read or written) by any other automaton (global variable), a new instance of the atom *intGlobalVariable* or *charGlobalVariable* is created, depending of the data type of  $v$  (the mapping between PLC data types and BIP data types was shown in Table 3.1). For every input variable, a random value will be assigned to the variable. This can be done using C language as BIP framework supports it. If the BIP language is used for this purpose, abstractions and restrictions on the assignments have to be applied as currently BIP does not support random values. An example is shown in Listing 3.28. In the atom  $a$  the variables  $B1$  and  $B2$  are coming from a different atom. In addition,  $B1Var$  and  $B2Var$  (instances of *intGlobalVariable*) are created representing global variables, and the variable

$A1$  represents an input variable in the atom  $a$ , therefore it is initialized to random at the beginning of the PLC cycle.

```

1 ...
2 atom type intGlobalVariable()
3   data int Value
4   export port intDataPort r(Value)
5   export port intDataPort w(Value)
6   place 10
7   initial to 10
8   on r from 10 to 10 do{}
9   on w from 10 to 10 do{}
10  end
11
12 /* DEFINITION OF THE AUTOMATON */
13 atom type a()
14
15   /* Input VARIABLES */
16   data int A1
17
18   /* VARIABLES From different atom*/
19   data int B2
20
21   /* PORTS */
22   export port VoidPort p1()
23   export port VoidPort p2()
24   export port VoidPort p3()
25   export port intDataPort t1(B1)
26   export port intDataPort t2(B2)
27
28   /* STATES */
29   place 10, 11, 12, 13, 14
30
31   /* INITIAL STATE*/
32   initial to 10
33
34   /* TRANSITIONS */
35   on p1 from 10 to 11 do {
36     A1 = (rand() % 2);}
37
38   on t1 from 11 to 12
39
40   on p3 from 12 to 13 do {
41     B2 = !A1 & B1;}
42
43   on t2 from 13 to 14

```

```

44
45     on t2 from l4 to l0
46 end
47 ...
48     component intGlobalVariable B1Var()
49     component intGlobalVariable B2Var()

```

Listing 3.28: Representation of automaton  $a$  in BIP

**Rule BIP4** For every *transition*  $t = (l_1, g, amt, i, l_2)$  in automaton  $a$ , a new *voidPort* port and a transition labeled by a port name is added to the atom  $a$ . It will express that if the current location is  $l_1$ , the next location will be  $l_2$ , if guard  $g$  is true and synchronization  $i1$  is enabled. If there is no synchronization or guard connected to transition  $t$ , the corresponding condition can be omitted. If  $t$  does not contain a synchronization  $i1$  a *Singleton* connector instance is created. If  $t$  contains a synchronization  $i1$  a *SynchroAtoms* connector instance is created. If the transition  $t$  contains a variable assignment  $amt$  containing variables from a different automaton, extra transitions have to be created (See rule BIP6). Fig. 3.45 shows a basic transition in IM and the corresponding BIP code in Listing 3.29. The variable assignment is shown in Fig. 3.46 and the corresponding BIP code on Listing. 3.30.

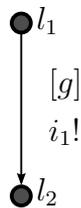


Figure 3.45: Automaton fragment for a transition

```

1 ...
2 export port VoidPort p1()
3 ...
4     on p1
5     from l1 to l2 provided (g ==
6         1)
7 ...
8 connector VoidPortConnector i1(a
9     .p1, b.p1)

```

Listing 3.29: Corresponding BIP code

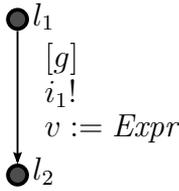


Figure 3.46: Automaton fragment for a variable assignment

```

1 ...
2 export port VoidPort p1()
3 ...
4   on p1
5     from l1 to l2 provided (g ==
6       1) do {v = Expr;}
7 ...
connector VoidPortConnector i1(a
  .p1, b.p1)

```

Listing 3.30: Corresponding BIP code

**Rule BIP5** For each *synchronization*  $i_1$  connecting a transition  $t_1$  which goes from location  $l_1$  to  $l_2$  in automaton  $a$  and a transition  $t_2$  which goes from location  $l_1$  to  $l_2$  in automaton  $b$ , a *SynchroAtoms* connector instance has to be created. The corresponding IM is shown Fig. 3.47 and the corresponding BIP code fragment can be seen on Fig. 3.31.

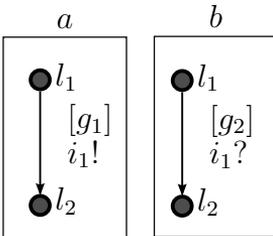


Figure 3.47: Automaton fragments for a synchronization

```

1 connector VoidPortConnector
  interaction1(a.p1, b.p2)

```

Listing 3.31: Corresponding BIP code

**Rule BIP6** For each *amt* in  $t$  that contains variables from different automaton, extra places, transition and ports are created. For each *Boolean* or *Unsigned Char* variable from different automaton or global variable from an *amt* in a transition  $t$ , an instance of *charDataPort* port type is created. For the rest of variables from a different automaton or global variable from an *amt* in a transition  $t$ , an instance of *intDataPort* port type is created. If a *Boolean* or *Unsigned Char* variable from a different automaton or global variable is read a transition labeled by a *charDataPort* and an extra place are created just before of the assignment. If a *Boolean* or *Unsigned Char* variable from a

different automaton or global variable is written, a transition labeled by a *charDataPort* and an extra place are created just after the assignment. If a non *Boolean* or *Unsigned Char* variable from a different automaton or global variable is read, a transition labeled by a *intDataPort* and an extra place are created just before the assignment. If a non *Boolean* or *Unsigned Char* variable from a different automaton or global variable is written, a transition labeled by a *intDataPort* and an extra place are created just after the assignment. An example of a BIP code fragment containing variable assignments is shown on Listing. 3.34.

Listing 3.32 shows an example of a small ST code and Fig. 3.32 its corresponding IM. The corresponding BIP code is shown in 3.33. The ST code contains a single function block implementing a counter without any function calls. The corresponding IM has a single automaton with no interactions. The BIP model generated from IM contains two modules: the module *main* for the network and the module *counter* for the single automaton.

```

1 FUNCTION_BLOCK counter
2   VAR_INPUT
3     enabled : BOOL;
4     reset : BOOL;
5   END_VAR
6   VAR_OUTPUT
7     cntr : INT := 0;
8   END_VAR
9   BEGIN
10    IF enabled THEN
11      cntr := cntr+1;
12    END_IF;
13    IF reset THEN
14      cntr := 0;
15    END_IF;
16  END_FUNCTION_BLOCK

```

Listing 3.32: PLC ST code

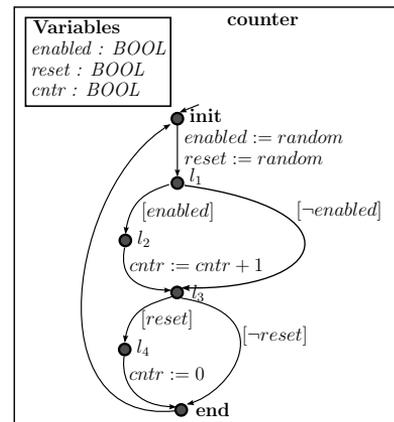


Figure 3.48: Corresponding IM

```

1 @cpp(include="stdio.h")
2 package PLCProgram
3 /* PORT TYPE DEFINITION */
4 port type VoidPort()
5 atom type counter()
6 /* VARIABLES */
7 data int enabled
8 data int reset
9 data int cntr

```

```

10  /* PORTS */
11  export port VoidPort p1()
12  export port VoidPort p2()
13  export port VoidPort p3()
14  export port VoidPort p4()
15  export port VoidPort p5()
16  export port VoidPort p6()
17  export port VoidPort p7()
18  export port VoidPort p8()
19  /* STATES */
20  place init, end, l1, l2, l3, l4
21  /* INITIAL STATE*/
22  initial to init do {}
23
24  /* TRANSITIONS */
25  on p1 from init to l1 do {
26    enabled = (rand() % 2);
27    reset = (rand() % 2);}
28  on p2 from l1 to l2 provided ((enabled == 1))
29  on p3 from l2 to l3 do {
30    cntr = cntr +1;}
31  on p4 from l1 to l3 provided ((enabled == 0))
32  on p5 from l3 to l4 provided ((reset == 1))
33  on p6 from l4 to end do {
34    cntr = 0;}
35  on p7 from l3 to end provided ((reset == 0))
36  on p8 from end to init
37  end
38
39  /* CONNECTOR TYPE DEFINITION */
40  connector type Singleton(voidPort p)
41    define p
42  end
43
44  /* COMPOUND */
45  compound type PlcCompound()
46    /* INSTANCES */
47    component MAINBLOCK counter()
48
49    /* CONNECTORS */
50    connector Singleton c1(MAINBLOCK.p1)
51    connector Singleton c2(MAINBLOCK.p2)
52    connector Singleton c3(MAINBLOCK.p3)
53    connector Singleton c4(MAINBLOCK.p4)
54    connector Singleton c5(MAINBLOCK.p5)
55    connector Singleton c6(MAINBLOCK.p6)
56    connector Singleton c7(MAINBLOCK.p7)
57    connector Singleton c8(MAINBLOCK.p8)
58  end
59  end

```

Listing 3.33: Corresponding BIP model

Another example of BIP Code, generated automatically from PLC code with two blocks is shown in Listing 3.34. The original PLC code contains the main program (OB1) and an instance of FB.

```

1 @cpp(include="stdio.h")
2 package PLCProgram
3 /* PORT TYPE DEFINITION */
4 port type VoidPort()
5 port type intDataPort(int data)
6
7 atom type intGlobalVariable()
8   data int Value
9   export port intDataPort r(Value)
10  export port intDataPort w(Value)
11  place l0
12  initial to l0
13  on r from l0 to l0 do{}
14  on w from l0 to l0 do{}
15 end
16
17 /* AUTOMATA*/
18 /* DEFINITION OF THE AUTOMATON */
19 atom type MAINBLOCK_MAINBLOCK()
20
21 /* VARIABLES */
22 data int A1
23 data int A2
24 data int A3
25
26 /* VARIABLES From different atom*/
27 data int B1
28 data int B2
29
30 /* PORTS */
31 export port VoidPort p1()
32 export port VoidPort p2()
33 export port VoidPort p3()
34 export port VoidPort p4()
35 export port VoidPort p5()
36 export port VoidPort p6()
37 export port VoidPort p7()
38 export port VoidPort p8()
39
40 export port intDataPort t1(B1)
41 export port intDataPort t2(B2)
42
43 /* STATES */
44 place INITIAL, END, l1, l2, l3, l4, l5, l6, l7
45
46 /* INITIAL STATE*/
47 initial to INITIAL do {
48   /* VARIABLE INITIALIZATIONS */
49   A2 = 0; // Default value
50   A3 = 0; // Default value
51 }
52
53 /* TRANSITIONS */
54 on p1 from INITIAL to l1 do {
55   A1 = (rand() % 2);
56 }
57

```

```

58   on p2 from l1 to l2 provided (A2 == 1)
59
60   on p3 from l1 to l5 provided (!(A2 == 1))
61
62   on p4
63   from l2 to l3 do {
64     B1 = !(A2 == 1);}
65
66   on t1 from l3 to l4
67
68   on p5 from l4 to l5
69
70   on p6 from l5 to l6 do {
71     A2 = !A2;}
72
73   on t2 from l6 to l7
74
75   on p7 from l6 to END do {
76     A3 = B2;}
77
78   on p8 from END to INITIAL
79
80 end
81
82 /* DEFINITION OF THE AUTOMATON */
83 atom type DB1_FB1()
84
85   /* VARIABLES */
86   data int B1
87   data int B2
88
89   /* PORTS */
90   export port VoidPort p9()
91   export port VoidPort p10()
92   export port VoidPort p11()
93
94   export port intDataPort t3(B1)
95   export port intDataPort t4(B2)
96
97   /* STATES */
98   place INITIAL, END, l1, l2, l3
99
100  /* INITIAL STATE*/
101  initial to INITIAL do {
102    /* VARIABLE INITIALIZATIONS */
103    B1 = 0; // Default value
104    B2 = 0; // Default value
105  }
106
107  /* TRANSITIONS */
108  on p10 from INITIAL to l1
109
110  on t3 from l1 to l2
111
112  on p11 from l2 to l1 do {
113    B2 = !(B1 == 1);}
114
115  on t4 from l1 to END

```

```

116
117     on p9 from END to INITIAL
118
119 end
120
121
122 /* CONNECTOR TYPE DEFINITION */
123 connector type SynchroAtoms(voidPort a, voidPort b)
124     define a b
125 end
126
127 connector type IntDataTransferAtoms(intDataPort a, intDataPort b)
128     define a b
129     on a b down {b.data = a.data;}
130 end
131
132 connector type Singleton(voidPort a)
133     define a
134 end
135
136 /* COMPOUND */
137 compound type PlcCompound()
138     /* INSTANCES */
139     component MAINBLOCK_MAINBLOCK MAINBLOCK()
140     component DB1_FB1 DB1()
141     component intGlobalVariable B1Var()
142     component intGlobalVariable B2Var()
143
144     /* CONNECTORS */
145     connector VoidPortConnector interaction1(MAINBLOCK.p5, DB1.p10)
146     connector VoidPortConnector interaction2(DB1.p9, MAINBLOCK.p6)
147     connector IntDataTransferAtoms transfer1(MAINBLOCK.t1, B1Var.w)
148     connector IntDataTransferAtoms transfer2(B2Var.r, MAINBLOCK.t2)
149     connector IntDataTransferAtoms transfer4(B1Var.r, DB1.t4)
150     connector IntDataTransferAtoms transfer3(DB1.t3, B2Var.w)
151     connector Singleton c1(MAINBLOCK.p1)
152     connector Singleton c2(MAINBLOCK.p2)
153     connector Singleton c3(MAINBLOCK.p3)
154     connector Singleton c4(MAINBLOCK.p4)
155     connector Singleton c5(MAINBLOCK.p7)
156     connector Singleton c6(MAINBLOCK.p8)
157     connector Singleton c7(DB1.p11)
158 end
159
160 end

```

Listing 3.34: BIP example

### 3.9 Modeling timing aspects of PLCs

This section presents how the timing aspects of PLCs are integrated in the proposed methodology. The modeling strategy for timers and time in PLCs has been published in Fernández Adiego et al. (2014a).

Timing operations are widely used in PLC control systems. This section is focused in the modeling strategy for the three most commonly used timers in PLC programs: TON, TOFF and TP timers.

A subset of the PLC timers were presented in detail in Section 2.2.4. Fig. 3.49 summarizes their behaviors to help understanding the proposed modeling strategies.

In this methodology, a PLC timer is represented by a separated automaton synchronized with the main program. Moreover, modeling PLC timers also implies to model the *TIME* data type. However, as it was mentioned previously, timed models usually contain a huge state space, which cannot be handled by model checkers. Based on this observation, two different approaches are proposed for the methodology:

1. The first is a realistic timer representation, which is close to the reality, enabling precise modeling and verification.
2. The second approach proposes an abstract time representation, which is less accurate, but drastically reduces the state space of the model.

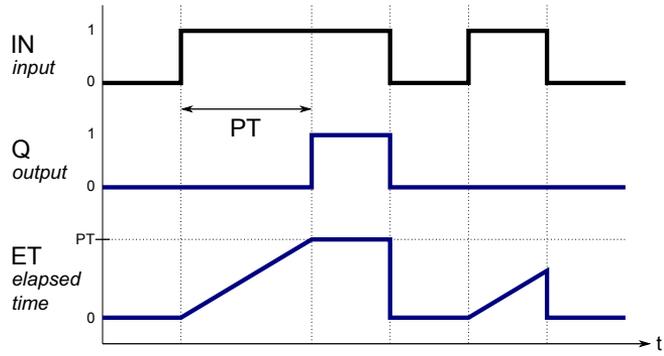
Both approaches are described hereafter focusing on the time and timer representation (for TON, TOFF and TP timers) and the property specification. In addition, the proof that the abstract approach simulates the realistic one is presented, thus guaranteeing that any property verified on the abstract approach also holds in the realistic one, even if it is much simpler.

### 3.9.1 Realistic approach

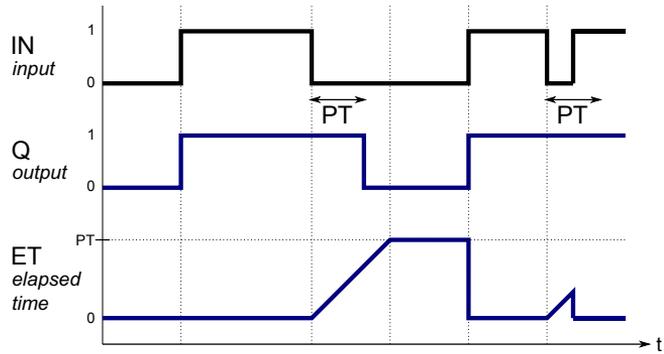
This first approach presents a realistic model of the PLC timers and the time handling. This approach allows to specify properties with explicit time in it. Having this realistic representation of the timer implies that *time* needs also to be modeled realistically.

#### 3.9.1.1 Time representation

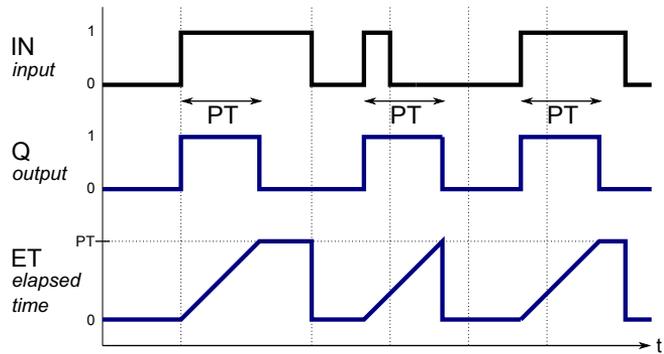
Three main characteristics are considered to model time for this approach:



(a) TON time diagram



(b) TOFF time diagram



(c) TP time diagram

Figure 3.49: Timer diagrams

1. **Time is modeled as a finite variable:** it represents with high fidelity the *TIME* data type in a PLC. However, instead of having a signed 32-bits integer variable (like in Siemens PLCs), a 16-bits variable is used to represent this data type. This range reduction is possible as the behavior of a PLC is cyclic and the cycle time, the delays of the timers, and the delays in the requirements are much smaller than the range of this 16 bit variable. The accuracy of this variable is 1 ms, as it is in real PLCs. Because of this time representation, overflow of time has to be considered when the timer is modeled, and also when the requirement to be checked is expressed (e.g. the current time can be smaller for a later event, see Fig. 3.50).
2. **Time is incremented by adding the cycle time:** In this representation, time is not incremented by individual units of time. It is instead incremented by the duration of the last PLC cycle at the end of it. This assumption obviously simplifies the global model and there is not any loss of accuracy when comparing with the real implementation in a PLC, considering that the timers are called at most once in each PLC cycle, which holds for our real cases.
3. **Cycle time is chosen non-deterministically:** in order to represent standard PLC with a varying cycle time, a random value is generated at the end of each cycle to represent this. The selected random values are between 5 ms and 100 ms, which is a valid assumption based on the PLC systems at CERN.

### 3.9.1.2 TON timer representation

Given this finite time representation, the behavior of the TON timer is represented in ST code as it is shown in Listing 3.35. In this code, the input variables are *IN* and *PT*, and the output variables are *Q* and *ET*. The variable *ctime* represents the current time and it is modeled as previously explained. In addition, two variables are added: *running* and *start*, where *running* is a Boolean variable representing when the timer is working after a rising edge on *IN* and *start* contains the value of *ctime* when *IN* has a rising edge.

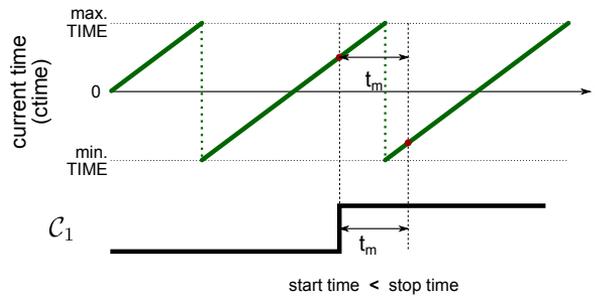


Figure 3.50: Consequences of finite time representation

```

1 FUNCTION_BLOCK TON
2 VAR_INPUT
3   PT : TIME;
4   IN : BOOL;
5 END_VAR
6 VAR_OUTPUT
7   Q : BOOL := FALSE;
8   ET : TIME; // elapsed time
9 END_VAR
10 VAR
11   running : BOOL;
12   start : TIME;
13 END_VAR
14 BEGIN
15 IF IN = FALSE THEN
16   Q := FALSE;
17   ET := 0;
18   running := FALSE; // t1
19 ELSIF running = FALSE THEN
20   start := CTIME;
21   running := TRUE; // t2
22 ELSIF CTIME - (start + PT) >= 0 THEN
23   Q := TRUE;
24   ET := PT; // t3
25 ELSE
26   IF NOT Q THEN
27     ET := CTIME - start;
28   END_IF; // t4
29 END_IF;

```

Listing 3.35: ST code of TON

By applying the extended methodology, the corresponding automaton of the TON ST code was produced. The equivalent state machine is shown in Fig. 3.51. (Note that the assignments of ET are omitted from the state machine to simplify the figure.) This state machine contains three states corresponding to the three original states of the TON: *NR* (not running; running=false, Q=false), *R* (running; running=true, Q=false) and *TO* (timeout; running=true, Q=true). The transitions in the state machine (labeled as  $t_1, t_2, t_3, t_4$ ) correspond to the conditional statements in the ST code.

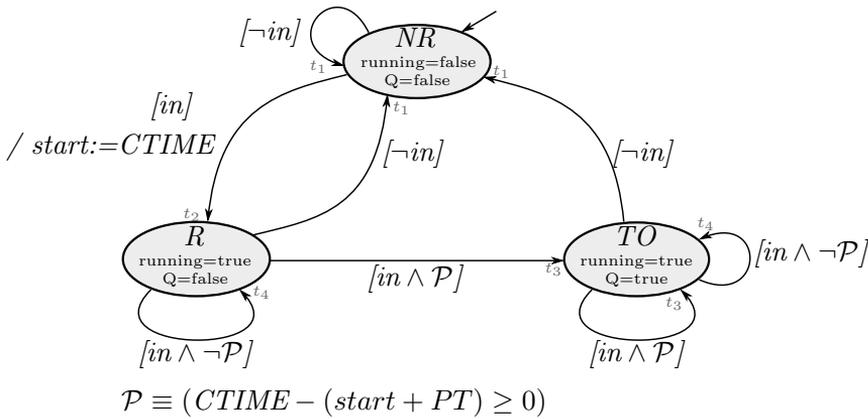


Figure 3.51: State machine of the realistic TON representation

At most one transition can happen in this state machine for every call of the timer. Therefore, the timer cannot go from state *NR* to state *TO* with one function call, which is valid if we assume that delays are always greater than zero ( $PT > 0$ ). According to the specification of the Siemens TON implementation (Siemens (1998b)), the parameter  $PT$  should be positive.

The potential state space (PSS) size of this timer representation is  $3.38 \cdot 10^{16}$  and its reachable state space (RSS) is  $5.91 \cdot 10^{15}$  without counting the variable *ctime* (as it is a global variable used by all the timers).

### 3.9.1.3 TOFF timer representation

The same strategy is applied to the TOFF timer. The corresponding ST code is shown in Listing 3.35. The input and output variables are

the same than for the TON. In this case, *running* is a Boolean variable representing when the timer is working after a falling edge on *IN* and *start* contains the value of *ctime* when *IN* has a falling edge.

```

1 FUNCTION_BLOCK TOFF
2 VAR_INPUT
3   PT : TIME;
4   IN : BOOL;
5 END_VAR
6 VAR_OUTPUT
7   Q : BOOL := FALSE;
8   ET : TIME; // elapsed time
9 END_VAR
10 VAR
11   running : BOOL;
12   start : TIME;
13 END_VAR
14 BEGIN
15 IF IN = TRUE THEN
16   Q := TRUE;
17   ET := 0;
18   running := FALSE;           // t1
19 ELSIF running = FALSE THEN
20   start := CTIME;
21   running := TRUE;           // t2
22 ELSIF CTIME - (start + PT) >= 0 THEN
23   Q := FALSE;
24   ET := PT;                 // t3
25 ELSE
26   IF Q THEN
27     ET := CTIME - start;
28   END_IF;                   // t4
29 END_IF;

```

Listing 3.36: ST code of TOFF

As well as for the TON, by applying the extended methodology, the corresponding automaton of the TOFF ST code was produced and the equivalent state machine is shown in Fig. 3.52. This state machine contains the same states and transitions as the previous timer.

#### 3.9.1.4 TP timer representation

The same strategy is applied to the TP timer. The corresponding ST code is shown in Listing 3.37. The input and output variables are the

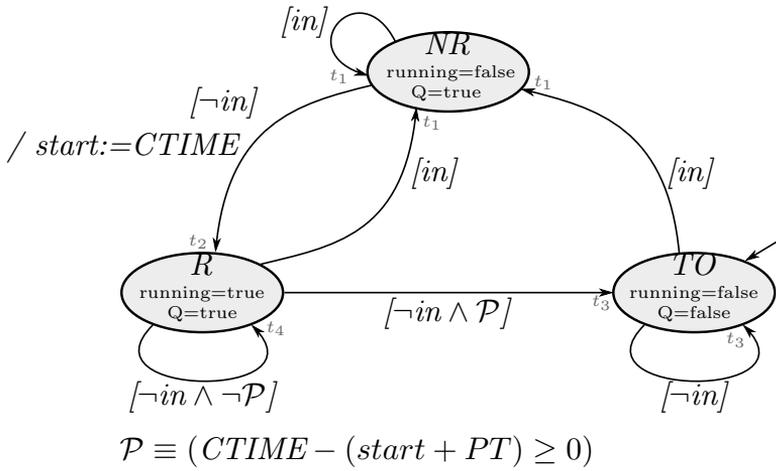


Figure 3.52: State machine of the realistic TOFF representation

same than for the previous timer as well.

```

1 FUNCTION_BLOCK TP
2 VAR_INPUT
3   PT : TIME;
4   IN : BOOL;
5 END_VAR
6 VAR_OUTPUT
7   Q : BOOL := FALSE;
8   ET : TIME;
9 END_VAR
10 VAR
11   running : BOOL;
12   start : TIME;
13 END_VAR
14 BEGIN
15 IF Q = FALSE AND running = FALSE THEN
16   IF IN THEN
17     start := CT;
18     running := TRUE;
19     Q := TRUE;           // t1
20   END_IF;
21 ELSIF Q = TRUE AND running = TRUE THEN
22   IF IN AND (CTIME - (start + PT) >= 0) THEN
23     running := TRUE;
24     Q := FALSE;        // t2
25   ELSIF NOT IN AND (CTIME - (start + PT) >= 0) THEN

```

```

26   running := FALSE;
27   Q := FALSE;           // t3
28   END_IF;
29   ELSIF Q = FALSE AND running = FALSE THEN
30     IF NOT IN THEN
31       running := FALSE;
32       Q := FALSE;       // t4
33     END_IF;
34   END_IF;
35   END_FUNCTION_BLOCK

```

Listing 3.37: ST code of TP

As well as for the previous timers, by applying the extended methodology, the corresponding automaton of the TP ST code was produced and the equivalent state machine is shown in Fig. 3.53. This state machine contains the same states and transitions as the previous timers.

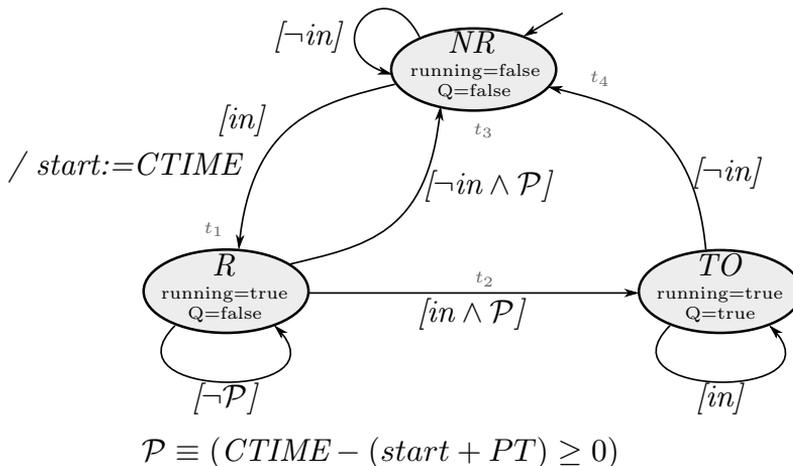


Figure 3.53: State machine of the realistic TP representation

### 3.9.1.5 Property specification

Using the methodology and the developed tool, input models for nuXmv are automatically generated. The experimental results presented here are produced using this model checker. The goal is to verify properties with explicit time in them, like:

“if  $\mathcal{C}_1$  is true, after  $t_m$  time  $\mathcal{C}_2$  will be true, if  $\mathcal{C}_1$  remained true.”

Where  $\mathcal{C}_1$  and  $\mathcal{C}_2$  are Boolean expressions,  $\mathcal{C}_1$  contains input variables and parameters and  $\mathcal{C}_2$  contains output variables.

CTL and LTL do not provide this expressiveness, for this reason a *monitor* or observer automata is added to the model. The goal of the monitor is to check  $\mathcal{C}_1$ , and if it is true for at least  $t_m$  time, the monitor output value *Mout* is set to true.

By using such monitor, the requirement is simplified as:

“if *Mout* is true, then  $\mathcal{C}_2$  should be true.”

This requirement can be formalized easily in CTL:

$$\text{AG}\left(\text{Mout} \rightarrow \mathcal{C}_2\right)$$

An example for this monitor usage can be seen in Fig. 3.54. As the computed values are assigned to the real outputs of the PLC only at the end of the PLC cycle, the requirements should be checked only at this point. The general CTL expression extended with it is the following:

$$\text{AG}\left(\text{EoC} \wedge (\text{Mout} \rightarrow \mathcal{C}_2)\right)$$

Where *EoC* is true, iff the execution is at the end of a PLC scan cycle.

The behavior of this monitor is similar to the TON timer, but it is independent of the rest of the program logic, therefore *Mout* will be true after  $t_m$  time, if  $\mathcal{C}_1$  is true and it can be used to verify if *outn* holds the property. Notice that it is not enough to save a “timestamp”  $t_{\mathcal{C}_1}$  when  $\mathcal{C}_1$  is true, and formalize the requirement as  $\text{AG}((\text{ctime} \geq t_{\mathcal{C}_1} + t_m) \rightarrow \mathcal{C}_2)$ , because the *ctime* variable is a finite integer, thus it can overflow, as it is illustrated by Fig. 3.50.

This technique for specification can be used with any timer models. Chapter 4 will provide some experimental results regarding this modeling approach.

### 3.9.2 Abstract approach

The models created by applying the realistic approach have a big state space even if only one TON is modeled. If this approach is applied

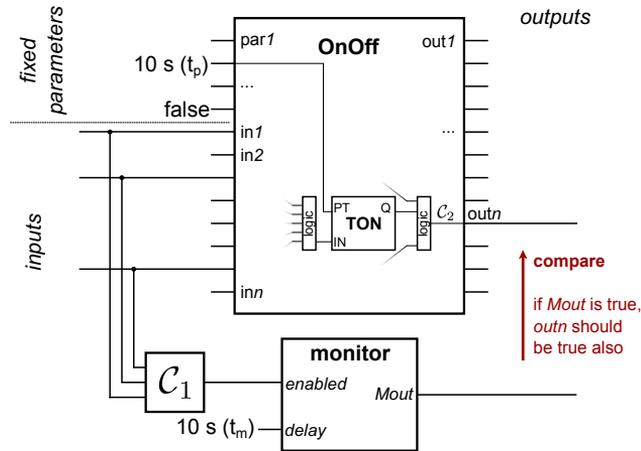


Figure 3.54: Verification configuration for OnOff model extended with monitor

to large models, probably verification would not be even possible. For that reason, a second approach is proposed, which is based on a data abstraction of the first approach.

### 3.9.2.1 Time representation

In this case, time is not represented explicitly in the model, the variable *ctime* representing the current time is not maintained. Therefore properties with explicit time cannot be validated.

### 3.9.2.2 TON timer representation

This timer representation approach consists in a non-deterministic model produced as an abstraction of the realistic approach. The corresponding state machine is presented in Figure 3.55.

Similarly to the realistic model, this model has three possible internal state: *NR* (not running), *R* (running) and *TO* (timed out). If the input *IN* is true, the TON will start to run (goes to state *R*). After that, the TON can stay in the state *R* or go to *TO* non-deterministically, which means, we do not know when the timer will stop. However, by adding a *fairness constraint* to the model, it is ensured that the timer cannot stay in state *R* for infinite time. Only one transition can be fired in one PLC cycle, i.e., the timer cannot go from

state *NR* to state *TO* with one call. This corresponds to the previously introduced  $PT > 0$  constraint. Figure 3.56 shows the timing diagram of the abstract approach compared with the realistic one. The size of the PSS and RSS for this model is 6 comprising the Boolean input variable, reducing significantly the size of the realistic approach but introducing some limitations of the property specification.

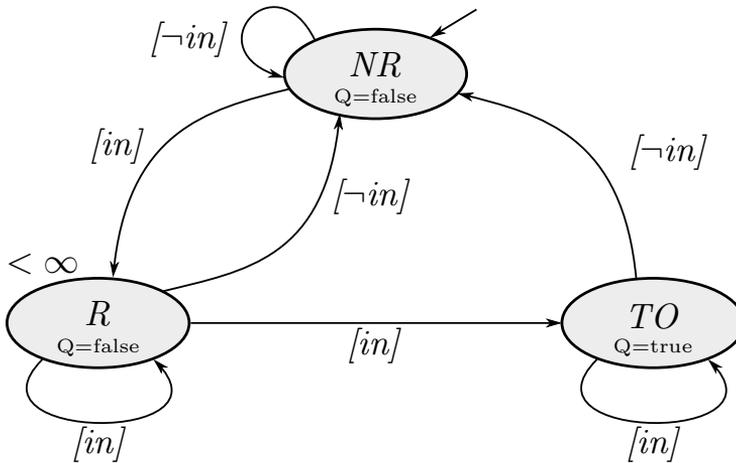


Figure 3.55: State machine of the abstract TON representation

This representation can result false positives, meaning that verification tools can give spurious counterexample that cannot occur in reality. However false negatives can never occur, therefore if a property holds in the abstract model, it holds in the real system.

### 3.9.2.3 TOFF timer representation

Following the same strategy, the TOFF abstract model is represented in Fig. 3.57.

### 3.9.2.4 TP timer representation

Following the same strategy, the TP abstract model is represented in Fig. 3.58.

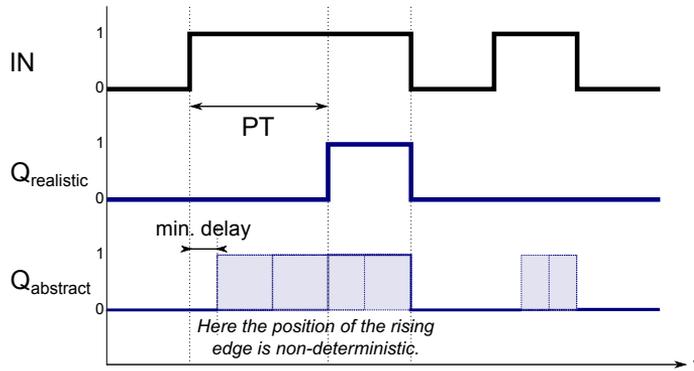


Figure 3.56: Timing diagram of TON modeled using different approaches

### 3.9.2.5 Property specification

Obviously, this abstract model implies certain limitations in the specification properties. Explicit delay times cannot be expressed in the requirement, but safety or liveness properties can be expressed, for example:

“if  $a$  is sometime true and remains true forever, eventually  $b$  will be true”.

Where  $a$  and  $b$  are variables affected by timers in the PLC program.

Chapter 4 will provide some experimental results using these models.

### 3.9.3 Refinement between the two approaches

We can verify that the realistic approach indeed refines the abstract approach. Using this proof, we are able to guarantee that requirements verified on the abstract model – where verification is easier – also hold on the realistic model – where automatic verification would be harder and more time consuming. However, the false result on the abstract model does not imply false result on the realistic model. Using more abstract models for verification purposes and lifting results to realistic models is a well known technique (Clarke et al. (1999); Loiseaux et al. (1995)) and we apply it to the PLC domain. Specifically to PLC timer

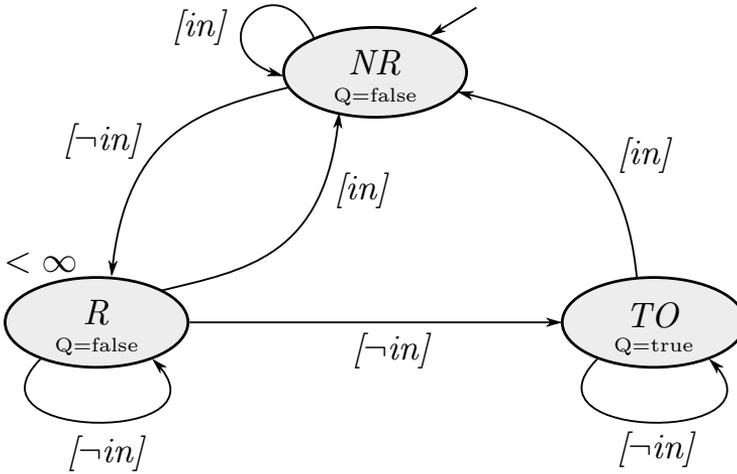


Figure 3.57: State machine of the abstract TOFF representation

models where we perform a proof in a similar fashion as the refinement proofs described in Blech and Grégoire (2011).

For example, for the TON model, the proof is done by first establishing a simulation relation  $S$  between realistic (Fig. 3.51) and abstract automaton (Fig. 3.55) that relates the states of the different automata with each other such that only the following holds:

$$\begin{aligned} &S(NR_{realistic}, NR_{abstract}) \\ &S(R_{realistic}, R_{abstract}) \\ &S(TO_{realistic}, TO_{abstract}) \end{aligned}$$

Furthermore, the simulation relation ensures that  $Q$  has the same value in all states. In order to finish the proof, we show that:

- The initial states are in the simulation relation:  $NR_{realistic}$  and  $NR_{abstract}$  are in the simulation relation and the value of  $Q$  is the same, false.
- Each pair of realistic and abstract model state transitions with a corresponding condition – regarding the value of  $in$  – from possible states in the simulation relation  $S$  lead to a pair of states for which  $S$  holds again. Most cases are trivial except: the transition of the  $[in]$ -guarded transition in the abstract model in the  $(TO_{realistic}, TO_{abstract})$  state pair has two corresponding

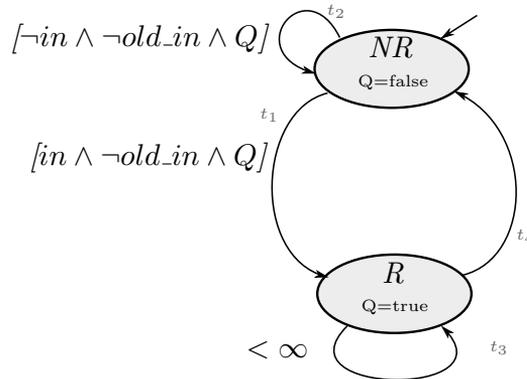


Figure 3.58: State machine of the abstract TP representation

transitions in the realistic model since we do not regard the value of  $\mathcal{P}$  in the abstract model.

Due to the fairness constraint in the abstract model, its state cannot be  $R_{abstract}$  for infinite time. However, this is also true for the corresponding  $NR_{realistic}$  state of the realistic model if it is called sufficiently often<sup>5</sup>, because the delay time  $PT$  is finite.

Based on the definition of the simulation relation, safety properties making use of the atomic elements – like conditions on the used variables  $Q$  and  $in$  – that are preserved in  $S$  are also preserved between abstract and realistic model. However, the abstract model can impose false positives, i.e. the given abstract counterexamples can never occur in the in the real system.

### 3.10 Process modeling

The previous sections were dedicated to describe the rules for building the formal models. However there is a missing part of the system, that was not considered: the controlled process. Process information

<sup>5</sup>It is not defined in the standard or in the manual, but our implementation works properly, if the elapsed time between two calls is less than the maximal value of the TIME data type, which is about 30 s represented on 16 bits or 24 days represented on 32 bits. Usually, the timers are called in every PLC cycle, thus this is not a limitation. This is also needed for the timers to work properly.

is important but building the process model is a very challenging and time consuming task (as extracting the information from the process is done manually). In addition, building a complete model of the controlled process increases significantly the state space of models.

Other authors (e.g. Machado (2006), Machado et al. (2003a), Machado et al. (2003b), Machado and Eurico (2013)) studied the use of process information to verify PLC programs. Their results show that some properties can only be verified using the process information and if this information is not included, false positives are produced, i.e. the verification tool will report a bug that can never occur in the real system.

Considering the following PLC code 3.38. It shows an example where the process information is necessary.

```
1 FUNCTION_BLOCK Valve
2 VAR_INPUT
3   AuOnR: BOOL := FALSE;
4   AuOffR: BOOL := FALSE;
5   HFOn: BOOL := FALSE;
6   HFOff: BOOL := FALSE;
7 END_VAR
8 VAR_OUTPUT
9   Status : BOOL;
10  Open: BOOL;
11  Close: BOOL;
12 END_VAR
13 BEGIN
14   IF AuOnR AND NOT AuOffR THEN
15     Open := TRUE;
16   END_IF;
17   IF AuOffR AND NOT AuOnR THEN
18     Close := TRUE;
19   END_IF;
20   IF Open AND HFON AND NOT HFOff THEN
21     Status := TRUE;
22   ELSIF Close AND HFOff AND NOT HFON THEN
23     Status := FALSE;
24   END_IF;
25 END_FUNCTION_BLOCK
```

Listing 3.38: ST code example

This PLC program consists in a single FB representing the behavior of a digital valve. It has 4 inputs and 3 outputs variables.

- *AuOnR* is an input variable that represents an order to open the valve.
- *AuOffR* is an input variable that represents an order to close the valve.
- *HFO<sub>n</sub>* is an input variable that represents the feedback from the valve, indicating that the valve is open.
- *HFO<sub>ff</sub>* is an input variable that represents the feedback from the valve, indicating that the valve is close.
- *Status* is an output variable that represents the status of this digital valve (information sent to the SCADA).
- *Open* is an output variable that represents the order to open the digital valve (information sent to the process).
- *Close* is an output variable that represents the order to close the digital valve (information sent to the process).

In this case, the requirement is the following:

“If *Open* is TRUE and *HFO<sub>n</sub>* is TRUE, *Status* will be true at the end of the PLC cycle.”

The corresponding CTL formula is:

$$\text{AG} \left( (EoC \wedge \text{Open} \wedge \text{HFO}_n) \rightarrow \text{Status} \right)$$

In this trivial example is obvious that the verification result by using nuXmv is FALSE, due to *HFO<sub>ff</sub>*. The relevant information extracted from the counterexample is presented in Table 3.5.

However, if the hardware of the valve is correct, the input variables *HFO<sub>n</sub>* and *HFO<sub>ff</sub>* can never be equal to true at the same time, as it means that the valve is open and close at the same moment. This information can be added to the methodology as an invariant:

$$\text{INVAR: } \neg(\text{HFO}_n \wedge \text{HFO}_{ff})$$

After adding the invariant, model checking is applied again and the result of the verification is TRUE. It shows that without some information from the controlled process, some properties cannot be proved

Table 3.5: Variables values from the counterexample at the end of the PLC cycle

<b>Variable</b>	<b>End of Cycle1</b>
AuOffR	FALSE
AuOnR	TRUE
<i>HFOff</i>	TRUE
<i>HFO<sub>n</sub></i>	TRUE
Close	FALSE
Open	TRUE
<i>Status</i>	FALSE

that they are TRUE so false positives are produced (the model checker gives as a result a spurious counterexample for the property). However, it also implies that by adding wrong invariants, we can hide bugs in the PLC program and produce false negatives in the verification, which is much more dangerous.

In general in this methodology, the formal model of the controlled process is not included, due to the complexity of building these models and because it can potentially increment the state space of the models. However, invariants with the process information can be manually added to the methodology when needed.

### 3.11 Verification and counterexample analysis

Once the input models for the verification tools are automatically generated and reduced, formal verification can be applied and the analysis of the results can be performed. This step of the methodology has been published in Fernández Adiego et al. (2014c).

### 3.11.1 Counterexample analysis

The counterexample produced by the model checkers when a property is not satisfied, provides the relevant information to identify the source of the problem in the PLC program.

The model checker generates detailed counterexamples, containing the value of each variable after each transition in the model (corresponding to the execution of an instruction in the PLC code). However, these counterexamples are usually too long. In some of our experiments, the counterexample produced by nuXmv contains around 37 500 lines. This information is too detailed for human analysis. However, it can be automatically reduced, as it is enough to know the value of the input and output variables at the *end of each PLC cycle*, thus the variable valuations of the intermediate states can be removed. By removing the unnecessary states and the temporary variables, the counterexample for the same requirement can be reduced to approximately 100 lines. This information can be given to the PLC program developer for human analysis.

The counterexample can also be used to demonstrate the problem in a real environment. Using the nuXmv's counterexample, a demonstrator PLC code can be easily generated by adding a module that simulates the variables from the counterexample and checks the expected result. By doing so, it can be proved that the counterexample is not caused by a mistake during the model generation or due to bad reductions.

Listing 3.39 shows an extract from the generated counterexample of a real PLC program and Listing 3.40 the corresponding part of the PLC demonstrator source code.

Furthermore, the PLC demonstrator can help us to reduce the counterexample. Usually, most of the input variables do not have any effect on the evaluated requirement. Those variables can be fixed to constant values thus helping to focus on the input value changes that causes the violation of the requirement.

```

1 -> State: 1.17 <-
2   inst.HLD = FALSE
3   inst.MANREG01[8] = TRUE
4   inst.MANREG01[9] = FALSE
5   inst.MANREG01[10] = FALSE
6   inst.MANREG01[11] = FALSE
7   inst.AUAUMOR = FALSE
8   inst.AUIHMMO = FALSE
9   inst.AUIHFOMO = FALSE
10  ...

```

Listing 3.39:  
Counterexample fragment

```

1 onoff1.HLD := FALSE;
2 onoff1.ManReg01 :=
3     2#00000000000000001;
4 onoff1.AuAuMoR := FALSE;
5 onoff1.AuIhMMo := FALSE;
6 onoff1.AuIhFoMo := FALSE;
7 ...
8 CPC_FB_ONOFF.onoff1();
9 // Check:
10 b := onoff1.AuMoSt;

```

Listing 3.40: Corresponding  
ST code demonstrator

In the current example, by using the counterexample and the reduced PLC demonstrator, we were able to identify the source of the problem. It turned out that from *Forced* mode the logic cannot switch to *Auto* mode. It can only be done by the operator. The corresponding part of the implementation can be seen in Fig. 3.41<sup>6</sup>. The code shows the transition from *Forced* mode to *Auto* mode, which does not take into account the *AuAuMoR* input. This behavior is correct and intentional, in this case the source of the problem is not the PLC code, the specification contains a mistake.

```

1 IF (MMoSt_aux AND (E_MAUmoR OR E_AuAuMoR)) OR
2   (FoMoSt_aux AND E_MAUmoR) OR
3   (SoftLDSt_aux AND E_MAUmoR) OR
4   (MMoSt_aux AND AuIhMMo) OR
5   (FoMoSt_aux AND AuIhFoMo) OR
6   NOT (AuMoSt_aux OR MMoSt_aux OR
7       FoMoSt_aux OR SoftLDSt_aux) THEN
8   (* Setting mode to Auto *)
9   AuMoSt_aux := TRUE;
10  MMoSt_aux := FALSE;
11  FoMoSt_aux := FALSE;
12  SoftLDSt_aux := FALSE;
13 END_IF;

```

Listing 3.41: Extract of the source code causing the violation of the

<sup>6</sup>In the source code, the variables starting with “E.” indicate the rising edges of the corresponding inputs. *AuMoSt\_aux* is true if the current mode is *Auto*. The meaning of variables *MMoSt\_aux*, *FoMoSt\_aux*, and *SoftLDSt\_aux* are similar, for the other modes.

requirement

In the bottom part of Fig. 3.59, it can be observed that the PLC demonstrator generation is integrated in the methodology.

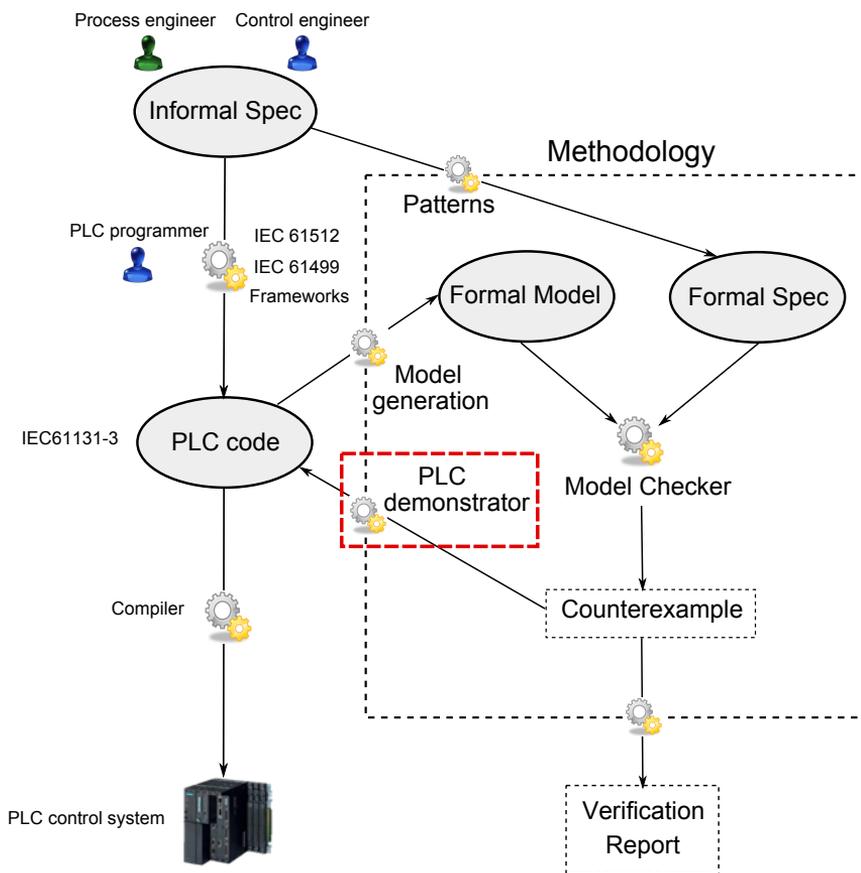


Figure 3.59: Integration of the proposed methodology on the PLC program development process.

Many safety and liveness requirements have been verified using this methodology and the number of discrepancies between the specification and PLC program were significant even if testing was previously applied. In all these cases, three possible situations were identified as the source of the problem: a bug in the implementation, a mistake in the specification or an incomplete specification.

## 3.12 Methodology CASE tool

This methodology has been implemented in a CASE tool using EMF (Eclipse Modeling Framework described in Steinberg et al. (2009)) and Xtext technologies (Eysholdt and Behrens (2010)). The grammar of the PLC languages is implemented as a Xtext grammar. The IM is implemented as an EMF metamodel as it can be seen in Fig. 3.60. The transformation rules from the PLC languages to the IM, the reduction techniques and the transformation rules from IM to nuXmv, UPPAAL and BIP have been implemented using Xtend (Xtend website (2014)) and Java. The Xtext framework provides a parser, an over linker, an interpreter, an editor, and many other features to help the developer (Xtext website (2014)).

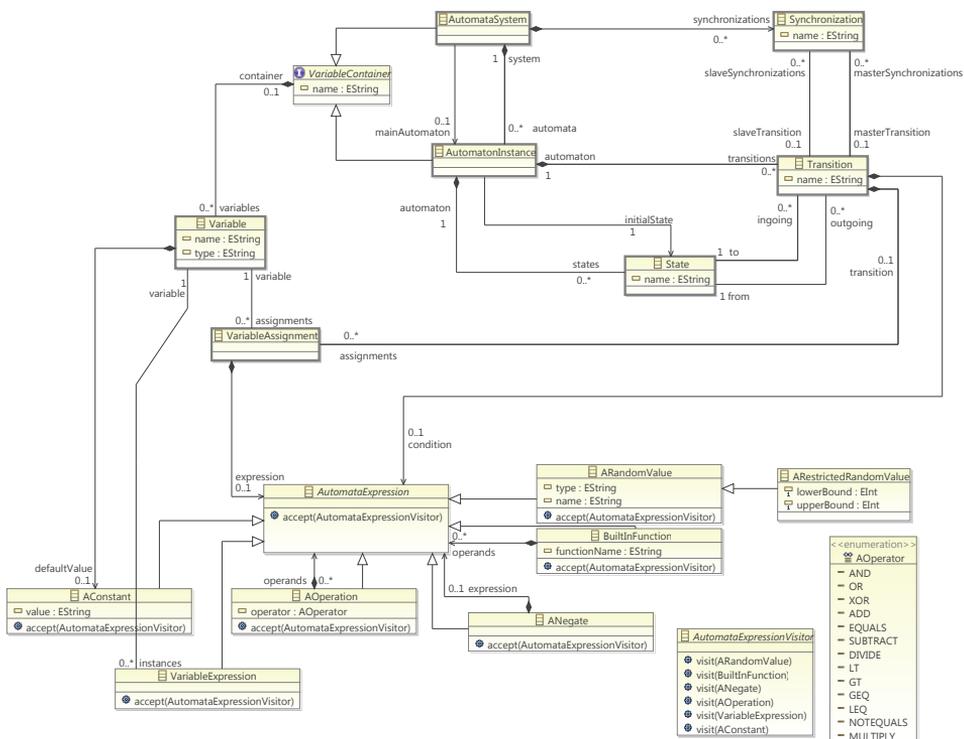


Figure 3.60: EMF implementation

The tool provides a graphical interface, where the user can create a verification project. The user has to provide the PLC code and the properties to verify, and the tool returns the verification result (TRUE

and FALSE) and a report with all the details of the counterexample analysis. Figures 3.61, 3.62, 3.63 and 3.64 show some screenshots of the tool.

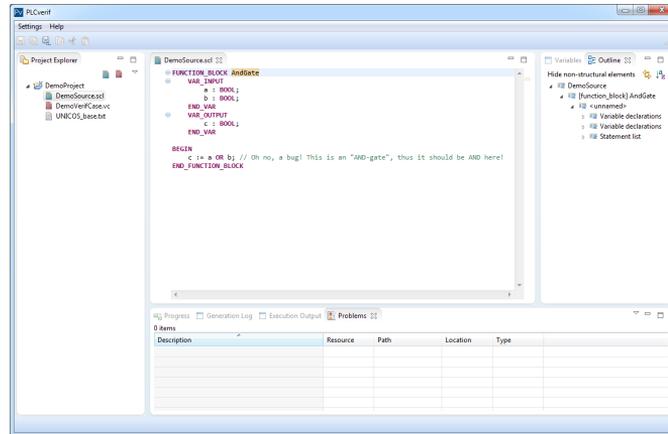


Figure 3.61: CASE tool editor for the PLC code

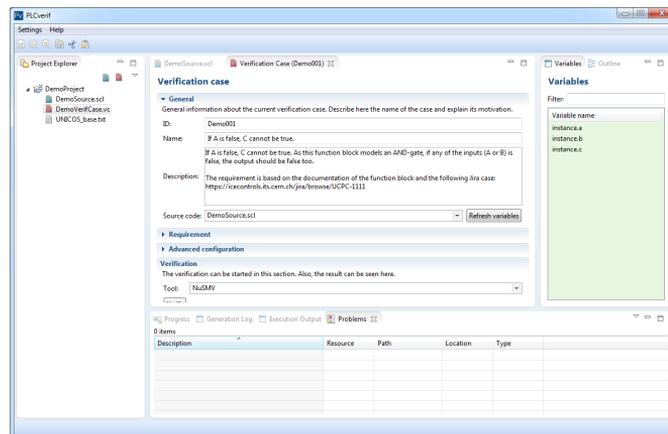


Figure 3.62: CASE tool property description

### 3.13 Summary of the chapter

This chapter presented the contribution of this thesis: a general approach to apply formal verification to PLC programs. Each step of

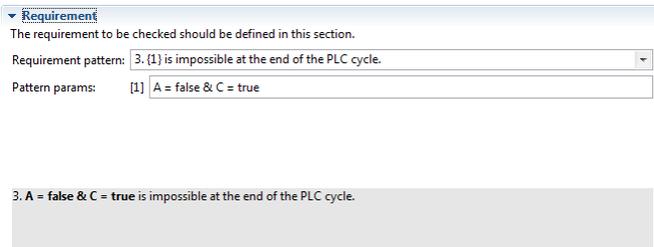


Figure 3.63: CASE tool property formalization by using patterns

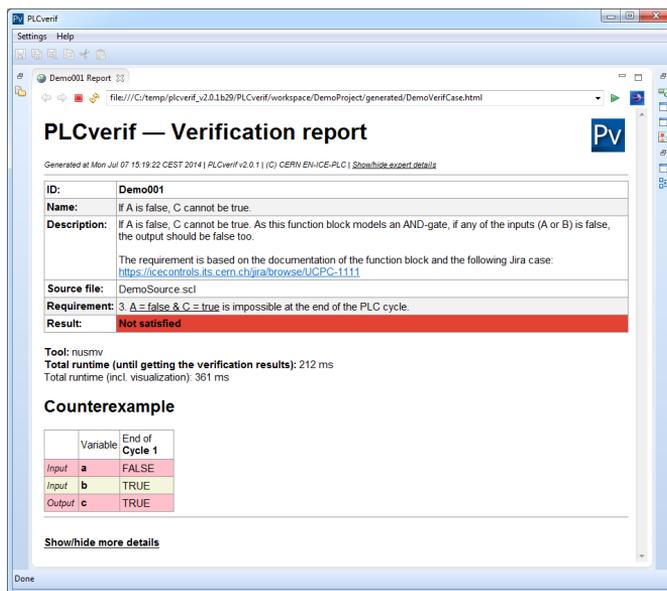


Figure 3.64: CASE tool verification report

the methodology has been described in a separate section as well as the syntax and semantics of the IM. These methodology hides any complexity related to formal methods from control engineers. For that purpose, the methodology provides a solution for the following challenges:

1. Requirements specification.
2. PLC hardware modeling.
3. PLC code – formal models transformation.

4. Time and timers modeling.
5. Process modeling.
6. Counterexample analysis.

In addition, a brief discussion about the CASE tool that supports this methodology is presented.

# Chapter 4

## Case studies and measurements

### 4.1 Introduction

This chapter presents the experimental results of applying the proposed methodology to real-life PLC programs. The selected systems have been developed and deployed at CERN using the UNICOS framework and Siemens S7 PLCs. The verification results presented in this chapter, are obtained using nuXmv as it is currently the verification tool which give us better results from the verification point of view. The measurements were performed on a PC with Intel® Core™ i5-3230M 2.60 GHz CPU, 8 GB RAM, on Windows 7 x64.

This chapter is divided in 3 main parts:

1. Section 4.2 introduces the UNICOS framework describing the control system architecture obtained when applying it.
2. Section 4.3 describes the experimental results of applying the methodology to a PLC program corresponding to one single UNICOS object. This PLC code consists of one main FB and some additional FCs and timers.
3. Finally, Section 4.4 presents the experimental results of a complete PLC program of the LHC cryogenics control system. This PLC program contains hundreds of FBs, FCs and DBs.

## 4.2 UNICOS framework

UNICOS is an object-based control system framework developed at CERN and based on the IEC 61512 (2009) standard. This framework provides a methodology, a library of objects and a set of code generation tools to produce control programs.

UNICOS was born more than a decade ago to provide a solution for the LHC cryogenics control system. The first reference about a control system developed with UNICOS can be found in Casas-Cubillos et al. (2002) and the first reference about the framework itself in Gayet and Barillère (2005). Since that first development, the framework has been enriched, becoming a more general and flexible framework. The last updates can be found in Blanco Viñuela et al. (2011).

When developing a control system using the UNICOS framework, control and process engineers design and describe the control strategy in a high level specification and the automatic generation tools produce the control code for the control layer and the configuration of the supervision layer. In addition, the necessary code for the communication between these layers is also automatically generated.

The framework produces control programs for different control devices, i.e. PLCs or industrial PCs. Regarding the type of process and the control devices used, UNICOS can be divided in different packages as it is show in Fig. 4.1.

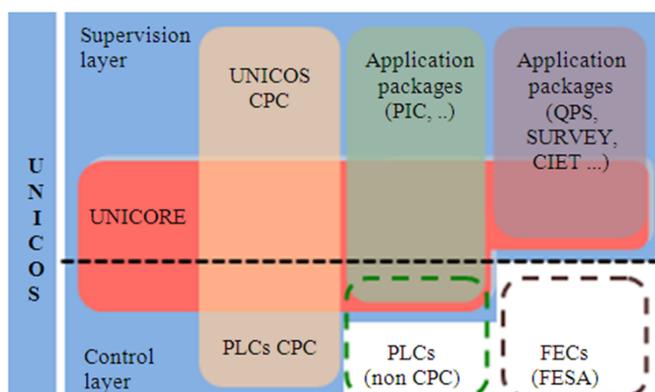


Figure 4.1: Different UNICOS packages

Here we focus on the CPC (Continuous Process Control) package, which produces control systems based on PLCs at the control

layer (currently Siemens and Schneider PLCs) and on the WinCC OA SCADA (from Siemens) at the supervision layer. This package produces mainly control systems for continuous processes. The common architecture of the control systems developed with UNICOS CPC is shown in Fig. 4.2. Details about the PLC code generated by the UNICOS CPC package are presented in the following paragraphs.

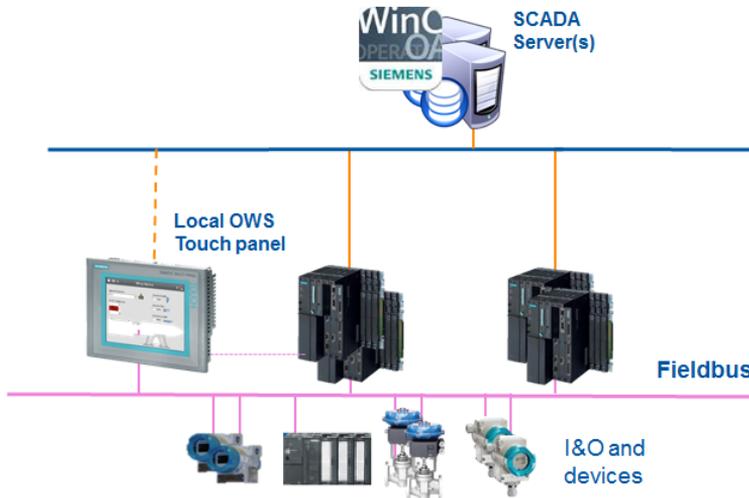


Figure 4.2: UNICOS CPC control system architecture

By using the UNICOS CPC package, the control engineer designs a model of the process units (e.g. sensors, actuators, subsystems, etc.). On this model, every element from the process instrumentation is represented by the UNICOS CPC objects (also called UNICOS CPC baseline objects). These objects are based on the UNICOS metamodel described in Copy et al. (2011) and classified in families. There are 4 UNICOS CPC object families:

1. *I/O object family*: they represent the inputs and output signals connecting the PLC inputs and outputs with sensors and actuators. For example, a temperature sensor (which is connected to the digital input card of the PLC) is represented by a “Digital-Input” object.
2. *Field object family*: they represent instrumentation equipment such as valves, pumps, motors, heaters, etc. Usually these devices are connected to the real PLC inputs and outputs through

several input or output objects (e.g. a digital valve may receive one signal to open the valve, one signal to close it and it may send two feedback signals to the PLC indicating if the valve is open or close). For example, a digital pump is represented by a “OnOff” object.

3. *Control object family*: they represent a group of instrumentation equipments or a subsystem operated independently from the rest of the process. The corresponding object in this case is called “ProcessControl” object. This object behaves as a master, sending orders to the *Field objects* that are part of the corresponding subsystem. This family also includes the objects that represent the PID controllers, the alarms and interlocks of the system.
4. *Interface object family*: they represent information communicated between the SCADA and the control layer. Typically parameters or status are part of this family. One example is the “AnalogParameter” object, which represents the analog parameters sent from the SCADA to the control layer (e.g. thresholds).

In the CPC package, each UNICOS CPC object has its corresponding PLC code, containing the behavior of the object. This PLC code consists in one FB (sometimes including some additional FCs).

The PLC code of a complete control system, developed with UNICOS CPC, consists in a common software architecture where the object instances are interconnected and the specific logic to control the process according with the high level specification is included.

Fig. 4.3 shows an example of this mapping between the process instrumentation and the UNICOS objects. In this high level model, each element on the process side (sensors, actuators, signals, subsystems, alarms, etc.) correspond to one UNICOS object and they are interconnected.

The PLC code is automatically generated from this high level model, according with a set of rules included in the generation tool (See Fernández Adiego et al. (2011)). The code contains the instance of the UNICOS CPC objects, the interconnection between them and functions containing the specific logic of the application (according with the specification).

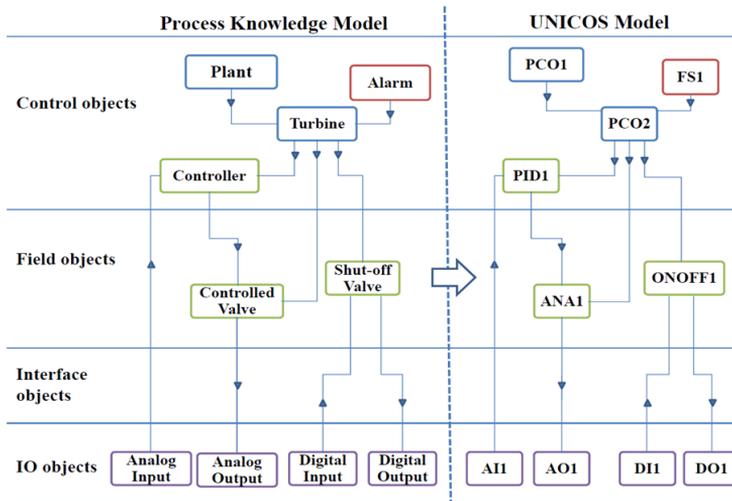


Figure 4.3: Mapping between the process instrumentation and the UNICOS objects example

Fig. 4.4 shows the static representation of the PLC code architecture for Siemens S7 PLCs. It shows the main program (i.e. OB1), a cyclic interrupt (i.e. OB35), an event interrupt (i.e. OB82) and a restart subroutine (i.e. OB100). All the baseline object FBs (the ones starting by “CPC\_FB\_”, e.g. *CPC\_FB\_DI*) are called from the main program as well as the specific logic of the control system. The PLC languages used in UNICOS CPC for Siemens PLCs are SCL (Structured Control Language) and Graph, which is similar to GRAFCET (Graphe Fonctionnel de Croissant Étape Transition). They are the Siemens implementation of ST and SFC from the IEC 61131 (2013) standard.

This framework has been used in many industrial installations, for example, the LHC cryogenics, cooling, HVAC (Heating, Ventilating, and Air Conditioning) and the LHC vacuum systems. It has become a standard in the control system development at CERN.

The following paragraphs present the verification case studies of UNICOS PLC programs.

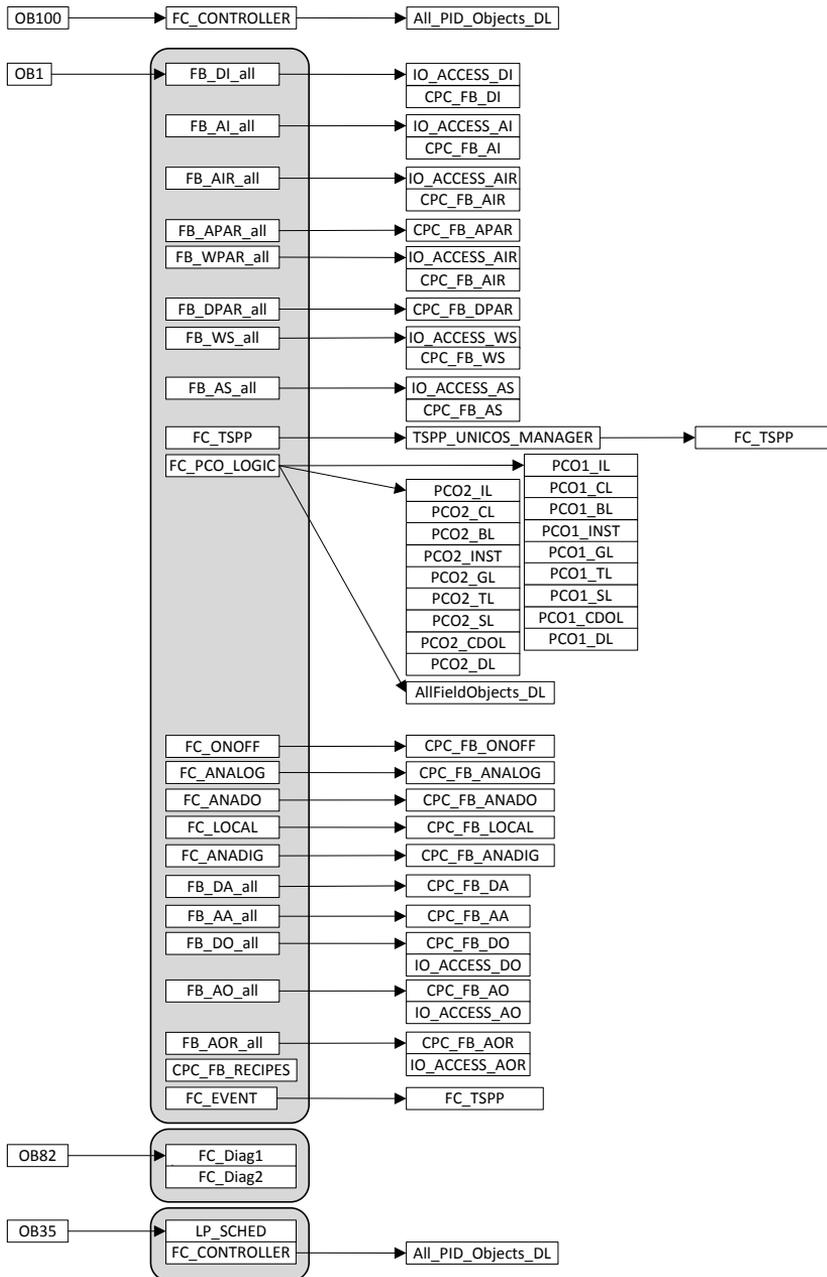


Figure 4.4: UNICOS PLC code architecture

## 4.3 UNICOS baseline object case study

The proposed methodology has been applied to verify the UNICOS CPC objects. The size and complexity in terms of coding logic of these objects varies from the simplest objects, such as the *DigitalInput* and the *DigitalParameter* objects to the most complex, such as the *Controller*.

The selected object to present the experimental results in this chapter is the so-called *OnOff* object. The *OnOff* object size and complexity are representative of the UNICOS CPC objects and many requirements (complex safety properties, liveness properties are time-related properties) were provided by the UNICOS developers. In most of the cases, verification of these properties was possible by applying the property preserving reduction techniques proposed in this thesis.

This section presents a detailed description of the experimental results and it is divided in the following subsections:

- Description and some metrics about the *OnOff* PLC implementation (Section 4.3.1).
- Experimental results regarding model generation (Section 4.3.2).
- Experimental results regarding verification of complex properties (Section 4.3.3).
- Experimental results regarding verification of time properties (Section 4.3.4).
- Example of a counterexample analysis (Section 4.3.5).

Part of these experimental results were published in Fernández Adiego et al. (2014c).

### 4.3.1 Object description and specification

This object represents a physical equipment driven by digital signals, e.g. an on-off valve, heater or motor. The corresponding PLC code, used for the experimental results, consists on one FB and some additional FCs written in SCL. Some key metrics about this object are shown in Table 4.1. The complete *OnOff* PLC code can be found in the Appendix A.1.

Table 4.1: Case Study metrics

Metric	OnOff PLC code
Lines of code	$\approx 820$
Program blocks	1 main FB, 2 timers and 3 FCs
Statements	418
Function calls	21
Input variables	29 (20 BOOL, 3 WORD, 2 ARRAY of 16 BOOL and 4 TIME)
Output variables	31 (27 BOOL, 2 WORD and 2 ARRAY of 16 BOOL)
Internal variables	82 (73 BOOL, 1 TIME, 2 TP timer instances, 1 TON timer instance, 1 REAL and 4 INT)
PLC data types	BOOL, INT, REAL, WORD, TIME, STRUCT and ARRAY
Timers	3 instances

Fig. 4.5 shows the *OnOff* signal interface. More details about the *OnOff* object and about UNICOS in general can be found in the UNICOS documentation website (UNICOS CPC Team (2013)).

### 4.3.2 Experimental results regarding model generation

As it was described in the methodology, the first step is the formalization of the requirements. This first step has an important impact on the generation of the models, as the performance of reduction techniques such as COI highly depends on the property requirements. About fifty requirements have been verified on the *OnOff* object, finding many discrepancies between the specification and the PLC program. As well as for the *OnOff* object, the methodology has also been successfully applied to rest of the UNICOS CPC baseline objects.

In this section, two verification cases are presented to illustrate how this methodology is applied to the *OnOff* object. Both requirements are connected to the so-called *Mode manager*.

The *Mode manager* specification describes the different operating modes in which the *OnOff* can be operated. This object can operate in one of the following five modes:

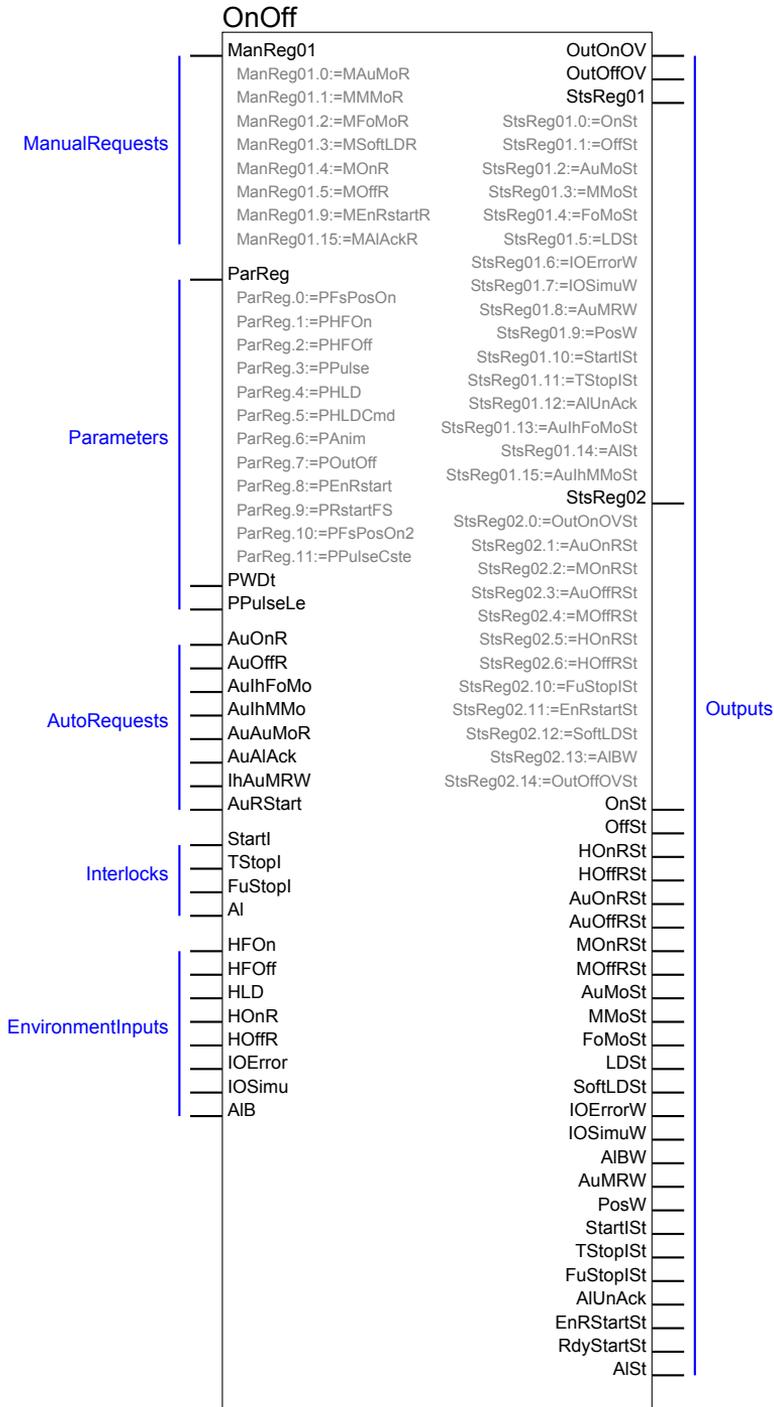


Figure 4.5: OnOff signals interface

- *Auto*: the object is driven by the control logic of a higher object of the hierarchy (e.g. a PCO (Process Control Object) from the control object family).
- *Manual*: the object is driven by the operator from the supervision and it can change to *Auto* mode with a control logic request.
- *Forced*: the object is driven by the operator from the supervision but it can only change to *Auto* mode with an operator request.
- *Software Local Drive*: the object is driven locally by a local software such a local touch panel.
- *Hardware Local Drive*: the object is driven locally by the process field.

The transitions between the operating modes are performed by “mode requests”, coming from the implemented logic, the operator or a local device. To perform an operation mode change, a rising edge is necessary on the corresponding mode request input. For example, one of these inputs is **AuAuMoR**, which stands for “Auto mode request initiated by the logic”. Similarly, there is a **MAuMoR** input which stands for “Auto mode request initiated by the operator”. Fig. 4.6 shows the state machine that specifies the behavior of the mode manager for the three main modes. In the figure, “RE( $x$ )” stands for rising edge on input  $x$ .

The PLC code corresponding to the mode manager is 4.1. However, it is affected by more variables from the rest of the PLC program.

```

1  (* MODE MANAGER *)
2  IF NOT (HLD AND PHLD) THEN
3
4  (* Forced Mode *)
5  IF (AuMoSt_aux OR MMoSt_aux OR SoftLDSt_aux) AND
6  E_MFoMoR AND NOT(AuIhFoMo) THEN
7  AuMoSt_aux := FALSE;
8  MMoSt_aux := FALSE;
9  FoMoSt_aux := TRUE;
10 SoftLDSt_aux := FALSE;
11 END_IF;
12
13 (* Manual Mode *)
14 IF (AuMoSt_aux OR FoMoSt_aux OR SoftLDSt_aux) AND
15 E_MMMoR AND NOT(AuIhMMo) THEN
16 AuMoSt_aux := FALSE;

```

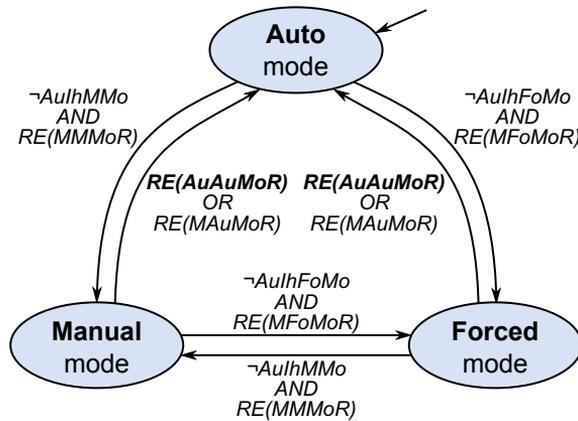


Figure 4.6: State machine of the OnOff mode manager from the UNICOS specifications

```

17  MMoSt_aux := TRUE;
18  FoMoSt_aux := FALSE;
19  SoftLDSt_aux := FALSE;
20  END_IF;
21
22  (* Auto Mode *)
23  IF (MMoSt_aux AND (E_MAuMoR OR E_AuAuMoR )) OR (FoMoSt_aux AND E_MAuMoR)
    OR (SoftLDSt_aux AND E_MAuMoR) OR (MMoSt_aux AND AuIhMMo) OR (
    FoMoSt_aux AND AuIhFoMo) OR (SoftLDSt_aux AND AuIhFoMo) OR NOT(
    AuMoSt_aux OR MMoSt_aux OR FoMoSt_aux OR SoftLDSt_aux) THEN
24  AuMoSt_aux := TRUE;
25  MMoSt_aux := FALSE;
26  FoMoSt_aux := FALSE;
27  SoftLDSt_aux := FALSE;
28  END_IF;
29
30  (* Software Local Mode *)
31  IF (AuMoSt_aux OR MMoSt_aux) AND E_MSoftLDR AND NOT AuIhFoMo THEN
32  AuMoSt_aux := FALSE;
33  MMoSt_aux := FALSE;
34  FoMoSt_aux := FALSE;
35  SoftLDSt_aux := TRUE;
36  END_IF;
37
38  (* Status setting *)
39  LDSt := FALSE;
40  AuMoSt := AuMoSt_aux;
41  MMoSt := MMoSt_aux;
42  FoMoSt := FoMoSt_aux;
43  SoftLDSt := SoftLDSt_aux;
44  ELSE
45  (* Local Drive Mode *)
46  AuMoSt := FALSE;
47  MMoSt := FALSE;
48  FoMoSt := FALSE;

```

```

49 LDSt := TRUE;
50 SoftLDSt:= FALSE;
51 END_IF;

```

Listing 4.1: OnOff mode manager program

### 4.3.2.1 First requirement

The first requirement, that is derived from the specification of the object, is shown in Fig. 4.6, is the following:

“If there is a rising edge on the AuAuMoR (Auto mode request initiated by the logic) input, the mode will be changed to Auto.”

This requirement is ambiguous, thus some refinement is needed in order to be able to be formalized and verified:

“If the current mode is Auto, Manual or Forced mode and there is a rising edge on the AuAuMoR (Auto mode request initiated by the logic) input, but there are no other mode change requests, the mode will be Auto at the end of the cycle.”

The corresponding pattern for this requirement is the pattern TL7, defined in Section 3.4:

If \_\_\_\_\_ [1] is true at the end of cycle N and \_\_\_\_\_ [2] is true at the end of cycle N+1, then \_\_\_\_\_ [3] is always true at the end of cycle N+1.

and this is the corresponding LTL formula:

$$\begin{aligned}
 & G \left( (EoC \wedge [1] \wedge \times (\neg EoC \cup (EoC \wedge [2]))) \right. \\
 & \left. \rightarrow \times (\neg EoC \cup (EoC \wedge [3])) \right)
 \end{aligned}$$

with the corresponding PLC variables:

- [1]: not AuAuMoR and (AuMoSt or MMoSt or FoMoSt)
- [2]: AuAuMoR and (not MAuMoR and not MMMoR and not HLD and not MSoftLDR and not MFoMoR)
- [3]: AuAuMoR

Where  $EoC$  means “end of PLC cycle” and is true only at the end of PLC cycles.

The resulting LTL formula is the following:

$$\begin{aligned} & \mathbf{G} \left( (EoC \wedge (\neg AuAuMoR \wedge (AuMoSt \vee MMoSt \vee FoMoSt)) \right. \\ & \wedge (\neg MAuMoR \wedge \neg MMMoR \wedge \neg HLD \\ & \wedge \neg MSoftLDR \wedge \neg MFoMoR)) \\ & \wedge \mathbf{X} (\neg EoC \mathbf{U} (EoC \wedge AuAuMoR \\ & \wedge (AuMoSt \vee MMoSt \vee FoMoSt) \\ & \wedge (\neg MAuMoR \wedge \neg MMMoR \wedge \\ & \neg HLD \wedge \neg MSoftLDR \wedge \neg MFoMoR))) \\ & \left. \rightarrow \mathbf{X} (\neg EoC \mathbf{U} (EoC \wedge AuMoSt)) \right) \end{aligned}$$

Note that in the PLC code  $MAuMoR$  corresponds to  $ManReg01[8]$ ,  $MMMoR$  corresponds to  $ManReg01[9]$ ,  $MSoftLDR$  corresponds to  $ManReg01[10]$  and  $MFoMoR$  corresponds to  $ManReg01[11]$ .  $ManReg01$  is an input variable which contains the manual requests from the SCADA operator.

Once the requirement is formalized, the formal models are produced. Table 4.2 shows some key metrics regarding the size of the models and the generation times for the case study. Three different models are compared here:

- $M_1$ : a “non-reduced model” without any model reductions.
- $M_2$ : a “reduced model”, which is still general (not property specific) as COI has not been applied.
- $M_3$ : a model tailored to the current example requirement, which is only suitable for the evaluation of a reduced set of requirements (requirements which contain the same variables). COI has been applied.

By using property preserving reduction techniques, the size of the PSS size of the *OnOff* object can be reduced from  $1.61 \cdot 10^{218}$  to  $3.65 \cdot 10^{10}$ , without affecting the verification result.

The run time of the model generation including the ST code parsing, the reductions, and the generation of the nuXmv inputs does not

Table 4.2: Model generation metrics

IM metrics	Models		
	No red. ( $M_1$ )	+ General red. ( $M_2$ )	+ COI ( $M_3$ )
Variables	259	118	33
Automata	9	1	1
locations	460	323	17
PSS	$1.61 \cdot 10^{218}$	$4.57 \cdot 10^{36}$	$3.65 \cdot 10^{10}$
Generation time	0.3 s	11.3 s	12.6 s

take more than 10–15 seconds, which is orders of magnitude lower than the achieved reduction in the verification time.

The verification results for this requirement are presented in Section 4.3.3.

#### 4.3.2.2 Second requirement

The second experimental result corresponds with the following requirement:

“If the object is controlled locally (is in the so-called Hardware Local mode) and there is no interlock, nor explicit output change request valid in this mode, the output keeps its value”

The requirement is formalized using the pattern TL7 (the same as the previous requirement), and it uses the following values:

If \_\_\_\_\_ [1] is true at the end of cycle N and \_\_\_\_\_ [2] is true at the end of cycle N+1, then \_\_\_\_\_ [3] is always true at the end of cycle N+1.

with the corresponding PLC variables:

[1]: not OutOnOV and not TStopI and not FuStopI and not StartI

[2]: HLD and not HOnR and not HOffR and not TStopI and not FuStopI and not StartI

[3]: not OutOnOV

Table 4.3: Model generation metrics

IM metrics	Models		
	No red. ( $M_1$ )	+ General red. ( $M_2$ )	+ COI ( $M_3$ )
Variables	259	118	33
Automata	9	1	1
locations	460	323	17
PSS	$1.61 \cdot 10^{218}$	$4.3 \cdot 10^{36}$	$4.3 \cdot 10^{26}$
Generation time	0.2 s	0.4 s	0.6 s

The corresponding LTL formula is the following:

$$\mathbf{G} \left( \left( \mathbf{E}oC \wedge \neg \mathbf{O}ut\mathbf{O}n\mathbf{O}V \wedge \neg \mathbf{T}Stop\mathbf{I} \wedge \neg \mathbf{F}u\mathbf{S}top\mathbf{I} \wedge \neg \mathbf{S}tart\mathbf{I} \wedge \right. \right. \\ \left. \left. \mathbf{H}LD \wedge \mathbf{X}(\neg \mathbf{E}oC \mathbf{U}(\mathbf{E}oC \wedge \mathbf{H}LD \wedge \neg \mathbf{H}O\mathbf{n}R \wedge \neg \mathbf{H}O\mathbf{f}fR \wedge \right. \right. \\ \left. \left. \neg \mathbf{T}Stop\mathbf{I} \wedge \neg \mathbf{F}u\mathbf{S}top\mathbf{I} \wedge \neg \mathbf{S}tart\mathbf{I})) \right) \rightarrow \right. \\ \left. \mathbf{X}(\neg \mathbf{E}oC \mathbf{U}(\mathbf{E}oC \wedge \neg \mathbf{O}ut\mathbf{O}n\mathbf{O}V)) \right)$$

Table 4.3 shows the metrics for this second requirement. Before the reductions, the PSS size of this baseline object is  $1.6 \cdot 10^{218}$ . After the general and requirement-dependent reductions, the PSS contains  $4.3 \cdot 10^{26}$  states of which  $4.9 \cdot 10^{14}$  are reachable. The generation of the model including all the reductions takes 0.6 s.

### 4.3.3 Experimental results regarding verification of complex properties

When talking about complex properties, we reference properties that need several temporal operators to be formalized, for example, the formulas presented in the previous subsection. These requirements represent sequences of values from different PLC cycles and this was one of the main goals when designing this methodology. The two requirements presented before were verified using the nuXmv model checker, integrated in the methodology.

nuXmv, as well as NuSMV, provides various parameters to fine-tune the verification hence the run time highly depends on the used parameters. Without any given parameters, nuXmv will explore the

full state space before the evaluation of the requirement. However, if this is not necessary for the verification, it can be disabled by the *-df* parameter. The *-dcx* parameter disables the generation of the counterexample that makes the verification time smaller, but we cannot profit from the information given by the counterexample. The *-dynamic* parameter enables the dynamic reordering of the variables, which can drastically reduce the memory consumption of the tool.

Table 4.4 shows verification time results for the first requirement on the *OnOff* object. As it is shown in the table, the verification of the non-reduced model ( $M_1$ ) was not successful. The requirement can be checked on the general reduced model, taking 160 s with counterexample generation. However, much smaller verification times can be achieved by using the reduced model specific to the given requirement ( $M_3$ ). In this case, the verification and the counterexample generation took less than 1 second together. (The symbol “—” indicates an execution longer than 24 hours). This table shows the difference of verification performance when using the different nuXmv parameters.

For the introduced example requirement, the result of the model checker is false, thus the requirement is not satisfied.

Table 4.4: Verification run time for the first requirement from Fernández Adiego et al. (2014c)

Metric	Non-reduced model ( $M_1$ )	Reduced model ( $M_2$ )	Specific model ( $M_3$ )
no parameters	—	—	8.398 s
-dynamic	—	$\approx 7$ h	1.334 s
-dynamic -df	—	160.8 s	0.547 s
-dynamic -df -dcx	—	3.787 s	0.141 s

For the second requirement, as it is shown in Table 4.5, the evaluation of the requirement takes 6.1 s using nuXmv without counterexample generation, which showed that the requirement is not satisfied. If the counterexample is generated too, the run time is 19.4 s.

In general, for the baseline objects the property preserving reduction techniques (*COI*, the *general rule-based* reductions and *mode selection*) were enough to verify these objects. For this case study, the *variable abstraction* technique was not applied. In Section 4.4, the variable abstraction technique has been applied to verify safety prop-

Table 4.5: Verification run time for the second requirement

Metric	Non-reduced model ( $M_1$ )	Reduced model ( $M_2$ )	Specific model ( $M_2$ )
Verification (-dynamic -df -dcx)	—	—	19.4 s (6.1 s without c.ex.)

erties on complete UNICOS PLC programs and the results are presented.

As it was mentioned before, more than fifty different requirements, stated by the UNICOS developers, have been verified on the *OnOff* baseline object. These experiments identified several faults in this well-tested base object used all over the CERN control systems.

### 4.3.4 Experimental results regarding verification of time properties

Verification of time related requirements on the *OnOff* are presented in this section. These results can be found in the publication Fernández Adiego et al. (2014a). As it was described on Section 3.9, two different approaches for modeling the timing aspects of PLCs are included in the methodology. If the requirement contains explicit time in it, the realistic approach is applied. If it does not, the abstract approach is applied. Experiments with both approaches are presented.

#### 4.3.4.1 Realistic Approach

Having the following requirement:

“If there is a rising edge on  $\mathcal{C}_1$ , after  $t_m$  units of time  $\mathcal{C}_2$  will be true, if  $\mathcal{C}_1$  remained true”.

where,  $\mathcal{C}_1$ ,  $t_m$  and  $\mathcal{C}_2$  corresponds with the signals shown in Fig. 4.7. the selected pattern is the *TL10 (TON-like property)*, where  $\mathcal{C}_1$  and  $\mathcal{C}_2$  corresponds to the following signals of the *OnOff* object:

$\mathcal{C}_1 = \text{HFOff}$  and  $\text{HFOn}$  and  $\text{not}(\text{Re\_OutOvSt\_aux})$  and  $\text{not}(\text{Fe\_OutOvSt\_aux})$ ;

$C_2 = \text{PosW}$ ;

$t_m$  is the monitor delay time. 3 different experiments are performed for this requirement: 10 s, 9 s and 1 s.

In addition, the following parameters (mode selection) are set:

PPulse = FALSE;

POutOff = FALSE;

PHLD = FALSE;

PHFOff = TRUE;

PHFO<sub>n</sub> = TRUE;

PFsPosOn = TRUE;

PWDt = 10 s;

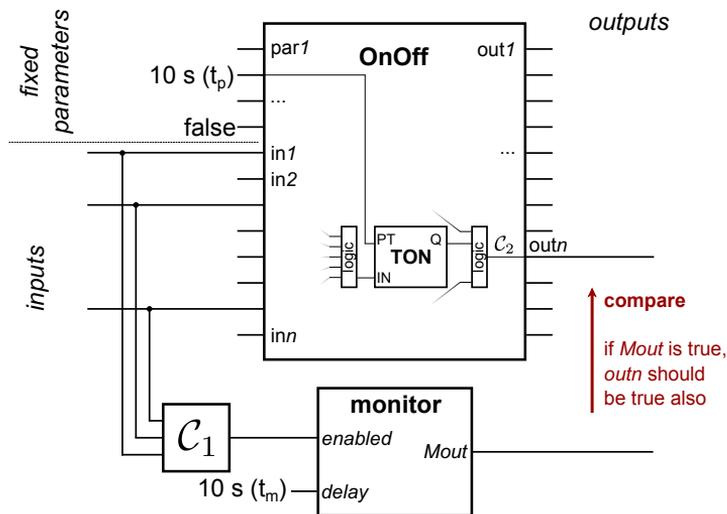


Figure 4.7: Verification configuration for OnOff model extended with monitor

Table 4.6 presents different state spaces depending on which abstraction and reduction techniques are applied.

To be able to verify this property, the original model was shrink from a PSS of  $1.6 \cdot 10^{218}$  states to a PSS of  $1.1 \cdot 10^{36}$  as can be seen in Table 4.6. In order to do so, reductions, fixed parameters (focusing on a specific scenario), and the COI technique (which eliminated 2 timers out of 3) have been applied. As it was mentioned before, time values were modeled as 16-bits variables instead of 32-bits.

Table 4.6: State space of the OnOff model with realistic timer representation from Fernández Adiego et al. (2014a)

Time	Monitor	Reductions	PSS	#Vars
32 bit	–	none	$1.6 \cdot 10^{218}$	255
16 bit	–	none	$2.5 \cdot 10^{160}$	255
16 bit	–	general	$1.6 \cdot 10^{134}$	185
16 bit	+	general	$8.5 \cdot 10^{139}$	189
16 bit	+	general, COI	$1.1 \cdot 10^{85}$	143
16 bit	+	gen., fix params, COI	$1.1 \cdot 10^{36}$	86

Table 4.7 presents verification results for the *OnOff* model after applying all the previously mentioned reduction techniques. The requirement has been checked with three different configurations in terms of timer delay ( $t_p$ ) and monitor delay ( $t_m$ ). The first uses  $t_p = t_m = 10$  s values, which means the parameter  $t_p$  of OnOff and the delay parameter  $t_m$  of the monitor were both 10 s. In this case, the result is true, which is given by nuXmv in 11.4 s. The other two configurations use monitor delay times smaller than the parameter  $t_p$ , therefore the output is expected too early, thus the result should be false. As it can be seen in Table 4.7, in both cases the verification time was around 5 to 20 s, but the generation of the counterexample (C.ex. gen. time) took significantly more time. In the case when we used  $t_m = 9$  s, the counterexample was longer (3876 steps), and its generation time was around 15 times bigger than when we used  $t_m = 1$  s. Notice that these verification results can have a timing error not bigger than the maximal cycle time, 100 ms in this case, because the current time is incremented in quanta.

#### 4.3.4.2 Abstract Approach

This approach has also been applied to the *OnOff* object from the UNICOS library. Three experiments are presented in this section to evaluate the abstract approach.

For Experiment 1 (see Table 4.9), the same reduction techniques and the same fixed parameters as for the realistic approach have been used. The PSS of the *OnOff* model became  $5.5 \cdot 10^{24}$  (instead of

Table 4.7: Verification time on OnOff model with realistic time representation from Fernández Adiego et al. (2014a)

Timer delay ( $t_p$ )	Monitor delay ( $t_m$ )	Result	Run time	C.ex. gen. time	C.ex. length
10 s	10 s	true	11.4 s	—	—
10 s	9 s	false	19.5 s	2513.2 s	3 876
2 s	1 s	false	5.2 s	123.0 s	510

$1.1 \cdot 10^{36}$ ). Representing the requirement checked on the realistic model is not possible in this approach as the time is not counted explicitly. Instead, we can check the following liveness property (pattern TL):

“if  $\mathcal{C}_1$  is sometime true and remains true forever, eventually  $\mathcal{C}_2$  will be true”.

The equivalent LTL property is:

$$F\left(G(EoC \rightarrow \mathcal{C}_1) \rightarrow F(\mathcal{C}_2)\right)$$

In this case, no monitor is needed and  $\mathcal{C}_1$  and  $\mathcal{C}_2$  corresponds to the same signals as for the realistic approach:

$$\begin{aligned} \mathcal{C}_1 &= \text{HFOff} \text{ and } \text{HFOn} \text{ and } \text{not}(\text{Re\_OutOvSt\_aux}) \text{ and } \\ &\text{not}(\text{Fe\_OutOvSt\_aux}); \\ \mathcal{C}_2 &= \text{PosW}; \end{aligned}$$

Two other experiments on the model with abstract time representation are presented (Exp. 2, 3). In these cases, no parameters were fixed and all the three timers had effect on the requirement. The requirement 2 was a simple safety requirement in CTL ( $AG(\mathcal{C}_3 \rightarrow \mathcal{C}_4)$ ), while requirement 3 was a bit more complex:  $AG(\mathcal{C}_5 \rightarrow AF(\mathcal{C}_6))$ .

The experiments (c.f. Table 4.9) show that these requirements can be checked using the abstract time representation, even without fixing any parameters. With the realistic time representation, the state space would be too large to be verified using nuXmv.

Table 4.8 presents different state spaces depending on which reductions techniques are applied and what requirement was verified. Table 4.9 shows the measured run times.

Table 4.8: State space of the OnOff model with abstract time representation from Fernández Adiego et al. (2014a)

Reductions	PSS	#Vars
none	$1.1 \cdot 10^{131}$	243
general	$4.7 \cdot 10^{112}$	164
general, COI, fix params (Exp. 1)	$5.5 \cdot 10^{24}$	77
general, COI (Exp. 2, 3)	$1.1 \cdot 10^{40}$	126

Table 4.9: Verification time on OnOff model with abstract time representation from Fernández Adiego et al. (2014a)

Exp.	#Timers	Result	Run time	C.ex.
(1)	1	true	6.1 s	—
(2)	3	false	294.2 s	18.6 s
(3)	3	false	4 201.9 s	12 020.9 s

### 4.3.5 Counterexample analysis

An example of last step of the methodology is presented here, the counterexample analysis. The selected counterexample corresponds to the verification case presented previously (See Section 4.3.3). This experiment was also published in Fernández Adiego et al. (2014c). The requirement for the verification case is the following:

$$\begin{aligned}
& \mathbf{G} \left( (EoC \wedge (\neg AuAuMoR \wedge (AuMoSt \vee MMoSt \vee FoMoSt)) \right. \\
& \wedge (\neg MAuMoR \wedge \neg MMMoR \wedge \neg HLD \\
& \wedge \neg MSoftLDR \wedge \neg MFoMoR)) \\
& \wedge \mathbf{X} (\neg EoC \cup (EoC \wedge AuAuMoR \\
& \wedge (AuMoSt \vee MMoSt \vee FoMoSt) \\
& \wedge (\neg MAuMoR \wedge \neg MMMoR \wedge \\
& \neg HLD \wedge \neg MSoftLDR \wedge \neg MFoMoR))) \\
& \left. \rightarrow \mathbf{X} (\neg EoC \cup (EoC \wedge AuMoSt)) \right)
\end{aligned}$$

The generated counterexample by nuXmv contains the value of each variable after each transition in the model (corresponding to the execution of one instruction in the PLC code). This detailed counterexample contains about 5000 lines for the specific model ( $M_3$ ) and

37500 lines for the generic model ( $M_2$ ). These are too detailed for human analysis. However, they can be reduced, as it is enough to know the value of the input and output variables at the end of each PLC cycle, thus the variable valuations of the intermediate states can be discarded. By removing the unnecessary states and the temporary variables, the counterexample of the current requirement can be reduced to approximately 100 lines.

The counterexample can also be used to demonstrate the problem in a real environment. Using the nuXmv's counterexample, a demonstrator PLC code can be easily generated by adding a module simulating the variables from the counterexample and checking the expected result. By doing so, it can be demonstrated that the counterexample is not caused by a mistake during the model generation or due to bad reductions. In this case study, the counterexample given by nuXmv was confirmed in a real PLC, showing that the model and the real system are equivalent for this specific requirement.

Listing 4.2 and Listing 4.3 show an extract from the generated counterexample and the corresponding part of the PLC demonstrator source code.

Furthermore, the PLC demonstrator can help the engineer to reduce the counterexample. Usually, most of the input variables do not have any effect on the evaluated requirement. Those variables can be fixed to constant values that can help focusing on the input value changes that causes the violation of the requirement.

```

1 -> State: 1.17 <-
2   inst.HLD = FALSE
3   inst.ManReg01[8] = TRUE
4   inst.ManReg01[9] = FALSE
5   inst.ManReg01[10] = FALSE
6   inst.ManReg01[11] = FALSE
7   inst.AuAuMoR = FALSE
8   inst.AUIHMMO = FALSE
9   inst.AUIHFOMO = FALSE
10  ...

```

Listing 4.2: Extract from the counterexample

```

1 onoff1.HLD := FALSE;
2 onoff1.ManReg01 :=
3     2#00000000000000001;
4 onoff1.AuAuMoR := FALSE;
5 onoff1.AuIhMMo := FALSE;
6 onoff1.AuIhFoMo := FALSE;
7 ...
8 CPC_FB_ONOFF.onoff1();
9 // Check:
10 b := onoff1.AuMoSt;

```

Listing 4.3: Extract from the PLC demonstrator

In the current case, by using the counterexample and the reduced PLC demonstrator we were able to identify the source of the problem.

It turned out that from *Forced mode* the logic cannot switch to *Auto mode*, it can only be done by the operator. The corresponding part of the implementation can be seen in Listing 4.4<sup>1</sup>. The highlighted part shows the code implementing the transition from Forced mode to Auto mode, which does not take into account the AuAuMoR input. This behavior is correct and intentional, in this case the specification contained a mistake.

```

1 IF (MMoSt_aux AND (E_MAUmoR OR E_AuAuMoR)) OR
2   (FoMoSt_aux AND E_MAUmoR) OR
3   (SoftLDSt_aux AND E_MAUmoR) OR
4   (MMoSt_aux AND AuIhMMo) OR
5   (FoMoSt_aux AND AuIhFoMo) OR
6 NOT (AuMoSt_aux OR MMoSt_aux OR
7     FoMoSt_aux OR SoftLDSt_aux) THEN
8   (* Setting mode to Auto *)
9   AuMoSt_aux := TRUE;
10  MMoSt_aux := FALSE;
11  FoMoSt_aux := FALSE;
12  SoftLDSt_aux := FALSE;
13 END_IF;

```

Listing 4.4: Extract of the source code causing the violation of the requirement

---

<sup>1</sup>In the source code, the variables starting with “E\_” indicate the rising edges of the corresponding inputs. AuMoSt\_aux is true if the current mode is Auto. The meaning of variables MMoSt\_aux, FoMoSt\_aux, and SoftLDSt\_aux are similar, with the other modes.

## 4.4 Full UNICOS application case study

This section presents the experimental results obtained on one of LHC cryogenics control system. The goal of this section is to complement the experimental results presented in Section 4.3 for large PLC programs where the property preserving reduction techniques included in the methodology are not enough to verify the program. In this section, the *variable abstraction* technique is applied to verify the PLC program. The section is divided in two main parts:

- Description of the QSDN control system (Section 4.4.1).
- Experimental results of safety properties (Section 4.4.2).

### 4.4.1 Process description and specification

The so-called QSDN<sup>2</sup> application has been selected as a case study of a complete PLC program. This application controls a subsystem of the LHC cryogenics process.

Fig. 4.8 represents the QSDN subsystem consisting in two nitrogen storage vessels. These vessels provide liquid nitrogen to the cryogenic plants via two digital valves (xPV409). In addition, each vessel is filled from a nitrogen truck and the internal pressure of vessels is regulated by an electrical heater (xEH400). Each vessel has also a gas outlet (xPV408) to provide warm gaseous nitrogen.

The size of QSDN PLC program is rather representative for medium UNICOS control systems. It contains 110 functions and function blocks, and consists of approximately 17,500 lines of code. Before reductions, these results in a huge generated model: the IM contains 302 automata, and the PSS size is  $10^{31985}$  (see  $M_1$  in Table 4.10).

When verifying full UNICOS control systems, we assume that the base objects from the UNICOS library are already verified, i.e. their specification is satisfied. Therefore the goal is to check the application-specific logic of the PLC program. This program is implemented using ST and SFC languages (SCL and Graph for Siemens PLCs). This logic is described in the *UNICOS Functional Analysis*, which is a semi-formal textual specification. We extracted the requirements from this

---

<sup>2</sup>QSDN is a codename for Cryogenics Surface Liquid Nitrogen Storage System.

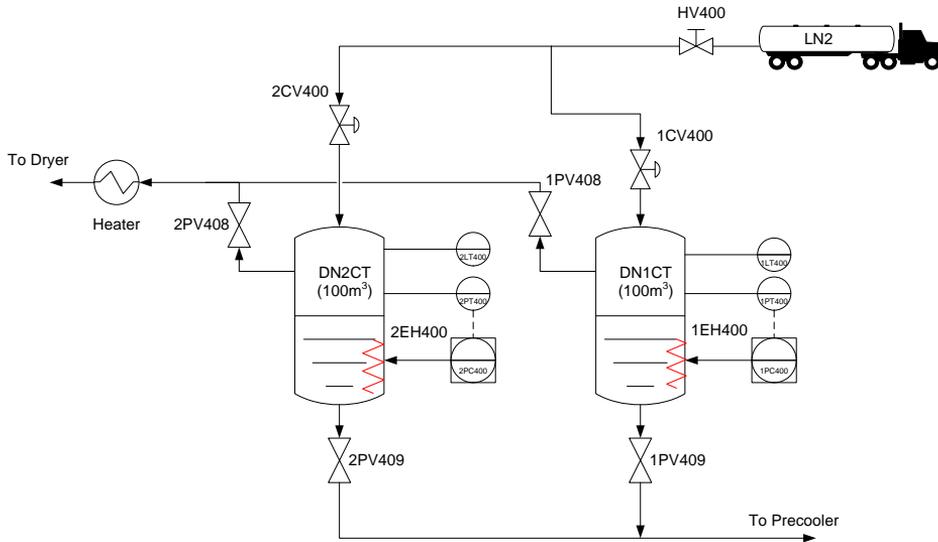


Figure 4.8: QSDN process

specification regarding to specific logic. The extracted requirements are safety properties to check the specific logic of this PLC program.

#### 4.4.2 Experimental results regarding verification of safety properties

This section presents the experimental results of three safety requirements, extracted from the *Functional Analysis* of the QSDN control system. These requirements are typically similar to the following structure:

“if  $\alpha$  is TRUE, then  $\beta$  is TRUE”

The corresponding CTL formula for this example is:

$$AG(\alpha \rightarrow \beta)$$

Where the Greek letters mark Boolean expressions of variables. The property preserving reduction techniques (i.e. COI, general rule-based reduction and mode selection) remove all the unnecessary variables regarding the given property and the automata network is simplified. The requirements presented in this section correspond to the

so-called “dependent logic” of one of the actuator of the QSDN system: the QSDN\_4\_1PV408, which corresponds to a on-off valve (modeled by an *OnOff* object for the UNICOS model).

Listing 4.5 shows the relevant part of the QSDN PLC code related to the three requirements. However, this piece of code is not enough to prove that the requirements hold or not on this QSDN PLC program, so information from the rest of the program is also needed.

```

1 (* Position Management*)
2 QSDN_4_1PV408.AuOnR:=
3     QSDN_4_DN1CT_SEQ_DB.ValvesOn.x OR
4     QSDN_4_DN1CT_SEQ_DB.Run.x OR
5     QSDN_4_DN1CT_SEQ_DB.OkSignalOff.x ;
6
7 QSDN_4_1PV408.AuOffR:= NOT QSDN_4_1PV408.AuOnR;

```

Listing 4.5: Excerpt of QSDN code relevant to the case study

#### 4.4.2.1 First requirement

The first requirement to be checked is the following:

“If QSDN\_4\_DN1CT\_SEQ\_DB.Stop.x is true (at the end of a scan cycle), QSDN\_4\_1PV408.AuOffR should be true also.”

The corresponding patterns is TL1 and the corresponding temporal logic formula  $p$  is:

$$\text{AG} \left( (EoC \wedge \text{QSDN\_4\_DN1CT\_SEQ\_DB.Stop.x}) \rightarrow \text{QSDN\_4\_1PV408.AuOffR} \right)$$

Table 4.10 presents relevant metrics about the generated model. The original state space is huge and the original model obviously cannot be verified (see  $M_1$ ). After the COI reduction 3757 variables are kept in the reduced model (see model  $M_2$  with a PSS size of  $10^{5048}$  states). Formal verification is still not possible. Therefore the *variable abstraction* technique has to be applied.

Fig. 4.9 shows the variable dependency graph for the given requirement with the distance between variables. The gray variables are part of the requirement (i.e. QSDN\_4\_1PV408.AuOffR and QSDN\_4\_DN1CT\_SEQ\_DB.Stop.x). The red edges represent assignment

dependencies (e.g.  $a := b$  will imply a “ $b \rightarrow a$ ” red edge). The blue edges represent conditional dependencies (e.g. *IF a THEN  $b := c$*  will imply a “ $b \rightarrow a$ ” blue edge).

Table 4.10: Metrics of the models of QSDN

Metric	Non-reduced model ( $M_1$ )	Reduced model ( $M_2$ )
PSS	$10^{31985}$	$10^{5048}$
Variables	31,402	3757
Generation	4.2 s	15.3 s

Fig. 4.10 shows the relationship between the abstract models and the maximum number of reachability properties to be checked if *variable abstraction* is applied (or the maximum number of possible invariants).

Table 4.11 shows the verification results when applying the *variable abstraction* technique. With this technique described in Section 3.7.4, it is proved that the property is TRUE. The experiment is performed with a TO of 30 s. The answer is given by the tool in 45.309 s (which 30 of these seconds corresponds to the TO obtained when checking the reachability on the  $OM''$  model).

To verify this requirement, six steps of the variable abstraction technique are needed (according with the steps defined in the subsection 3.7.4):

- **Step 1:**  $p$  is verified on the first abstraction  $AM'_1$  and the result is FALSE.
- **Step 2:**  $q$  is verified on  $AM'_1$  and the result is FALSE.  $q$  corresponds with the following CTL property:

$$\text{AG} \left( (EoC \wedge \text{QSDN\_4\_DN1CT\_SEQ\_DB.Stop.x}) \right. \\ \left. \rightarrow \neg \text{QSDN\_4\_1PV408.AuOffR} \right)$$

- **Step 3:** The extracted counterexample  $c$  is transformed in a reachability property  $r$  and verified on  $OM''$ . After 30 s a TO

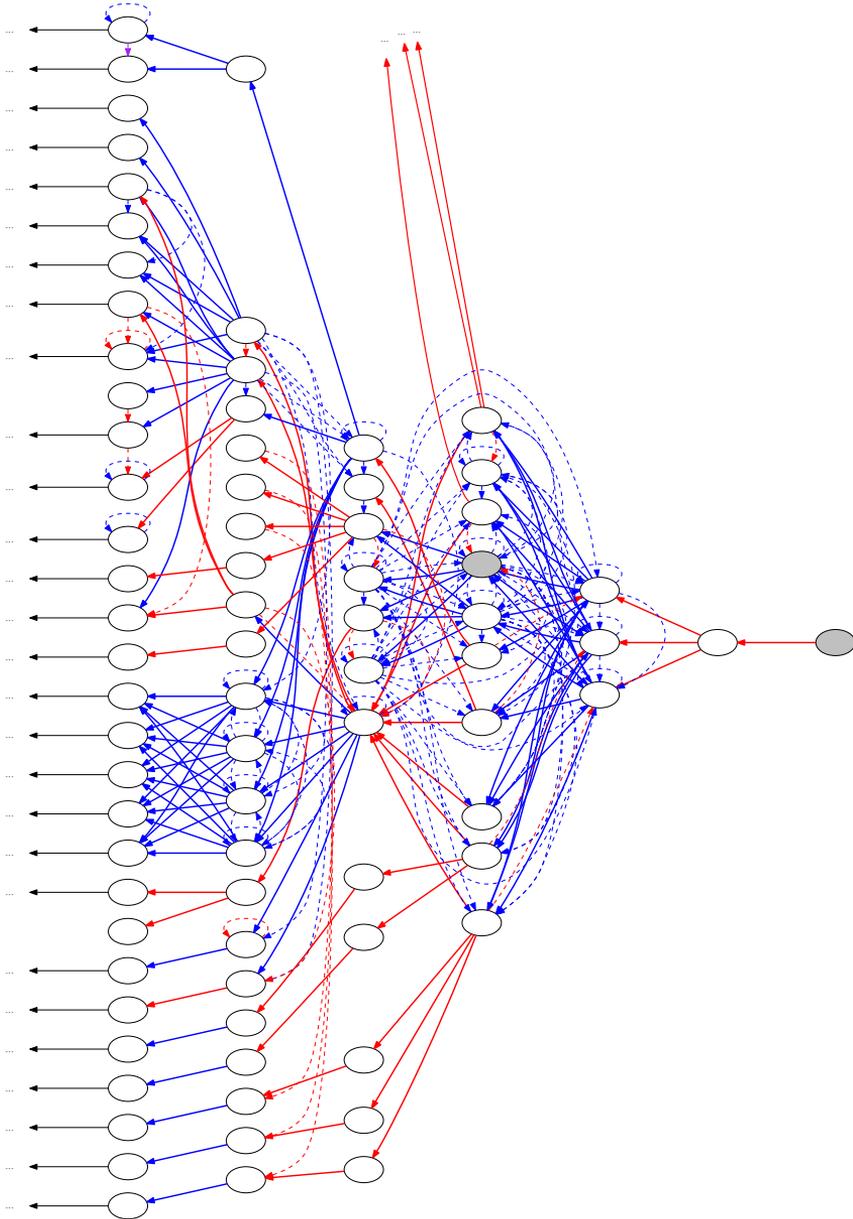


Figure 4.9: Variable dependency graph for the given requirement of QSDN

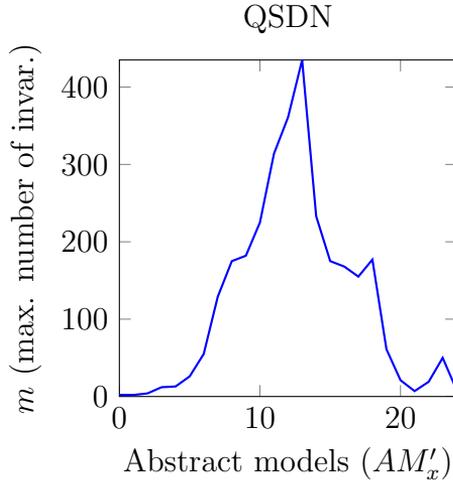


Figure 4.10: Relationship between the abstract models and the maximum number of reachability properties for the variable abstraction technique  $m$

is reached by nuXmv.  $r$  corresponds with the following CTL property:

$$\text{EF} \left( EoC \wedge \text{QSDN\_4\_DN1CT\_SEQ\_DB.Stop.x} \wedge \text{QSDN\_4\_1PV408.AuOnR} \right)$$

- **Step 4:** The same reachability property is verified on  $AM'_1$  (first abstraction) and the result is TRUE. A new iteration is needed.
- **Step 4:** The same reachability property is verified on  $AM''_2$  (second abstraction) and the result is FALSE. It means the first counterexample is spurious.
- **Step 5:** the number of potential invariants for  $AM'_1$  is smaller than 10 (see Fig. 4.10), then an invariant is added to  $AM'_1$ .
- *New iteration.* **Step 1:**  $p$  is verified again on  $AM'_1$  with the new invariant. The result is TRUE, proving that the property also holds on the original model  $OM'$ . This abstract model is shown in the Listing B.2 in the Appendix B.

Table 4.11: Verification time on *QSDN* model

Model	PSS	Model gen.	nuXmv run-time	Ver. result
$p$ on $AM'_1$	$2.4 \cdot 10^1$	7.549 s	0.030 s	FALSE
$q$ on $AM'_1$	$2.4 \cdot 10^1$	0 s	0.033 s	FALSE
$r$ on $OM''$	$7.8 \cdot 10^{4872}$	18.301 s	TO	—
$r$ on $AM''_1$	$5.9 \cdot 10^6$	9.147 s	0.049 s	TRUE
$r$ on $AM''_2$	$4.28 \cdot 10^{30}$	8.220 s	1.340 s	FALSE
$p$ on $AM'_1 + \text{invar.}$	$2.4 \cdot 10^1$	0 s	0.053	TRUE

#### 4.4.2.2 Second requirement

The second requirement extracted from the functional analysis is the following:

“If `QSDN_4_DN1CT_SEQ_DB.Run.x` is true (at the end of a scan cycle), `QSDN_4_1PV408.AuOnR` should be true also.”

The corresponding patterns is 3.4.1.1 and the corresponding temporal logic formula is:

$$\text{AG} \left( (EoC \wedge \text{QSDN\_4\_DN1CT\_SEQ\_DB.Run.x}) \rightarrow \text{QSDN\_4\_1PV408.AuOnR} \right)$$

The model after the property preserving reduction techniques has a PSS size of  $10^{5048}$  as the previous example. The Table 4.12 shows the verification performance for each step when applying this technique.

- **Step 1:**  $p$  is verified on the first abstraction  $AM'_1$  and the result is TRUE. This abstract model is shown in the Listing B.3 in the Appendix B.

In this case, the answer is given by the tool in 7.544 s as the property holds in first abstraction of this technique.

#### 4.4.2.3 Third requirement

The third requirement extracted from the functional analysis is presented to show how the algorithm can prove that a property does not hold. The requirement is the following:

Table 4.12: Verification time on *QSDN* model

Model	PSS	Model gen.	nuXmv run-time	Ver. result
$p$ on $AM'_1$	$4.8 \cdot 10^1$	7.521 s	0.023 s	TRUE

“If  $QSDN\_4\_DN1CT\_SEQ\_DB.Run.x$  is true (at the end of a scan cycle),  $QSDN\_4\_1PV408.AuOffR$  should be true also.”

The corresponding patterns is 3.4.1.1 and the corresponding temporal logic formula is:

$$\begin{aligned} & \text{AG} \left( (EoC \wedge QSDN\_4\_DN1CT\_SEQ\_DB.Run.x) \right. \\ & \left. \rightarrow QSDN\_4\_1PV408.AuOffR \right) \end{aligned}$$

The model after COI has a PSS size of  $10^{5048}$  as well as the previous examples. In this case, this technique can confirm that this property does not hold and a counterexample was provided in 126.573 s. The summary of all the steps performed to evaluate this property is the following:

- **Step 1:**  $p$  is verified on the first abstraction  $AM'_1$  and the result is FALSE.
- **Step 2:**  $q$  is verified on  $AM'_1$  and the result is FALSE.  $q$  corresponds with the following CTL property:

$$\begin{aligned} & \text{AG} \left( (EoC \wedge QSDN\_4\_DN1CT\_SEQ\_DB.Run.x) \right. \\ & \left. \rightarrow \neg QSDN\_4\_1PV408.AuOffR \right) \end{aligned}$$

- **Step 3:** The extracted counterexample  $c$  is transformed in a reachability property  $r$  and verified on  $OM''$ . After 30 s a TO is reached by nuXmv.  $r$  corresponds with the following CTL property:

$$\begin{aligned} & \text{EF} \left( EoC \wedge QSDN\_4\_DN1CT\_SEQ\_DB.Run.x \wedge \right. \\ & \left. QSDN\_4\_1PV408.AUOnR \right) \end{aligned}$$

- **Step 4:** The same reachability property is verified on  $AM_1'$  (first abstraction) and the result is TRUE. A new iteration is needed.
- **Step 4:** The same reachability property is verified on  $AM_2''$  (second abstraction) and the result is TRUE. A new iteration is needed.
- **Step 4:** The same reachability property is verified on  $AM_3''$  (third abstraction) and a TO is reached and no result is provided.
- New iteration. **Step 1:**  $p$  is verified on a new abstraction  $AM_2'$ . The result is FALSE and a new counterexample  $c$  is produced.
- **Step 2:**  $q$  is verified on  $AM_2'$  and the result is TRUE. These results implies that  $p$  does not hold on  $AM_2'$  and on  $OM'$ . The counterexample  $c$  is real. This abstract model is shown in the Listing B.4 in the Appendix B.

The Table 4.13 shows the verification performance for each step when applying this technique.

Table 4.13: Verification time on *QSDN* model

Model	PSS	Model gen.	nuXmv run-time	Ver. result
$p$ on $AM_1'$	$2.4 \cdot 10^1$	9.471 s	0.107 s	FALSE
$q$ on $AM_1'$	$2.4 \cdot 10^1$	0 s	0.028 s	FALSE
$r$ on $OM''$	$7.8 \cdot 10^{4872}$	19.580 s	TO	—
$r$ on $AM_1''$	$2.95 \cdot 10^6$	9.510 s	0.076 s	TRUE
$r$ on $AM_2''$	$1.34 \cdot 10^{29}$	8.232 s	1.260 s	TRUE
$r$ on $AM_3''$	$5.11 \cdot 10^{80}$	10.436 s	TO	—
$p$ on $AM_2'$	$4.8 \cdot 10^1$	7.825 s	0.026 s	FALSE
$q$ on $AM_2'$	$4.8 \cdot 10^1$	0 s	0.022 s	TRUE

As can be seen, our verification method can be used for isolated verification of modules or for verification of complete PLC applications. Approximately 30 different requirement were extracted from the QSDN functional analysis document.

## 4.5 Summary of the chapter

This chapter presented the experimental results of applying the proposed methodology to two real-life PLC programs developed at CERN using the UNICOS framework.

The first PLC program corresponds with one of the UNICOS CPC baseline objects, the so-called *OnOff* object. Complex properties, including time properties and sequences were verified on the *OnOff* program. The proposed property preserving reduction techniques have been applied to be able to verify the object with successful results.

The second PLC program controls a subsystem of the LHC Cryogenics process at CERN. The significant size of this PLC program with 17,500 lines of code and a generated model with a PSS size of  $10^{31985}$ , makes the verification very challenging. In this case, safety properties were verified on this program and the “variable abstraction” technique was applied.

The experimental results show that the methodology is useful and PLC programs with a huge state space can be verified. Obviously, there are still some limitation and the combination of the property preserving reduction techniques and the variable abstraction may not be enough in some other cases. Providing new abstraction techniques is one of the future works for this project.

In addition, any complexity linked to formal verification or formal methods is hidden from the user of the methodology. The next chapter will give a deep analysis of these results, focusing in each steps of the methodology.



# Chapter 5

## Evaluation and analysis

This chapter presents an analysis about the different steps of the methodology and about the experimental results obtained when applying the designed methodology to CERN PLC programs.

### 5.1 Introduction

Different kind of requirements have been verified on UNICOS CPC baseline objects using the proposed methodology of this thesis, including complex properties (with several temporal logic operators), safety and liveness properties and finally “time-related” properties. The number of discrepancies that were found between the specifications and PLC programs was significant, even if these objects have been previously tested.

In addition, the methodology has been applied to complete UNICOS CPC PLC programs. The PLC program used for the experimental results in this thesis contains approximately 17,500 lines of code. In this case, a subset of safety properties (i.e.  $\text{AG}(\alpha \rightarrow \beta)$ ) was verified on this PLC program.

The usability of the methodology was demonstrated, even for large PLC programs. Any complexity related to formal methods was completely hidden from control engineers.

This chapter analyzes all the steps of the methodology regarding three different aspects: completeness, usability and limitations. The content of this chapter is divided in the following sections:

- Section 5.2 analyzes the solution for requirement specification provided by the methodology.
- Section 5.3 analyzes the modeling strategy for modeling the PLC platform and the controlled process.
- Section 5.4 analyzes the transformation rules from the PLC code to the IM.
- Section 5.5 analyzes the reduction techniques applied to the IM that make formal verification possible.
- Section 5.6 analyzes the transformation from the IM to the modeling language of the verification tools.
- Section 5.7 analyzes the verification results provided to the user of these methodology.
- Section 5.8 analyzes briefly the correctness of the different model transformations included in the methodology.

## 5.2 Evaluation of requirements formalization

This methodology gives a solution for the formalization of requirements for control and process engineers. This solution is based on patterns that rely on the experience of the PLC program developers of the EN-ICE group at CERN <sup>1</sup>.

These patterns, written in English, provide a natural language that can be easily used by control and process engineers to express requirements without any need of knowledge about formal methods. The list of patterns presented in Chapter 3 covers the current needs of PLC developers of the EN-ICE group at CERN. It covers a quite large range of properties, as it includes common safety and liveness properties but also properties involving several PLC cycles and also “time properties”. However this list can be extended if a new type of requirement is needed for verification purposes.

---

<sup>1</sup>EN-ICE web: <http://www.cern.ch/enice/>

These patterns have a well-defined semantics and they can be automatically translated to temporal logic (either CTL or LTL) for formal verification purposes.

In terms of limitations, the expressiveness of the requirements is limited by the expressiveness of the temporal logic formalisms implemented in the verification tools included in our methodology. Currently, nuXmv provides the biggest range of property specification expressiveness as it fully supports CTL and LTL. UPPAAL is restricted to a subset of CTL and BIP to deadlock and safety properties.

Another limitation on the specification phase is provoked by the “variable abstraction” reduction technique, which only allows to verify a subset of safety properties. This technique has a very aggressive abstraction strategy and is focused on one specific type of safety properties. This restriction is a consequence of the verification requirements for full UNICOS PLC programs that are currently safety properties. For that reason, this reduction is only applied when our property preserving reduction techniques are not enough for verification. The variable abstraction technique could be modified to accept ACTL properties instead of the current safety properties (i.e.  $\text{AG}(\alpha \rightarrow \beta)$ ) if the requirement specification expressiveness needs to be extended. However this would imply to have a less aggressive abstraction strategy and it could imply a worse performance of this technique.

This proposal for requirement specification is useful and it can help control and process engineers to find bugs in their programs. However, this is not the final solution for requirement specification in control software engineering. Using patterns does not guarantee that the PLC program is complete and fully verified. This approach only guaranties that PLC programs are compliant with the list of requirements, which are created using these patterns. A future work in this direction will have the goal of providing a complete, unambiguous and still useful (easy to understand by non-formal methods experts) formal specification for PLC control code.

In conclusion, we can summarize the specification approach of this methodology emphasizing these three aspects:

- Specification by using patterns is complete for the current needs of the UNICOS PLC developers at CERN, comprising safety, liveness and “time-related” properties. More patterns can be added to this list if required.

- The approach is useful as it provides a very simple natural language to specify.
- The expressiveness limitations are given by the verification tools included in the methodology and by the “variable abstraction” reduction technique.
- The current approach is appropriate to find bugs in PLC programs, but it is not the final solution for control software specification.

### 5.3 Evaluation of PLC hardware and process modeling

Formal verification of PLC programs implies building a global model of the whole system. The whole system is composed, not only by the control software but also the hardware of the control system and the controlled process (e.g. water treatment, petroleum, ventilation systems, etc.) In this case, this global model should include information about the PLC execution platform and about the controlled process.

Regarding the PLC execution platform, this methodology models the *scan cycle* of a PLC providing the skeleton of the generated models. It also includes a strategy to identify PLC inputs when parsing the PLC program, as it is a challenging task when these inputs are parameters or they come from the supervision or from a decentralized periphery. The methodology can generate models for both safety and standard PLCs and both models only differ when verifying time properties with explicit time in it. Finally, the methodology also models PLC interrupts and restarts but assuming that there are no concurrency problems on the PLC program.

This modeling strategy is a compromise between the two extreme approaches for modeling the PLC execution platform. The first one would be providing a very detailed model for the execution platform where even distributed systems are modeled and concurrency problems can be detected by the verification tools, but producing very heavy models with a huge state space. The second approach would be to provide a very simple model with a small state space, but it would provoke many false positives in the verification. If these models do

not represent properly the real process can cause false negatives as well.

In conclusion, we can summarize the PLC hardware modeling approach of this methodology emphasizing the following aspects:

- The methodology models PLC interrupts and restarts, however concurrency problems should be checked before using a different technique, for example the static analysis of the PLC code.
- It currently targets centralized PLC control systems, but not distributed control systems. Modeling distributed systems obviously implies a higher complexity in terms of concurrency problems and a bigger size of the generated models.
- This modeling approach is completely hidden from the control and process engineers and contains enough information for PLC program verification purposes.

Regarding the controlled process, this methodology allows the control and process engineers to add invariants to the formal models, including information about the process. However, it does not include a complete model of the process because it would break one of the main goals of this project: hiding the complexity related to formal methods. Including a complete model of the controlled process would imply to build it manually, as in industry there is normally no formal specification of the controlled process.

Not adding any invariant to the formal model may produce false positives (spurious counterexamples) in the verification phase. False positives do not break the safety of the system, as the user gets a report of an error in the model which does not exist in the real system. However, adding a wrong invariant can hide a bug that may occur in the real PLC program (false negative), which is a much bigger problem than a false positive. Therefore adding invariants to the models is a critical step of the methodology.

In conclusion, we can summarize the controlled process modeling approach of this methodology emphasizing the following aspects:

- Adding invariants is a very simple and useful mechanism. The user of the methodology can include invariants in the models in

order to verify a property. It reduces the number of false positives and it can improve the verification performance (it depends on the implementation of this mechanism in the model checker) as it reduces the RSS.

- Adding a wrong invariant implies false negatives, which is a much more dangerous situation than false positives for our goal of finding bugs in PLC programs.
- This approach does not include a complete model of the controlled process. This is because it would imply to create the model manually, as typically in industry there is no formal specification of the process. Modeling manually the controlled process is a challenging and time consuming task and it could hide potential bugs in the real code due to models that do not represent properly the real processes.

## 5.4 Evaluation of the PLC program – IM transformation

In this methodology, the transformation rules from the PLC code to the modeling languages of the verification tools are split in two groups: PLC - IM transformation rules, and IM- modeling languages of the verification tools transformation rules.

The first group includes the transformation rules that allow translating ST and SFC programs into the IM, already presented in Section 3.6. These transformations are rule-based and they include several assumptions that simplify the rules and improve the verification performance. Even with these assumptions, the generated models are reliable and they represent with high fidelity the real PLC program. For example, the assumption 3, which assumes the non concurrency of PLC blocks, includes some limitations from the verification point of view as concurrency problems cannot be detected. Currently, the assumptions included in the methodology are related to PLC interrupts, recursion, variable access, function calls, pointers and no parallel branches in the SFC code.

These transformations have been implemented in our CASE tool. This tool is based on EMF and Xtext and the transformations have

been implemented in Java and Xtend. However the complete ST and SFC grammars are not yet included. Currently, the implemented grammars and set of rules cover a large set of the full grammars and it is enough for parsing and translating the UNICOS CPC programs to the IM. One example of the features that are not included is the pointers (the use of pointers is not recommended, although it is not forbidden). These rules and tool can be extended to support the full grammar of ST and SFC languages.

In addition, other programming languages from the IEC 61131 (2013) standard (e.g. IL) can be added to the methodology. Adding new languages to the methodology may imply new specific assumptions related to the new language. For example we could assume that IL instructions are atomic and they cannot be interrupted, although in reality an IL instruction can be compiled as several *machine code* instructions, therefore an IL instruction can be interrupted.

In conclusion, we can summarize the PLC code - IM transformation rules of this methodology emphasizing the following aspects:

- The proposed transformation rules are implemented in our CASE tool and allow to translate automatically PLC code to the IM without any interaction of the user.
- Currently the PLC grammars and transformation rules implemented in our tool do not cover 100% of the ST and SFC grammars defined by the IEC 61131 (2013) standard. However, they support all the needed features of the UNICOS CPC PLC programs, which is a large subset of the grammars.
- A set of assumptions is included to simplify the transformation rules and improve the verification performance but obtaining reliable models. This assumptions imply some limitations from the verification point of view.
- Currently the methodology includes the transformation rules for ST and SFC languages. Other languages can be included, e.g. IL language.

Time-related transformation rules are analyzed in detail in the following paragraphs as they are a particular case of the transformation of PLC code to IM.

### 5.4.1 Evaluation of the time-related transformation rules

This methodology provides two different approaches to model PLC time and timers.

If the property contains explicit time in it, a realistic model of PLC time and timers is provided. An example of this kind properties is:

“if  $a$  is true, after 5 seconds  $b$  will be true, if  $a$  remained true.”

If the property does not contain explicit time in it, an abstract model of PLC time and timers is provided. An example of this kind properties is:

“if  $a$  is sometime true and remains true forever, eventually  $b$  will be true”.

The first approach represents with high fidelity the timer implementation on a real PLC (e.g. TON, TOFF and TP). This representation also implies to model time. The accuracy of the model is high enough, using as unit of logic time 1 ms for timers (as in a real PLC). In terms of specification, this approach allows to express properties with explicit time by using CTL and a monitor. However the resulting state space size is very large. Even if some abstraction techniques are applied, verification time can become very long and in some cases model checkers may not be able to provide a verification result.

The second approach solves the problem of state space explosion by proposing a simplified model of the first approach. The resulting model has a non-deterministic nature, so the accuracy is reduced and it can produce false positives. In terms of specification, it is not possible to verify properties with explicit time, however it can verify properties that guarantee that the timer gives a response (liveness property).

Table 5.1 summarizes the main differences of both approaches.

The first approach is thus needed when properties with explicit time have to be verified although there is a higher chance of having a state explosion problem. The second approach is suitable for timed properties without explicit time, like certain before/after properties (e.g. “the variable  $b$  has to be true at the end of the PLC cycle, after the variable  $a$  becomes false”), and for non-timed properties (for example safety or liveness properties), where the variables linked to

Table 5.1: Overview of the timer representation approaches from Fernández Adiego et al. (2014a)

	<b>Realistic</b>	<b>Abstract</b>
Size of PSS	huge (e.g. $5.91 \cdot 10^{15}$ for the TON model)	moderate (e.g. 6 for the TON model)
Requirement expressivity	rich requirements with explicit time, CTL/LTL + monitors	requirements without explicit time, CTL/LTL is enough
Constraints	time incremented by PLC cycle time	no explicit time, false positives

the property are affected by a timer in the real system and this timer cannot be eliminated by using reduction techniques, such as cone of influence. Although false positive results can occur using this approach, false negative can never occur.

## 5.5 Evaluation of the reduction techniques

In this methodology, the reduction techniques are applied to the IM and then all the verification tools can profit from these reductions.

Two groups of reduction techniques are currently included in the methodology: the so-called property preserving reduction techniques and the variable abstraction technique.

The first group includes the following reductions: COI, general rule-based reduction and mode selection. By using these reductions the resulting and original models are equivalent for the given property.

The variable abstraction technique produces abstract models that are an overapproximation of the original one. These models are produced based on the variable dependency graph of the variables included in the property to be verified.

The rest of this section presents some measurements to evaluate these reduction techniques.

### 5.5.1 Property preserving reduction techniques

As it was mentioned in Section 3.7, the theoretical concepts of some of the reduction techniques included in this group are already implemented in many verification tools. A clear example is *COI*, which is implemented for example in the nuXmv model checker.

However, these algorithms were not really effective on our models, so this was the motivation for implementing them at the IM level, which contains a higher level of information making our algorithms more effective.

The rest of the reduction techniques included in this group are domain specific and they benefit from the PLC domain knowledge.

The experimental results performed in PLC programs corresponding to the UNICOS CPC baseline objects were very satisfactory, but we experienced some limitations when verifying complete UNICOS CPC programs. Sometimes the number of variables kept on the model by COI was very large and the state space of the reduced model was still too large to perform formal verification.

An analysis of effectiveness of these techniques regarding the state space and the run-time reduction is presented in this section. To do so, some measurements are presented, showing how the different reduction techniques can complement each other. The measurements also compare our results with the nuXmv's reduction techniques.

The measurements correspond with the experimental results applied to the UNICOS CPC object *OnOff*, presented in Section 4.3.

### 5.5.2 State space

In the case study presented in Section 4.3, the generated model of the *OnOff* object has a PSS of  $1.61 \cdot 10^{218}$  states without any reductions. Fig. 5.1 shows measurements about the size of the *OnOff* model PSS without any reduction, using nuXmv's COI and our COI. The measurements are split in two groups: in the first group, on the left part of the figure, the rule-based reductions are not applied. In the second group, on the right part of the figure, the rule-based reductions are applied. As it can be seen, nuXmv's COI can reduce the state space significantly, however our implementation provides much better results. In addition, applying the rule-based reductions improves significantly the performance of the reductions. For example, if the

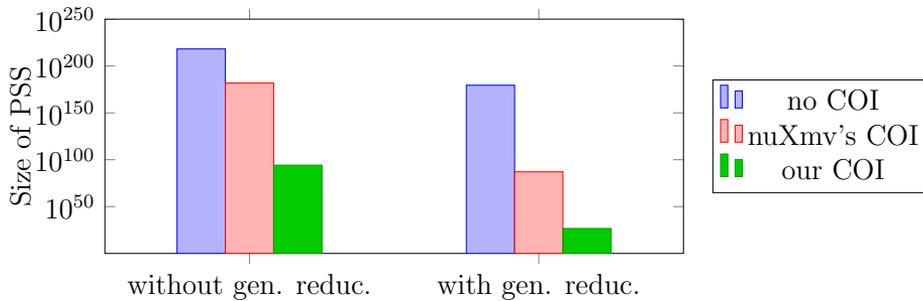


Figure 5.1: Measurements comparing the different COI solutions from Darvas et al. (2014)

rule-based reductions are enabled, the size of the PSS is  $1.2 \cdot 10^{87}$  with nuXmv's COI, and  $4.3 \cdot 10^{26}$  with our own COI implementation.

We can conclude that applying the COI to higher level models, i.e. the IM, significantly improves the reduction performance. In addition, we can observe that by using the rule-based reductions the effectiveness of both COI algorithms (nuXmv and ours) is much higher. In this experiment, formal verification of the given requirement would not have been possible without the rule-based reductions.

### 5.5.3 Verification run-time

The difference between the COI implementations can also be observed on the run-time of the verification. Two tables are presented to compare the effectiveness of our COI implementation on the IM and nuXmv's COI. These tables show verification run-time measurements and measurements about the internal BDD data structures of nuXmv that represent the state space. In these cases, four different real-life requirements were evaluated:

- Req. 1 is an LTL expression with form of  $G(\alpha U \beta)$ .
- Req. 2 and 3 are CTL safety expressions ( $AG(\alpha)$ ).
- Req. 4 is a complex LTL expression ( $G((\alpha \wedge X(\beta U \gamma)) \rightarrow X(\beta U \delta))$ ) describing a real requirement coming from the UNICOS developers (it represents a sequence of variable values from different cycles).

In these requirements, the Greek letters represent Boolean logical expressions containing multiple (usually 1–5) variables.

Table 5.2 shows the measurements for the nuXmv’s COI algorithm and Table 5.3 shows the measurements for our COI algorithm.

Without our rule-based reductions (even if nuXmv’s COI is used), none of the requirements could have been verified in a day, thus these reductions are inevitable. These measurements show that by using our COI implementation, the verification run-time can be reduced by 1–3 orders of magnitude compared to the COI of nuXmv. The same reduction can be observed in the number of allocated BDD nodes ( $\#Node$ ). The reduction in the peak number of live BDD nodes ( $\#PNode$ ) is smaller, but significant. This comparison shows the efficiency of the proposed method.

Table 5.2: Requirement evaluation measurements with nuXmv COI from Darvas et al. (2014)

Req.	no red. +	rule-based red. + nuXmv COI		
	nuXmv COI	Runtime	$\#PNode$	$\#Node$
1	—	896 s	$8.8 \cdot 10^5$	$1.8 \cdot 10^8$
2	—	1,250 s	$9.9 \cdot 10^5$	$8.8 \cdot 10^8$
3	—	19,300 s	$3.4 \cdot 10^6$	$1.6 \cdot 10^{10}$
4	—	649 s	$9.0 \cdot 10^5$	$5.5 \cdot 10^8$

Table 5.3: Requirement evaluation measurements with our COI from Darvas et al. (2014)

Req.	no reduct.+	rule-based reduc. + our COI		
	nuXmv COI	Runtime	$\#PNode$	$\#Node$
1	—	2.5 s	$2.2 \cdot 10^5$	$1.1 \cdot 10^6$
2	—	19.0 s	$4.6 \cdot 10^5$	$1.4 \cdot 10^7$
3	—	1,440 s	$1.3 \cdot 10^6$	$1.6 \cdot 10^9$
4	—	2.3 s	$2.2 \cdot 10^5$	$9.2 \cdot 10^5$

### 5.5.4 Variable abstraction

The last reduction technique included in the methodology, the variable abstraction technique, was designed to give an answer to verification cases when the property preserving reduction techniques were not enough. In our experiments, this situation appeared when verifying full UNICOS PLC programs, such as the one presented in Section 4.4 with 17,500 lines of code and an IM of PSS size of  $10^{31985}$  (without reductions). The UNICOS developers provided a set of requirements for these programs. These requirements were safety properties of the following nature:  $\text{AG}(\alpha \rightarrow \beta)$ , where the Greek letters represent Boolean logical expressions containing multiple (usually up to 5) variables.

For that reason, this iterative abstraction technique was designed with a very aggressive abstraction strategy tailored just for  $\text{AG}(\alpha \rightarrow \beta)$  properties. In addition, the algorithm heuristic is easy to automatize and include in our CASE tool.

This technique produces abstract models based on the variable dependency graph. In the PLC program presented in Section 4.4, after applying the properties preserving reduction techniques for the three different requirements, the resulting models have a PSS size around  $10^{5048}$ .

For the first experiment, the algorithm demonstrated in 45.309 s (including the model generation, the verification time and a timeout of 30 s) that the property holds in the model after one iteration on the algorithm, where one invariant was extracted.

For the second experiment, the algorithm demonstrated in 7.544 s (including the model generation and the verification time) that the requirement holds in the model in the first abstraction.

For the third experiment, the algorithm demonstrated in 126.573 s (including the model generation, the verification time, the counterexample generation and two timeouts of 30 s) that the requirement does not hold in the model using two iterations.

These experimental results show that this algorithm is useful to verify properties in models with a huge PSS. However, this technique has also some limitations:

- The algorithm is restricted to safety properties like  $\text{AG}(\alpha \rightarrow \beta)$ . This helps to provide an aggressive abstraction strategy.

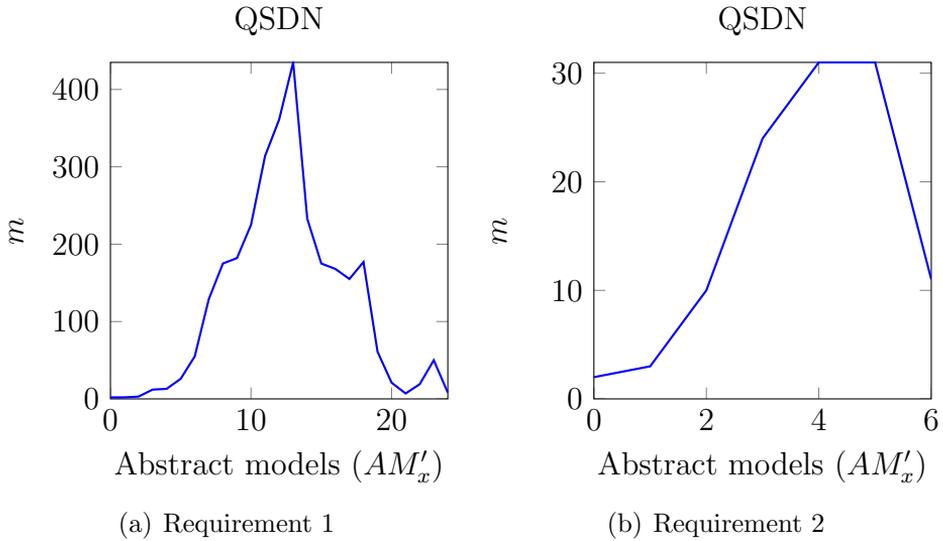


Figure 5.2: Relationship between the abstract models and the maximum number of reachability properties for the variable abstraction technique

- Extracting invariants is sometimes hopeless when the number of potential invariants ( $m$ ) is very high. For that reason, we included the *step 5* in the algorithm, which only extracts invariants if for an abstract model  $AM'_n$  corresponding with a  $\delta' = n$ , the value of  $m \leq 10$  (usually this corresponds with  $\delta' \leq 2$ , see an example of the QSDN PLC program on Fig.5.2).
- One of the weakness of this method is the *step 3*, when verifying the reachability property  $r$  on the original model  $OM''$ . If verifying  $p$  on  $OM'$  ended in a timeout, there are a lot of chances of getting also a timeout in this step because the size of the models  $OM'$  and  $OM''$  may be similar. *Bounded model checking* seems to be a good alternative to check  $r$  on  $OM''$ , as bounded model checking allows to check properties on a part of the state space. nuXmv includes bounded model checking algorithms, however, the experiments with our models were not successful and the tool crashes with an error after small bounds and no answer were obtained.

To conclude, we have demonstrated that this technique is useful

for large PLC programs where our property preserving reduction techniques are not enough. But obviously, we can have models where a timeout is obtained when verifying the safety properties  $p$  on an abstract model  $AM'_n$  in any of the iterations  $\delta' = n$  of the algorithm. In this situation, no answer is provided and we cannot prove that the counterexamples obtained in previous iterations ( $AM'_{n-1}$ ,  $AM'_{n-2}$ , ...,  $AM'_1$ ) are real or spurious.

## 5.6 Evaluation of the IM – verification tools transformation

The intermediate step (IM) on the transformation process from the PLC programs to the modeling languages of the verification tools gives a lot of flexibility to the methodology and new verification tools can easily be added and compared. Currently nuXmv, UPPAAL and BIP are included, and new tools are being investigated, e.g. mCRL2<sup>2</sup>. This tool has been developed at the Department of Mathematics and Computer Science of the Technical University of Eindhoven, in collaboration with other institutes.

The selected automata-based formalism for the IM simplifies the transformations from IM to the modeling languages of the verification tools. The current verification tools that are integrated in the methodology have state-based modeling languages. These formalism are very close to the IM formalism, although special strategies have to be applied for each verification, like for example, the synchronization between two automata or modules in nuXmv or how to model global variables in BIP.

Obviously, it is easier to integrate in the methodology state-based modeling languages but different formalisms can be included, e.g. mCRL2 has an action-based modeling language. In this case, transformation rules are more complicated but once the strategy is defined it can be equally automatized.

---

<sup>2</sup><http://www.mcrl2.org/>

## 5.7 Evaluation of the verification results

The experimental results confirmed that this methodology can be applied to real PLC programs without any knowledge about formal methods. UNICOS CPC baseline objects were verified and even experiments with complete UNICOS PLC programs were performed. These experiments reported many discrepancies between these PLC programs and the specifications. But some other relevant conclusions can be extracted in terms of verification performance. The methodology is flexible and different verification tools can be added and compared. Regarding the verification performance of nuXmv, UPPAAL and BIP, we can extract the following conclusions:

- nuXmv provides the best performance in terms of verification for our models.
- BIP provides an infrastructure for code generation that can be applied for model-based testing and we did some experiments in this direction (see Fernández Adiego et al. (2013b)). The verification solution is very promising for our models as compositional verification algorithms are included, but the verification tool (DFinder) does not support currently data transfer between atoms, therefore it cannot be applied to our models. Future work on this direction is planned.
- UPPAAL provides the best environment for model simulation. However, when big models are automatically generated by our tool, UPPAAL cannot even load and visualize the model. In terms of verification performance, UPPAAL provides worse results than nuXmv for our models.

For that reason, for verification purposes most of the experiments were performed using nuXmv.

Due to multiple model to model transformations, the question whether the methodology itself contains errors arises. Section 5.8 presents a brief discussion on this topic.

However in this last step of the methodology, it can be demonstrated that a bug found in the model also exists in the real PLC code. When a counterexample is produced it can be used to produce a PLC demonstrator and check if this counterexample produces the

expected error in the real code. By doing this, errors in the model to model transformation can be detected without proving mathematically these transformations, always assuming that the PLC compiler and the verification tools do not contain bugs.

## 5.8 Correctness of our approach

Proving the correctness of all the model transformations is a challenging itself. This would require to prove all the model to model transformations, i.e. the PLC code - IM transformations, the IM reductions and the IM- input languages of every verification tools. But also the correctness of the verification tool itself and the correctness of the PLC compiler as they are all part of the methodology.

Proving the model transformations is possible and ideally this methodology could be certified, however, the huge effort that this implies was not a real option (not a priority at least) at the current stage of this project. For that reason, we adopted a more practical approach where we can prove that a bug, found by the verification tool on the model, really exists in the real PLC program. The counterexample provided by the verification tool can be used to prove its existence on the real PLC code as described in Section 3.11. However, in the following paragraphs the steps to guarantee mathematically the correctness of this approach are sketched.

Possible sources of errors in our verification methodology may detect false positives, i.e. may report errors that do not exist in the real system. On the other hand, it may not detect all errors (false negatives), i.e. state that a property that is intend to hold for a system is verified although it does not hold in the real system. The latter one is more dangerous since it potentially states the correctness of an erroneous system.

To ensure correctness in our methodology, we have to investigate the following potential sources of errors:

1. *Formalization of properties*: Errors can be made when translating an informal requirement into a logical formula suitable for processing in verification tools.
2. *Implementation of verification tools*: The implementation of the model checker may contain errors.

3. *Transformation of models*: The translation from SFC and ST code into the IM, the IM reductions as well as the translation from the IM into nuXmv, BIP and UPPAAL models may contain errors.
4. *Interpretation of verification tool results*: A counterexample may be interpreted in a wrong way.

The error source (1) is addressed by thoroughly analyzing our stated properties. We cannot guarantee the absence of errors in the used verification tools in error source (2), however, the used model checkers are well established and employed in a variety of projects. Error source (4) can be avoided by formulating the properties in a way that they yield simple yes/no answers as verification results.

Regarding the error source (3), it comprises the model transformations that we developed for this work. We sketch the steps to guarantee correctness using mathematical sound proofs. An example of proof can be found in Fernández Adiego et al. (2014a), which it was conducted for proving the correctness of abstraction rules for timers. The correctness proof requires the establishment of formal semantics for the involved languages, i.e. operational semantics (cf. Plotkin (2004) for the classical structured way) for all languages involved: the model checker languages, the IM and the programming languages defined by the IEC 61131 (2013) standard (cf. Blech and Biha (2011) for IL and FBD). A simulation relation between the original and transformed model can be established. States in one model are related with semantical equivalent states in the transformed model. The simulation relation must preserve the properties that we ultimately want to verify (abstractions can alter the classes of properties that are preserved, cf. Bozga et al. (2008)). The correctness proof comprises the verification of a base case stating that the simulation holds for the initial states of the original and transformed model, and a simulation step, stating that if it holds for two states it will also hold for succeeding states.

## 5.9 Summary of the chapter

This chapter presented an analysis of the different steps of the proposed methodology from the completeness, usability, and limitations

point of view. The experimental results confirm the usability of the proposed methodology on real life PLC programs. In addition, the complexity of applying formal methods is completely hidden from the engineers.

This analysis gives also an idea of the future work to be done, which is presented in detail in Chapter 6.



# Chapter 6

## Conclusions and future work

This final chapter includes the conclusions of this thesis, presenting the list of contributions, limitations and benefits of this research. In addition, a brief discussion about the future work of this project is presented. This chapter is divided in three sections:

- Section 6.1 presents a discussion about the evolution of the thesis, from the initial requirements until the adopted solution to satisfy them.
- Section 6.2 presents the contributions, the limitations and the benefits of this research.
- Finally, Section 6.3 suggests the future research lines of this project.

### 6.1 Discussion

This research project was set out to provide a solution to one of the main challenges of PLC program developers and designers in the automation industry, a mechanism to guarantee that the PLC code is compliant with the project specification.

This challenge is particularly important for Safety Instrumented Systems, which are in charge of protecting people, environment and installations, as stated in the IEC 61508 (2010) standard. However this is obviously a desirable feature for any control system.

Nowadays in industry, automated and manual testing of PLC programs are the most popular solutions adopted, but some well-known drawbacks are inherent to these approaches.

This thesis provides a solution for applying automated formal verification to PLC programs. At the beginning of the project, two main goals were set to ensure the applicability of this proposal in real-life PLC programs:

1. Hiding any complexity related to formal verification and formal methods from the automation engineers.
2. Being able to verify new and existing PLC programs. This requires that the proposed solution should not modify the development process of any PLC program.

Satisfying these requirements implied to adopt the solution presented in this PhD thesis: a general automated methodology for formal verification of PLC programs. The term “general” stands for a flexible methodology, in which different verification tools can be added to the methodology. The term “automated” indicates that all the internal steps do not require any human intervention.

Formal models for different verification tools are created automatically out of the PLC code. The methodology is based on an intermediate step, the so-called intermediate model (IM), used as a pivot between all the PLC and formal modeling languages included in the methodology. This approach potentially covers all PLC languages. The current implementation supports ST and SFC and the support for IL is under development.

All the steps of the methodology were presented in Chapter 3: requirements formalization, automatic translation of PLC programs to the IM, reduction of the IM, translation of the IM to the modeling languages of the verification tools and finally the counterexample analysis.

The experimental results presented in Chapter 4 show the applicability of this approach in real-life PLC programs. Many discrepancies between the specification and the PLC programs were found during these experiments, even if these programs were previously tested with other techniques.

The practical and theoretical contributions of this methodology are discussed in the following section.

## 6.2 Contributions

This PhD thesis is not certainly the first attempt to apply formal verification to PLC programs. Section 2.4 showed the related work to this field. However, none of the approaches satisfied completely our initial requirements, or strong limitations were necessary to apply their solutions. The theoretical and practical contributions of this thesis were enumerated in Section 1.3. Here, more details about these contributions are presented according to the experimental results.

### 6.2.1 Methodology

The first contribution is the adopted methodology itself. The transformation from different PLC languages to different modeling languages of the verification tools, passing through this intermediate step, already exists in other fields of application but is an innovative approach on the industrial automation community, in particular for the PLC environment. Regarding the requirement formalizations, an approach based on patterns can be found before this document as presented in Section 2.4, but in this case, new patterns are provided in this research to cope with the real-life needs from the PLC program developers at CERN. The counterexample analysis and the automatically generated report, which is provided to the engineer when a bug is found, is a practical contribution which is very helpful for control engineers. The model checker usually provides a huge amount of information when a bug is found in a real-life PLC program, but only the relevant information is extracted and provided to the engineer in order to find the source of the problem.

### 6.2.2 IM syntax and semantics

The definition of the syntax and semantics of the IM is the second contribution. The goal was to provide a simple formalism where all the features of PLC programs can be modeled with this formalism. After the experimental results, no limitations due to the IM expressiveness were found. The adopted assumptions to translate the PLC code into the IM were taken to simplify the transformation rules and to reduce the state space but obtaining meaningful models. In addition, the IM

formalism is close to the formalisms adopted by several verification tools, simplifying the transformation and therefore facilitating to add new verification tools in order to check their verification performance and simulation facilities for our models.

### **6.2.3 *PLC code - IM transformation rules***

The third contribution is the set of transformation rules from ST and SFC languages to the IM. Although the full grammars are not supported yet, the most relevant transformations are presented which fully cover the needs for the transformation of UNICOS CPC PLC programs.

### **6.2.4 *IM- input verification tools transformation rules***

Currently three verification tools are supported and compared in the methodology. The transformation from IM to the nuXmv, UPPAAL and BIP are the fourth contribution. The analysis of these three tools, done in Chapter 5, shows that in terms of verification performance, nuXmv provides the best results of the three tools for our models.

### **6.2.5 *Reduction techniques***

The reduction and abstraction techniques proposed in this thesis are a fundamental part of the methodology as they allow to apply formal verification to real-life PLC programs. They are the fifth contribution of this thesis. Two groups of reduction techniques are proposed, the property preserving reduction techniques and the variable abstraction technique:

1. For the first group, some of the applied theoretical concepts were previously defined, like Cone of Influence (COI). However, by using the PLC knowledge and applying the reductions to the IM level, a better verification performance than other implementations was achieved. This thesis compares our property preserving reduction techniques with the COI algorithm implemented in nuXmv.

2. The variable abstraction technique is a novel technique which allowed us to verify safety properties, like  $\text{AG}(\alpha \rightarrow \beta)$ , on the PLC programs where the first group of reductions was not effective by its own. It has some similarities with the CEGAR technique but it differs in the following aspects: the heuristics of this technique are easier to automatize in our methodology, the abstract models are automatically created using the variable dependency graph of the variables included in the requirement and as we target  $\text{AG}(\alpha \rightarrow \beta)$  properties, a more aggressive abstraction strategy can be applied.

### 6.2.6 Modeling the timing aspects of PLCs

The transformation of the timing aspects of PLC programs are part of the set of rules to translate the PLC programs to the IM. This is a very special and challenging aspect for which this thesis gives a solution. This is the sixth contribution of this thesis. Actually two different solutions are proposed depending on the nature of the requirement to be verified. In the first option, time is modeled as a 16-bits variable. Besides, PLC timers (i.e. TON, TOFF and TP) are modeled with a realistic approach, very close to the reality which allows to verify requirements with explicit time in it. The second option does not model time and an abstract model of the timers is provided instead. This approach allows only to check “before/after” timed properties, but the verification performance is significantly better than the previous approach as the state space is much smaller.

### 6.2.7 PLC behavior analysis

The last contribution of this thesis is the analysis of the internal behavior of PLCs, focusing in Siemens PLCs. This is not a theoretical contribution of this thesis, however this analysis was a fundamental step in the design of the modeling strategy and also in the definition of the semantics of the IM. One example of a large impact on the modeling strategy was the analysis of the timing aspects of PLCs programs, i.e. the *TIME* data type and PLC timers. When defining the syntax and semantics of the IM, one option was to provide a timed automata-based formalism, similar to the UPPAAL modeling

language (described in Amnell et al. (2001)), in order to be able to model these timing aspects as some other authors did before. However, after analyzing the internal PLC behavior, the adopted strategy was to use an automata-based formalism. This is a simpler formalism than timed automata and the timing aspects of PLCs can be properly modeled using this modeling strategy (as described in Section 3.9).

## 6.2.8 Applicability on CERN control systems

In addition to the contributions, this thesis presents the experiments performed on real-life PLC programs. These systems, which were developed at CERN, have significantly large PLC programs and bugs were found on these systems. To the best of the author's knowledge, none of the previous works presented experimental results with PLC programs of similar size using a completely automated procedure.

The current limitations of this methodology were presented in Chapter 5. Some features, like the support of the full grammar of ST and SFC, impose currently some small restrictions in the methodology. However, this kind of issues are not theoretical limitations and can be solved with a relatively small effort, comparing with the following limitations.

The three main limitations of this methodology are: the state space explosion of the models (common problem in the formal verification domain), the verification of concurrency problems in PLCs and the requirements specification.

The state space explosion problem is still the main limitation in this methodology, as it is for any model checking application. In this approach, when property preserving reductions and the variable abstraction technique do not provide an answer, the limits of this approach are met. Although the experiments show good results in real-life PLC programs, there is obviously room for improvement and extensions in the reduction techniques.

This approach assumes that no concurrency problems can occur in PLC programs. However, concurrency problems may happen in reality, so this approach suggests to check this kind of problems before applying formal verification of the programs. The reason of this assumption is to simplify the transformation rules and improve the

performance of the verification process. For instance, if a very realistic model of the PLC interrupts were created (where an interrupt can stop the main program at any time), the state space of the models would be huge, thus preventing the use of formal verification. The solution to this problem is very challenging. Instead of formal verification, this approach currently suggests to apply different techniques to check concurrency problem, for example, static analysis techniques. To check this kind of properties using formal verification would probably require using different verification tools than the ones currently included in the methodology. Another solution could be producing more abstract models out of the PLC programs to deal with this kind of analysis.

The last but not least big limitation of this approach is the requirements specification. The approach currently provides patterns to formalize the requirements. This is useful for finding bugs in the PLC programs but it is not the final solution for requirements specification in engineering. For example, with this approach the completeness of the requirements is not guaranteed. The challenge of providing a mechanism to produce complete, unambiguous and coherent control systems requirements is still unsolved. In addition, the fact that control and process engineers are usually not familiar with formal methods makes this task still more challenging.

## 6.3 Future work

The future research work is related to overcome the main drawbacks described before. There are three main research topics among others:

1. Abstraction techniques.
2. Requirement specification.
3. Applying different formal verification algorithms to our models.

The first field is the reduction and abstraction techniques applied to the methodology. New and improved reduction and abstraction techniques (e.g. predicate abstraction) would be necessary to verify any PLC program as an ultimate goal.

The second field of research is the design of a specification mechanism for control systems. This mechanism has to provide a formal, unambiguous and easy-to-understand language. Nowadays, it does not exist a unified mechanism or language for requirement specification in control systems as there are unified programming languages. In a near future, unified “formal or pseudo-formal” specification mechanisms should be highly recommended by any of the automation standards in order to develop more robust and safe control systems. Having a complete and unambiguous specification will open the door to the “correctness by construction” approach in industrial automation and it could complement the current approach proposed in this thesis, where formal verification can be applied at different levels of the control system development process: at the specification and control code levels, which would reduce the potential bugs of the systems to the minimum.

Finally, a nice feature of this methodology is that different formal verification techniques can be included and their results can be compared. From this point of view, it would be very interesting to check the benefits of the SAT-based techniques on our models. We started to investigate these techniques and the development and results are an ongoing work. Bounded model checking could also improve the verification performance in some cases for our models. In particular, in the variable abstraction technique, one of the current “bottleneck” of this technique is the verification of reachability properties on very large models, bounded model checking is suitable for the reachability analysis. In addition, compositional formal verification seems to be a very promising verification strategy for our models, specially when verifying full UNICOS PLC programs. This research could open the door to the verification of distributed control systems. The BIP framework provides a solution for compositional verification, although the verification tool does not support yet data transfer between components.

As a final conclusion, I consider that the original requirements defined at the beginning of this project were met during this research. The relevance of the practical and theoretical contributions is demonstrated in the experimental results of real-life PLC programs at CERN. The automation and formal methods communities are getting closer

thanks to the contributions from this thesis and from other researchers.

However, there is still a long way to have formal verification applied extensively in the automation industry. This thesis has triggered new research topics that hopefully will remove the gap once and for all between these two scientific worlds.



# Capítulo 7

## Conclusiones y trabajo futuro

Este capítulo final incluye las conclusiones de esta tesis, presentando la lista de contribuciones, limitaciones y beneficios de esta investigación. Además, una breve discusión sobre el futuro de este proyecto es presentado. Este capítulo está dividido en tres secciones:

- La Sección 7.1 presenta una discusión sobre la evolución de la tesis, desde los objetivos iniciales hasta la solución adoptada para lograr dichos objetivos.
- La Sección 7.2 presenta las contribuciones, las limitaciones y los beneficios de esta investigación.
- Finalmente, la Sección 7.3 sugiere las futuras líneas de investigación para este proyecto.

### 7.1 Discusión

Este proyecto de investigación fue diseñado para proporcionar una solución a uno de los principales retos de los diseñadores y desarrolladores de programas PLC en la industria de automatización, un mecanismo para garantizar que el código PLC respeta las especificaciones de diseño.

Este reto es particularmente importante para los sistemas de seguridad, encargados de la protección de personas, medioambiente e

instalaciones, tal y como es indicado en el estándar IEC 61508 (2010). Sin embargo, esto es un elemento deseado para cualquier sistema de control.

Hoy en día en industria, testeo automático y manual de los programas PLC son las soluciones adoptadas más populares, pero tienen limitaciones bien conocidas.

Esta tesis propone una solución para el empleo de verificación formal de forma automática en programas PLC. Al principio de este proyecto, dos objetivos principales fueron establecidos para garantizar que esta estrategia sea útil para la verificación de programas PLC reales:

1. Ocultar cualquier dificultad relacionada con tareas de verificación formal a los ingenieros de automatización.
2. Ser capaces de verificar nuevos programas PLC o programas ya existentes. Esto requiere que la metodología propuesta no debería modificar ningún proceso de desarrollo de programas PLC.

Lograr esos objetivos supuso adoptar la solución presentada en esta tesis: una metodología genérica y automatizada para aplicar verificación formal de programas PLC. El término “general” hace referencia a una metodología flexible, en la cual diferentes herramientas de verificación pueden ser incluidas en la metodología. El término “automatizada” indica que las etapas internas no necesitan de intervención humana.

Los modelos formales para diferentes herramientas de verificación son creados de manera automática a partir del código PLC. La metodología está basada en una etapa intermedia, el llamado modelo intermedio (IM por sus siglas en inglés: *Intermediate Model*) usado como eje central entre todos los lenguajes PLC y de las herramientas de verificación incluidas en la metodología. Esta estrategia puede ser aplicada potencialmente a todos los programas PLC. El estado actual de la metodología incluye los lenguajes ST y SFC, el lenguaje IL está bajo desarrollo.

Todas las etapas de la metodología son presentadas en el Capítulo 3: formalización de los requerimientos, traducción automática de los programas PLC al IM, reducción del IM, traducción del IM a los lenguajes de las herramientas de verificación y finalmente el análisis del contraejemplo.

Los resultados experimentales presentados en el capítulo 4 muestran que esta solución es válida para programas PLC reales. Muchas discrepancias entre las especificaciones y los programas PLC fueron encontradas durante estos experimentos, incluso si estos programas fueron testeados con otras técnicas.

Las contribuciones prácticas y teóricas de esta metodología son analizadas en la próxima sección.

## 7.2 Contribuciones

Esta tesis doctoral no es el primer intento de aplicar verificación formal con programas PLC. La Sección 2.4 analiza el estado del arte en este campo de investigación. Sin embargo, ninguna de esas propuestas cumplía completamente con los requisitos iniciales de este proyecto, o imponían limitaciones muy grandes en su aplicación. Las contribuciones teóricas y prácticas de esta tesis fueron enumeradas en la Sección 1.3. En esta sección, más detalles sobre dichas contribuciones son presentados en base a los resultados experimentales obtenidos.

### 7.2.1 Metodología

La primera contribución es la propia metodología. La transformación de diferentes lenguajes PLC a diferentes lenguajes de modelado de las herramientas de verificación, pasando por este paso intermedio, ya existe en otros campos de aplicación pero es un enfoque innovador en la comunidad de automatización industrial, particularmente en el campo de PLCs. En cuanto a la formalización de los requerimientos, un enfoque basado en patrones puede ser encontrado antes de este documento tal y como es presentado en la Sección 2.4, pero en este caso nuevos patrones son incluidos en esta investigación para enfrentarse a las necesidades de los sistemas reales desarrollados por los ingenieros de control del CERN. El análisis de los contraejemplos y los informes generados, los cuales se proporcionan al ingeniero cuando un error es encontrado, es una contribución práctica la cual es muy útil para los ingenieros de control. Las herramientas de verificación (model checkers) normalmente proporcionan una gran cantidad de información cuando un error es encontrado en programas de PLC reales, pero solamente

la información relevante es extraída y proporcionada al ingeniero para que pueda encontrar el origen del problema.

### **7.2.2 Sintaxis y semántica del IM**

La definición de la sintaxis y de la semántica del IM es la segunda contribución. El objetivo fue proporcionar un formalismo simple donde todos los aspectos de los programas PLC puedan ser modelados. Después de los resultados experimentales, no se encontraron limitaciones debidas a la expresividad del IM. Las suposiciones adoptadas para traducir el código PLC al IM ayudan a simplificar las reglas de transformación y a reducir el espacio de estados pero consiguiendo modelos adecuados al sistema real. Además, el formalismo del IM es cercano a los formalismos adoptados por varias herramientas de verificación, lo cual simplifica la transformación y por lo tanto facilita la posibilidad de añadir nuevas herramientas de verificación para comprobar cuáles son sus rendimientos y posibilidades de simulación para nuestros modelos.

### **7.2.3 Reglas de transformación *código PLC - IM***

La tercera contribución es el conjunto de reglas de transformación de los lenguajes ST y SFC al IM. Aunque las gramáticas completas no han sido incluidas aún, las transformaciones más relevantes son presentadas. Estas reglas de transformación cubren todas las necesidades para transformar programas UNICOS CPC desarrollados en el CERN.

### **7.2.4 Reglas de transformación *IM- modelo para las herramientas de verificación***

Actualmente tres herramientas de verificación son incluidas y comparadas en la metodología. La transformación del IM a los lenguajes de modelado de nuXmv, UPPAAL y BIP es la cuarta contribución. El análisis de estas tres herramientas, presentado en el Capítulo 5, muestra que en términos de rendimiento de los algoritmos de verificación, nuXmv proporciona los mejores resultados de las tres herramientas para nuestros modelos.

### 7.2.5 Técnicas de reducción

Las técnicas de reducción y abstracción propuestas en esta tesis son una parte fundamental de la metodología ya que permiten aplicar verificación formal a programas PLC reales. Estas técnicas son la quinta contribución de esta tesis. Dos grupos de técnicas de reducción son propuestos, las técnicas de reducción que preservan las propiedades y la técnica de abstracción de variables:

1. Para el primer grupo, algunos de los conceptos teóricos de estas técnicas fueron previamente definidos por otros autores, como por ejemplo “Cone of Influence” (COI). Sin embargo, al utilizar el conocimiento del funcionamiento de los PLCs y aplicando estas técnicas en el IM, un mejor rendimiento en términos de verificación fue logrado para nuestros modelos. Esta tesis compara nuestras técnicas de reducción que preservan las propiedades con el algoritmo implementado en nuXmv.
2. La técnica de abstracción de variables es una técnica innovadora la cual permite verificar propiedades de seguridad, como  $\text{AG}(\alpha \rightarrow \beta)$ , en aquellos programas PLC donde el primer grupo de reducciones no es lo suficientemente efectivo. Tiene ciertas similitudes con la técnica CEGAR pero difiere en los siguientes aspectos: la heurística de esta técnica es más sencilla para ser automatizada en nuestra metodología, los modelos abstractos son creados automáticamente usando el árbol de variables dependientes de las variables incluidas en la propiedad a verificar y como el objetivo es verificar propiedades como  $\text{AG}(\alpha \rightarrow \beta)$ , una estrategia de abstracción más agresiva puede ser aplicada.

### 7.2.6 Modelado de los aspectos temporales de los PLCs

La transformación de los aspectos temporales de los programas PLC forman parte del conjunto de reglas para traducir los programas PLC al IM. Esta tesis proporciona una solución para este caso muy particular y complicado de la transformación de programas PLC al IM. Esta es la sexta contribución de esta tesis. Actualmente dos soluciones diferentes son propuestas dependiendo de la naturaleza de la propiedad

que se va a verificar. En la primera opción, el tiempo es modelado como una variable de 16 bits. Además, los temporizadores TON, TOFF y TP son modelados con un enfoque realista muy cercano a la realidad, que permite verificar propiedades que incluyen tiempo explícitamente. La segunda opción por el contrario no modela el tiempo y un modelo abstracto es creado para representar los temporizadores. Este enfoque permite verificar solamente propiedades temporales del tipo “antes/después”, pero el rendimiento es significativamente mejor que en el enfoque anterior ya que el espacio de estados es mucho menor.

### 7.2.7 Análisis del funcionamiento de un PLC

La última contribución de la tesis es el análisis del funcionamiento interno de un PLC, en particular de Siemens PLCs. No se trata en este caso de una contribución teórica de esta tesis, sin embargo este análisis fue una etapa fundamental en el diseño de la estrategia de modelado y también en la definición de la semántica del IM.

Un ejemplo del gran impacto en dicha estrategia fue el análisis de los aspectos temporales en los programas PLC, es decir, del tipo de dato *TIME* y de los temporizadores. Cuando se definió la sintaxis y la semántica del IM, una opción era proporcionar un formalismo basado en automata temporizado, similar al lenguaje de modelado de UPPAAL (descrito en Amnell et al. (2001)), para poder modelar estos aspectos temporales tal y como otros autores lo propusieron previamente. Sin embargo, después de analizar el funcionamiento interno del PLC, la estrategia adoptada fue usar un formalismo simplemente basado en automata. Este formalismo es más sencillo que el formalismo basado en automata temporizado y los aspectos temporales de los PLCs pueden ser modelados adecuadamente usando esta estrategia (tal y como fue descrita en la Sección 3.9).

### 7.2.8 Aplicación en sistemas de control desarrollados en el CERN

Además de las contribuciones, esta tesis presenta los experimentos realizados en programas PLC reales. Estos sistemas, los cuales han sido desarrollados en el CERN, tienen programas PLC de un tamaño muy significativo y errores fueron encontrados en dichos sistemas. En

ninguno de los trabajos encontrados que son previos a esta tesis, los experimentos fueron realizados en programas de la entidad de los experimentos presentados en esta tesis de una manera completamente automatizada.

Las actuales limitaciones de esta metodología fueron presentadas en el Capítulo 5. Algunos aspectos, como incluir la gramática completa de los lenguajes ST y SFC, implica actualmente pequeñas restricciones en la metodología. Sin embargo, este tipo de problemas no son limitaciones teóricas y pueden ser resueltas con un esfuerzo relativamente pequeño, comparado con las siguientes limitaciones.

Las tres principales limitaciones de esta metodología son: la explosión de estados de los modelos (problema común en la comunidad de verificación formal), la verificación de problemas de concurrencia en los PLCs y la especificación de los requisitos.

El problema de explosión de estados es aún la principal limitación en esta metodología, tal y como ocurre en cualquier aplicación de model checking. En esta metodología, cuando las técnicas de reducción que preservan las propiedades y la técnica de abstracción de variables no proporcionan ninguna respuesta, el usuario de la misma no obtiene ningún resultado. Aunque los experimentos presentan buenos resultados en programas PLC reales, existe obviamente margen para mejoras y extensiones en las técnicas de reducción.

Esta metodología asume que no pueden ocurrir problemas de concurrencia en los programas PLC. Sin embargo, esos problemas pueden ocurrir en la realidad, por lo que la estrategia recomendada sugiere analizar dichos problemas antes de aplicar verificación formal a los programas. La razón de esta suposición es simplificar las reglas de transformación y mejorar el rendimiento del proceso de verificación. Por ejemplo, si un modelo muy fiel a la implementación real del PLC fuese creado (donde una interrupción puede parar el programa principal en cualquier momento para ser ejecutado), el espacio de estados de los modelos sería enorme y por lo tanto no sería posible aplicar verificación formal a dichos modelos.

La solución a este problema no es trivial. En lugar de verificación formal, esta metodología propone actualmente aplicar diferentes técnicas para analizar los problemas de concurrencia, por ejemplo, técnicas de análisis estático. Para chequear este tipo de propiedades usando

verificación formal, probablemente sería necesario usar diferentes herramientas de verificación que las que son incluidas en la metodología hoy en día. Otra posible solución podría ser la de producir modelos mucho más abstractos de los programas PLC para poder analizar este tipo de propiedades.

La última pero no menos importante gran limitación de esta metodología es la especificación de los requisitos funcionales que el programa debe cumplir. El enfoque actual proporciona patrones para formalizar los requisitos. Esta es una solución útil para encontrar errores en los programas PLC pero no es la solución final para la especificación de requisitos en ingeniería. Por ejemplo, con esta solución no se puede garantizar que la especificación de los requisitos sea completa. El objetivo de proporcionar un mecanismo para producir una especificación de requisitos completa, sin ambigüedades y coherente para sistemas de control no se ha conseguido aún. Además, el hecho que los ingenieros de control y de proceso no están en la mayoría de los casos familiarizados con métodos formales hace esta tarea aún más complicada.

## 7.3 Trabajo futuro

El futuro trabajo de investigación está orientado a mejorar o resolver las limitaciones descritas anteriormente. Se puede hablar de tres campos de investigación principales:

1. Técnicas de abstracción.
2. Especificación de requisitos.
3. La aplicación de diferentes algoritmos de verificación.

El primer campo de investigación son las técnicas de reducción y abstracción que pueden ser aplicadas a esta metodología. Nuevas y mejoradas técnicas de reducción y abstracción (por ejemplo “predicate abstraction”) son necesarias para que la metodología pudiera verificar cualquier programa PLC, ya que este es el objetivo final de esta metodología.

El segundo campo de investigación es el diseño de un mecanismo de especificación para sistemas de control. Este mecanismo tiene

que proporcionar un language sin ambigüedades y fácil de utilizar. Hoy en día, no existe un mecanismo o language para la especificación de requisitos en sistemas de control tal y como existen languages de programación estandarizados. En un futuro cercano, mecanismos de especificación formales o pseudo-formales deberían ser altamente recomendados por los estándares de automática para desarrollar sistemas de control más seguros y robustos. Disponer de una especificación completa y sin ambigüedades significaría la posibilidad de emplear el enfoque de “exactitud en la construcción (correctness by construction)” en la automatización industrial. Esto podría complementar el enfoque actual de esta tesis, donde verificación formal puede ser aplicada en los diferentes niveles del proceso de desarrollo de sistemas de control: en la especificación y el código de control, lo cual reduciría el número de errores potenciales al mínimo.

Finalmente, un factor muy interesante de esta metodología es que permite incluir diferentes técnicas de verificación formal en ella y los resultados pueden ser comparados. Desde este punto de vista, sería muy interesante incluir técnicas de verificación basadas en SAT para nuestros modelos. Durante el desarrollo de esta tesis, empezamos a estudiar estas técnicas y es un trabajo de investigación que va en proceso. “Bounded model checking” podría también mejorar el rendimiento de la verificación de nuestros modelos. En concreto, en la técnica de abstracción de variables, uno de los actuales cuellos de botella de esta técnica es la verificación de propiedades de “reachability” en modelos muy grandes, “bounded model checking” es una técnica adecuada para este tipo de análisis. Además, “compositional formal verification” es una estrategia de verificación muy prometedora para nuestros modelos, especialmente para programas UNICOS CPC. Esta investigación podría abrir la puerta a la verificación formal de sistemas de control distribuidos. La infraestructura de BIP proporciona una solución para “compositional formal verification”, aunque actualmente su herramienta de verificación no permite la transferencia de datos entre componentes.

Como conclusión final, considero que la tesis da una respuesta a los requisitos originales definidos al inicio de este proyecto. La relevancia de las contribuciones tanto teóricas como prácticas ha sido demostrada en los resultados experimentales usando programas PLC desarrollados

en el CERN. Las comunidades de automática y métodos formales están cada vez más cerca gracias a las contribuciones de esta tesis y las contribuciones de otros investigadores en este mismo campo.

Sin embargo, existe aún un largo camino para disponer de verificación formal aplicada ampliamente en esta industria. El desarrollo de esta tesis ha desencadenado nuevas líneas de investigación que contribuirán a que el espacio que separa estos dos mundos sea eliminado de una vez por todas.

# Appendix A

## PLC programs

The following piece of ST code corresponds to the Siemens implementation of the OnOff UNICOS CPC object.

```
1 //UNICOS
2 // Copyright CERN 2013 all rights reserved
3 (* ON/OFF OBJECT FUNCTION BLOCK ***** )
4 FUNCTION_BLOCK CPC_FB_ONOFF
5 TITLE = 'CPC_FB_ONOFF'
6 //
7 // ONOFF Object
8 //
9 VERSION: '6.5'
10 AUTHOR: 'EN/ICE'
11 NAME: 'OBJECT'
12 FAMILY: 'FO'
13
14 VAR_INPUT
15     HFOn:          BOOL;
16     HFOff:         BOOL;
17     HLD:           BOOL;
18     IOError:      BOOL;
19     IOSimu:       BOOL;
20     ALB:          BOOL;
21     Manreg01:     WORD;
22     Manreg01b AT Manreg01: ARRAY [0..15] OF BOOL;
23     HOnR:         BOOL;
24     HOffR:        BOOL;
25     StartI:      BOOL;
26     TStopI:      BOOL;
27     FuStopI:     BOOL;
28     Al:          BOOL;
29     AuOnR:       BOOL;
30     AuOffR:      BOOL;
31     AuAuMoR:    BOOL;
32     AuIhMMo:    BOOL;
33     AuIhFoMo:   BOOL;
34     AuAlAck:    BOOL;
35     IhAuMRW:    BOOL;
```

```

36   AuRstart:          BOOL;
37   POnOff:           CPC_ONOFF_PARAM;
38   POnOffb AT POnOff: STRUCT
39   ParRegb:          ARRAY [0..15] OF BOOL;
40   PPulseLeb:        TIME;
41   PWDtb:            TIME;
42   END_STRUCT;
43
44 END_VAR
45 VAR_OUTPUT
46   Stsreg01:          WORD;
47   Stsreg01b AT Stsreg01: ARRAY [0..15] OF BOOL;
48   Stsreg02:          WORD;
49   Stsreg02b AT Stsreg02: ARRAY [0..15] OF BOOL;
50   OutOnOV:          BOOL;
51   OutOffOV:         BOOL;
52   OnSt:              BOOL;
53   OffSt:             BOOL;
54   AuMoSt:            BOOL;
55   MMoSt:             BOOL;
56   LDSt:              BOOL;
57   SoftLDSt:         BOOL;
58   FoMoSt:            BOOL;
59   AuOnRSt:           BOOL;
60   AuOffRSt:          BOOL;
61   MOnRSt:            BOOL;
62   MOffRSt:           BOOL;
63   HOnRSt:            BOOL;
64   HOffRSt:           BOOL;
65   IOErrorW:          BOOL;
66   IOSimuW:           BOOL;
67   AuMRW:             BOOL;
68   AUnAck:            BOOL;
69   PosW:              BOOL;
70   StartISt:          BOOL;
71   TStopISt:          BOOL;
72   FuStopISt:         BOOL;
73   AlSt:              BOOL;
74   AlBW:              BOOL;
75   EnRstartSt:        BOOL := TRUE;
76   RdyStartSt:        BOOL;
77 END_VAR
78 VAR //Internal Variables
79     //Variables for Edge detection
80     E_MAuMoR:         BOOL;
81     E_MMMoR:          BOOL;
82     E_MFoMoR:         BOOL;
83     E_MOnR:           BOOL;
84     E_MOffR:          BOOL;
85     E_MAlAckR:        BOOL;
86     E_StartI:          BOOL;
87     E_TStopI:          BOOL;
88     E_FuStopI:         BOOL;
89     E_Al:              BOOL;
90     E_AuAuMoR:         BOOL;
91     E_AuAlAck:         BOOL;
92     E_MSoftLDR:        BOOL;
93     E_MEnRstartR:     BOOL;

```

```

94 RE_AlUnAck:      BOOL;
95 FE_AlUnAck:      BOOL;
96 RE_PulseOn:      BOOL;
97 FE_PulseOn:      BOOL;
98 RE_PulseOff:     BOOL;
99 RE_OutOVSt_aux:  BOOL;
100 FE_OutOVSt_aux:  BOOL;
101 FE_InterlockR:   BOOL;
102
103 //Variables for old values
104 MAuMoR_old:      BOOL;
105 MMMoR_old:       BOOL;
106 MFoMoR_old:      BOOL;
107 MOnR_old:        BOOL;
108 MOffR_old:       BOOL;
109 MAlAckR_old:     BOOL;
110 AuAuMoR_old:     BOOL;
111 AuAlAck_old:     BOOL;
112 StartI_old:      BOOL;
113 TStopI_old:      BOOL;
114 FuStopI_old:     BOOL;
115 Al_old:          BOOL;
116 AlUnAck_old:     BOOL;
117 MSoftLDR_old:    BOOL;
118 MEnRstartR_old:  BOOL;
119 RE_PulseOn_old:  BOOL;
120 FE_PulseOn_old:  BOOL;
121 RE_PulseOff_old: BOOL;
122 RE_OutOVSt_aux_old:  BOOL;
123 FE_OutOVSt_aux_old:  BOOL;
124 FE_InterlockR_old:  BOOL;
125
126 //General internal variables
127 PFsPosOn:        BOOL;
128 PFsPosOn2:       BOOL;
129 PHFOn:           BOOL;
130 PHFOff:          BOOL;
131 PPulse:          BOOL;
132 PPulseCste:     BOOL;
133 PHLD:            BOOL;
134 PHLDCmd:        BOOL;
135 PAnim:           BOOL;
136 POutOff:        BOOL;
137 PEnRstart:      BOOL;
138 PRstartFS:      BOOL;
139 OutOnOVSt:      BOOL;
140 OutOffOVSt:     BOOL;
141 AuMoSt_aux:     BOOL;
142 MMoSt_aux:      BOOL;
143 FoMoSt_aux:     BOOL;
144 SoftLDSt_aux:   BOOL;
145 PulseOn:        BOOL;
146 PulseOff:       BOOL;
147 PosW_aux:       BOOL;
148 OutOVSt_aux:    BOOL;
149 fullNotAcknowledged:  BOOL;
150 PulseOnR:       BOOL;
151 PulseOffR:      BOOL;

```

```

152 InterlockR:      BOOL;
153
154 //Variables for IEC Timers
155 Time_Warning:    TIME;
156 Timer_PulseOn:  TP;
157 Timer_PulseOff: TP;
158 Timer_Warning:  TON;
159
160 //Variables for interlock Sttus delay handling
161 PulseWidth:     REAL;
162 FSIinc:         INT;
163 TSIinc:         INT;
164 SIinc:          INT;
165 Alinc:          INT;
166 WTStopISt:     BOOL;
167 WStartISt:     BOOL;
168 WA1St:         BOOL;
169 WFuStopISt:    BOOL;
170 END_VAR
171
172 BEGIN
173 (* INPUT MANAGER *)
174
175 E_MaMoR := R_EDGE(new:=ManReg01b[8],old:=MAuMoR_old);      (*
176     Manual Auto Mode Request *
177 E_MMMoR := R_EDGE(new:=ManReg01b[9],old:=MMMOR_old);      (*
178     Manual Manual Mode Request *
179 E_MFoMoR := R_EDGE(new:=ManReg01b[10],old:=MFoMoR_old);   (*
180     Manual Forced Mode Request *
181 E_MSoftLDR := R_EDGE(new:=ManReg01b[11],old:=MSoftLDR_old); (*
182     Manual Software Local Drive Request *
183 E_MOnR := R_EDGE(new:=ManReg01b[12],old:=MOnR_old);       (*
184     Manual On/Open Request *
185 E_MOffR := R_EDGE(new:=ManReg01b[13],old:=MOffR_old);    (*
186     Manual Off/close Request *
187 E_MEnRstartR := R_EDGE(new:=ManReg01b[1],old:=MEnRstartR_old); (*
188     Manual Restart after full stop Request *
189 E_MAlAckR := R_EDGE(new:=ManReg01b[7],old:=MAlAckR_old);  (*
190     Manual Alarm Ack. Request *
191
192 PFsPosOn := POnOffb.ParRegb[8];                             (* 1st
193     Parameter bit to define Fail safe position behaviour *
194 PHFOn := POnOffb.ParRegb[9];                                 (*
195     Hardware feedback On present*
196 PHFOff := POnOffb.ParRegb[10];                              (*
197     Hardware feedback Off present*
198 PPulse := POnOffb.ParRegb[11];                              (*
199     Object is pulsed pulse duration : POnOff.PulseLe*
200 PHLD := POnOffb.ParRegb[12];                                 (* Local
201     Drive mode Allowed *
202 PHLDCmd := POnOffb.ParRegb[13];                              (* Local
203     Drive Command allowed *
204 PAnim := POnOffb.ParRegb[14];                                (*
205     Inverted Output*
206 POutOff := POnOffb.ParRegb[15];
207 PEnRstart := POnOffb.ParRegb[0];                             (*
208     Enable Restart after Full Stop *
209 PRstartFS := POnOffb.ParRegb[1];                             (*

```

```

194      Enable Restart when Full Stop still active *)
194      PFsPosOn2 := POnOffb.ParRegb[2];          (* 2nd
195      Parameter bit to define Fail safe position behaviour *)
195      PPulseCste := POnOffb.ParRegb[3];        (* Pulse Constant
196      duration irrespective of the feedback status *)
196
197      E_AuAuMoR := R_EDGE(new:=AuAuMoR,old:=AuAuMoR_old);    (* Auto
198      Auto Mode Request *)
198      E_AuAlAck := R_EDGE(new:=AuAlAck,old:=AuAlAck_old);    (* Auto
199      Alarm Ack. Request *)
199
200      E_StartI := R_EDGE(new:=StartI,old:=StartI_old);
201      E_TStopI := R_EDGE(new:=TStopI,old:=TStopI_old);
202      E_FuStopI := R_EDGE(new:=FuStopI,old:=FuStopI_old);
203      E_Al := R_EDGE(new:=Al,old:=Al_old);
204
205      StartISt := StartI;                        (* Start
206      Interlock present *)
206      TStopISt := TStopI;                        (*
207      Temporary Stop Interlock present *)
207      FuStopISt := FuStopI;                      (* Full
208      Stop Interlock present *)
208
209      (* INTERLOCK & ACKNOWLEDGE *)
210
211      IF (E_MAlAckR OR E_AuAlAck) THEN
212          fullNotAcknowledged := FALSE;
213          AlUnAck := FALSE;
214      ELSIF (E_TStopI OR E_StartI OR E_FuStopI OR E_Al) THEN
215          AlUnAck := TRUE;
216      END_IF;
217
218      IF (PEnrstart AND (E_MEnrstartR OR AuRstart) AND NOT FuStopISt) OR (
219          PEnrstart AND PRstartFS AND (E_MEnrstartR OR AuRstart)) AND NOT
220          fullNotAcknowledged THEN
221          EnrstartSt := TRUE;
222      END_IF;
223
224      IF E_FuStopI THEN
225          fullNotAcknowledged := TRUE;
226      END_IF;
227      IF PEnrstart THEN
228          EnrstartSt := FALSE;
229      END_IF;
230
231      InterlockR := TStopISt OR FuStopISt OR FullNotAcknowledged OR NOT
232          EnrstartSt OR
233          (StartISt AND NOT POutOff AND NOT OutOnOV) OR
234          (StartISt AND POutOff AND ((PFsPosOn AND OutOVSt_aux) OR (
235              NOT PFsPosOn AND NOT OutOVSt_aux)));
236
237      FE_InterlockR := F_EDGE (new:=InterlockR,old:=FE_InterlockR_old);
238
239      (* MODE MANAGER *)
240
241      IF NOT (HLD AND PHLD) THEN
242
243          (* Forced Mode *)

```

```

240         IF (AuMoSt_aux OR MMoSt_aux OR SoftLDSt_aux) AND
241             E_MFoMoR AND NOT(AuIhFoMo) THEN
242             AuMoSt_aux := FALSE;
243             MMoSt_aux := FALSE;
244             FoMoSt_aux := TRUE;
245             SoftLDSt_aux := FALSE;
246         END_IF;
247
248         (* Manual Mode *)
249         IF (AuMoSt_aux OR FoMoSt_aux OR SoftLDSt_aux) AND
250             E_MMMoR AND NOT(AuIhMMo) THEN
251             AuMoSt_aux := FALSE;
252             MMoSt_aux := TRUE;
253             FoMoSt_aux := FALSE;
254             SoftLDSt_aux := FALSE;
255         END_IF;
256
257         (* Auto Mode *)
258         IF (MMoSt_aux AND (E_MAUmoR OR E_AuAUmoR )) OR
259             (FoMoSt_aux AND E_MAUmoR) OR
260             (SoftLDSt_aux AND E_MAUmoR) OR
261             (MMoSt_aux AND AuIhMMo) OR
262             (FoMoSt_aux AND AuIhFoMo) OR
263             (SoftLDSt_aux AND AuIhFoMo) OR
264             NOT(AuMoSt_aux OR MMoSt_aux OR FoMoSt_aux OR SoftLDSt_aux)
265             THEN
266             AuMoSt_aux := TRUE;
267             MMoSt_aux := FALSE;
268             FoMoSt_aux := FALSE;
269             SoftLDSt_aux := FALSE;
270         END_IF;
271
272         (* Software Local Mode *)
273         IF (AuMoSt_aux OR MMoSt_aux) AND E_MSoftLDR AND NOT AuIhFoMo
274             THEN
275             AuMoSt_aux := FALSE;
276             MMoSt_aux := FALSE;
277             FoMoSt_aux := FALSE;
278             SoftLDSt_aux:= TRUE;
279         END_IF;
280
281         (* Status setting *)
282         LDSt := FALSE;
283         AuMoSt := AuMoSt_aux;
284         MMoSt := MMoSt_aux;
285         FoMoSt := FoMoSt_aux;
286         SoftLDSt := SoftLDSt_aux;
287
288     ELSE
289         (* Local Drive Mode *)
290         AuMoSt := FALSE;
291         MMoSt := FALSE;
292         FoMoSt := FALSE;
293         LDSt := TRUE;
294         SoftLDSt:= FALSE;
295     END_IF;

```

(\* LIMIT MANAGER \*)

```

296  (* On/Open Evaluation *)
297  OnSt:= (HFOn AND PHFOn) OR (*Feedback ON
    present*)
298  (NOT PHFOn AND PHFOff AND PAnim AND NOT HFOff) OR (*Feedback
    ON not present and PAnim = TRUE*)
299  (NOT PHFOn AND NOT PHFOff AND OutOVSt_aux);
300
301
302  (* Off/Closed Evaluation *)
303  OffSt:=(HFOff AND PHFOff) OR (*Feedback
    OFF present*)
304  (NOT PHFOff AND PHFOn AND PAnim AND NOT HFOn) OR (*Feedback
    OFF not present and PAnim = TRUE*)
305  (NOT PHFOn AND NOT PHFOff AND NOT OutOVSt_aux);
306
307
308  (* REQUEST MANAGER *)
309
310  (* Auto On/Off Request*)
311
312  IF AuOffR THEN
313  AuOnRSt := FALSE;
314  ELSIF AuOnR THEN
315  AuOnRSt := TRUE;
316  ELSIF fullNotAcknowledged OR FuStopISt OR NOT EnRstartSt THEN
317  AuOnRSt := PFsPosOn;
318  END_IF;
319  AuOffRSt:= NOT AuOnRSt;
320
321  (* Manual On/Off Request*)
322
323  IF (((E_MoffR AND (MMoSt OR FoMoSt OR SoftLDSt))
324  OR (AuOffRSt AND AuMoSt)
325  OR (LDSt AND PHLDCmd AND HOffRSt)
326  OR (FE_PulseOn AND PPulse AND NOT POutOff) AND EnRstartSt)
327  OR (E_FuStopI AND NOT PFsPosOn)) THEN
328
329  MOnRSt := FALSE;
330
331  ELSIF (((E_MOnR AND (MMoSt OR FoMoSt OR SoftLDSt))
332  OR (AuOnRSt AND AuMoSt)
333  OR (LDSt AND PHLDCmd AND HOnRSt) AND EnRstartSt)
334  OR (E_FuStopI AND PFsPosOn)) THEN
335
336  MOnRSt := TRUE;
337  END_IF;
338
339  MOffRSt:= NOT MOnRSt;
340
341  (* Local Drive Request *)
342
343  IF HOffR THEN
344  HOnRSt := FALSE;
345  ELSE IF HOnR THEN
346  HOnRSt := TRUE;
347  END_IF;
348  END_IF;
349  HOffRSt := NOT(HOnRSt);

```

```

350
351
352  (* PULSE REQUEST MANAGER*)
353  IF PPulse THEN
354      IF InterlockR THEN
355          PulseOnR:= (PFsPosOn AND NOT PFsPosOn2) OR (PFsPosOn AND
356                    PFsPosOn2);
357          PulseOffR:= (NOT PFsPosOn AND NOT PFsPosOn2) OR (PFsPosOn AND
358                    PFsPosOn2);
359          ELSIF FE_InterlockR THEN      (*Clear PulseOnR/PulseOffR to be sure you
360                    get a new pulse after InterlockR*)
361              PulseOnR:= FALSE;
362              PulseOffR:= FALSE;
363              Timer_PulseOn (IN:=FALSE,PT:=T#0s);
364              Timer_PulseOff (IN:=FALSE,PT:=T#0s);
365          ELSIF (MOffrSt AND (MMoSt OR FoMoSt OR SoftLDSt)) OR (AuOffrSt AND
366                    AuMoSt) OR (HOffr AND LDSt AND PHLDCmd) THEN //Off Request
367              PulseOnR:= FALSE;
368              PulseOffR:= TRUE;
369          ELSIF (MOnRSt AND (MMoSt OR FoMoSt OR SoftLDSt)) OR (AuOnRSt AND
370                    AuMoSt) OR (HOnR AND LDSt AND PHLDCmd) THEN //On Request
371              PulseOnR:= TRUE;
372              PulseOffR:= FALSE;
373          ELSE
374              PulseOnR:= FALSE;
375              PulseOffR:= FALSE;
376          END_IF;
377
378  //Pulse functions
379  Timer_PulseOn (IN:= PulseOnR,PT:=POnOffb.PPulseLeb);
380  Timer_PulseOff (IN:=PulseOffR,PT:=POnOffb.PPulseLeb);
381
382  RE_PulseOn := R_EDGE(new:=PulseOn,old:=RE_PulseOn_old);
383  FE_PulseOn := F_EDGE(new:=PulseOn,old:=FE_PulseOn_old);
384  RE_PulseOff := R_EDGE(new:=PulseOff,old:=RE_PulseOff_old);
385
386  //The pulse functions have to be reset when changing from On to Off
387  IF RE_PulseOn THEN
388      Timer_PulseOff (IN:=FALSE,PT:=T#0s);
389  END_IF;
390
391  IF RE_PulseOff THEN
392      Timer_PulseOn (IN:=FALSE,PT:=T#0s);
393  END_IF;
394
395  IF PPulseCste THEN      (* Pulse constant duration irrespective of feedback
396                    status *)
397      PulseOn := Timer_PulseOn.Q AND NOT PulseOffR;
398      PulseOff := Timer_PulseOff.Q AND NOT PulseOnR;
399  ELSE
400      PulseOn := Timer_PulseOn.Q AND NOT PulseOffR AND (NOT PHFOn OR (
401                    PHFOn AND NOT HFOn));
402      PulseOff := Timer_PulseOff.Q AND NOT PulseOnR AND (NOT PHFOff OR (
403                    PHFOff AND NOT HFOff));
404  END_IF;
405  END_IF;

```

```

400  (* Output On Request *)
401  OutOnOVSt := (PPulse AND PulseOn) OR
402             (NOT PPulse AND ((MOnRSt AND (MMoSt OR FoMoSt OR SoftLDSt
403             )) OR
404             (AuOnRSt AND AuMoSt) OR
405             (HOnRST AND LDSt AND PHLDCmd)));
406  (* Output Off Request *)
407  IF POutOff THEN
408      OutOffOVSt := (PulseOff AND PPulse) OR
409                  (NOT(PPulse) AND ((MOffRSt AND (MMoSt OR FoMoSt OR
410                  SoftLDSt)) OR (AuOffrSt AND AuMoSt) OR (HOffrSt
411                  AND LDSt AND PHLDCmd)));
412  END_IF;
413  (* Interlocks / FailSafe *)
414  IF POutOff THEN
415      IF InterlockR THEN
416          IF PPulse AND NOT PFsPosOn2 THEN
417              IF PFsPosOn THEN
418                  OutOnOVSt := PulseOn;
419                  OutOffOVSt := FALSE;
420              ELSE
421                  OutOnOVSt := FALSE;
422                  OutOffOVSt := PulseOff;
423              END_IF;
424          ELSE
425              OutOnOVSt := (PFsPosOn AND NOT PFsPosOn2) OR (PFsPosOn
426              AND PFsPosOn2);
427              OutOffOVSt := (NOT PFsPosOn AND NOT PFsPosOn2) OR (
428              PFsPosOn AND PFsPosOn2);
429          END_IF;
430      ELSE
431          IF InterlockR THEN
432              OutOnOVSt := PFsPosOn;
433          END_IF;
434      END_IF;
435  (* Ready to Start Status *)
436  RdyStartSt := NOT InterlockR;
437  (* Alarms *)
438  AlSt := Al;
439  (* SURVEILLANCE *)
440  (* I/O Warning *)
441  IOErrorW := IOError;
442  IOSimuW := IOSimu;
443  (* Auto<> Manual Warning *)
444  AuMRW := (MMoSt OR FoMoSt OR SoftLDSt) AND
445           ((AuOnRSt XOR MOnRSt) OR (AuOffrSt XOR MOffrSt)) AND NOT
446           IhAuMRW;

```

```

452
453
454
455 (* OUTPUT_MANAGER AND OUTPUT REGISTER *)
456 IF NOT POutOff THEN
457     IF PFsPosOn THEN
458         OutOnOV := NOT OutOnOVSt;
459     ELSE
460         OutOnOV := OutOnOVSt;
461     END_IF;
462 ELSE
463     OutOnOV := OutOnOVSt;
464     OutOffOV := OutOffOVSt;
465 END_IF;
466
467 (* Position warning *)
468
469     (* Set reset of the OutOnOVSt *)
470     IF OutOnOVSt OR (PPulse AND PulseOnR) THEN
471         OutOVSt_aux := TRUE;
472     END_IF;
473     IF (OutOffOVSt AND POutOff) OR (NOT OutOnOVSt AND NOT POutOff) OR (
474         PPulse AND PulseOffR) THEN
475         OutOVSt_aux := FALSE;
476     END_IF;
477
478     RE_OutOVSt_aux := R_EDGE(new:=OutOVSt_aux,old:=RE_OutOVSt_aux_old);
479     FE_OutOVSt_aux := F_EDGE(new:=OutOVSt_aux,old:=FE_OutOVSt_aux_old);
480
481     IF ((OutOVSt_aux AND ((PHFOn AND NOT OnSt) OR (PHFOff AND OffSt)))
482         OR (NOT OutOVSt_aux AND ((PHFOff AND NOT OffSt) OR (PHFON AND OnSt)))
483         )
484         OR (OffSt AND OnSt))
485         AND (NOT PPulse OR (POutOff AND PPulse AND NOT OutOnOV AND NOT
486             OutOffOV))
487     THEN
488         PosW_aux := TRUE;
489     END_IF;
490
491     IF NOT ((OutOVSt_aux AND ((PHFOn AND NOT OnSt) OR (PHFOff AND OffSt)))
492         OR (NOT OutOVSt_aux AND ((PHFOff AND NOT OffSt) OR (PHFON AND OnSt)))
493         )
494         OR (OffSt AND OnSt))
495         OR RE_OutOVSt_aux
496         OR FE_OutOVSt_aux
497         OR (PPulse AND POutOff AND OutOnOV)
498         OR (PPulse AND POutOff AND OutOffOV)
499     THEN
500         PosW_aux := FALSE;
501     END_IF;
502
503     Timer_Warning(IN := PosW_aux,
504                 PT := POnOffb.PWdtb);
505
506     PosW := Timer_Warning.Q;
507     Time_Warning := Timer_Warning.ET;
508
509 (* Alarm Blocked Warning*)

```

```

506
507     ALBW := ALB;
508
509     (* Maintain Interlock status 1.5s in Stsreg for PVSS *)
510
511     PulseWidth := 1500 (* msec*) / DINT_TO_REAL(TIME_TO_DINT(T_CYCLE));
512
513
514     IF FuStopISt OR FSIinc > 0 THEN
515         FSIinc := FSIinc + 1;
516         WFuStopISt := TRUE;
517     END_IF;
518
519     IF FSIinc > PulseWidth OR (NOT FuStopISt AND FSIinc = 0) THEN
520         FSIinc := 0;
521         WFuStopISt := FuStopISt;
522     END_IF;
523
524     IF TStopISt OR TSIinc > 0 THEN
525         TSIinc := TSIinc + 1;
526         WTStopISt := TRUE;
527     END_IF;
528
529     IF TSIinc > PulseWidth OR (NOT TStopISt AND TSIinc = 0) THEN
530         TSIinc := 0;
531         WTStopISt := TStopISt;
532     END_IF;
533
534     IF StartISt OR SIinc > 0 THEN
535         SIinc := SIinc + 1;
536         WStartISt := TRUE;
537     END_IF;
538
539     IF SIinc > PulseWidth OR (NOT StartISt AND SIinc = 0) THEN
540         SIinc := 0;
541         WStartISt := StartISt;
542     END_IF;
543
544     IF AlSt OR Alinc > 0 THEN
545         Alinc := Alinc + 1;
546         WALSt := TRUE;
547     END_IF;
548
549     IF Alinc > PulseWidth OR (NOT AlSt AND Alinc = 0) THEN
550         Alinc := 0;
551         WALSt := AlSt;
552     END_IF;
553
554
555     (* STATUS REGISTER *)
556
557     Stsreg01b[8] := OnSt;           //StsReg01 Bit 00
558     Stsreg01b[9] := OffSt;        //StsReg01 Bit 01
559     Stsreg01b[10] := AuMoSt;      //StsReg01 Bit 02
560     Stsreg01b[11] := MMoSt;       //StsReg01 Bit 03
561     Stsreg01b[12] := FoMoSt;      //StsReg01 Bit 04
562     Stsreg01b[13] := LDSt;        //StsReg01 Bit 05
563     Stsreg01b[14] := IOErrorW;    //StsReg01 Bit 06

```

```

564 Stsreg01b[15] := IOSimuW;           //StsReg01 Bit 07
565 stsreg01b[0] := AuMRW;             //StsReg01 Bit 08
566 Stsreg01b[1] := PosW;              //StsReg01 Bit 09
567 Stsreg01b[2] := WStartISt;         //StsReg01 Bit 10
568 Stsreg01b[3] := WTStopISt;         //StsReg01 Bit 11
569 Stsreg01b[4] := AlUnAck;           //StsReg01 Bit 12
570 Stsreg01b[5] := AuIhFoMo;         //StsReg01 Bit 13
571 Stsreg01b[6] := WAlSt;             //StsReg01 Bit 14
572 Stsreg01b[7] := AuIhMMo;          //StsReg01 Bit 15
573
574 Stsreg02b[8] := OutOnOVSt;          //StsReg02 Bit 00
575 Stsreg02b[9] := AuOnRSt;           //StsReg02 Bit 01
576 Stsreg02b[10] := MOnRSt;          //StsReg02 Bit 02
577 Stsreg02b[11] := AuOffRSt;        //StsReg02 Bit 03
578 Stsreg02b[12] := MOffRSt;         //StsReg02 Bit 04
579 Stsreg02b[13] := HOOnRSt;         //StsReg02 Bit 05
580 Stsreg02b[14] := HOffRSt;         //StsReg02 Bit 06
581 Stsreg02b[15] := 0;               //StsReg02 Bit 07
582 stsreg02b[0] := 0;                //StsReg02 Bit 08
583 Stsreg02b[1] := 0;                //StsReg02 Bit 09
584 Stsreg02b[2] := WFuStopISt ;      //StsReg02 Bit 10
585 Stsreg02b[3] := EnRstartSt;        //StsReg02 Bit 11
586 Stsreg02b[4] := SoftLDSt;         //StsReg02 Bit 12
587 Stsreg02b[5] := AlBW;             //StsReg02 Bit 13
588 Stsreg02b[6] := OutOffOVSt;       //StsReg02 Bit 14
589 Stsreg02b[7] := 0;                //StsReg02 Bit 15
590
591 (* Edges *)
592
593 DETECT_EDGE(new:=AlUnAck,old:=AlUnAck_old,re:=RE_AlUnAck,fe:=FE_AlUnAck);
594
595
596 END_FUNCTION_BLOCK
597
598
599 // Common functions for UNICOS applications + implementation of platform FBs.
600 //UNICOS Copyright CERN 2013 all rights reserved
601
602 (* DATA STRUCTURES ***** )
603 TYPE CPC_DB_PA_STATUS
604 TITLE = 'CPC_DB_PA_STATUS' AUTHOR : 'EN/ICE'
605 FAMILY : 'Base'
606 STRUCT
607     ReadBack : REAL; //position of the valve
608     ReadBack_status : BYTE; //status of readback
609     Pos_D : BYTE; //position discrete
610     Pos_D_Status : BYTE; //status of Pos_D
611     CheckBack0 : BYTE; //checkback status 0
612     CheckBack1 : BYTE; //checkback status 1
613     CheckBack2 : BYTE; //checkback status 2
614 END_STRUCT; END_TYPE
615 TYPE CPC_DB_COMM
616 TITLE = 'CPC_DB_COMM'
617 //
618 // Type for the communication DB
619 //
620 AUTHOR : 'EN/ICE'
621 NAME : 'DataType'

```

```

622 FAMILY : 'Base'
623   STRUCT
624     status_DB : INT;
625     status_DB_old : INT;
626     size : INT;
627   END_STRUCT
628 END_TYPE
629 TYPE CPC_ONOFF_PARAM
630 TITLE = 'CPC_ONOFF_PARAM'
631 //
632 // Parameters of ONOFF
633 //
634 AUTHOR : 'EN/ICE'
635 NAME : 'DataType'
636 FAMILY : 'Base'
637   STRUCT
638     ParReg : WORD;
639     PPulseLe : TIME;
640     PWDt : TIME;
641   END_STRUCT
642 END_TYPE
643 TYPE CPC_LOCAL_PARAM
644 TITLE = 'CPC_LOCAL_PARAM'
645 //
646 // Parameters of LOCAL
647 //
648 AUTHOR : 'EN/ICE'
649 NAME : 'DataType'
650 FAMILY : 'Base'
651   STRUCT
652     ParReg : WORD;
653   END_STRUCT
654 END_TYPE
655 TYPE CPC_ANALOGALARM_PARAM
656 TITLE = 'CPC_ANALOGALARM_PARAM'
657 //
658 // Parameters of Analog Alarm
659 //
660 AUTHOR : 'EN/ICE'
661 NAME : 'DataType'
662 FAMILY : 'Base'
663   STRUCT
664     ParReg : WORD;
665   END_STRUCT
666 END_TYPE
667 TYPE CPC_PCO_PARAM
668 TITLE = 'CPC_PCO_PARAM'
669 //
670 // Parameters of PCO
671 //
672 AUTHOR : 'EN/ICE'
673 NAME : 'DataType'
674 FAMILY : 'Base'
675   STRUCT
676     ParReg : WORD;
677   END_STRUCT
678 END_TYPE
679 TYPE CPC_ANALOG_PARAM

```

```

680 TITLE = 'CPC_ANALOG_PARAM'
681 //
682 // Parameters of Analog and Anadig Objects
683 //
684 AUTHOR : 'EN/ICE'
685 NAME : 'DataType'
686 FAMILY : 'Base'
687 STRUCT
688     ParReg : WORD;
689     PMaxRan : REAL;
690     PMinRan : REAL;
691     PMStpInV : REAL;
692     PMStpDeV : REAL;
693     PMInSpd : REAL;
694     PMDeSpd : REAL;
695     PWDt : TIME;
696     PWDb : REAL;
697 END_STRUCT
698 END_TYPE
699 TYPE CPC_ANADIG_PWM_PARAM
700 TITLE = 'PARAM_PWM'
701 //
702 // Parameters of the Pulse Wave Modulation of Anadig objects
703 //
704 AUTHOR : 'EN/ICE'
705 NAME : 'DataType'
706 FAMILY : 'Base'
707 STRUCT
708     PTPeriod : TIME;
709     PTMin : TIME;
710     PInMax : REAL;
711 END_STRUCT
712 END_TYPE
713 TYPE CPC_PID_LIB_PARAM
714 TITLE = 'PARAM_PID_LIB'
715 //
716 // Parameters of the PID
717 //
718 AUTHOR : 'EN/ICE'
719 NAME : 'DataType'
720 FAMILY : 'Base'
721 STRUCT
722     Kc : REAL;
723     Ti : REAL;
724     Td : REAL;
725     Tds : REAL;
726     SPH : REAL;
727     SPL : REAL;
728     OutH : REAL;
729     OutL : REAL;
730     EKc : BOOL;
731     ETi : BOOL;
732     ETd : BOOL;
733     ETds : BOOL;
734     ESPH : BOOL;
735     ESPL : BOOL;
736     EOutH : BOOL;
737     EOutL : BOOL;

```

```

738   END_STRUCT
739 END_TYPE
740 TYPE CPC_PID_PARAM
741 TITLE = 'PARAM_PID'
742 //
743 // Parameters of controller objects
744 //
745 AUTHOR : 'EN/ICE'
746 NAME : 'DataType'
747 FAMILY : 'Base'
748   STRUCT
749     PMinRan : REAL;
750     PMaxRan : REAL;
751     POutMinRan : REAL;
752     POutMaxRan : REAL;
753     MVFiltTime : TIME;
754     PIDCycle : TIME;
755     ScaMethod : INT;
756     RA : BOOL;
757   END_STRUCT
758 END_TYPE
759 TYPE CPC_RAMP_PARAM
760 TITLE = 'PARAM_RAMP'
761 //
762 // Ramp parameters for PID
763 //
764 AUTHOR : 'EN/ICE'
765 NAME : 'DataType'
766 FAMILY : 'Base'
767   STRUCT
768     InSpd : REAL;
769     DeSpd : REAL;
770   END_STRUCT
771 END_TYPE
772 TYPE CPC_IOERROR
773 TITLE = 'TYPE_ERR'
774 //
775 // Used for IOError in AI/DI/AO/DO
776 //
777 AUTHOR : 'EN/ICE'
778 NAME : 'DataType'
779 FAMILY : 'Base'
780   STRUCT
781     ADDR : REAL; //Channel Address
782     Err : BOOL; //Error
783   END_STRUCT
784 END_TYPE
785
786 (* OTHER FUNCTIONS *****)
787
788 (* Version DB*)
789 DATA_BLOCK CPC_DB_VERSION
790 TITLE = 'DB_CPC_VERSION' AUTHOR : 'UNICOS'
791 NAME : 'Version'
792 FAMILY : 'Version'
793 STRUCT
794   Baseline_version : REAL := 6.6; //Version of the baseline used
795 END_STRUCT

```

```

796 BEGIN
797 END_DATA_BLOCK
798
799 (*Rising Edge*)
800 FUNCTION R_EDGE : BOOL
801 TITLE = 'R_EDGE'
802 //
803 // Detect a Rising Edge on a signal
804 //
805 AUTHOR : 'EN/ICE'
806 NAME : 'Function'
807 FAMILY : 'Base'
808 VAR_INPUT
809     new : BOOL;
810 END_VAR
811 VAR_IN_OUT
812     old : BOOL;
813 END_VAR
814 BEGIN
815
816     IF (new = 1 AND old = 0) THEN //Raising edge detected
817         R_EDGE := 1;
818         old := 1;
819     ELSE R_EDGE := 0;
820         old := new;
821     END_IF;
822 END_FUNCTION
823
824 (*Falling Edge*)
825 FUNCTION F_EDGE : BOOL
826 TITLE = 'F_EDGE'
827 //
828 // Detect a Falling Edge on a signal
829 //
830 AUTHOR : 'EN/ICE'
831 NAME : 'Function'
832 FAMILY : 'Base'
833 VAR_INPUT
834     new : BOOL;
835 END_VAR
836 VAR_IN_OUT
837     old : BOOL;
838 END_VAR
839 BEGIN
840
841     IF (new = 0 AND old = 1) THEN //Falling edge detected
842         F_EDGE := 1;
843         old := 0;
844     ELSE F_EDGE := 0;
845         old := new;
846     END_IF;
847 END_FUNCTION
848
849 (*Rising and Falling Edge*)
850 FUNCTION DETECT_EDGE : VOID
851 TITLE = 'DETECT_EDGE'
852 //
853 // Detect a Rising and Falling Edge of a signal

```

```

854 //
855 AUTHOR : 'EN/ICE'
856 NAME : 'Function'
857 FAMILY : 'Base'
858 VAR_INPUT
859     new : BOOL;
860 END_VAR
861 VAR_IN_OUT
862     old : BOOL;
863 END_VAR
864 VAR_OUTPUT
865     re : BOOL;
866     fe : BOOL;
867 END_VAR
868
869 BEGIN
870     IF new <> old THEN
871         IF new = TRUE THEN // Raising edge
872             re := TRUE;
873             fe := FALSE;
874         ELSE // Falling edge
875             re := FALSE;
876             fe := TRUE;
877         END_IF;
878         old := new; // shift new to old
879
880     ELSE re := FALSE; // reset edge detection
881         fe := FALSE;
882     END_IF;
883 END_FUNCTION
884
885 // TIMERS
886 /// GLOBALVAR _GLOBAL_TIME : TIME;
887 /// GLOBALVAR T_CYCLE : UINT;
888
889 // Pulse timer
890 FUNCTION_BLOCK TP
891 VAR_INPUT
892     PT : TIME;
893     IN : BOOL;
894 END_VAR
895 VAR_OUTPUT
896     Q : BOOL := FALSE;
897     ET : TIME;
898 END_VAR
899 VAR
900     running : BOOL;
901     Start : TIME;
902 END_VAR
903 BEGIN
904     IF Q = FALSE AND running = FALSE THEN
905         IF IN THEN
906             Start := CT;
907             running := TRUE;
908             Q := TRUE; // t1
909         END_IF;
910     ELSIF Q = TRUE AND running = TRUE THEN
911         IF IN AND (CTIME - (start + PT) >= 0) THEN

```

```

912     running := TRUE;
913     Q := FALSE;           // t2
914     ELSIF NOT IN AND (CTIME - (start + PT) >= 0) THEN
915         running := FALSE;
916         Q := FALSE;       // t3
917     END_IF;
918 ELSIF Q = FALSE AND running = FALSE THEN
919     IF NOT IN THEN
920         running := FALSE;
921         Q := FALSE;       // t4
922     END_IF;
923 END_IF;
924 END_FUNCTION_BLOCK
925
926
927 // On-delay timer
928 FUNCTION_BLOCK TON
929 VAR_INPUT
930     PT : TIME;
931     IN : BOOL;
932 END_VAR
933 VAR_OUTPUT
934     Q : BOOL := FALSE;
935     ET : TIME; // elapsed time
936 END_VAR
937 VAR
938     running : BOOL;
939     start : BOOL;
940 END_VAR
941 BEGIN
942 IF IN = FALSE THEN
943     Q := FALSE;
944     ET := 0;
945     running := FALSE; // t1
946 ELSIF running = FALSE THEN
947     start := CTIME;
948     running := TRUE; // t2
949 ELSIF CTIME - (start + PT) >= 0 THEN
950     Q := TRUE;
951     ET := PT; // t3
952 ELSE
953     IF NOT Q THEN
954         ET := CTIME - start;
955     END_IF; // t4
956 END_IF;

```

Listing A.1: OnOff PLC program

# Appendix B

## nuXmv models

The following piece of NuSMV code corresponds to the automatically generated model from the OnOff UNICOS CPC object (whithout applying any reductions).

```
1 -- V1.1
2 -- Generated model from OnOff.scl by bfernand
3
4 MODULE module_INSTANCE(interaction, main)
5   VAR
6     loc : {initial, end, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 110, 111
      , 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124
      , 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137
      , 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150
      , 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163
      , 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176
      , 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189
      , 190, 191, 192, 193, 194, 195, 196, 197, 198, 199, 1100, 1101,
      1102, 1103, 1104, 1105, 1106, 1107, 1108, 1109, 1110, 1111, 1112,
      1113, 1114, 1115, 1116, 1117, 1118, 1119, 1120, 1121, 1122, 1123
      , 1124, 1125, 1126, 1127, 1128, 1129, 1130, 1131, 1132, 1133,
      1134, 1135, 1136, 1137, 1138, 1139, 1140, 1141, 1142, 1143, 1144,
      1145, 1146, 1147, 1148, 1149, 1150, 1151, 1152, 1153, 1154, 1155
      , 1156, 1157, 1158, 1159, 1160, 1161, 1162, 1163, 1164, 1165,
      1166, 1167, 1168, 1169, 1170, 1171, 1172, 1173, 1174, 1175, 1176,
      1177, 1178, 1179, 1180, 1181, 1182, 1183, 1184, 1185, 1186, 1187
      , 1188, 1189, 1190, 1191, 1192, 1193, 1194, 1195, 1196, 1197,
      1198, 1199, 1200, 1201, 1202, 1203, 1204, 1205, 1206, 1207, 1208,
      1209, 1210, 1211, 1212, 1213, 1214, 1215, 1216, 1217, 1218, 1219
      , 1220, 1221, 1222, 1223, 1224, 1225, 1226, 1227, 1228, 1229,
      1230, 1231, 1232, 1233, 1234, 1235, 1236, 1237, 1238, 1239, 1240,
      1241, 1242, 1243, 1244, 1245, 1246, 1247, 1248, 1249, 1250, 1251
      , 1252, 1253, 1254, 1255, 1256, 1257, 1258, 1259, 1260, 1261,
      1262, 1263, 1264, 1265, 1266, 1267, 1268, 1269, 1270, 1271, 1272,
      1273, 1274, 1275, 1276, 1277, 1278, 1279, 1280, 1281, 1282, 1283
      , 1284, 1285, 1286, 1287, 1288, 1289, 1290, 1291, 1292, 1293,
      1294, 1295, 1296, 1297, 1298, 1299, 1300, 1301, 1302, 1303, 1304,
      1305, 1306, 1307, 1308, 1309, 1310, 1311, 1312, 1313, 1314, 1315
      , 1316, 1317, 1318, 1319, 1320, 1321, 1322, 1323, 1324, 1325,
```

```

1326, 1327, 1328, 1329, 1330, 1331, 1332, 1333, 1334, 1335, 1336,
1337, 1338, 1339, 1340, 1341, 1342, 1343, 1344, 1345, 1346, 1347
, 1348, 1349, 1350, 1351, 1352, 1353, 1354, 1355, 1356, 1357,
1358, 1359, 1360, 1361, 1362, 1363, 1364, 1365, 1366, 1367, 1368,
1369, 1370, 1371, 1372, 1373, 1374, 1375, 1376, 1377, 1378, 1379
, 1380, 1381, 1382, 1383, 1384, 1385, 1386, 1387, 1388, 1389,
1390, 1391, 1392, 1393, 1394, 1395, 1396, 1397, 1398, 1399, 1400,
1401, 1402, 1403, 1404, 1405, 1406, 1407, 1408, 1409, 1410, 1411
, 1412, 1413, 1414, 1415, 1416, 1417, 1418, 1419, 1420, 1421,
1422, 1423, 1424, 1425, 1426, 1427, 1428, 1429, 1430, 1431, 1432,
1433, 1434, 1435, 1436, 1437, 1438, 1439, 1440, 1441, 1442, 1443
, 1444, 1445, 1446, 1447, 1448, 1449, 1450, 1127_elsif_1,
1179_elsif_1, 1179_elsif_2, 1184_elsif_1, 1194_elsif_1,
1194_elsif_2, 1194_elsif_3};
7  HFON : boolean;
8  HFOFF : boolean;
9  HLD : boolean;
10 IOERROR : boolean;
11 IOSIMU : boolean;
12 ALB : boolean;
13 MANREG01_0__ : boolean;
14 MANREG01_1__ : boolean;
15 MANREG01_2__ : boolean;
16 MANREG01_3__ : boolean;
17 MANREG01_4__ : boolean;
18 MANREG01_5__ : boolean;
19 MANREG01_6__ : boolean;
20 MANREG01_7__ : boolean;
21 MANREG01_8__ : boolean;
22 MANREG01_9__ : boolean;
23 MANREG01_10__ : boolean;
24 MANREG01_11__ : boolean;
25 MANREG01_12__ : boolean;
26 MANREG01_13__ : boolean;
27 MANREG01_14__ : boolean;
28 MANREG01_15__ : boolean;
29 HONR : boolean;
30 HOFFR : boolean;
31 STARTI : boolean;
32 TSTOPI : boolean;
33 FUSTOPI : boolean;
34 AL : boolean;
35 AUONR : boolean;
36 AUOFFR : boolean;
37 AUAUMOR : boolean;
38 AUIHMMO : boolean;
39 AUIHFOMO : boolean;
40 AUALACK : boolean;
41 IHAUMRW : boolean;
42 AURSTART : boolean;
43 PONOFF.PARREG_0__ : boolean;
44 PONOFF.PARREG_1__ : boolean;
45 PONOFF.PARREG_2__ : boolean;
46 PONOFF.PARREG_3__ : boolean;
47 PONOFF.PARREG_4__ : boolean;
48 PONOFF.PARREG_5__ : boolean;
49 PONOFF.PARREG_6__ : boolean;
50 PONOFF.PARREG_7__ : boolean;

```

```
51  PONOFF.PARREG_8__ : boolean;
52  PONOFF.PARREG_9__ : boolean;
53  PONOFF.PARREG_10__ : boolean;
54  PONOFF.PARREG_11__ : boolean;
55  PONOFF.PARREG_12__ : boolean;
56  PONOFF.PARREG_13__ : boolean;
57  PONOFF.PARREG_14__ : boolean;
58  PONOFF.PARREG_15__ : boolean;
59  PONOFF.PPULSESE : signed word[32];
60  PONOFF.PWDT : signed word[32];
61  STSREG01_0__ : boolean;
62  STSREG01_1__ : boolean;
63  STSREG01_2__ : boolean;
64  STSREG01_3__ : boolean;
65  STSREG01_4__ : boolean;
66  STSREG01_5__ : boolean;
67  STSREG01_6__ : boolean;
68  STSREG01_7__ : boolean;
69  STSREG01_8__ : boolean;
70  STSREG01_9__ : boolean;
71  STSREG01_10__ : boolean;
72  STSREG01_11__ : boolean;
73  STSREG01_12__ : boolean;
74  STSREG01_13__ : boolean;
75  STSREG01_14__ : boolean;
76  STSREG01_15__ : boolean;
77  STSREG02_0__ : boolean;
78  STSREG02_1__ : boolean;
79  STSREG02_2__ : boolean;
80  STSREG02_3__ : boolean;
81  STSREG02_4__ : boolean;
82  STSREG02_5__ : boolean;
83  STSREG02_6__ : boolean;
84  STSREG02_7__ : boolean;
85  STSREG02_8__ : boolean;
86  STSREG02_9__ : boolean;
87  STSREG02_10__ : boolean;
88  STSREG02_11__ : boolean;
89  STSREG02_12__ : boolean;
90  STSREG02_13__ : boolean;
91  STSREG02_14__ : boolean;
92  STSREG02_15__ : boolean;
93  OUTONOV : boolean;
94  OUTOFFOV : boolean;
95  ONST : boolean;
96  OFFST : boolean;
97  AUMOST : boolean;
98  MMOST : boolean;
99  LDST : boolean;
100  SOFTLDST : boolean;
101  FOMOST : boolean;
102  AUONRST : boolean;
103  AUOFFRST : boolean;
104  MONRST : boolean;
105  MOFFRST : boolean;
106  HONRST : boolean;
107  HOFFRST : boolean;
108  IOERRORW : boolean;
```

```
109 IOSIMUW : boolean;
110 AUMRW : boolean;
111 ALUNACK : boolean;
112 POSW : boolean;
113 STARTIST : boolean;
114 TSTOPIST : boolean;
115 FUSTOPIST : boolean;
116 ALST : boolean;
117 ALBW : boolean;
118 ENRSTARTST : boolean;
119 RDYSTARTST : boolean;
120 E_MAUMOR : boolean;
121 E_MMMOR : boolean;
122 E_MFOMOR : boolean;
123 E_MONR : boolean;
124 E_MOFFR : boolean;
125 E_MALACKR : boolean;
126 E_STARTI : boolean;
127 E_TSTOPI : boolean;
128 E_FUSTOPI : boolean;
129 E_AL : boolean;
130 E_AUAUMOR : boolean;
131 E_AUALACK : boolean;
132 E_MSOFTLDR : boolean;
133 E_MENRSTARTR : boolean;
134 RE_ALUNACK : boolean;
135 FE_ALUNACK : boolean;
136 RE_PULSEON : boolean;
137 FE_PULSEON : boolean;
138 RE_PULSEOFF : boolean;
139 RE_OUTOVST_AUX : boolean;
140 FE_OUTOVST_AUX : boolean;
141 FE_INTERLOCKR : boolean;
142 MAUMOR_OLD : boolean;
143 MMMOR_OLD : boolean;
144 MFOMOR_OLD : boolean;
145 MONR_OLD : boolean;
146 MOFFR_OLD : boolean;
147 MALACKR_OLD : boolean;
148 AUAUMOR_OLD : boolean;
149 AUALACK_OLD : boolean;
150 STARTI_OLD : boolean;
151 TSTOPI_OLD : boolean;
152 FUSTOPI_OLD : boolean;
153 AL_OLD : boolean;
154 ALUNACK_OLD : boolean;
155 MSOFTLDR_OLD : boolean;
156 MENRSTARTR_OLD : boolean;
157 RE_PULSEON_OLD : boolean;
158 FE_PULSEON_OLD : boolean;
159 RE_PULSEOFF_OLD : boolean;
160 RE_OUTOVST_AUX_OLD : boolean;
161 FE_OUTOVST_AUX_OLD : boolean;
162 FE_INTERLOCKR_OLD : boolean;
163 PFSPOSON : boolean;
164 PFSPOSON2 : boolean;
165 PHFON : boolean;
166 PHFOFF : boolean;
```

```
167 PPULSE : boolean;
168 PPULSECSTE : boolean;
169 PHLD : boolean;
170 PHLDCMD : boolean;
171 PANIM : boolean;
172 POUTOFF : boolean;
173 PENRSTART : boolean;
174 PRSTARTFS : boolean;
175 OUTONOVST : boolean;
176 OUTOFFOVST : boolean;
177 AUMOST_AUX : boolean;
178 MMOST_AUX : boolean;
179 FOMOST_AUX : boolean;
180 SOFTLDST_AUX : boolean;
181 PULSEON : boolean;
182 PULSEOFF : boolean;
183 POSW_AUX : boolean;
184 OUTOVST_AUX : boolean;
185 FULLNOTACKNOWLEDGED : boolean;
186 PULSEONR : boolean;
187 PULSEOFFR : boolean;
188 INTERLOCKR : boolean;
189 TIME_WARNING : signed word[32];
190 PULSEWIDTH : signed word[32];
191 FSIINC : signed word[16];
192 TSIINC : signed word[16];
193 SIINC : signed word[16];
194 ALINC : signed word[16];
195 WTSTOPIST : boolean;
196 WSTARTIST : boolean;
197 WALST : boolean;
198 WFUSTOPIST : boolean;
199 INLINED_R_EDGE_1__NEW : boolean;
200 INLINED_R_EDGE_1__OLD : boolean;
201 INLINED_R_EDGE_2__NEW : boolean;
202 INLINED_R_EDGE_2__OLD : boolean;
203 INLINED_R_EDGE_3__NEW : boolean;
204 INLINED_R_EDGE_3__OLD : boolean;
205 INLINED_R_EDGE_4__NEW : boolean;
206 INLINED_R_EDGE_4__OLD : boolean;
207 INLINED_R_EDGE_5__NEW : boolean;
208 INLINED_R_EDGE_5__OLD : boolean;
209 INLINED_R_EDGE_6__NEW : boolean;
210 INLINED_R_EDGE_6__OLD : boolean;
211 INLINED_R_EDGE_7__NEW : boolean;
212 INLINED_R_EDGE_7__OLD : boolean;
213 INLINED_R_EDGE_8__NEW : boolean;
214 INLINED_R_EDGE_8__OLD : boolean;
215 INLINED_R_EDGE_9__NEW : boolean;
216 INLINED_R_EDGE_9__OLD : boolean;
217 INLINED_R_EDGE_10__NEW : boolean;
218 INLINED_R_EDGE_10__OLD : boolean;
219 INLINED_R_EDGE_11__NEW : boolean;
220 INLINED_R_EDGE_11__OLD : boolean;
221 INLINED_R_EDGE_12__NEW : boolean;
222 INLINED_R_EDGE_12__OLD : boolean;
223 INLINED_R_EDGE_13__NEW : boolean;
224 INLINED_R_EDGE_13__OLD : boolean;
```

```

225     INLINED_R_EDGE_14__NEW : boolean;
226     INLINED_R_EDGE_14__OLD : boolean;
227     INLINED_F_EDGE_15__NEW : boolean;
228     INLINED_F_EDGE_15__OLD : boolean;
229     INLINED_TIMER_PULSEON__PT : signed word[32];
230     INLINED_TIMER_PULSEON__IN : boolean;
231     INLINED_TIMER_PULSEON__Q : boolean;
232     INLINED_TIMER_PULSEON__ET : signed word[32];
233     INLINED_TIMER_PULSEON__OLD_IN : boolean;
234     INLINED_TIMER_PULSEON__DUE : signed word[32];
235     INLINED_TIMER_PULSEOFF__PT : signed word[32];
236     INLINED_TIMER_PULSEOFF__IN : boolean;
237     INLINED_TIMER_PULSEOFF__Q : boolean;
238     INLINED_TIMER_PULSEOFF__ET : signed word[32];
239     INLINED_TIMER_PULSEOFF__OLD_IN : boolean;
240     INLINED_TIMER_PULSEOFF__DUE : signed word[32];
241     INLINED_R_EDGE_20__NEW : boolean;
242     INLINED_R_EDGE_20__OLD : boolean;
243     INLINED_F_EDGE_21__NEW : boolean;
244     INLINED_F_EDGE_21__OLD : boolean;
245     INLINED_R_EDGE_22__NEW : boolean;
246     INLINED_R_EDGE_22__OLD : boolean;
247     INLINED_R_EDGE_25__NEW : boolean;
248     INLINED_R_EDGE_25__OLD : boolean;
249     INLINED_F_EDGE_26__NEW : boolean;
250     INLINED_F_EDGE_26__OLD : boolean;
251     INLINED_TIMER_WARNING__PT : signed word[32];
252     INLINED_TIMER_WARNING__IN : boolean;
253     INLINED_TIMER_WARNING__Q : boolean;
254     INLINED_TIMER_WARNING__ET : signed word[32];
255     INLINED_TIMER_WARNING__RUNNING : boolean;
256     INLINED_TIMER_WARNING__START : signed word[32];
257     INLINED_DETECT_EDGE_28__NEW : boolean;
258     INLINED_DETECT_EDGE_28__OLD : boolean;
259     INLINED_DETECT_EDGE_28__RE : boolean;
260     INLINED_DETECT_EDGE_28__FE : boolean;
261
262     ASSIGN
263     init(loc) := initial;
264     next(loc) := case
265         loc = end : initial;
266         loc = initial : 10;
267         loc = 10 : 11;
268         loc = 11 : 12;
269         loc = 12 & (((INLINED_R_EDGE_1__NEW = TRUE) & (INLINED_R_EDGE_1__OLD
270             = FALSE))) : 13;
271         loc = 12 & (!( (((INLINED_R_EDGE_1__NEW = TRUE) & (
272             INLINED_R_EDGE_1__OLD = FALSE)))))) : 15;
273         loc = 13 : 14;
274         loc = 14 : 17;
275         loc = 15 : 16;
276         loc = 16 : 17;
277         loc = 17 : 18;
278         loc = 18 : 19;
279         loc = 19 : 110;
280         loc = 110 & (((INLINED_R_EDGE_2__NEW = TRUE) & (
281             INLINED_R_EDGE_2__OLD = FALSE))) : 111;
282         loc = 110 & (!( (((INLINED_R_EDGE_2__NEW = TRUE) & (

```

```

    INLINED_R_EDGE_2__OLD = FALSE)))) : 113;
280     loc = 111 : 112;
281     loc = 112 : 115;
282     loc = 113 : 114;
283     loc = 114 : 115;
284     loc = 115 : 116;
285     loc = 116 : 117;
286     loc = 117 : 118;
287     loc = 118 & (((INLINED_R_EDGE_3__NEW = TRUE) & (
    INLINED_R_EDGE_3__OLD = FALSE))) : 119;
288     loc = 118 & (!(((INLINED_R_EDGE_3__NEW = TRUE) & (
    INLINED_R_EDGE_3__OLD = FALSE)))) : 121;
289     loc = 119 : 120;
290     loc = 120 : 123;
291     loc = 121 : 122;
292     loc = 122 : 123;
293     loc = 123 : 124;
294     loc = 124 : 125;
295     loc = 125 : 126;
296     loc = 126 & (((INLINED_R_EDGE_4__NEW = TRUE) & (
    INLINED_R_EDGE_4__OLD = FALSE))) : 127;
297     loc = 126 & (!(((INLINED_R_EDGE_4__NEW = TRUE) & (
    INLINED_R_EDGE_4__OLD = FALSE)))) : 129;
298     loc = 127 : 128;
299     loc = 128 : 131;
300     loc = 129 : 130;
301     loc = 130 : 131;
302     loc = 131 : 132;
303     loc = 132 : 133;
304     loc = 133 : 134;
305     loc = 134 & (((INLINED_R_EDGE_5__NEW = TRUE) & (
    INLINED_R_EDGE_5__OLD = FALSE))) : 135;
306     loc = 134 & (!(((INLINED_R_EDGE_5__NEW = TRUE) & (
    INLINED_R_EDGE_5__OLD = FALSE)))) : 137;
307     loc = 135 : 136;
308     loc = 136 : 139;
309     loc = 137 : 138;
310     loc = 138 : 139;
311     loc = 139 : 140;
312     loc = 140 : 141;
313     loc = 141 : 142;
314     loc = 142 & (((INLINED_R_EDGE_6__NEW = TRUE) & (
    INLINED_R_EDGE_6__OLD = FALSE))) : 143;
315     loc = 142 & (!(((INLINED_R_EDGE_6__NEW = TRUE) & (
    INLINED_R_EDGE_6__OLD = FALSE)))) : 145;
316     loc = 143 : 144;
317     loc = 144 : 147;
318     loc = 145 : 146;
319     loc = 146 : 147;
320     loc = 147 : 148;
321     loc = 148 : 149;
322     loc = 149 : 150;
323     loc = 150 & (((INLINED_R_EDGE_7__NEW = TRUE) & (
    INLINED_R_EDGE_7__OLD = FALSE))) : 151;
324     loc = 150 & (!(((INLINED_R_EDGE_7__NEW = TRUE) & (
    INLINED_R_EDGE_7__OLD = FALSE)))) : 153;
325     loc = 151 : 152;
326     loc = 152 : 155;

```

```

327     loc = 153 : 154;
328     loc = 154 : 155;
329     loc = 155 : 156;
330     loc = 156 : 157;
331     loc = 157 : 158;
332     loc = 158 & (((INLINED_R_EDGE_8__NEW = TRUE) & (
        INLINED_R_EDGE_8__OLD = FALSE))) : 159;
333     loc = 158 & (!(((INLINED_R_EDGE_8__NEW = TRUE) & (
        INLINED_R_EDGE_8__OLD = FALSE)))) : 161;
334     loc = 159 : 160;
335     loc = 160 : 163;
336     loc = 161 : 162;
337     loc = 162 : 163;
338     loc = 163 : 164;
339     loc = 164 : 165;
340     loc = 165 : 166;
341     loc = 166 : 167;
342     loc = 167 : 168;
343     loc = 168 : 169;
344     loc = 169 : 170;
345     loc = 170 : 171;
346     loc = 171 : 172;
347     loc = 172 : 173;
348     loc = 173 : 174;
349     loc = 174 : 175;
350     loc = 175 : 176;
351     loc = 176 : 177;
352     loc = 177 : 178;
353     loc = 178 & (((INLINED_R_EDGE_9__NEW = TRUE) & (
        INLINED_R_EDGE_9__OLD = FALSE))) : 179;
354     loc = 178 & (!(((INLINED_R_EDGE_9__NEW = TRUE) & (
        INLINED_R_EDGE_9__OLD = FALSE)))) : 181;
355     loc = 179 : 180;
356     loc = 180 : 183;
357     loc = 181 : 182;
358     loc = 182 : 183;
359     loc = 183 : 184;
360     loc = 184 : 185;
361     loc = 185 : 186;
362     loc = 186 & (((INLINED_R_EDGE_10__NEW = TRUE) & (
        INLINED_R_EDGE_10__OLD = FALSE))) : 187;
363     loc = 186 & (!(((INLINED_R_EDGE_10__NEW = TRUE) & (
        INLINED_R_EDGE_10__OLD = FALSE)))) : 189;
364     loc = 187 : 188;
365     loc = 188 : 191;
366     loc = 189 : 190;
367     loc = 190 : 191;
368     loc = 191 : 192;
369     loc = 192 : 193;
370     loc = 193 : 194;
371     loc = 194 & (((INLINED_R_EDGE_11__NEW = TRUE) & (
        INLINED_R_EDGE_11__OLD = FALSE))) : 195;
372     loc = 194 & (!(((INLINED_R_EDGE_11__NEW = TRUE) & (
        INLINED_R_EDGE_11__OLD = FALSE)))) : 197;
373     loc = 195 : 196;
374     loc = 196 : 199;
375     loc = 197 : 198;
376     loc = 198 : 199;

```

```

377     loc = 199 : 1100;
378     loc = 1100 : 1101;
379     loc = 1101 : 1102;
380     loc = 1102 & (((INLINED_R_EDGE_12__NEW = TRUE) & (
        INLINED_R_EDGE_12__OLD = FALSE))) : 1103;
381     loc = 1102 & (!(((INLINED_R_EDGE_12__NEW = TRUE) & (
        INLINED_R_EDGE_12__OLD = FALSE)))) : 1105;
382     loc = 1103 : 1104;
383     loc = 1104 : 1107;
384     loc = 1105 : 1106;
385     loc = 1106 : 1107;
386     loc = 1107 : 1108;
387     loc = 1108 : 1109;
388     loc = 1109 : 1110;
389     loc = 1110 & (((INLINED_R_EDGE_13__NEW = TRUE) & (
        INLINED_R_EDGE_13__OLD = FALSE))) : 1111;
390     loc = 1110 & (!(((INLINED_R_EDGE_13__NEW = TRUE) & (
        INLINED_R_EDGE_13__OLD = FALSE)))) : 1113;
391     loc = 1111 : 1112;
392     loc = 1112 : 1115;
393     loc = 1113 : 1114;
394     loc = 1114 : 1115;
395     loc = 1115 : 1116;
396     loc = 1116 : 1117;
397     loc = 1117 : 1118;
398     loc = 1118 & (((INLINED_R_EDGE_14__NEW = TRUE) & (
        INLINED_R_EDGE_14__OLD = FALSE))) : 1119;
399     loc = 1118 & (!(((INLINED_R_EDGE_14__NEW = TRUE) & (
        INLINED_R_EDGE_14__OLD = FALSE)))) : 1121;
400     loc = 1119 : 1120;
401     loc = 1120 : 1123;
402     loc = 1121 : 1122;
403     loc = 1122 : 1123;
404     loc = 1123 : 1124;
405     loc = 1124 : 1125;
406     loc = 1125 : 1126;
407     loc = 1126 : 1127;
408     loc = 1127 & ((E_MALACKR | E_AUALACK)) : 1128;
409     loc = 1127 & (!((E_MALACKR | E_AUALACK))) : 1127_elsif_1;
410     loc = 1127_elsif_1 & (((E_TSTOPI | E_STARTI | E_FUSTOPI) | E_AL)
        : 1130;
411     loc = 1127_elsif_1 & (!(((E_TSTOPI | E_STARTI) | E_FUSTOPI) | E_AL)
        )) : 1131;
412     loc = 1128 : 1129;
413     loc = 1129 : 1131;
414     loc = 1130 : 1131;
415     loc = 1131 & (((PENRSTART & (E_MENRSTARTR | AURSTART)) & !(
        FUSTOPIST)) | (((PENRSTART & PRSTARTFS) & (E_MENRSTARTR |
        AURSTART)) & !(FULLNOTACKNOWLEDGED)))) : 1132;
416     loc = 1131 & (!(((PENRSTART & (E_MENRSTARTR | AURSTART)) & !(
        FUSTOPIST)) | (((PENRSTART & PRSTARTFS) & (E_MENRSTARTR |
        AURSTART)) & !(FULLNOTACKNOWLEDGED)))) : 1133;
417     loc = 1132 : 1133;
418     loc = 1133 & (E_FUSTOPI) : 1134;
419     loc = 1133 & !(E_FUSTOPI) : 1137;
420     loc = 1134 : 1135;
421     loc = 1135 & (PENRSTART) : 1136;
422     loc = 1135 & !(PENRSTART) : 1137;

```

```

423     loc = 1136 : 1137;
424     loc = 1137 : 1138;
425     loc = 1138 : 1139;
426     loc = 1139 : 1140;
427     loc = 1140 & (((INLINED_F_EDGE_15__NEW = FALSE) & (
INLINED_F_EDGE_15__OLD = TRUE))) : 1141;
428     loc = 1140 & (!(((INLINED_F_EDGE_15__NEW = FALSE) & (
INLINED_F_EDGE_15__OLD = TRUE)))) : 1143;
429     loc = 1141 : 1142;
430     loc = 1142 : 1145;
431     loc = 1143 : 1144;
432     loc = 1144 : 1145;
433     loc = 1145 : 1146;
434     loc = 1146 & (!(HLD & PHLD))) : 1147;
435     loc = 1146 & (!(HLD & PHLD))) : 1172;
436     loc = 1147 & (((((AUMOST_AUX | MMOST_AUX) | SOFTLDST_AUX) & E_MFOMOR
) & !(AUIHFOMO))) : 1148;
437     loc = 1147 & (!((((AUMOST_AUX | MMOST_AUX) | SOFTLDST_AUX) &
E_MFOMOR) & !(AUIHFOMO)))) : 1152;
438     loc = 1148 : 1149;
439     loc = 1149 : 1150;
440     loc = 1150 : 1151;
441     loc = 1151 : 1152;
442     loc = 1152 & (((((AUMOST_AUX | FOMOST_AUX) | SOFTLDST_AUX) & E_MMMOR
) & !(AUIHMMO))) : 1153;
443     loc = 1152 & (!((((AUMOST_AUX | FOMOST_AUX) | SOFTLDST_AUX) &
E_MMMOR) & !(AUIHMMO)))) : 1157;
444     loc = 1153 : 1154;
445     loc = 1154 : 1155;
446     loc = 1155 : 1156;
447     loc = 1156 : 1157;
448     loc = 1157 & (((((((MMOST_AUX & (E_MAUMOR | E_AUAUMOR)) | (
FOMOST_AUX & E_MAUMOR)) | (SOFTLDST_AUX & E_MAUMOR)) | (
MMOST_AUX & AUIHMMO)) | (FOMOST_AUX & AUIHFOMO)) | (
SOFTLDST_AUX & AUIHFOMO)) | (!(AUMOST_AUX | MMOST_AUX) |
FOMOST_AUX) | SOFTLDST_AUX)))) : 1158;
449     loc = 1157 & (!(((((((((MMOST_AUX & (E_MAUMOR | E_AUAUMOR)) | (
FOMOST_AUX & E_MAUMOR)) | (SOFTLDST_AUX & E_MAUMOR)) | (
MMOST_AUX & AUIHMMO)) | (FOMOST_AUX & AUIHFOMO)) | (
SOFTLDST_AUX & AUIHFOMO)) | (!(AUMOST_AUX | MMOST_AUX) |
FOMOST_AUX) | SOFTLDST_AUX)))))) : 1162;
450     loc = 1158 : 1159;
451     loc = 1159 : 1160;
452     loc = 1160 : 1161;
453     loc = 1161 : 1162;
454     loc = 1162 & (((AUMOST_AUX | MMOST_AUX) & E_MSOFTLDR) & !(AUIHFOMO
)) : 1163;
455     loc = 1162 & (!((((AUMOST_AUX | MMOST_AUX) & E_MSOFTLDR) & !(
AUIHFOMO)))) : 1167;
456     loc = 1163 : 1164;
457     loc = 1164 : 1165;
458     loc = 1165 : 1166;
459     loc = 1166 : 1167;
460     loc = 1167 : 1168;
461     loc = 1168 : 1169;
462     loc = 1169 : 1170;
463     loc = 1170 : 1171;
464     loc = 1171 : 1177;

```

```

465     loc = 1172 : 1173;
466     loc = 1173 : 1174;
467     loc = 1174 : 1175;
468     loc = 1175 : 1176;
469     loc = 1176 : 1177;
470     loc = 1177 : 1178;
471     loc = 1178 : 1179;
472     loc = 1179 & (AUOFFR) : 1180;
473     loc = 1179 & (!(AUOFFR)) : 1179_elsif_1;
474     loc = 1179_elsif_1 & (AUONR) : 1181;
475     loc = 1179_elsif_1 & (!(AUONR)) : 1179_elsif_2;
476     loc = 1179_elsif_2 & (((FULLNOTACKNOWLEDGED | FUSTOPIST) | !(
477         ENRSTARTST))) : 1182;
478     loc = 1179_elsif_2 & (!(((FULLNOTACKNOWLEDGED | FUSTOPIST) | !(
479         ENRSTARTST)))) : 1183;
480     loc = 1180 : 1183;
481     loc = 1181 : 1183;
482     loc = 1182 : 1183;
483     loc = 1183 : 1184;
484     loc = 1184 & ((((((E_MOFFR & ((MMOST | FOMOST) | SOFTLDST)) | (
485         AUOFFRST & AUMOST)) | ((LDST & PHLD CMD) & HOFFRST)) | (((
486         FE_PULSEON & PPULSE) & !(POUTOFF)) & ENRSTARTST)) | (E_FUSTOPI
487         & !(PFSPON)))))) : 1185;
488     loc = 1184 & (!(((((((E_MOFFR & ((MMOST | FOMOST) | SOFTLDST)) | (
489         AUOFFRST & AUMOST)) | ((LDST & PHLD CMD) & HOFFRST)) | (((
490         FE_PULSEON & PPULSE) & !(POUTOFF)) & ENRSTARTST)) | (E_FUSTOPI
491         & !(PFSPON)))))) : 1184_elsif_1;
492     loc = 1184_elsif_1 & ((((((E_MONR & ((MMOST | FOMOST) | SOFTLDST)) |
493         (AUONRST & AUMOST)) | ((LDST & PHLD CMD) & HONRST) & ENRSTARTST
494         )) | (E_FUSTOPI & PFSPON))) : 1186;
495     loc = 1184_elsif_1 & (!(((((((E_MONR & ((MMOST | FOMOST) | SOFTLDST))
496         | (AUONRST & AUMOST)) | ((LDST & PHLD CMD) & HONRST) &
497         ENRSTARTST)) | (E_FUSTOPI & PFSPON)))) : 1187;
498     loc = 1185 : 1187;
499     loc = 1186 : 1187;
500     loc = 1187 : 1188;
501     loc = 1188 & (HOFFR) : 1189;
502     loc = 1188 & !(HOFFR) : 1190;
503     loc = 1189 : 1192;
504     loc = 1190 & (HONR) : 1191;
505     loc = 1190 & !(HONR) : 1192;
506     loc = 1191 : 1192;
507     loc = 1192 : 1193;
508     loc = 1193 & (PPULSE) : 1194;
509     loc = 1193 & !(PPULSE) : 1310;
510     loc = 1194 & (INTERLOCKR) : 1195;
511     loc = 1194 & !(INTERLOCKR) : 1194_elsif_1;
512     loc = 1194_elsif_1 & (FE_INTERLOCKR) : 1197;
513     loc = 1194_elsif_1 & !(FE_INTERLOCKR) : 1194_elsif_2;
514     loc = 1194_elsif_2 & (((MOFFRST & ((MMOST | FOMOST) | SOFTLDST)) |
515         (AUOFFRST & AUMOST)) | ((HOFFR & LDST) & PHLD CMD)) : 1223;
516     loc = 1194_elsif_2 & (!((((MOFFRST & ((MMOST | FOMOST) | SOFTLDST))
517         | (AUOFFRST & AUMOST)) | ((HOFFR & LDST) & PHLD CMD)))) :
518         1194_elsif_3;
519     loc = 1194_elsif_3 & (((MONRST & ((MMOST | FOMOST) | SOFTLDST)) | (
520         AUONRST & AUMOST)) | ((HONR & LDST) & PHLD CMD)) : 1225;
521     loc = 1194_elsif_3 & (!((((MONRST & ((MMOST | FOMOST) | SOFTLDST)) |
522         (AUONRST & AUMOST)) | ((HONR & LDST) & PHLD CMD)))) : 1227;

```

```

506     loc = 1195 : 1196;
507     loc = 1196 : 1229;
508     loc = 1197 : 1198;
509     loc = 1198 : 1199;
510     loc = 1199 : 1200;
511     loc = 1200 : 1201;
512     loc = 1201 & (((INLINED_TIMER_PULSEON__IN & !(
        INLINED_TIMER_PULSEON__OLD_IN)) & !(INLINED_TIMER_PULSEON__Q)))
        : 1202;
513     loc = 1201 & (!(((INLINED_TIMER_PULSEON__IN & !(
        INLINED_TIMER_PULSEON__OLD_IN)) & !(INLINED_TIMER_PULSEON__Q)))
        ) : 1203;
514     loc = 1202 : 1203;
515     loc = 1203 & ((main.__GLOBAL_TIME <= INLINED_TIMER_PULSEON__DUE)) :
        1204;
516     loc = 1203 & (!(main.__GLOBAL_TIME <= INLINED_TIMER_PULSEON__DUE))
        : 1206;
517     loc = 1204 : 1205;
518     loc = 1205 : 1210;
519     loc = 1206 : 1207;
520     loc = 1207 & (INLINED_TIMER_PULSEON__IN) : 1208;
521     loc = 1207 & !(INLINED_TIMER_PULSEON__IN) : 1209;
522     loc = 1208 : 1210;
523     loc = 1209 : 1210;
524     loc = 1210 : 1211;
525     loc = 1211 : 1212;
526     loc = 1212 : 1213;
527     loc = 1213 & (((INLINED_TIMER_PULSEOFF__IN & !(
        INLINED_TIMER_PULSEOFF__OLD_IN)) & !(INLINED_TIMER_PULSEOFF__Q)
        )) : 1214;
528     loc = 1213 & (!(((INLINED_TIMER_PULSEOFF__IN & !(
        INLINED_TIMER_PULSEOFF__OLD_IN)) & !(INLINED_TIMER_PULSEOFF__Q)
        )))) : 1215;
529     loc = 1214 : 1215;
530     loc = 1215 & ((main.__GLOBAL_TIME <= INLINED_TIMER_PULSEOFF__DUE)) :
        1216;
531     loc = 1215 & (!(main.__GLOBAL_TIME <= INLINED_TIMER_PULSEOFF__DUE))
        ) : 1218;
532     loc = 1216 : 1217;
533     loc = 1217 : 1222;
534     loc = 1218 : 1219;
535     loc = 1219 & (INLINED_TIMER_PULSEOFF__IN) : 1220;
536     loc = 1219 & !(INLINED_TIMER_PULSEOFF__IN) : 1221;
537     loc = 1220 : 1222;
538     loc = 1221 : 1222;
539     loc = 1222 : 1229;
540     loc = 1223 : 1224;
541     loc = 1224 : 1229;
542     loc = 1225 : 1226;
543     loc = 1226 : 1229;
544     loc = 1227 : 1228;
545     loc = 1228 : 1229;
546     loc = 1229 : 1230;
547     loc = 1230 : 1231;
548     loc = 1231 & (((INLINED_TIMER_PULSEON__IN & !(
        INLINED_TIMER_PULSEON__OLD_IN)) & !(INLINED_TIMER_PULSEON__Q)))
        : 1232;
549     loc = 1231 & (!(((INLINED_TIMER_PULSEON__IN & !(

```

```

        INLINED_TIMER_PULSEON__OLD_IN)) & !(INLINED_TIMER_PULSEON__Q)))
    ) : 1233;
550     loc = 1232 : 1233;
551     loc = 1233 & ((main.__GLOBAL_TIME <= INLINED_TIMER_PULSEON__DUE)) :
        1234;
552     loc = 1233 & (!(main.__GLOBAL_TIME <= INLINED_TIMER_PULSEON__DUE)))
        : 1236;
553     loc = 1234 : 1235;
554     loc = 1235 : 1240;
555     loc = 1236 : 1237;
556     loc = 1237 & (INLINED_TIMER_PULSEON__IN) : 1238;
557     loc = 1237 & !(INLINED_TIMER_PULSEON__IN)) : 1239;
558     loc = 1238 : 1240;
559     loc = 1239 : 1240;
560     loc = 1240 : 1241;
561     loc = 1241 : 1242;
562     loc = 1242 : 1243;
563     loc = 1243 : 1244;
564     loc = 1244 & (((INLINED_TIMER_PULSEOFF__IN & !(
        INLINED_TIMER_PULSEOFF__OLD_IN)) & !(INLINED_TIMER_PULSEOFF__Q)
        )) : 1245;
565     loc = 1244 & (!(((INLINED_TIMER_PULSEOFF__IN & !(
        INLINED_TIMER_PULSEOFF__OLD_IN)) & !(INLINED_TIMER_PULSEOFF__Q)
        ))) : 1246;
566     loc = 1245 : 1246;
567     loc = 1246 & ((main.__GLOBAL_TIME <= INLINED_TIMER_PULSEOFF__DUE)) :
        1247;
568     loc = 1246 & (!(main.__GLOBAL_TIME <= INLINED_TIMER_PULSEOFF__DUE))
        ) : 1249;
569     loc = 1247 : 1248;
570     loc = 1248 : 1253;
571     loc = 1249 : 1250;
572     loc = 1250 & (INLINED_TIMER_PULSEOFF__IN) : 1251;
573     loc = 1250 & !(INLINED_TIMER_PULSEOFF__IN)) : 1252;
574     loc = 1251 : 1253;
575     loc = 1252 : 1253;
576     loc = 1253 : 1254;
577     loc = 1254 : 1255;
578     loc = 1255 : 1256;
579     loc = 1256 : 1257;
580     loc = 1257 & (((INLINED_R_EDGE_20__NEW = TRUE) & (
        INLINED_R_EDGE_20__OLD = FALSE))) : 1258;
581     loc = 1257 & (!(((INLINED_R_EDGE_20__NEW = TRUE) & (
        INLINED_R_EDGE_20__OLD = FALSE)))) : 1260;
582     loc = 1258 : 1259;
583     loc = 1259 : 1262;
584     loc = 1260 : 1261;
585     loc = 1261 : 1262;
586     loc = 1262 : 1263;
587     loc = 1263 : 1264;
588     loc = 1264 : 1265;
589     loc = 1265 & (((INLINED_F_EDGE_21__NEW = FALSE) & (
        INLINED_F_EDGE_21__OLD = TRUE))) : 1266;
590     loc = 1265 & (!(((INLINED_F_EDGE_21__NEW = FALSE) & (
        INLINED_F_EDGE_21__OLD = TRUE)))) : 1268;
591     loc = 1266 : 1267;
592     loc = 1267 : 1270;
593     loc = 1268 : 1269;

```

```

594     loc = 1269 : 1270;
595     loc = 1270 : 1271;
596     loc = 1271 : 1272;
597     loc = 1272 : 1273;
598     loc = 1273 & (((INLINED_R_EDGE_22__NEW = TRUE) & (
        INLINED_R_EDGE_22__OLD = FALSE))) : 1274;
599     loc = 1273 & (!(((INLINED_R_EDGE_22__NEW = TRUE) & (
        INLINED_R_EDGE_22__OLD = FALSE)))) : 1276;
600     loc = 1274 : 1275;
601     loc = 1275 : 1278;
602     loc = 1276 : 1277;
603     loc = 1277 : 1278;
604     loc = 1278 : 1279;
605     loc = 1279 & (RE_PULSEON) : 1280;
606     loc = 1279 & (!(RE_PULSEON)) : 1292;
607     loc = 1280 : 1281;
608     loc = 1281 : 1282;
609     loc = 1282 & (((INLINED_TIMER_PULSEOFF__IN & !(
        INLINED_TIMER_PULSEOFF__OLD_IN)) & !(INLINED_TIMER_PULSEOFF__Q)
        )) : 1283;
610     loc = 1282 & (!(((INLINED_TIMER_PULSEOFF__IN & !(
        INLINED_TIMER_PULSEOFF__OLD_IN)) & !(INLINED_TIMER_PULSEOFF__Q)
        ))) : 1284;
611     loc = 1283 : 1284;
612     loc = 1284 & ((main.__GLOBAL_TIME <= INLINED_TIMER_PULSEOFF__DUE)) :
        1285;
613     loc = 1284 & (!((main.__GLOBAL_TIME <= INLINED_TIMER_PULSEOFF__DUE))
        ) : 1287;
614     loc = 1285 : 1286;
615     loc = 1286 : 1291;
616     loc = 1287 : 1288;
617     loc = 1288 & (INLINED_TIMER_PULSEOFF__IN) : 1289;
618     loc = 1288 & (!(INLINED_TIMER_PULSEOFF__IN)) : 1290;
619     loc = 1289 : 1291;
620     loc = 1290 : 1291;
621     loc = 1291 : 1292;
622     loc = 1292 & (RE_PULSEOFF) : 1293;
623     loc = 1292 & (!(RE_PULSEOFF)) : 1305;
624     loc = 1293 : 1294;
625     loc = 1294 : 1295;
626     loc = 1295 & (((INLINED_TIMER_PULSEON__IN & !(
        INLINED_TIMER_PULSEON__OLD_IN)) & !(INLINED_TIMER_PULSEON__Q)))
        : 1296;
627     loc = 1295 & (!(((INLINED_TIMER_PULSEON__IN & !(
        INLINED_TIMER_PULSEON__OLD_IN)) & !(INLINED_TIMER_PULSEON__Q)))
        ) : 1297;
628     loc = 1296 : 1297;
629     loc = 1297 & ((main.__GLOBAL_TIME <= INLINED_TIMER_PULSEON__DUE)) :
        1298;
630     loc = 1297 & (!((main.__GLOBAL_TIME <= INLINED_TIMER_PULSEON__DUE)))
        : 1300;
631     loc = 1298 : 1299;
632     loc = 1299 : 1304;
633     loc = 1300 : 1301;
634     loc = 1301 & (INLINED_TIMER_PULSEON__IN) : 1302;
635     loc = 1301 & (!(INLINED_TIMER_PULSEON__IN)) : 1303;
636     loc = 1302 : 1304;
637     loc = 1303 : 1304;

```

```

638     loc = 1304 : 1305;
639     loc = 1305 & (PPULSECSTE) : 1306;
640     loc = 1305 & (!(PPULSECSTE)) : 1308;
641     loc = 1306 : 1307;
642     loc = 1307 : 1310;
643     loc = 1308 : 1309;
644     loc = 1309 : 1310;
645     loc = 1310 : 1311;
646     loc = 1311 & (POUTOFF) : 1312;
647     loc = 1311 & (!(POUTOFF)) : 1313;
648     loc = 1312 : 1313;
649     loc = 1313 & (POUTOFF) : 1314;
650     loc = 1313 & (!(POUTOFF)) : 1323;
651     loc = 1314 & (INTERLOCKR) : 1315;
652     loc = 1314 & (!(INTERLOCKR)) : 1325;
653     loc = 1315 & ((PPULSE & !(PFSPSON2))) : 1316;
654     loc = 1315 & (!(PPULSE & !(PFSPSON2))) : 1321;
655     loc = 1316 & (PFSPSON) : 1317;
656     loc = 1316 & (!(PFSPSON)) : 1319;
657     loc = 1317 : 1318;
658     loc = 1318 : 1325;
659     loc = 1319 : 1320;
660     loc = 1320 : 1325;
661     loc = 1321 : 1322;
662     loc = 1322 : 1325;
663     loc = 1323 & (INTERLOCKR) : 1324;
664     loc = 1323 & (!(INTERLOCKR)) : 1325;
665     loc = 1324 : 1325;
666     loc = 1325 : 1326;
667     loc = 1326 : 1327;
668     loc = 1327 : 1328;
669     loc = 1328 : 1329;
670     loc = 1329 : 1330;
671     loc = 1330 & (!(POUTOFF)) : 1331;
672     loc = 1330 & (!(!(POUTOFF))) : 1334;
673     loc = 1331 & (PFSPSON) : 1332;
674     loc = 1331 & (!(PFSPSON)) : 1333;
675     loc = 1332 : 1336;
676     loc = 1333 : 1336;
677     loc = 1334 : 1335;
678     loc = 1335 : 1336;
679     loc = 1336 & ((OUTONOVST | (PPULSE & PULSEONR))) : 1337;
680     loc = 1336 & (!(OUTONOVST | (PPULSE & PULSEONR))) : 1338;
681     loc = 1337 : 1338;
682     loc = 1338 & (((OUTOFFOVST & POUTOFF) | (!(OUTONOVST) & !(POUTOFF))
        ) | (PPULSE & PULSEOFFR)) : 1339;
683     loc = 1338 & (!((((OUTOFFOVST & POUTOFF) | (!(OUTONOVST) & !(POUTOFF)
        ))) | (PPULSE & PULSEOFFR))) : 1340;
684     loc = 1339 : 1340;
685     loc = 1340 : 1341;
686     loc = 1341 : 1342;
687     loc = 1342 & (((INLINED_R_EDGE_25__NEW = TRUE) & (
        INLINED_R_EDGE_25__OLD = FALSE))) : 1343;
688     loc = 1342 & (!((((INLINED_R_EDGE_25__NEW = TRUE) & (
        INLINED_R_EDGE_25__OLD = FALSE)))))) : 1345;
689     loc = 1343 : 1344;
690     loc = 1344 : 1347;
691     loc = 1345 : 1346;

```

```

692     loc = 1346 : 1347;
693     loc = 1347 : 1348;
694     loc = 1348 : 1349;
695     loc = 1349 : 1350;
696     loc = 1350 & (((INLINED_F_EDGE_26__NEW = FALSE) & (
        INLINED_F_EDGE_26__OLD = TRUE))) : 1351;
697     loc = 1350 & (!(((INLINED_F_EDGE_26__NEW = FALSE) & (
        INLINED_F_EDGE_26__OLD = TRUE)))) : 1353;
698     loc = 1351 : 1352;
699     loc = 1352 : 1355;
700     loc = 1353 : 1354;
701     loc = 1354 : 1355;
702     loc = 1355 : 1356;
703     loc = 1356 & (((((OUTOVST_AUX & ((PHFON & !(ONST)) | (PHFOFF & OFFST
        ))) | (!(OUTOVST_AUX) & ((PHFOFF & !(OFFST)) | (PHFON & ONST)))
        ) | (OFFST & ONST)) & !(PPULSE) | ((POUTOFF & PPULSE) & !(
        OUTONOV)) & !(OUTOFFOV)))) : 1357;
704     loc = 1356 & (!((((OUTOVST_AUX & ((PHFON & !(ONST)) | (PHFOFF &
        OFFST))) | (!(OUTOVST_AUX) & ((PHFOFF & !(OFFST)) | (PHFON &
        ONST)))) | (OFFST & ONST)) & !(PPULSE) | ((POUTOFF & PPULSE)
        & !(OUTONOV)) & !(OUTOFFOV)))) : 1358;
705     loc = 1357 : 1358;
706     loc = 1358 & ((((!(((OUTOVST_AUX & ((PHFON & !(ONST)) | (PHFOFF &
        OFFST))) | (!(OUTOVST_AUX) & ((PHFOFF & !(OFFST)) | (PHFON &
        ONST)))) | (OFFST & ONST))) | RE_OUTOVST_AUX | FE_OUTOVST_AUX)
        | ((PPULSE & POUTOFF) & OUTONOV)) | ((PPULSE & POUTOFF) &
        OUTOFFOV)) : 1359;
707     loc = 1358 & (!(((((!(((OUTOVST_AUX & ((PHFON & !(ONST)) | (PHFOFF
        & OFFST))) | (!(OUTOVST_AUX) & ((PHFOFF & !(OFFST)) | (PHFON &
        ONST)))) | (OFFST & ONST))) | RE_OUTOVST_AUX | FE_OUTOVST_AUX)
        | ((PPULSE & POUTOFF) & OUTONOV)) | ((PPULSE & POUTOFF) &
        OUTOFFOV)))) : 1360;
708     loc = 1359 : 1360;
709     loc = 1360 : 1361;
710     loc = 1361 : 1362;
711     loc = 1362 & ((INLINED_TIMER_WARNING__IN = FALSE)) : 1363;
712     loc = 1362 & (!(INLINED_TIMER_WARNING__IN = FALSE)) : 1366;
713     loc = 1363 : 1364;
714     loc = 1364 : 1365;
715     loc = 1365 : 1375;
716     loc = 1366 & ((INLINED_TIMER_WARNING__RUNNING = FALSE)) : 1367;
717     loc = 1366 & (!(INLINED_TIMER_WARNING__RUNNING = FALSE)) : 1370;
718     loc = 1367 : 1368;
719     loc = 1368 : 1369;
720     loc = 1369 : 1375;
721     loc = 1370 & (!(((main.__GLOBAL_TIME - (INLINED_TIMER_WARNING__START
        + INLINED_TIMER_WARNING__PT)) >= 0sd32_0))) : 1371;
722     loc = 1370 & (!((!((main.__GLOBAL_TIME - (
        INLINED_TIMER_WARNING__START + INLINED_TIMER_WARNING__PT)) >= 0
        sd32_0)))) : 1373;
723     loc = 1371 & (!(INLINED_TIMER_WARNING__Q)) : 1372;
724     loc = 1371 & (!(INLINED_TIMER_WARNING__Q)) : 1375;
725     loc = 1372 : 1375;
726     loc = 1373 : 1374;
727     loc = 1374 : 1375;
728     loc = 1375 : 1376;
729     loc = 1376 : 1377;
730     loc = 1377 : 1378;

```

```

731     loc = 1378 : 1379;
732     loc = 1379 & ((FUSTOPIST | (FSIINC > 0sd16_0))) : 1380;
733     loc = 1379 & (!(FUSTOPIST | (FSIINC > 0sd16_0))) : 1382;
734     loc = 1380 : 1381;
735     loc = 1381 : 1382;
736     loc = 1382 & (((extend(FSIINC, 16) > PULSEWIDTH) | (!(FUSTOPIST) & (
       FSIINC = 0sd16_0)))) : 1383;
737     loc = 1382 & (!(((extend(FSIINC, 16) > PULSEWIDTH) | (!(FUSTOPIST) &
       (FSIINC = 0sd16_0)))))) : 1385;
738     loc = 1383 : 1384;
739     loc = 1384 : 1385;
740     loc = 1385 & ((TSTOPIST | (TSIINC > 0sd16_0))) : 1386;
741     loc = 1385 & (!(TSTOPIST | (TSIINC > 0sd16_0))) : 1388;
742     loc = 1386 : 1387;
743     loc = 1387 : 1388;
744     loc = 1388 & (((extend(TSIINC, 16) > PULSEWIDTH) | (!(TSTOPIST) & (
       TSIINC = 0sd16_0)))) : 1389;
745     loc = 1388 & (!(((extend(TSIINC, 16) > PULSEWIDTH) | (!(TSTOPIST) &
       (TSIINC = 0sd16_0)))))) : 1391;
746     loc = 1389 : 1390;
747     loc = 1390 : 1391;
748     loc = 1391 & ((STARTIST | (SIINC > 0sd16_0))) : 1392;
749     loc = 1391 & (!(STARTIST | (SIINC > 0sd16_0))) : 1394;
750     loc = 1392 : 1393;
751     loc = 1393 : 1394;
752     loc = 1394 & (((extend(SIINC, 16) > PULSEWIDTH) | (!(STARTIST) & (
       SIINC = 0sd16_0)))) : 1395;
753     loc = 1394 & (!(((extend(SIINC, 16) > PULSEWIDTH) | (!(STARTIST) & (
       SIINC = 0sd16_0)))))) : 1397;
754     loc = 1395 : 1396;
755     loc = 1396 : 1397;
756     loc = 1397 & ((ALST | (ALINC > 0sd16_0))) : 1398;
757     loc = 1397 & (!(ALST | (ALINC > 0sd16_0))) : 1400;
758     loc = 1398 : 1399;
759     loc = 1399 : 1400;
760     loc = 1400 & (((extend(ALINC, 16) > PULSEWIDTH) | (!(ALST) & (ALINC
       = 0sd16_0)))) : 1401;
761     loc = 1400 & (!(((extend(ALINC, 16) > PULSEWIDTH) | (!(ALST) & (
       ALINC = 0sd16_0)))))) : 1403;
762     loc = 1401 : 1402;
763     loc = 1402 : 1403;
764     loc = 1403 : 1404;
765     loc = 1404 : 1405;
766     loc = 1405 : 1406;
767     loc = 1406 : 1407;
768     loc = 1407 : 1408;
769     loc = 1408 : 1409;
770     loc = 1409 : 1410;
771     loc = 1410 : 1411;
772     loc = 1411 : 1412;
773     loc = 1412 : 1413;
774     loc = 1413 : 1414;
775     loc = 1414 : 1415;
776     loc = 1415 : 1416;
777     loc = 1416 : 1417;
778     loc = 1417 : 1418;
779     loc = 1418 : 1419;
780     loc = 1419 : 1420;

```

```

781     loc = 1420 : 1421;
782     loc = 1421 : 1422;
783     loc = 1422 : 1423;
784     loc = 1423 : 1424;
785     loc = 1424 : 1425;
786     loc = 1425 : 1426;
787     loc = 1426 : 1427;
788     loc = 1427 : 1428;
789     loc = 1428 : 1429;
790     loc = 1429 : 1430;
791     loc = 1430 : 1431;
792     loc = 1431 : 1432;
793     loc = 1432 : 1433;
794     loc = 1433 : 1434;
795     loc = 1434 : 1435;
796     loc = 1435 : 1436;
797     loc = 1436 : 1437;
798     loc = 1437 : 1438;
799     loc = 1438 : 1439;
800     loc = 1439 & ((INLINED_DETECT_EDGE_28__NEW !=
801         INLINED_DETECT_EDGE_28__OLD)) : 1440;
802     loc = 1439 & (!(INLINED_DETECT_EDGE_28__NEW !=
803         INLINED_DETECT_EDGE_28__OLD)) : 1446;
804     loc = 1440 & ((INLINED_DETECT_EDGE_28__NEW = TRUE)) : 1441;
805     loc = 1440 & (!(INLINED_DETECT_EDGE_28__NEW = TRUE)) : 1443;
806     loc = 1441 : 1442;
807     loc = 1442 : 1445;
808     loc = 1443 : 1444;
809     loc = 1444 : 1445;
810     loc = 1445 : 1448;
811     loc = 1446 : 1447;
812     loc = 1447 : 1448;
813     loc = 1448 : 1449;
814     loc = 1449 : 1450;
815     loc = 1450 : end;
816     TRUE: loc;
817     esac;
818
819     next(HFON) := case
820         loc = initial : {TRUE, FALSE};
821         TRUE : HFON;
822     esac;
823
824     next(HFOFF) := case
825         loc = initial : {TRUE, FALSE};
826         TRUE : HFOFF;
827     esac;
828
829     next(HLD) := case
830         loc = initial : {TRUE, FALSE};
831         TRUE : HLD;
832     esac;
833
834     next(IOERROR) := case
835         loc = initial : {TRUE, FALSE};
836         TRUE : IOERROR;
837     esac;
838
839     next(IOSIMU) := case
840         loc = initial : {TRUE, FALSE};
841         TRUE : IOSIMU;
842     esac;

```

```
837 next(ALB) := case
838     loc = initial : {TRUE, FALSE};
839     TRUE : ALB;
840 esac;
841 next(MANREG01_0__) := case
842     loc = initial : {TRUE, FALSE};
843     TRUE : MANREG01_0__;
844 esac;
845 next(MANREG01_1__) := case
846     loc = initial : {TRUE, FALSE};
847     TRUE : MANREG01_1__;
848 esac;
849 next(MANREG01_2__) := case
850     loc = initial : {TRUE, FALSE};
851     TRUE : MANREG01_2__;
852 esac;
853 next(MANREG01_3__) := case
854     loc = initial : {TRUE, FALSE};
855     TRUE : MANREG01_3__;
856 esac;
857 next(MANREG01_4__) := case
858     loc = initial : {TRUE, FALSE};
859     TRUE : MANREG01_4__;
860 esac;
861 next(MANREG01_5__) := case
862     loc = initial : {TRUE, FALSE};
863     TRUE : MANREG01_5__;
864 esac;
865 next(MANREG01_6__) := case
866     loc = initial : {TRUE, FALSE};
867     TRUE : MANREG01_6__;
868 esac;
869 next(MANREG01_7__) := case
870     loc = initial : {TRUE, FALSE};
871     TRUE : MANREG01_7__;
872 esac;
873 next(MANREG01_8__) := case
874     loc = initial : {TRUE, FALSE};
875     TRUE : MANREG01_8__;
876 esac;
877 next(MANREG01_9__) := case
878     loc = initial : {TRUE, FALSE};
879     TRUE : MANREG01_9__;
880 esac;
881 next(MANREG01_10__) := case
882     loc = initial : {TRUE, FALSE};
883     TRUE : MANREG01_10__;
884 esac;
885 next(MANREG01_11__) := case
886     loc = initial : {TRUE, FALSE};
887     TRUE : MANREG01_11__;
888 esac;
889 next(MANREG01_12__) := case
890     loc = initial : {TRUE, FALSE};
891     TRUE : MANREG01_12__;
892 esac;
893 next(MANREG01_13__) := case
894     loc = initial : {TRUE, FALSE};
```

```

895     TRUE : MANREG01_13__;
896   esac;
897   next(MANREG01_14__) := case
898     loc = initial : {TRUE, FALSE};
899     TRUE : MANREG01_14__;
900   esac;
901   next(MANREG01_15__) := case
902     loc = initial : {TRUE, FALSE};
903     TRUE : MANREG01_15__;
904   esac;
905   next(HONR) := case
906     loc = initial : {TRUE, FALSE};
907     TRUE : HONR;
908   esac;
909   next(HOFFR) := case
910     loc = initial : {TRUE, FALSE};
911     TRUE : HOFFR;
912   esac;
913   next(STARTI) := case
914     loc = initial : {TRUE, FALSE};
915     TRUE : STARTI;
916   esac;
917   next(TSTOPI) := case
918     loc = initial : {TRUE, FALSE};
919     TRUE : TSTOPI;
920   esac;
921   next(FUSTOPI) := case
922     loc = initial : {TRUE, FALSE};
923     TRUE : FUSTOPI;
924   esac;
925   next(AL) := case
926     loc = initial : {TRUE, FALSE};
927     TRUE : AL;
928   esac;
929   next(AUONR) := case
930     loc = initial : {TRUE, FALSE};
931     TRUE : AUONR;
932   esac;
933   next(AUOFFR) := case
934     loc = initial : {TRUE, FALSE};
935     TRUE : AUOFFR;
936   esac;
937   next(AUAUMOR) := case
938     loc = initial : {TRUE, FALSE};
939     TRUE : AUAUMOR;
940   esac;
941   next(AUIHMMO) := case
942     loc = initial : {TRUE, FALSE};
943     TRUE : AUIHMMO;
944   esac;
945   next(AUIHFOMO) := case
946     loc = initial : {TRUE, FALSE};
947     TRUE : AUIHFOMO;
948   esac;
949   next(AUALACK) := case
950     loc = initial : {TRUE, FALSE};
951     TRUE : AUALACK;
952   esac;

```

```
953 next(IHAUMRW) := case
954   loc = initial : {TRUE, FALSE};
955   TRUE : IHAUMRW;
956 esac;
957 next(AURSTART) := case
958   loc = initial : {TRUE, FALSE};
959   TRUE : AURSTART;
960 esac;
961 next(PONOFF.PARREG_0__) := case
962   loc = initial : {TRUE, FALSE};
963   TRUE : PONOFF.PARREG_0__;
964 esac;
965 next(PONOFF.PARREG_1__) := case
966   loc = initial : {TRUE, FALSE};
967   TRUE : PONOFF.PARREG_1__;
968 esac;
969 next(PONOFF.PARREG_2__) := case
970   loc = initial : {TRUE, FALSE};
971   TRUE : PONOFF.PARREG_2__;
972 esac;
973 next(PONOFF.PARREG_3__) := case
974   loc = initial : {TRUE, FALSE};
975   TRUE : PONOFF.PARREG_3__;
976 esac;
977 next(PONOFF.PARREG_4__) := case
978   loc = initial : {TRUE, FALSE};
979   TRUE : PONOFF.PARREG_4__;
980 esac;
981 next(PONOFF.PARREG_5__) := case
982   loc = initial : {TRUE, FALSE};
983   TRUE : PONOFF.PARREG_5__;
984 esac;
985 next(PONOFF.PARREG_6__) := case
986   loc = initial : {TRUE, FALSE};
987   TRUE : PONOFF.PARREG_6__;
988 esac;
989 next(PONOFF.PARREG_7__) := case
990   loc = initial : {TRUE, FALSE};
991   TRUE : PONOFF.PARREG_7__;
992 esac;
993 next(PONOFF.PARREG_8__) := case
994   loc = initial : {TRUE, FALSE};
995   TRUE : PONOFF.PARREG_8__;
996 esac;
997 next(PONOFF.PARREG_9__) := case
998   loc = initial : {TRUE, FALSE};
999   TRUE : PONOFF.PARREG_9__;
1000 esac;
1001 next(PONOFF.PARREG_10__) := case
1002   loc = initial : {TRUE, FALSE};
1003   TRUE : PONOFF.PARREG_10__;
1004 esac;
1005 next(PONOFF.PARREG_11__) := case
1006   loc = initial : {TRUE, FALSE};
1007   TRUE : PONOFF.PARREG_11__;
1008 esac;
1009 next(PONOFF.PARREG_12__) := case
1010   loc = initial : {TRUE, FALSE};
```

```

1011     TRUE : PONOFF.PARREG_12__;
1012   esac;
1013   next(PONOFF.PARREG_13__) := case
1014     loc = initial : {TRUE, FALSE};
1015     TRUE : PONOFF.PARREG_13__;
1016   esac;
1017   next(PONOFF.PARREG_14__) := case
1018     loc = initial : {TRUE, FALSE};
1019     TRUE : PONOFF.PARREG_14__;
1020   esac;
1021   next(PONOFF.PARREG_15__) := case
1022     loc = initial : {TRUE, FALSE};
1023     TRUE : PONOFF.PARREG_15__;
1024   esac;
1025   next(PONOFF.PPULSELE) := case
1026     loc = initial : main.random_r53;
1027     TRUE : PONOFF.PPULSELE;
1028   esac;
1029   next(PONOFF.PWDT) := case
1030     loc = initial : main.random_r54;
1031     TRUE : PONOFF.PWDT;
1032   esac;
1033   init(STSREG01_0__) := FALSE;
1034   next(STSREG01_0__) := case
1035     loc = 1411 : AUMRW;
1036     TRUE : STSREG01_0__;
1037   esac;
1038   init(STSREG01_1__) := FALSE;
1039   next(STSREG01_1__) := case
1040     loc = 1412 : POSW;
1041     TRUE : STSREG01_1__;
1042   esac;
1043   init(STSREG01_2__) := FALSE;
1044   next(STSREG01_2__) := case
1045     loc = 1413 : WSTARTIST;
1046     TRUE : STSREG01_2__;
1047   esac;
1048   init(STSREG01_3__) := FALSE;
1049   next(STSREG01_3__) := case
1050     loc = 1414 : WTSTOPIST;
1051     TRUE : STSREG01_3__;
1052   esac;
1053   init(STSREG01_4__) := FALSE;
1054   next(STSREG01_4__) := case
1055     loc = 1415 : ALUNACK;
1056     TRUE : STSREG01_4__;
1057   esac;
1058   init(STSREG01_5__) := FALSE;
1059   next(STSREG01_5__) := case
1060     loc = 1416 : AUIHFOMO;
1061     TRUE : STSREG01_5__;
1062   esac;
1063   init(STSREG01_6__) := FALSE;
1064   next(STSREG01_6__) := case
1065     loc = 1417 : WALST;
1066     TRUE : STSREG01_6__;
1067   esac;
1068   init(STSREG01_7__) := FALSE;

```

```
1069 next(STSREG01_7__) := case
1070     loc = 1418 : AUIHMMO;
1071     TRUE : STSREG01_7__;
1072 esac;
1073 init(STSREG01_8__) := FALSE;
1074 next(STSREG01_8__) := case
1075     loc = 1403 : ONST;
1076     TRUE : STSREG01_8__;
1077 esac;
1078 init(STSREG01_9__) := FALSE;
1079 next(STSREG01_9__) := case
1080     loc = 1404 : OFFST;
1081     TRUE : STSREG01_9__;
1082 esac;
1083 init(STSREG01_10__) := FALSE;
1084 next(STSREG01_10__) := case
1085     loc = 1405 : AUMOST;
1086     TRUE : STSREG01_10__;
1087 esac;
1088 init(STSREG01_11__) := FALSE;
1089 next(STSREG01_11__) := case
1090     loc = 1406 : MMOST;
1091     TRUE : STSREG01_11__;
1092 esac;
1093 init(STSREG01_12__) := FALSE;
1094 next(STSREG01_12__) := case
1095     loc = 1407 : FOMOST;
1096     TRUE : STSREG01_12__;
1097 esac;
1098 init(STSREG01_13__) := FALSE;
1099 next(STSREG01_13__) := case
1100     loc = 1408 : LDST;
1101     TRUE : STSREG01_13__;
1102 esac;
1103 init(STSREG01_14__) := FALSE;
1104 next(STSREG01_14__) := case
1105     loc = 1409 : IOERRORW;
1106     TRUE : STSREG01_14__;
1107 esac;
1108 init(STSREG01_15__) := FALSE;
1109 next(STSREG01_15__) := case
1110     loc = 1410 : IOSIMUW;
1111     TRUE : STSREG01_15__;
1112 esac;
1113 init(STSREG02_0__) := FALSE;
1114 next(STSREG02_0__) := case
1115     loc = 1427 : FALSE;
1116     TRUE : STSREG02_0__;
1117 esac;
1118 init(STSREG02_1__) := FALSE;
1119 next(STSREG02_1__) := case
1120     loc = 1428 : FALSE;
1121     TRUE : STSREG02_1__;
1122 esac;
1123 init(STSREG02_2__) := FALSE;
1124 next(STSREG02_2__) := case
1125     loc = 1429 : WFUSTOPIST;
1126     TRUE : STSREG02_2__;
```

```

1127     esac;
1128     init(STSREG02_3__) := FALSE;
1129     next(STSREG02_3__) := case
1130         loc = 1430 : ENRSTARTST;
1131         TRUE : STSREG02_3__;
1132     esac;
1133     init(STSREG02_4__) := FALSE;
1134     next(STSREG02_4__) := case
1135         loc = 1431 : SOFTLDST;
1136         TRUE : STSREG02_4__;
1137     esac;
1138     init(STSREG02_5__) := FALSE;
1139     next(STSREG02_5__) := case
1140         loc = 1432 : ALBW;
1141         TRUE : STSREG02_5__;
1142     esac;
1143     init(STSREG02_6__) := FALSE;
1144     next(STSREG02_6__) := case
1145         loc = 1433 : OUTOFFOVST;
1146         TRUE : STSREG02_6__;
1147     esac;
1148     init(STSREG02_7__) := FALSE;
1149     next(STSREG02_7__) := case
1150         loc = 1434 : FALSE;
1151         TRUE : STSREG02_7__;
1152     esac;
1153     init(STSREG02_8__) := FALSE;
1154     next(STSREG02_8__) := case
1155         loc = 1419 : OUTONOVST;
1156         TRUE : STSREG02_8__;
1157     esac;
1158     init(STSREG02_9__) := FALSE;
1159     next(STSREG02_9__) := case
1160         loc = 1420 : AUONRST;
1161         TRUE : STSREG02_9__;
1162     esac;
1163     init(STSREG02_10__) := FALSE;
1164     next(STSREG02_10__) := case
1165         loc = 1421 : MONRST;
1166         TRUE : STSREG02_10__;
1167     esac;
1168     init(STSREG02_11__) := FALSE;
1169     next(STSREG02_11__) := case
1170         loc = 1422 : AUOFFRST;
1171         TRUE : STSREG02_11__;
1172     esac;
1173     init(STSREG02_12__) := FALSE;
1174     next(STSREG02_12__) := case
1175         loc = 1423 : MOFFRST;
1176         TRUE : STSREG02_12__;
1177     esac;
1178     init(STSREG02_13__) := FALSE;
1179     next(STSREG02_13__) := case
1180         loc = 1424 : HONRST;
1181         TRUE : STSREG02_13__;
1182     esac;
1183     init(STSREG02_14__) := FALSE;
1184     next(STSREG02_14__) := case

```

```

1185     loc = 1425 : HOFFRST;
1186     TRUE : STSREG02_14__;
1187   esac;
1188   init(STSREG02_15__) := FALSE;
1189   next(STSREG02_15__) := case
1190     loc = 1426 : FALSE;
1191     TRUE : STSREG02_15__;
1192   esac;
1193   init(OUTONOV) := FALSE;
1194   next(OUTONOV) := case
1195     loc = 1332 : !(OUTONOVST);
1196     loc = 1333 : OUTONOVST;
1197     loc = 1334 : OUTONOVST;
1198     TRUE : OUTONOV;
1199   esac;
1200   init(OUTOFFOV) := FALSE;
1201   next(OUTOFFOV) := case
1202     loc = 1335 : OUTOFFOVST;
1203     TRUE : OUTOFFOV;
1204   esac;
1205   init(ONST) := FALSE;
1206   next(ONST) := case
1207     loc = 1177 : (((HFON & PHFON) | (((!(PHFON) & PHFOFF) & PANIM) & !(
1208       HFOFF))) | ((!(PHFON) & !(PHFOFF)) & OUTOVST_AUX));
1209     TRUE : ONST;
1210   esac;
1211   init(OFFST) := FALSE;
1212   next(OFFST) := case
1213     loc = 1178 : (((HFOFF & PHFOFF) | (((!(PHFOFF) & PHFON) & PANIM) &
1214       !(HFON))) | ((!(PHFON) & !(PHFOFF)) & !(OUTOVST_AUX)));
1215     TRUE : OFFST;
1216   esac;
1217   init(AUMOST) := FALSE;
1218   next(AUMOST) := case
1219     loc = 1168 : AUMOST_AUX;
1220     loc = 1172 : FALSE;
1221     TRUE : AUMOST;
1222   esac;
1223   init(MMOST) := FALSE;
1224   next(MMOST) := case
1225     loc = 1169 : MMOST_AUX;
1226     loc = 1173 : FALSE;
1227     TRUE : MMOST;
1228   esac;
1229   init(LDST) := FALSE;
1230   next(LDST) := case
1231     loc = 1167 : FALSE;
1232     loc = 1175 : TRUE;
1233     TRUE : LDST;
1234   esac;
1235   init(SOFTLDST) := FALSE;
1236   next(SOFTLDST) := case
1237     loc = 1171 : SOFTLDST_AUX;
1238     loc = 1176 : FALSE;
1239     TRUE : SOFTLDST;
1240   esac;
1241   init(FOMOST) := FALSE;
1242   next(FOMOST) := case

```

```

1241     loc = 1170 : FOMOST_AUX;
1242     loc = 1174 : FALSE;
1243     TRUE : FOMOST;
1244   esac;
1245   init(AUONRST) := FALSE;
1246   next(AUONRST) := case
1247     loc = 1180 : FALSE;
1248     loc = 1181 : TRUE;
1249     loc = 1182 : PFSPON;
1250     TRUE : AUONRST;
1251   esac;
1252   init(AUOFFRST) := FALSE;
1253   next(AUOFFRST) := case
1254     loc = 1183 : !(AUONRST);
1255     TRUE : AUOFFRST;
1256   esac;
1257   init(MONRST) := FALSE;
1258   next(MONRST) := case
1259     loc = 1185 : FALSE;
1260     loc = 1186 : TRUE;
1261     TRUE : MONRST;
1262   esac;
1263   init(MOFFRST) := FALSE;
1264   next(MOFFRST) := case
1265     loc = 1187 : !(MONRST);
1266     TRUE : MOFFRST;
1267   esac;
1268   init(HONRST) := FALSE;
1269   next(HONRST) := case
1270     loc = 1189 : FALSE;
1271     loc = 1191 : TRUE;
1272     TRUE : HONRST;
1273   esac;
1274   init(HOFFRST) := FALSE;
1275   next(HOFFRST) := case
1276     loc = 1192 : !(HONRST);
1277     TRUE : HOFFRST;
1278   esac;
1279   init(IOERRORW) := FALSE;
1280   next(IOERRORW) := case
1281     loc = 1327 : IOERROR;
1282     TRUE : IOERRORW;
1283   esac;
1284   init(IOSIMUW) := FALSE;
1285   next(IOSIMUW) := case
1286     loc = 1328 : IOSIMU;
1287     TRUE : IOSIMUW;
1288   esac;
1289   init(AUMRW) := FALSE;
1290   next(AUMRW) := case
1291     loc = 1329 : (((MMOST | FOMOST) | SOFTLDST) & ((AUONRST xor MONRST)
1292       | (AUOFFRST xor MOFFRST))) & !(IHAUMRW));
1293     TRUE : AUMRW;
1294   esac;
1295   init(ALUNACK) := FALSE;
1296   next(ALUNACK) := case
1297     loc = 1129 : FALSE;
1298     loc = 1130 : TRUE;

```

```
1298     TRUE : ALUNACK;
1299   esac;
1300   init(POSW) := FALSE;
1301   next(POSW) := case
1302     loc = 1375 : INLINED_TIMER_WARNING__Q;
1303     TRUE : POSW;
1304   esac;
1305   init(STARTIST) := FALSE;
1306   next(STARTIST) := case
1307     loc = 1124 : STARTI;
1308     TRUE : STARTIST;
1309   esac;
1310   init(TSTOPIST) := FALSE;
1311   next(TSTOPIST) := case
1312     loc = 1125 : TSTOPI;
1313     TRUE : TSTOPIST;
1314   esac;
1315   init(FUSTOPIST) := FALSE;
1316   next(FUSTOPIST) := case
1317     loc = 1126 : FUSTOPI;
1318     TRUE : FUSTOPIST;
1319   esac;
1320   init(ALST) := FALSE;
1321   next(ALST) := case
1322     loc = 1326 : AL;
1323     TRUE : ALST;
1324   esac;
1325   init(ALBW) := FALSE;
1326   next(ALBW) := case
1327     loc = 1377 : ALB;
1328     TRUE : ALBW;
1329   esac;
1330   init(ENRSTARTST) := TRUE;
1331   next(ENRSTARTST) := case
1332     loc = 1132 : TRUE;
1333     loc = 1136 : FALSE;
1334     TRUE : ENRSTARTST;
1335   esac;
1336   init(RDYSTARTST) := FALSE;
1337   next(RDYSTARTST) := case
1338     loc = 1325 : !(INTERLOCKR);
1339     TRUE : RDYSTARTST;
1340   esac;
1341   init(E_MAUMOR) := FALSE;
1342   next(E_MAUMOR) := case
1343     loc = 13 : TRUE;
1344     loc = 15 : FALSE;
1345     TRUE : E_MAUMOR;
1346   esac;
1347   init(E_MMMOR) := FALSE;
1348   next(E_MMMOR) := case
1349     loc = 111 : TRUE;
1350     loc = 113 : FALSE;
1351     TRUE : E_MMMOR;
1352   esac;
1353   init(E_MFOMOR) := FALSE;
1354   next(E_MFOMOR) := case
1355     loc = 119 : TRUE;
```

```

1356     loc = 121 : FALSE;
1357     TRUE : E_MFOMOR;
1358   esac;
1359   init(E_MONR) := FALSE;
1360   next(E_MONR) := case
1361     loc = 135 : TRUE;
1362     loc = 137 : FALSE;
1363     TRUE : E_MONR;
1364   esac;
1365   init(E_MOFFR) := FALSE;
1366   next(E_MOFFR) := case
1367     loc = 143 : TRUE;
1368     loc = 145 : FALSE;
1369     TRUE : E_MOFFR;
1370   esac;
1371   init(E_MALACKR) := FALSE;
1372   next(E_MALACKR) := case
1373     loc = 159 : TRUE;
1374     loc = 161 : FALSE;
1375     TRUE : E_MALACKR;
1376   esac;
1377   init(E_STARTI) := FALSE;
1378   next(E_STARTI) := case
1379     loc = 195 : TRUE;
1380     loc = 197 : FALSE;
1381     TRUE : E_STARTI;
1382   esac;
1383   init(E_TSTOPI) := FALSE;
1384   next(E_TSTOPI) := case
1385     loc = 1103 : TRUE;
1386     loc = 1105 : FALSE;
1387     TRUE : E_TSTOPI;
1388   esac;
1389   init(E_FUSTOPI) := FALSE;
1390   next(E_FUSTOPI) := case
1391     loc = 1111 : TRUE;
1392     loc = 1113 : FALSE;
1393     TRUE : E_FUSTOPI;
1394   esac;
1395   init(E_AL) := FALSE;
1396   next(E_AL) := case
1397     loc = 1119 : TRUE;
1398     loc = 1121 : FALSE;
1399     TRUE : E_AL;
1400   esac;
1401   init(E_AUAUMOR) := FALSE;
1402   next(E_AUAUMOR) := case
1403     loc = 179 : TRUE;
1404     loc = 181 : FALSE;
1405     TRUE : E_AUAUMOR;
1406   esac;
1407   init(E_AUALACK) := FALSE;
1408   next(E_AUALACK) := case
1409     loc = 187 : TRUE;
1410     loc = 189 : FALSE;
1411     TRUE : E_AUALACK;
1412   esac;
1413   init(E_MSOFTLDR) := FALSE;

```

```
1414 next(E_MSOFTLDR) := case
1415     loc = 127 : TRUE;
1416     loc = 129 : FALSE;
1417     TRUE : E_MSOFTLDR;
1418 esac;
1419 init(E_MENRSTARTR) := FALSE;
1420 next(E_MENRSTARTR) := case
1421     loc = 151 : TRUE;
1422     loc = 153 : FALSE;
1423     TRUE : E_MENRSTARTR;
1424 esac;
1425 init(RE_ALUNACK) := FALSE;
1426 next(RE_ALUNACK) := case
1427     loc = 1449 : INLINED_DETECT_EDGE_28__RE;
1428     TRUE : RE_ALUNACK;
1429 esac;
1430 init(FE_ALUNACK) := FALSE;
1431 next(FE_ALUNACK) := case
1432     loc = 1450 : INLINED_DETECT_EDGE_28__FE;
1433     TRUE : FE_ALUNACK;
1434 esac;
1435 init(RE_PULSEON) := FALSE;
1436 next(RE_PULSEON) := case
1437     loc = 1258 : TRUE;
1438     loc = 1260 : FALSE;
1439     TRUE : RE_PULSEON;
1440 esac;
1441 init(FE_PULSEON) := FALSE;
1442 next(FE_PULSEON) := case
1443     loc = 1266 : TRUE;
1444     loc = 1268 : FALSE;
1445     TRUE : FE_PULSEON;
1446 esac;
1447 init(RE_PULSEOFF) := FALSE;
1448 next(RE_PULSEOFF) := case
1449     loc = 1274 : TRUE;
1450     loc = 1276 : FALSE;
1451     TRUE : RE_PULSEOFF;
1452 esac;
1453 init(RE_OUTOVST_AUX) := FALSE;
1454 next(RE_OUTOVST_AUX) := case
1455     loc = 1343 : TRUE;
1456     loc = 1345 : FALSE;
1457     TRUE : RE_OUTOVST_AUX;
1458 esac;
1459 init(FE_OUTOVST_AUX) := FALSE;
1460 next(FE_OUTOVST_AUX) := case
1461     loc = 1351 : TRUE;
1462     loc = 1353 : FALSE;
1463     TRUE : FE_OUTOVST_AUX;
1464 esac;
1465 init(FE_INTERLOCKR) := FALSE;
1466 next(FE_INTERLOCKR) := case
1467     loc = 1141 : TRUE;
1468     loc = 1143 : FALSE;
1469     TRUE : FE_INTERLOCKR;
1470 esac;
1471 init(MAUMOR_OLD) := FALSE;
```

```

1472     next(MAUMOR_OLD) := case
1473         loc = 17 : INLINED_R_EDGE_1__OLD;
1474         TRUE : MAUMOR_OLD;
1475     esac;
1476     init(MMMOR_OLD) := FALSE;
1477     next(MMMOR_OLD) := case
1478         loc = 115 : INLINED_R_EDGE_2__OLD;
1479         TRUE : MMMOR_OLD;
1480     esac;
1481     init(MFOMOR_OLD) := FALSE;
1482     next(MFOMOR_OLD) := case
1483         loc = 123 : INLINED_R_EDGE_3__OLD;
1484         TRUE : MFOMOR_OLD;
1485     esac;
1486     init(MONR_OLD) := FALSE;
1487     next(MONR_OLD) := case
1488         loc = 139 : INLINED_R_EDGE_5__OLD;
1489         TRUE : MONR_OLD;
1490     esac;
1491     init(MOFFR_OLD) := FALSE;
1492     next(MOFFR_OLD) := case
1493         loc = 147 : INLINED_R_EDGE_6__OLD;
1494         TRUE : MOFFR_OLD;
1495     esac;
1496     init(MALACKR_OLD) := FALSE;
1497     next(MALACKR_OLD) := case
1498         loc = 163 : INLINED_R_EDGE_8__OLD;
1499         TRUE : MALACKR_OLD;
1500     esac;
1501     init(AUAUMOR_OLD) := FALSE;
1502     next(AUAUMOR_OLD) := case
1503         loc = 183 : INLINED_R_EDGE_9__OLD;
1504         TRUE : AUAUMOR_OLD;
1505     esac;
1506     init(AUALACK_OLD) := FALSE;
1507     next(AUALACK_OLD) := case
1508         loc = 191 : INLINED_R_EDGE_10__OLD;
1509         TRUE : AUALACK_OLD;
1510     esac;
1511     init(STARTI_OLD) := FALSE;
1512     next(STARTI_OLD) := case
1513         loc = 199 : INLINED_R_EDGE_11__OLD;
1514         TRUE : STARTI_OLD;
1515     esac;
1516     init(TSTOPI_OLD) := FALSE;
1517     next(TSTOPI_OLD) := case
1518         loc = 1107 : INLINED_R_EDGE_12__OLD;
1519         TRUE : TSTOPI_OLD;
1520     esac;
1521     init(FUSTOPI_OLD) := FALSE;
1522     next(FUSTOPI_OLD) := case
1523         loc = 1115 : INLINED_R_EDGE_13__OLD;
1524         TRUE : FUSTOPI_OLD;
1525     esac;
1526     init(AL_OLD) := FALSE;
1527     next(AL_OLD) := case
1528         loc = 1123 : INLINED_R_EDGE_14__OLD;
1529         TRUE : AL_OLD;

```

```
1530     esac;
1531     init(ALUNACK_OLD) := FALSE;
1532     next(ALUNACK_OLD) := case
1533         loc = 1448 : INLINED_DETECT_EDGE_28__OLD;
1534         TRUE : ALUNACK_OLD;
1535     esac;
1536     init(MSOFTLDR_OLD) := FALSE;
1537     next(MSOFTLDR_OLD) := case
1538         loc = 131 : INLINED_R_EDGE_4__OLD;
1539         TRUE : MSOFTLDR_OLD;
1540     esac;
1541     init(MENRSTARTR_OLD) := FALSE;
1542     next(MENRSTARTR_OLD) := case
1543         loc = 155 : INLINED_R_EDGE_7__OLD;
1544         TRUE : MENRSTARTR_OLD;
1545     esac;
1546     init(RE_PULSEON_OLD) := FALSE;
1547     next(RE_PULSEON_OLD) := case
1548         loc = 1262 : INLINED_R_EDGE_20__OLD;
1549         TRUE : RE_PULSEON_OLD;
1550     esac;
1551     init(FE_PULSEON_OLD) := FALSE;
1552     next(FE_PULSEON_OLD) := case
1553         loc = 1270 : INLINED_F_EDGE_21__OLD;
1554         TRUE : FE_PULSEON_OLD;
1555     esac;
1556     init(RE_PULSEOFF_OLD) := FALSE;
1557     next(RE_PULSEOFF_OLD) := case
1558         loc = 1278 : INLINED_R_EDGE_22__OLD;
1559         TRUE : RE_PULSEOFF_OLD;
1560     esac;
1561     init(RE_OUTOVST_AUX_OLD) := FALSE;
1562     next(RE_OUTOVST_AUX_OLD) := case
1563         loc = 1347 : INLINED_R_EDGE_25__OLD;
1564         TRUE : RE_OUTOVST_AUX_OLD;
1565     esac;
1566     init(FE_OUTOVST_AUX_OLD) := FALSE;
1567     next(FE_OUTOVST_AUX_OLD) := case
1568         loc = 1355 : INLINED_F_EDGE_26__OLD;
1569         TRUE : FE_OUTOVST_AUX_OLD;
1570     esac;
1571     init(FE_INTERLOCKR_OLD) := FALSE;
1572     next(FE_INTERLOCKR_OLD) := case
1573         loc = 1145 : INLINED_F_EDGE_15__OLD;
1574         TRUE : FE_INTERLOCKR_OLD;
1575     esac;
1576     init(PFSPOSON) := FALSE;
1577     next(PFSPOSON) := case
1578         loc = 164 : PONOFF.PARREG_8__;
1579         TRUE : PFSPOSON;
1580     esac;
1581     init(PFSPOSON2) := FALSE;
1582     next(PFSPOSON2) := case
1583         loc = 174 : PONOFF.PARREG_2__;
1584         TRUE : PFSPOSON2;
1585     esac;
1586     init(PHFON) := FALSE;
1587     next(PHFON) := case
```

```

1588     loc = 165 : PONOFF.PARREG_9__;
1589     TRUE : PHFON;
1590   esac;
1591   init(PHFOFF) := FALSE;
1592   next(PHFOFF) := case
1593     loc = 166 : PONOFF.PARREG_10__;
1594     TRUE : PHFOFF;
1595   esac;
1596   init(PPULSE) := FALSE;
1597   next(PPULSE) := case
1598     loc = 167 : PONOFF.PARREG_11__;
1599     TRUE : PPULSE;
1600   esac;
1601   init(PPULSESECSTE) := FALSE;
1602   next(PPULSESECSTE) := case
1603     loc = 175 : PONOFF.PARREG_3__;
1604     TRUE : PPULSESECSTE;
1605   esac;
1606   init(PHLD) := FALSE;
1607   next(PHLD) := case
1608     loc = 168 : PONOFF.PARREG_12__;
1609     TRUE : PHLD;
1610   esac;
1611   init(PHLDCMD) := FALSE;
1612   next(PHLDCMD) := case
1613     loc = 169 : PONOFF.PARREG_13__;
1614     TRUE : PHLDCMD;
1615   esac;
1616   init(PANIM) := FALSE;
1617   next(PANIM) := case
1618     loc = 170 : PONOFF.PARREG_14__;
1619     TRUE : PANIM;
1620   esac;
1621   init(POUTOFF) := FALSE;
1622   next(POUTOFF) := case
1623     loc = 171 : PONOFF.PARREG_15__;
1624     TRUE : POUTOFF;
1625   esac;
1626   init(PENRSTART) := FALSE;
1627   next(PENRSTART) := case
1628     loc = 172 : PONOFF.PARREG_0__;
1629     TRUE : PENRSTART;
1630   esac;
1631   init(PRSTARTFS) := FALSE;
1632   next(PRSTARTFS) := case
1633     loc = 173 : PONOFF.PARREG_1__;
1634     TRUE : PRSTARTFS;
1635   esac;
1636   init(OUTONOVST) := FALSE;
1637   next(OUTONOVST) := case
1638     loc = 1310 : ((PPULSE & PULSEON) | (!(PPULSE) & ((MONRST & ((MOST
      | FOMOST) | SOFTLDST)) | (AUONRST & AUMOST)) | ((HONRST & LDST)
      & PHLDCMD)))));
1639     loc = 1317 : PULSEON;
1640     loc = 1319 : FALSE;
1641     loc = 1321 : ((PFSPON & !(PFSPON2)) | (PFSPON & PFSPON2));
1642     loc = 1324 : PFSPON;
1643     TRUE : OUTONOVST;

```

```

1644     esac;
1645     init(OUTOFFOVST) := FALSE;
1646     next(OUTOFFOVST) := case
1647         loc = 1312 : ((PULSEOFF & PPULSE) | (!(PPULSE) & ((MOFFRST & ((
                MMOST | FOMOST) | SOFTLDST)) | (AUOFFRST & AUMOST)) | ((HOFFRST
                & LDST) & PHLDCMD)))));
1648         loc = 1318 : FALSE;
1649         loc = 1320 : PULSEOFF;
1650         loc = 1322 : ((!(PFSPON) & !(PFSPON2)) | (PFSPON & PFSPON2))
                ;
1651         TRUE : OUTOFFOVST;
1652     esac;
1653     init(AUMOST_AUX) := FALSE;
1654     next(AUMOST_AUX) := case
1655         loc = 1148 : FALSE;
1656         loc = 1153 : FALSE;
1657         loc = 1158 : TRUE;
1658         loc = 1163 : FALSE;
1659         TRUE : AUMOST_AUX;
1660     esac;
1661     init(MMOST_AUX) := FALSE;
1662     next(MMOST_AUX) := case
1663         loc = 1149 : FALSE;
1664         loc = 1154 : TRUE;
1665         loc = 1159 : FALSE;
1666         loc = 1164 : FALSE;
1667         TRUE : MMOST_AUX;
1668     esac;
1669     init(FOMOST_AUX) := FALSE;
1670     next(FOMOST_AUX) := case
1671         loc = 1150 : TRUE;
1672         loc = 1155 : FALSE;
1673         loc = 1160 : FALSE;
1674         loc = 1165 : FALSE;
1675         TRUE : FOMOST_AUX;
1676     esac;
1677     init(SOFTLDST_AUX) := FALSE;
1678     next(SOFTLDST_AUX) := case
1679         loc = 1151 : FALSE;
1680         loc = 1156 : FALSE;
1681         loc = 1161 : FALSE;
1682         loc = 1166 : TRUE;
1683         TRUE : SOFTLDST_AUX;
1684     esac;
1685     init(PULSEON) := FALSE;
1686     next(PULSEON) := case
1687         loc = 1306 : (INLINED_TIMER_PULSEON__Q & !(PULSEOFFR));
1688         loc = 1308 : ((INLINED_TIMER_PULSEON__Q & !(PULSEOFFR)) & !(PHFON
                | (PHFON & !(HFON))));
1689         TRUE : PULSEON;
1690     esac;
1691     init(PULSEOFF) := FALSE;
1692     next(PULSEOFF) := case
1693         loc = 1307 : (INLINED_TIMER_PULSEOFF__Q & !(PULSEONR));
1694         loc = 1309 : ((INLINED_TIMER_PULSEOFF__Q & !(PULSEONR)) & !(PHFOFF
                | (PHFOFF & !(HFOFF))));
1695         TRUE : PULSEOFF;
1696     esac;

```

```

1697     init(POSW_AUX) := FALSE;
1698     next(POSW_AUX) := case
1699         loc = 1357 : TRUE;
1700         loc = 1359 : FALSE;
1701         TRUE : POSW_AUX;
1702     esac;
1703     init(OUTOVST_AUX) := FALSE;
1704     next(OUTOVST_AUX) := case
1705         loc = 1337 : TRUE;
1706         loc = 1339 : FALSE;
1707         TRUE : OUTOVST_AUX;
1708     esac;
1709     init(FULLNOTACKNOWLEDGED) := FALSE;
1710     next(FULLNOTACKNOWLEDGED) := case
1711         loc = 1128 : FALSE;
1712         loc = 1134 : TRUE;
1713         TRUE : FULLNOTACKNOWLEDGED;
1714     esac;
1715     init(PULSEONR) := FALSE;
1716     next(PULSEONR) := case
1717         loc = 1195 : ((PFSPON & !(PFSPON2)) | (PFSPON & PFSPON2));
1718         loc = 1197 : FALSE;
1719         loc = 1223 : FALSE;
1720         loc = 1225 : TRUE;
1721         loc = 1227 : FALSE;
1722         loc = 1241 : INLINED_TIMER_PULSEON__IN;
1723         TRUE : PULSEONR;
1724     esac;
1725     init(PULSEOFFR) := FALSE;
1726     next(PULSEOFFR) := case
1727         loc = 1196 : (((!(PFSPON) & !(PFSPON2)) | (PFSPON & PFSPON2))
1728             ;
1729         loc = 1198 : FALSE;
1730         loc = 1224 : TRUE;
1731         loc = 1226 : FALSE;
1732         loc = 1228 : FALSE;
1733         loc = 1254 : INLINED_TIMER_PULSEOFF__IN;
1734         TRUE : PULSEOFFR;
1735     esac;
1736     init(INTERLOCKR) := FALSE;
1737     next(INTERLOCKR) := case
1738         loc = 1137 : (((((TSTOPIST | FUSTOPIST) | FULLNOTACKNOWLEDGED) | !(
1739             ENRSTARTST)) | ((STARTIST & !(POUTOFF)) & !(OUTONOV))) | ((
1740             STARTIST & POUTOFF) & ((PFSPON & OUTOVST_AUX) | (!(PFSPON)
1741             & !(OUTOVST_AUX)))));
1742         TRUE : INTERLOCKR;
1743     esac;
1744     init(TIME_WARNING) := Osd32_0;
1745     next(TIME_WARNING) := case
1746         loc = 1376 : INLINED_TIMER_WARNING__ET;
1747         TRUE : TIME_WARNING;
1748     esac;
1749     init(PULSEWIDTH) := Osd32_0;
1750     next(PULSEWIDTH) := case
1751         loc = 1378 : (Osd32_150000 * Osd32_100) / (signed(extend(main.
1752             T_CYCLE, 16)) * Osd32_100);
1753         TRUE : PULSEWIDTH;
1754     esac;

```

```

1750  init(FSIINC) := Osd16_0;
1751  next(FSIINC) := case
1752    loc = 1380 : (FSIINC + Osd16_1);
1753    loc = 1383 : Osd16_0;
1754    TRUE : FSIINC;
1755  esac;
1756  init(TSIINC) := Osd16_0;
1757  next(TSIINC) := case
1758    loc = 1386 : (TSIINC + Osd16_1);
1759    loc = 1389 : Osd16_0;
1760    TRUE : TSIINC;
1761  esac;
1762  init(SIINC) := Osd16_0;
1763  next(SIINC) := case
1764    loc = 1392 : (SIINC + Osd16_1);
1765    loc = 1395 : Osd16_0;
1766    TRUE : SIINC;
1767  esac;
1768  init(ALINC) := Osd16_0;
1769  next(ALINC) := case
1770    loc = 1398 : (ALINC + Osd16_1);
1771    loc = 1401 : Osd16_0;
1772    TRUE : ALINC;
1773  esac;
1774  init(WTSTOPIST) := FALSE;
1775  next(WTSTOPIST) := case
1776    loc = 1387 : TRUE;
1777    loc = 1390 : TSTOPIST;
1778    TRUE : WTSTOPIST;
1779  esac;
1780  init(WSTARTIST) := FALSE;
1781  next(WSTARTIST) := case
1782    loc = 1393 : TRUE;
1783    loc = 1396 : STARTIST;
1784    TRUE : WSTARTIST;
1785  esac;
1786  init(WALST) := FALSE;
1787  next(WALST) := case
1788    loc = 1399 : TRUE;
1789    loc = 1402 : ALST;
1790    TRUE : WALST;
1791  esac;
1792  init(WFUSTOPIST) := FALSE;
1793  next(WFUSTOPIST) := case
1794    loc = 1381 : TRUE;
1795    loc = 1384 : FUSTOPIST;
1796    TRUE : WFUSTOPIST;
1797  esac;
1798  init(INLINED_R_EDGE_1__NEW) := FALSE;
1799  next(INLINED_R_EDGE_1__NEW) := case
1800    loc = end : FALSE;
1801    loc = 11 : MANREG01_8__;
1802    TRUE : INLINED_R_EDGE_1__NEW;
1803  esac;
1804  init(INLINED_R_EDGE_1__OLD) := FALSE;
1805  next(INLINED_R_EDGE_1__OLD) := case
1806    loc = end : FALSE;
1807    loc = 10 : MAUMOR_OLD;

```

```

1808     loc = 14 : TRUE;
1809     loc = 16 : INLINED_R_EDGE_1__NEW;
1810     TRUE : INLINED_R_EDGE_1__OLD;
1811   esac;
1812   init(INLINED_R_EDGE_2__NEW) := FALSE;
1813   next(INLINED_R_EDGE_2__NEW) := case
1814     loc = end : FALSE;
1815     loc = 19 : MANREGO1_9__;
1816     TRUE : INLINED_R_EDGE_2__NEW;
1817   esac;
1818   init(INLINED_R_EDGE_2__OLD) := FALSE;
1819   next(INLINED_R_EDGE_2__OLD) := case
1820     loc = end : FALSE;
1821     loc = 18 : MMMOR_OLD;
1822     loc = 112 : TRUE;
1823     loc = 114 : INLINED_R_EDGE_2__NEW;
1824     TRUE : INLINED_R_EDGE_2__OLD;
1825   esac;
1826   init(INLINED_R_EDGE_3__NEW) := FALSE;
1827   next(INLINED_R_EDGE_3__NEW) := case
1828     loc = end : FALSE;
1829     loc = 117 : MANREGO1_10__;
1830     TRUE : INLINED_R_EDGE_3__NEW;
1831   esac;
1832   init(INLINED_R_EDGE_3__OLD) := FALSE;
1833   next(INLINED_R_EDGE_3__OLD) := case
1834     loc = end : FALSE;
1835     loc = 116 : MFOMOR_OLD;
1836     loc = 120 : TRUE;
1837     loc = 122 : INLINED_R_EDGE_3__NEW;
1838     TRUE : INLINED_R_EDGE_3__OLD;
1839   esac;
1840   init(INLINED_R_EDGE_4__NEW) := FALSE;
1841   next(INLINED_R_EDGE_4__NEW) := case
1842     loc = end : FALSE;
1843     loc = 125 : MANREGO1_11__;
1844     TRUE : INLINED_R_EDGE_4__NEW;
1845   esac;
1846   init(INLINED_R_EDGE_4__OLD) := FALSE;
1847   next(INLINED_R_EDGE_4__OLD) := case
1848     loc = end : FALSE;
1849     loc = 124 : MSOFTLDR_OLD;
1850     loc = 128 : TRUE;
1851     loc = 130 : INLINED_R_EDGE_4__NEW;
1852     TRUE : INLINED_R_EDGE_4__OLD;
1853   esac;
1854   init(INLINED_R_EDGE_5__NEW) := FALSE;
1855   next(INLINED_R_EDGE_5__NEW) := case
1856     loc = end : FALSE;
1857     loc = 133 : MANREGO1_12__;
1858     TRUE : INLINED_R_EDGE_5__NEW;
1859   esac;
1860   init(INLINED_R_EDGE_5__OLD) := FALSE;
1861   next(INLINED_R_EDGE_5__OLD) := case
1862     loc = end : FALSE;
1863     loc = 132 : MONR_OLD;
1864     loc = 136 : TRUE;
1865     loc = 138 : INLINED_R_EDGE_5__NEW;

```

```

1866     TRUE : INLINED_R_EDGE_5__OLD;
1867   esac;
1868   init(INLINED_R_EDGE_6__NEW) := FALSE;
1869   next(INLINED_R_EDGE_6__NEW) := case
1870     loc = end : FALSE;
1871     loc = 141 : MANREGO1_13__;
1872     TRUE : INLINED_R_EDGE_6__NEW;
1873   esac;
1874   init(INLINED_R_EDGE_6__OLD) := FALSE;
1875   next(INLINED_R_EDGE_6__OLD) := case
1876     loc = end : FALSE;
1877     loc = 140 : MOFFR_OLD;
1878     loc = 144 : TRUE;
1879     loc = 146 : INLINED_R_EDGE_6__NEW;
1880     TRUE : INLINED_R_EDGE_6__OLD;
1881   esac;
1882   init(INLINED_R_EDGE_7__NEW) := FALSE;
1883   next(INLINED_R_EDGE_7__NEW) := case
1884     loc = end : FALSE;
1885     loc = 149 : MANREGO1_1__;
1886     TRUE : INLINED_R_EDGE_7__NEW;
1887   esac;
1888   init(INLINED_R_EDGE_7__OLD) := FALSE;
1889   next(INLINED_R_EDGE_7__OLD) := case
1890     loc = end : FALSE;
1891     loc = 148 : MENRSTARTR_OLD;
1892     loc = 152 : TRUE;
1893     loc = 154 : INLINED_R_EDGE_7__NEW;
1894     TRUE : INLINED_R_EDGE_7__OLD;
1895   esac;
1896   init(INLINED_R_EDGE_8__NEW) := FALSE;
1897   next(INLINED_R_EDGE_8__NEW) := case
1898     loc = end : FALSE;
1899     loc = 157 : MANREGO1_7__;
1900     TRUE : INLINED_R_EDGE_8__NEW;
1901   esac;
1902   init(INLINED_R_EDGE_8__OLD) := FALSE;
1903   next(INLINED_R_EDGE_8__OLD) := case
1904     loc = end : FALSE;
1905     loc = 156 : MALACKR_OLD;
1906     loc = 160 : TRUE;
1907     loc = 162 : INLINED_R_EDGE_8__NEW;
1908     TRUE : INLINED_R_EDGE_8__OLD;
1909   esac;
1910   init(INLINED_R_EDGE_9__NEW) := FALSE;
1911   next(INLINED_R_EDGE_9__NEW) := case
1912     loc = end : FALSE;
1913     loc = 177 : AUAUMOR;
1914     TRUE : INLINED_R_EDGE_9__NEW;
1915   esac;
1916   init(INLINED_R_EDGE_9__OLD) := FALSE;
1917   next(INLINED_R_EDGE_9__OLD) := case
1918     loc = end : FALSE;
1919     loc = 176 : AUAUMOR_OLD;
1920     loc = 180 : TRUE;
1921     loc = 182 : INLINED_R_EDGE_9__NEW;
1922     TRUE : INLINED_R_EDGE_9__OLD;
1923   esac;

```

```

1924     init(INLINED_R_EDGE_10__NEW) := FALSE;
1925     next(INLINED_R_EDGE_10__NEW) := case
1926         loc = end : FALSE;
1927         loc = 185 : AUALACK;
1928         TRUE : INLINED_R_EDGE_10__NEW;
1929     esac;
1930     init(INLINED_R_EDGE_10__OLD) := FALSE;
1931     next(INLINED_R_EDGE_10__OLD) := case
1932         loc = end : FALSE;
1933         loc = 184 : AUALACK_OLD;
1934         loc = 188 : TRUE;
1935         loc = 190 : INLINED_R_EDGE_10__NEW;
1936         TRUE : INLINED_R_EDGE_10__OLD;
1937     esac;
1938     init(INLINED_R_EDGE_11__NEW) := FALSE;
1939     next(INLINED_R_EDGE_11__NEW) := case
1940         loc = end : FALSE;
1941         loc = 193 : STARTI;
1942         TRUE : INLINED_R_EDGE_11__NEW;
1943     esac;
1944     init(INLINED_R_EDGE_11__OLD) := FALSE;
1945     next(INLINED_R_EDGE_11__OLD) := case
1946         loc = end : FALSE;
1947         loc = 192 : STARTI_OLD;
1948         loc = 196 : TRUE;
1949         loc = 198 : INLINED_R_EDGE_11__NEW;
1950         TRUE : INLINED_R_EDGE_11__OLD;
1951     esac;
1952     init(INLINED_R_EDGE_12__NEW) := FALSE;
1953     next(INLINED_R_EDGE_12__NEW) := case
1954         loc = end : FALSE;
1955         loc = 1101 : TSTOPI;
1956         TRUE : INLINED_R_EDGE_12__NEW;
1957     esac;
1958     init(INLINED_R_EDGE_12__OLD) := FALSE;
1959     next(INLINED_R_EDGE_12__OLD) := case
1960         loc = end : FALSE;
1961         loc = 1100 : TSTOPI_OLD;
1962         loc = 1104 : TRUE;
1963         loc = 1106 : INLINED_R_EDGE_12__NEW;
1964         TRUE : INLINED_R_EDGE_12__OLD;
1965     esac;
1966     init(INLINED_R_EDGE_13__NEW) := FALSE;
1967     next(INLINED_R_EDGE_13__NEW) := case
1968         loc = end : FALSE;
1969         loc = 1109 : FUSTOPI;
1970         TRUE : INLINED_R_EDGE_13__NEW;
1971     esac;
1972     init(INLINED_R_EDGE_13__OLD) := FALSE;
1973     next(INLINED_R_EDGE_13__OLD) := case
1974         loc = end : FALSE;
1975         loc = 1108 : FUSTOPI_OLD;
1976         loc = 1112 : TRUE;
1977         loc = 1114 : INLINED_R_EDGE_13__NEW;
1978         TRUE : INLINED_R_EDGE_13__OLD;
1979     esac;
1980     init(INLINED_R_EDGE_14__NEW) := FALSE;
1981     next(INLINED_R_EDGE_14__NEW) := case

```

```

1982     loc = end : FALSE;
1983     loc = 1117 : AL;
1984     TRUE : INLINED_R_EDGE_14__NEW;
1985     esac;
1986     init(INLINED_R_EDGE_14__OLD) := FALSE;
1987     next(INLINED_R_EDGE_14__OLD) := case
1988         loc = end : FALSE;
1989         loc = 1116 : AL_OLD;
1990         loc = 1120 : TRUE;
1991         loc = 1122 : INLINED_R_EDGE_14__NEW;
1992         TRUE : INLINED_R_EDGE_14__OLD;
1993     esac;
1994     init(INLINED_F_EDGE_15__NEW) := FALSE;
1995     next(INLINED_F_EDGE_15__NEW) := case
1996         loc = end : FALSE;
1997         loc = 1139 : INTERLOCKR;
1998         TRUE : INLINED_F_EDGE_15__NEW;
1999     esac;
2000     init(INLINED_F_EDGE_15__OLD) := FALSE;
2001     next(INLINED_F_EDGE_15__OLD) := case
2002         loc = end : FALSE;
2003         loc = 1138 : FE_INTERLOCKR_OLD;
2004         loc = 1142 : FALSE;
2005         loc = 1144 : INLINED_F_EDGE_15__NEW;
2006         TRUE : INLINED_F_EDGE_15__OLD;
2007     esac;
2008     init(INLINED_TIMER_PULSEON__PT) := 0sd32_0;
2009     next(INLINED_TIMER_PULSEON__PT) := case
2010         loc = end : 0sd32_0;
2011         loc = 1199 : 0sd32_0;
2012         loc = 1229 : PONOFF.PPULSELE;
2013         loc = 1293 : 0sd32_0;
2014         TRUE : INLINED_TIMER_PULSEON__PT;
2015     esac;
2016     init(INLINED_TIMER_PULSEON__IN) := FALSE;
2017     next(INLINED_TIMER_PULSEON__IN) := case
2018         loc = end : FALSE;
2019         loc = 1200 : FALSE;
2020         loc = 1230 : PULSEONR;
2021         loc = 1294 : FALSE;
2022         TRUE : INLINED_TIMER_PULSEON__IN;
2023     esac;
2024     init(INLINED_TIMER_PULSEON__Q) := FALSE;
2025     next(INLINED_TIMER_PULSEON__Q) := case
2026         loc = end : FALSE;
2027         loc = 1204 : TRUE;
2028         loc = 1206 : FALSE;
2029         loc = 1234 : TRUE;
2030         loc = 1236 : FALSE;
2031         loc = 1298 : TRUE;
2032         loc = 1300 : FALSE;
2033         TRUE : INLINED_TIMER_PULSEON__Q;
2034     esac;
2035     init(INLINED_TIMER_PULSEON__ET) := 0sd32_0;
2036     next(INLINED_TIMER_PULSEON__ET) := case
2037         loc = end : 0sd32_0;
2038         loc = 1205 : (INLINED_TIMER_PULSEON__PT - (
                INLINED_TIMER_PULSEON__DUE - main.__GLOBAL_TIME));

```

```

2039     loc = 1208 : INLINED_TIMER_PULSEON__PT;
2040     loc = 1209 : Osd32_0;
2041     loc = 1235 : (INLINED_TIMER_PULSEON__PT - (
                INLINED_TIMER_PULSEON__DUE - main.__GLOBAL_TIME));
2042     loc = 1238 : INLINED_TIMER_PULSEON__PT;
2043     loc = 1239 : Osd32_0;
2044     loc = 1299 : (INLINED_TIMER_PULSEON__PT - (
                INLINED_TIMER_PULSEON__DUE - main.__GLOBAL_TIME));
2045     loc = 1302 : INLINED_TIMER_PULSEON__PT;
2046     loc = 1303 : Osd32_0;
2047     TRUE : INLINED_TIMER_PULSEON__ET;
2048   esac;
2049   init(INLINED_TIMER_PULSEON__OLD_IN) := FALSE;
2050   next(INLINED_TIMER_PULSEON__OLD_IN) := case
2051     loc = end : FALSE;
2052     loc = 1210 : INLINED_TIMER_PULSEON__IN;
2053     loc = 1240 : INLINED_TIMER_PULSEON__IN;
2054     loc = 1304 : INLINED_TIMER_PULSEON__IN;
2055     TRUE : INLINED_TIMER_PULSEON__OLD_IN;
2056   esac;
2057   init(INLINED_TIMER_PULSEON__DUE) := Osd32_0;
2058   next(INLINED_TIMER_PULSEON__DUE) := case
2059     loc = end : Osd32_0;
2060     loc = 1202 : (main.__GLOBAL_TIME + INLINED_TIMER_PULSEON__PT);
2061     loc = 1232 : (main.__GLOBAL_TIME + INLINED_TIMER_PULSEON__PT);
2062     loc = 1296 : (main.__GLOBAL_TIME + INLINED_TIMER_PULSEON__PT);
2063     TRUE : INLINED_TIMER_PULSEON__DUE;
2064   esac;
2065   init(INLINED_TIMER_PULSEOFF__PT) := Osd32_0;
2066   next(INLINED_TIMER_PULSEOFF__PT) := case
2067     loc = end : Osd32_0;
2068     loc = 1211 : Osd32_0;
2069     loc = 1242 : PONOFF.PPULSELE;
2070     loc = 1280 : Osd32_0;
2071     TRUE : INLINED_TIMER_PULSEOFF__PT;
2072   esac;
2073   init(INLINED_TIMER_PULSEOFF__IN) := FALSE;
2074   next(INLINED_TIMER_PULSEOFF__IN) := case
2075     loc = end : FALSE;
2076     loc = 1212 : FALSE;
2077     loc = 1243 : PULSEOFFR;
2078     loc = 1281 : FALSE;
2079     TRUE : INLINED_TIMER_PULSEOFF__IN;
2080   esac;
2081   init(INLINED_TIMER_PULSEOFF__Q) := FALSE;
2082   next(INLINED_TIMER_PULSEOFF__Q) := case
2083     loc = end : FALSE;
2084     loc = 1216 : TRUE;
2085     loc = 1218 : FALSE;
2086     loc = 1247 : TRUE;
2087     loc = 1249 : FALSE;
2088     loc = 1285 : TRUE;
2089     loc = 1287 : FALSE;
2090     TRUE : INLINED_TIMER_PULSEOFF__Q;
2091   esac;
2092   init(INLINED_TIMER_PULSEOFF__ET) := Osd32_0;
2093   next(INLINED_TIMER_PULSEOFF__ET) := case
2094     loc = end : Osd32_0;

```

```

2095     loc = 1217 : (INLINED_TIMER_PULSEOFF__PT - (
                INLINED_TIMER_PULSEOFF__DUE - main.__GLOBAL_TIME));
2096     loc = 1220 : INLINED_TIMER_PULSEOFF__PT;
2097     loc = 1221 : Osd32_0;
2098     loc = 1248 : (INLINED_TIMER_PULSEOFF__PT - (
                INLINED_TIMER_PULSEOFF__DUE - main.__GLOBAL_TIME));
2099     loc = 1251 : INLINED_TIMER_PULSEOFF__PT;
2100     loc = 1252 : Osd32_0;
2101     loc = 1286 : (INLINED_TIMER_PULSEOFF__PT - (
                INLINED_TIMER_PULSEOFF__DUE - main.__GLOBAL_TIME));
2102     loc = 1289 : INLINED_TIMER_PULSEOFF__PT;
2103     loc = 1290 : Osd32_0;
2104     TRUE : INLINED_TIMER_PULSEOFF__ET;
2105     esac;
2106     init(INLINED_TIMER_PULSEOFF__OLD_IN) := FALSE;
2107     next(INLINED_TIMER_PULSEOFF__OLD_IN) := case
2108         loc = end : FALSE;
2109         loc = 1222 : INLINED_TIMER_PULSEOFF__IN;
2110         loc = 1253 : INLINED_TIMER_PULSEOFF__IN;
2111         loc = 1291 : INLINED_TIMER_PULSEOFF__IN;
2112         TRUE : INLINED_TIMER_PULSEOFF__OLD_IN;
2113     esac;
2114     init(INLINED_TIMER_PULSEOFF__DUE) := Osd32_0;
2115     next(INLINED_TIMER_PULSEOFF__DUE) := case
2116         loc = end : Osd32_0;
2117         loc = 1214 : (main.__GLOBAL_TIME + INLINED_TIMER_PULSEOFF__PT);
2118         loc = 1245 : (main.__GLOBAL_TIME + INLINED_TIMER_PULSEOFF__PT);
2119         loc = 1283 : (main.__GLOBAL_TIME + INLINED_TIMER_PULSEOFF__PT);
2120         TRUE : INLINED_TIMER_PULSEOFF__DUE;
2121     esac;
2122     init(INLINED_R_EDGE_20__NEW) := FALSE;
2123     next(INLINED_R_EDGE_20__NEW) := case
2124         loc = end : FALSE;
2125         loc = 1256 : PULSEON;
2126         TRUE : INLINED_R_EDGE_20__NEW;
2127     esac;
2128     init(INLINED_R_EDGE_20__OLD) := FALSE;
2129     next(INLINED_R_EDGE_20__OLD) := case
2130         loc = end : FALSE;
2131         loc = 1255 : RE_PULSEON_OLD;
2132         loc = 1259 : TRUE;
2133         loc = 1261 : INLINED_R_EDGE_20__NEW;
2134         TRUE : INLINED_R_EDGE_20__OLD;
2135     esac;
2136     init(INLINED_F_EDGE_21__NEW) := FALSE;
2137     next(INLINED_F_EDGE_21__NEW) := case
2138         loc = end : FALSE;
2139         loc = 1264 : PULSEON;
2140         TRUE : INLINED_F_EDGE_21__NEW;
2141     esac;
2142     init(INLINED_F_EDGE_21__OLD) := FALSE;
2143     next(INLINED_F_EDGE_21__OLD) := case
2144         loc = end : FALSE;
2145         loc = 1263 : FE_PULSEON_OLD;
2146         loc = 1267 : FALSE;
2147         loc = 1269 : INLINED_F_EDGE_21__NEW;
2148         TRUE : INLINED_F_EDGE_21__OLD;
2149     esac;

```

```

2150     init(INLINED_R_EDGE_22__NEW) := FALSE;
2151     next(INLINED_R_EDGE_22__NEW) := case
2152         loc = end : FALSE;
2153         loc = 1272 : PULSEOFF;
2154         TRUE : INLINED_R_EDGE_22__NEW;
2155     esac;
2156     init(INLINED_R_EDGE_22__OLD) := FALSE;
2157     next(INLINED_R_EDGE_22__OLD) := case
2158         loc = end : FALSE;
2159         loc = 1271 : RE_PULSEOFF_OLD;
2160         loc = 1275 : TRUE;
2161         loc = 1277 : INLINED_R_EDGE_22__NEW;
2162         TRUE : INLINED_R_EDGE_22__OLD;
2163     esac;
2164     init(INLINED_R_EDGE_25__NEW) := FALSE;
2165     next(INLINED_R_EDGE_25__NEW) := case
2166         loc = end : FALSE;
2167         loc = 1341 : OUTOVST_AUX;
2168         TRUE : INLINED_R_EDGE_25__NEW;
2169     esac;
2170     init(INLINED_R_EDGE_25__OLD) := FALSE;
2171     next(INLINED_R_EDGE_25__OLD) := case
2172         loc = end : FALSE;
2173         loc = 1340 : RE_OUTOVST_AUX_OLD;
2174         loc = 1344 : TRUE;
2175         loc = 1346 : INLINED_R_EDGE_25__NEW;
2176         TRUE : INLINED_R_EDGE_25__OLD;
2177     esac;
2178     init(INLINED_F_EDGE_26__NEW) := FALSE;
2179     next(INLINED_F_EDGE_26__NEW) := case
2180         loc = end : FALSE;
2181         loc = 1349 : OUTOVST_AUX;
2182         TRUE : INLINED_F_EDGE_26__NEW;
2183     esac;
2184     init(INLINED_F_EDGE_26__OLD) := FALSE;
2185     next(INLINED_F_EDGE_26__OLD) := case
2186         loc = end : FALSE;
2187         loc = 1348 : FE_OUTOVST_AUX_OLD;
2188         loc = 1352 : FALSE;
2189         loc = 1354 : INLINED_F_EDGE_26__NEW;
2190         TRUE : INLINED_F_EDGE_26__OLD;
2191     esac;
2192     init(INLINED_TIMER_WARNING__PT) := Osd32_0;
2193     next(INLINED_TIMER_WARNING__PT) := case
2194         loc = end : Osd32_0;
2195         loc = 1360 : PONOFF.PWDT;
2196         TRUE : INLINED_TIMER_WARNING__PT;
2197     esac;
2198     init(INLINED_TIMER_WARNING__IN) := FALSE;
2199     next(INLINED_TIMER_WARNING__IN) := case
2200         loc = end : FALSE;
2201         loc = 1361 : POSW_AUX;
2202         TRUE : INLINED_TIMER_WARNING__IN;
2203     esac;
2204     init(INLINED_TIMER_WARNING__Q) := FALSE;
2205     next(INLINED_TIMER_WARNING__Q) := case
2206         loc = end : FALSE;
2207         loc = 1363 : FALSE;

```

```

2208     loc = 1373 : TRUE;
2209     TRUE : INLINED_TIMER_WARNING__Q;
2210     esac;
2211     init(INLINED_TIMER_WARNING__ET) := Osd32_0;
2212     next(INLINED_TIMER_WARNING__ET) := case
2213         loc = end : Osd32_0;
2214         loc = 1364 : Osd32_0;
2215         loc = 1369 : Osd32_0;
2216         loc = 1372 : (main.__GLOBAL_TIME - INLINED_TIMER_WARNING__START);
2217         loc = 1374 : INLINED_TIMER_WARNING__PT;
2218         TRUE : INLINED_TIMER_WARNING__ET;
2219     esac;
2220     init(INLINED_TIMER_WARNING__RUNNING) := FALSE;
2221     next(INLINED_TIMER_WARNING__RUNNING) := case
2222         loc = end : FALSE;
2223         loc = 1365 : FALSE;
2224         loc = 1368 : TRUE;
2225         TRUE : INLINED_TIMER_WARNING__RUNNING;
2226     esac;
2227     init(INLINED_TIMER_WARNING__START) := Osd32_0;
2228     next(INLINED_TIMER_WARNING__START) := case
2229         loc = end : Osd32_0;
2230         loc = 1367 : main.__GLOBAL_TIME;
2231         TRUE : INLINED_TIMER_WARNING__START;
2232     esac;
2233     init(INLINED_DETECT_EDGE_28__NEW) := FALSE;
2234     next(INLINED_DETECT_EDGE_28__NEW) := case
2235         loc = end : FALSE;
2236         loc = 1438 : ALUNACK;
2237         TRUE : INLINED_DETECT_EDGE_28__NEW;
2238     esac;
2239     init(INLINED_DETECT_EDGE_28__OLD) := FALSE;
2240     next(INLINED_DETECT_EDGE_28__OLD) := case
2241         loc = end : FALSE;
2242         loc = 1437 : ALUNACK_OLD;
2243         loc = 1445 : INLINED_DETECT_EDGE_28__NEW;
2244         TRUE : INLINED_DETECT_EDGE_28__OLD;
2245     esac;
2246     init(INLINED_DETECT_EDGE_28__RE) := FALSE;
2247     next(INLINED_DETECT_EDGE_28__RE) := case
2248         loc = end : FALSE;
2249         loc = 1436 : RE_ALUNACK;
2250         loc = 1441 : TRUE;
2251         loc = 1443 : FALSE;
2252         loc = 1446 : FALSE;
2253         TRUE : INLINED_DETECT_EDGE_28__RE;
2254     esac;
2255     init(INLINED_DETECT_EDGE_28__FE) := FALSE;
2256     next(INLINED_DETECT_EDGE_28__FE) := case
2257         loc = end : FALSE;
2258         loc = 1435 : FE_ALUNACK;
2259         loc = 1442 : FALSE;
2260         loc = 1444 : TRUE;
2261         loc = 1447 : FALSE;
2262         TRUE : INLINED_DETECT_EDGE_28__FE;
2263     esac;
2264
2265 MODULE main

```

```

2266 VAR
2267     interaction : {NONE };
2268     INSTANCE : module_INSTANCE(interaction, self);
2269     random_random_t_cycle : unsigned word[8];
2270     random_r53 : signed word[32];
2271     random_r54 : signed word[32];
2272     __GLOBAL_TIME : signed word[32];
2273     T_CYCLE : unsigned word[16];
2274     CPC_DB_VERSION.BASELINE_VERSION : signed word[32];
2275
2276     INVAR (
2277         (INSTANCE.loc != initial -> random_r53 = 0sd32_0) &
2278         (INSTANCE.loc != initial -> random_r54 = 0sd32_0)
2279     ); --
2280
2281     ASSIGN
2282     init(__GLOBAL_TIME) := 0sd32_0;
2283     next(__GLOBAL_TIME) := case
2284         INSTANCE.loc = initial : (__GLOBAL_TIME + signed(extend(T_CYCLE, 16)))
2285         );
2286         TRUE : __GLOBAL_TIME;
2287     esac;
2288     init(T_CYCLE) := 0ud16_0;
2289     next(T_CYCLE) := case
2290         INSTANCE.loc = end : (((extend(random_random_t_cycle,8))) mod 0
2291             ud16_116 + 0ud16_5);
2292         TRUE : T_CYCLE;
2293     esac;
2294     init(CPC_DB_VERSION.BASELINE_VERSION) := 0sd32_660;
2295     next(CPC_DB_VERSION.BASELINE_VERSION) := case
2296         TRUE : CPC_DB_VERSION.BASELINE_VERSION;
2297     esac;
2298
2299     DEFINE
2300     PLC_START := (INSTANCE.loc = 10);
2301     PLC_END := (INSTANCE.loc = end);

```

Listing B.1: NuSMV model for the OnOff UNICOS object

The following piece of NuSMV code corresponds to the automatically generated abstract model from the QSDN UNICOS CPC program for the first requirement in Section 4.4.2. This abstract model is enough to verify the following safety property  $p$ :

$$\text{AG} \left( (EoC \wedge \text{QSDN\_4\_DN1CT\_SEQ\_DB.Stop.x}) \right. \\
 \left. \rightarrow \text{QSDN\_4\_1PV408.AuOffR} \right)$$

```

1 -- V1.1
2 -- Generated model from QSDN.scl by bfernand
3
4 MODULE module_INSTANCE(interaction, main)
5     VAR

```

```

6   loc : {initial, end, l1};
7   ASSIGN
8     init(loc) := initial;
9     next(loc) := case
10      loc = end : initial;
11      loc = initial : l1;
12      loc = l1 : end;
13      TRUE: loc;
14    esac;
15
16  MODULE main
17    VAR
18      interaction : {NONE };
19      INSTANCE : module_INSTANCE(interaction, self);
20      QSDN_4_DN1CT_SEQ_DB.STOP.X# : boolean;
21      QSDN_4_1PV408.AUONR : boolean;
22      QSDN_4_1PV408.AUOFFR : boolean;
23
24    ASSIGN
25      init(QSDN_4_DN1CT_SEQ_DB.STOP.X#) := FALSE;
26      next(QSDN_4_DN1CT_SEQ_DB.STOP.X#) := case
27        INSTANCE.loc = initial : {TRUE, FALSE};
28        TRUE : QSDN_4_DN1CT_SEQ_DB.STOP.X#;
29      esac;
30      init(QSDN_4_1PV408.AUONR) := FALSE;
31      next(QSDN_4_1PV408.AUONR) := case
32        INSTANCE.loc = initial : {TRUE, FALSE};
33        TRUE : QSDN_4_1PV408.AUONR;
34      esac;
35      init(QSDN_4_1PV408.AUOFFR) := FALSE;
36      next(QSDN_4_1PV408.AUOFFR) := case
37        INSTANCE.loc = l1 : !(QSDN_4_1PV408.AUONR);
38        TRUE : QSDN_4_1PV408.AUOFFR;
39      esac;
40
41    DEFINE
42      PLC_START := (INSTANCE.loc = l1);
43      PLC_END := (INSTANCE.loc = end);
44
45    INVAR
46      !(( QSDN_4_DN1CT_SEQ_DB.STOP.X# = TRUE ) & ( QSDN_4_1PV408.AUONR = TRUE )
47        );
48  CTLSPEC AG((PLC_END) -> (QSDN_4_DN1CT_SEQ_DB.STOP.X# -> QSDN_4_1PV408.
    AUOFFR)) ;

```

Listing B.2: Abstract model  $AM'_1$ , where  $p$  is verified

The following piece of NuSMV code corresponds to the automatically generated abstract model from the QSDN UNICOS CPC program for the second requirement in Section 4.4.2. This abstract model

is enough to verify the following safety property  $p$ :

$$\text{AG} \left( (EoC \wedge \text{QSDN\_4\_DN1CT\_SEQ\_DB.Run.x}) \rightarrow \text{QSDN\_4\_1PV408.AuOnR} \right)$$

```

1  -- V1.1
2  -- Generated model from qsdn_deopt.scl by bfernand
3
4  MODULE module_INSTANCE(interaction, main)
5  VAR
6    loc : {initial, end, l1};
7
8  ASSIGN
9    init(loc) := initial;
10   next(loc) := case
11     loc = end : initial;
12     loc = initial : l1;
13     loc = l1 : end;
14     TRUE: loc;
15   esac;
16
17  MODULE main
18  VAR
19    interaction : {NONE };
20    INSTANCE : module_INSTANCE(interaction, self);
21    QSDN_4_DN1CT_SEQ_DB.VALVESON.X# : boolean;
22    QSDN_4_DN1CT_SEQ_DB.RUN.X# : boolean;
23    QSDN_4_DN1CT_SEQ_DB.OKSIGNALOFF.X# : boolean;
24    QSDN_4_1PV408.AUONR : boolean;
25
26  ASSIGN
27    init(QSDN_4_DN1CT_SEQ_DB.VALVESON.X#) := FALSE;
28    next(QSDN_4_DN1CT_SEQ_DB.VALVESON.X#) := case
29      INSTANCE.loc = initial : {TRUE, FALSE};
30      TRUE : QSDN_4_DN1CT_SEQ_DB.VALVESON.X#;
31    esac;
32    init(QSDN_4_DN1CT_SEQ_DB.RUN.X#) := FALSE;
33    next(QSDN_4_DN1CT_SEQ_DB.RUN.X#) := case
34      INSTANCE.loc = initial : {TRUE, FALSE};
35      TRUE : QSDN_4_DN1CT_SEQ_DB.RUN.X#;
36    esac;
37    init(QSDN_4_DN1CT_SEQ_DB.OKSIGNALOFF.X#) := FALSE;
38    next(QSDN_4_DN1CT_SEQ_DB.OKSIGNALOFF.X#) := case
39      INSTANCE.loc = initial : {TRUE, FALSE};
40      TRUE : QSDN_4_DN1CT_SEQ_DB.OKSIGNALOFF.X#;
41    esac;
42    init(QSDN_4_1PV408.AUONR) := FALSE;
43    next(QSDN_4_1PV408.AUONR) := case
44      INSTANCE.loc = l1 : ((QSDN_4_DN1CT_SEQ_DB.VALVESON.X# |
45        QSDN_4_DN1CT_SEQ_DB.RUN.X#) | QSDN_4_DN1CT_SEQ_DB.OKSIGNALOFF.X#);
46      TRUE : QSDN_4_1PV408.AUONR;
47    esac;
48  DEFINE
49    PLC_START := (INSTANCE.loc = l1);

```

```

50  PLC_END := (INSTANCE.loc = end);
51
52  CTLSPEC AG((PLC_END) -> (QSDN_4_DN1CT_SEQ_DB.RUN.X# -> QSDN_4_1PV408.AUONR)
    ) ;

```

Listing B.3: Abstract model  $AM'_1$ , where  $p$  is verified

The following piece of NuSMV code corresponds to the automatically generated abstract model from the QSDN UNICOS CPC program for the third requirement in Section 4.4.2. This abstract model is enough to verify the following safety property  $p$ :

$$\text{AG} \left( (EoC \wedge \text{QSDN\_4\_DN1CT\_SEQ\_DB.Run.x}) \right. \\
 \left. \rightarrow \text{QSDN\_4\_1PV408.AuOffr} \right)$$

```

1  -- V1.1
2  -- Generated model from QSDN.scl by bfernand
3
4  MODULE module_INSTANCE(interaction, main)
5  VAR
6    loc : {initial, end, l1};
7
8  ASSIGN
9    init(loc) := initial;
10   next(loc) := case
11     loc = end : initial;
12     loc = initial : l1;
13     loc = l1 : end;
14     TRUE: loc;
15   esac;
16
17  MODULE main
18  VAR
19    interaction : {NONE };
20    INSTANCE : module_INSTANCE(interaction, self);
21    QSDN_4_DN1CT_SEQ_DB.VALVESON.X# : boolean;
22    QSDN_4_DN1CT_SEQ_DB.RUN.X# : boolean;
23    QSDN_4_DN1CT_SEQ_DB.OKSIGNALOFF.X# : boolean;
24    QSDN_4_1PV408.AUOFFR : boolean;
25
26
27  ASSIGN
28    init(QSDN_4_DN1CT_SEQ_DB.VALVESON.X#) := FALSE;
29    next(QSDN_4_DN1CT_SEQ_DB.VALVESON.X#) := case
30      INSTANCE.loc = initial : {TRUE, FALSE};
31      TRUE : QSDN_4_DN1CT_SEQ_DB.VALVESON.X#;
32    esac;
33    init(QSDN_4_DN1CT_SEQ_DB.RUN.X#) := FALSE;
34    next(QSDN_4_DN1CT_SEQ_DB.RUN.X#) := case
35      INSTANCE.loc = initial : {TRUE, FALSE};
36      TRUE : QSDN_4_DN1CT_SEQ_DB.RUN.X#;
37    esac;

```

```

38  init(QSDN_4_DN1CT_SEQ_DB.OKSIGNALOFF.X#) := FALSE;
39  next(QSDN_4_DN1CT_SEQ_DB.OKSIGNALOFF.X#) := case
40    INSTANCE.loc = initial : {TRUE, FALSE};
41    TRUE : QSDN_4_DN1CT_SEQ_DB.OKSIGNALOFF.X#;
42  esac;
43  init(QSDN_4_1PV408.AUOFFR) := FALSE;
44  next(QSDN_4_1PV408.AUOFFR) := case
45    INSTANCE.loc = l1 : (!((QSDN_4_DN1CT_SEQ_DB.VALVESON.X# |
46      QSDN_4_DN1CT_SEQ_DB.RUN.X#) | QSDN_4_DN1CT_SEQ_DB.OKSIGNALOFF.X#));
47    TRUE : QSDN_4_1PV408.AUOFFR;
48  esac;
49  DEFINE
50    PLC_START := (INSTANCE.loc = l1);
51    PLC_END := (INSTANCE.loc = end);
52
53  CTLSPEC AG((PLC_END) -> (QSDN_4_DN1CT_SEQ_DB.RUN.X# -> ! QSDN_4_1PV408.
54    AUOFFR)) ;
-- Original property p: CTLSPEC AG((PLC_END) -> (QSDN_4_DN1CT_SEQ_DB.RUN.X#
  -> QSDN_4_1PV408.AUOFFR)) ;

```

Listing B.4: Abstract model  $AM'_2$ , where  $p$  is verified

# List of Acronyms

**AM:** Abstract Model

**API:** Application Programming Interface

**AST:** Abstract Syntax Tree

**BE/CO:** Beams department / Controls group at CERN

**BDD:** Binary Decision Diagram

**BIP:** Behavior, Interaction and Priority

**CEGAR:** Counterexample guided Abstraction Refinement

**CENELEC:** European Committee for Electrotechnical Standardization

**CERN:** International Electrotechnical Commission

**CDD:** Clock Difference Diagram

**CFG:** Control Flow Graph

**COI:** Cone of Influence

**CORBA:** Common Object Request Broker Architecture

**CPC:** Continuous Process Control

**CTL:** Computation Tree Logic

**DB:** Data Block

**DCS:** Distributed Control System

- DESY:** Deutsches Elektronen-Synchrotron
- DIEECS:** Electric, Electronics, Computers and Systems Engineering
- EMF:** Eclipse Modeling Framework
- EN/ICE:** Engineering department / Industrial Controls & Engineering group at CERN
- EoC:** End of Cycle
- EPICS:** Experimental Physics and Industrial Controls
- ERP:** Enterprise Resource Planning
- ESRF:** European Synchrotron Radiation Facility
- FAT:** Factory Acceptance Test
- FB:** Function Block
- FBD:** Function Block Diagram
- FC:** Function
- FESA:** Front-End Software Architecture
- FPGA:** Field-Programmable Gate Array
- FSM:** Finite State Machine
- GRAFCET:** Graphe Fonctionnel de Croissant Étape Transition
- HVAC:** Heating, Ventilating, and Air Conditioning
- IEC:** International Electrotechnical Commission
- IL:** Instruction List
- IIM:** Input Image Memory
- IM:** Intermediate Model
- IOC:** Input-Output Controllers

**ISA:** International Electrotechnical Commission or System Engineering and Automation

**ISO:** International Organization for Standardization

**LEP:** Large Electron-Positron Collider

**LHC:** Large Hadron Collider

**LINAC:** LINear ACcelerators

**LTL:** Linear Temporal Logic

**LTS:** Labelled Transition System

**M:** Memory

**MDD:** Multivalued Decision Diagram

**MES:** Manufacturing Execution System

**NR:** Not Running

**OB:** Organization Block

**OIM:** Output Image Memory

**OM:** Original Model

**OPI:** Operator Interface

**PC:** Personal Computer

**PID:** Proportional Integral Derivative

**PLC:** Programmable Logic Controller

**PLTL:** Propositional Linear Temporal Logic

**PS:** Proton Synchrotron

**PSS:** Potential State Space

**PT:** Predefined delay

**R:** Running

**RiSD:** Rigorous System Design

**RMIT:** Royal Melbourne Institute of Technology

**RSS:** Reachable State Space

**SAT:** Satisfiability or Site Acceptance Test

**SCADA:** Supervisory Control and Data Acquisition

**SCL:** Structured Control Language

**SFC:** Sequential Function Chart

**SIF:** Safety Instrumented Function

**SIL:** Safety Integrity Level

**SIS:** Safety Instrumented System

**SMT:** Satisfiability Modulo Theories

**SPS:** Super Proton Synchrotron

**ST:** Structured Text

**STS:** State Transition System

**TCTL:** Timed Computation Tree Logic

**TO:** Time Out

**TON:** Timer On-delay

**TOFF:** Timer Off-delay

**TP:** Pulse Timer

**UNICOS:** Unified Industrial Control System

# List of Figures

1.1	CERN accelerator complex from <a href="http://en.wikipedia.org/wiki/CERN">http://en.wikipedia.org/wiki/CERN</a> . . . . .	4
2.1	Control system layers . . . . .	15
2.2	IEC 61499 FB representation . . . . .	19
2.3	EPICS architecture . . . . .	21
2.4	EPICS control system example . . . . .	21
2.5	TANGO architecture . . . . .	22
2.6	Example SFC . . . . .	28
2.7	Example SFC (Screenshot from SIMATIC tool by Siemens) . . . . .	29
2.8	Timer diagrams . . . . .	31
2.9	Consequences of finite time representation . . . . .	32
2.10	Verification schema . . . . .	36
2.11	LTL operators representation . . . . .	39
2.12	CTL operators representation . . . . .	40
3.1	Traditional PLC program development . . . . .	60
3.2	Methodology overview . . . . .	62
3.3	Two main approaches for the verification and design of PLC programs. . . . .	66
3.4	Integration of the proposed methodology in the PLC program development process. . . . .	67
3.5	Example of IM . . . . .	71
3.6	Example of representing an interactive automata network as a STS . . . . .	73
3.7	Example of IM . . . . .	80
3.8	Example of IM with one PLC interrupt . . . . .	84
3.9	FC – IM transformation . . . . .	87

3.10	FB instance – IM transformation . . . . .	87
3.11	FC call – IM transformation . . . . .	89
3.12	Corresponding automaton fragment for variable assignment . . . . .	90
3.13	Corresponding automaton fragment for for conditional statement . . . . .	91
3.14	Example of transformation from a FC call into the IM . . . . .	92
3.15	Example SFC (Screenshot from SIMATIC tool by Siemens) . . . . .	95
3.16	IM representation of the SFC example . . . . .	95
3.17	Reduction techniques applied to the IM and to the verification tool models . . . . .	97
3.18	Example for control flow graph representation of an ST code in NuSMV . . . . .	99
3.19	Example comparing our and the NuSMV’s cone of influence algorithm from Darvas et al. (2014) . . . . .	104
3.20	Example reduction rules from Darvas et al. (2014) . . . . .	106
3.21	Example CFG after our COI and the reductions from Darvas et al. (2014) . . . . .	107
3.22	Overview of the reduction workflow . . . . .	109
3.23	Verification cases for $\text{AG}(\alpha \rightarrow \beta)$ in abstract models . . . . .	110
3.24	Verification cases for $\text{EF}(\gamma \& \theta)$ in abstract models . . . . .	111
3.25	Variable dependency graph example . . . . .	113
3.26	Variable Abstraction Flow Diagram. Verification results: [1] $p$ holds on $OM'$ ; [2] $p$ does not hold on $OM'$ ; [3] no answer. . . . .	114
3.27	Variable Abstraction Flow Diagram. Verification results: [1] $p$ holds on $OM'$ ; [2] $p$ does not hold on $OM'$ ; [3] no answer. . . . .	116
3.28	Relationship between the verification cases for $\text{AG}(\alpha \rightarrow \neg\beta)$ and $\text{AG}(\alpha \rightarrow \beta)$ in abstract models . . . . .	119
3.29	Relationship between the abstract models and the maximum number of possible invariants for the variable abstraction technique . . . . .	123
3.30	Example of IM . . . . .	125
3.31	Variable dependency graph of the original model corresponding to the PLC program example . . . . .	125

3.32	Variable dependency graph of the abstract model corresponding to the PLC program example . . . . .	126
3.33	Abstraction strategy applied to the example . . . . .	127
3.34	Graphical representation as a FSM . . . . .	132
3.35	Automaton fragment for a transition . . . . .	134
3.36	Automaton fragment for a variable assignment . . . . .	134
3.37	Automaton fragment for an interaction . . . . .	134
3.38	Corresponding IM . . . . .	135
3.39	Screenshot of the UPPAAL model . . . . .	138
3.40	Automaton fragment for a transition . . . . .	141
3.41	Automaton fragment for a variable assignment . . . . .	142
3.42	Automaton fragments for a synchronization . . . . .	143
3.43	Corresponding IM . . . . .	144
3.44	BIP model with two components . . . . .	149
3.45	Automaton fragment for a transition . . . . .	154
3.46	Automaton fragment for a variable assignment . . . . .	155
3.47	Automaton fragments for a synchronization . . . . .	155
3.48	Corresponding IM . . . . .	156
3.49	Timer diagrams . . . . .	162
3.50	Consequences of finite time representation . . . . .	164
3.51	State machine of the realistic TON representation . . . . .	165
3.52	State machine of the realistic TOFF representation . . . . .	167
3.53	State machine of the realistic TP representation . . . . .	168
3.54	Verification configuration for OnOff model extended with monitor . . . . .	170
3.55	State machine of the abstract TON representation . . . . .	171
3.56	Timing diagram of TON modeled using different approaches . . . . .	172
3.57	State machine of the abstract TOFF representation . . . . .	173
3.58	State machine of the abstract TP representation . . . . .	174
3.59	Integration of the proposed methodology on the PLC program development process. . . . .	180
3.60	EMF implementation . . . . .	181
3.61	CASE tool editor for the PLC code . . . . .	182
3.62	CASE tool property description . . . . .	182
3.63	CASE tool property formalization by using patterns . . . . .	183
3.64	CASE tool verification report . . . . .	183

4.1	Different UNICOS packages . . . . .	186
4.2	UNICOS CPC control system architecture . . . . .	187
4.3	Mapping between the process instrumentation and the UNICOS objects example . . . . .	189
4.4	UNICOS PLC code architecture . . . . .	190
4.5	OnOff signals interface . . . . .	193
4.6	State machine of the OnOff mode manager from the UNICOS specifications . . . . .	195
4.7	Verification configuration for OnOff model extended with monitor . . . . .	202
4.8	QSDN process . . . . .	209
4.9	Variable dependency graph for the given requirement of QSDN . . . . .	212
4.10	Relationship between the abstract models and the max- imum number of reachability properties for the variable abstraction technique $m$ . . . . .	213
5.1	Measurements comparing the different COI solutions from Darvas et al. (2014) . . . . .	229
5.2	Relationship between the abstract models and the max- imum number of reachability properties for the variable abstraction technique . . . . .	232

# List of Tables

1.1	Software design and development: software architecture design (see Table 7.4.3 from the IEC 61508 (2010) standard) . . . . .	6
2.1	The Range of Formal Methods Options Summarized in Terms of (a) Levels of Formalization and (b) Scope of Formal Methods Use. . . . .	33
2.2	Related work . . . . .	53
2.3	Related work: Abstraction techniques . . . . .	55
2.4	Related work: Time modeling strategy . . . . .	57
3.1	Data type mapping . . . . .	85
3.2	Counterexample for $p$ on $AM'_1$ . . . . .	126
3.3	Counterexample for $p$ on $AM'_1 + 1$ invariant . . . . .	128
3.4	Counterexample for $p$ on $AM'_1 + 2$ invariants . . . . .	129
3.5	Variables values from the counterexample at the end of the PLC cycle . . . . .	177
4.1	Case Study metrics . . . . .	192
4.2	Model generation metrics . . . . .	198
4.3	Model generation metrics . . . . .	199
4.4	Verification run time for the first requirement from Fernández Adiego et al. (2014c) . . . . .	200
4.5	Verification run time for the second requirement . . . . .	201
4.6	State space of the OnOff model with realistic timer representation from Fernández Adiego et al. (2014a) . . . . .	203
4.7	Verification time on OnOff model with realistic time representation from Fernández Adiego et al. (2014a) . . . . .	204

4.8	State space of the OnOff model with abstract time representation from Fernández Adiego et al. (2014a) . . .	205
4.9	Verification time on OnOff model with abstract time representation from Fernández Adiego et al. (2014a) . .	205
4.10	Metrics of the models of QSDN . . . . .	211
4.11	Verification time on <i>QSDN</i> model . . . . .	214
4.12	Verification time on <i>QSDN</i> model . . . . .	215
4.13	Verification time on <i>QSDN</i> model . . . . .	216
5.1	Overview of the timer representation approaches from Fernández Adiego et al. (2014a) . . . . .	227
5.2	Requirement evaluation measurements with nuXmv COI from Darvas et al. (2014) . . . . .	230
5.3	Requirement evaluation measurements with our COI from Darvas et al. (2014) . . . . .	230

# Listings

2.1	Example of ST code . . . . .	27
2.2	Example of the textual representation of SFC code . . . . .	27
3.1	Variable assignment in ST code . . . . .	90
3.2	Conditional statement in ST code . . . . .	91
3.3	Example of ST code . . . . .	91
3.4	Example of the textual representation of SFC code . . . . .	94
3.5	Example of ST code . . . . .	124
3.6	NuSMV code example . . . . .	132
3.7	Representation of automata network $a$ in nuXmv . . . . .	132
3.8	Representation of automaton $a$ in nuXmv . . . . .	133
3.9	Next-value assignment statement . . . . .	133
3.10	Corresponding nuXmv code . . . . .	134
3.11	Corresponding nuXmv code . . . . .	134
3.12	Corresponding nuXmv code . . . . .	134
3.13	PLC ST code . . . . .	135
3.14	nuXmv code . . . . .	135
3.15	nuXmv model example . . . . .	136
3.16	UPPAAL code example . . . . .	139
3.17	Representation of automata network in UPPAAL . . . . .	140
3.18	Representation of automaton $a$ in UPPAAL . . . . .	140
3.19	Variables of automaton $a$ in UPPAAL . . . . .	141
3.20	Corresponding UPPAAL code . . . . .	141
3.21	Corresponding UPPAAL code . . . . .	142
3.22	Corresponding UPPAAL code . . . . .	143
3.23	PLC ST code . . . . .	144
3.24	Corresponding UPPAAL model . . . . .	144
3.25	Synchronization of two automaton in UPPAAL . . . . .	146
3.26	Representation of automata network in BIP . . . . .	150
3.27	Representation of automaton $a$ in BIP . . . . .	151

3.28	Representation of automaton $a$ in BIP . . . . .	153
3.29	Corresponding BIP code . . . . .	154
3.30	Corresponding BIP code . . . . .	155
3.31	Corresponding BIP code . . . . .	155
3.32	PLC ST code . . . . .	156
3.33	Corresponding BIP model . . . . .	156
3.34	BIP example . . . . .	158
3.35	ST code of TON . . . . .	164
3.36	ST code of TOFF . . . . .	166
3.37	ST code of TP . . . . .	167
3.38	ST code example . . . . .	175
3.39	Counterexample fragment . . . . .	179
3.40	Corresponding ST code demonstrator . . . . .	179
3.41	Extract of the source code causing the violation of the requirement . . . . .	179
4.1	OnOff mode manager program . . . . .	194
4.2	Extract from the counterexample . . . . .	206
4.3	Extract from the PLC demonstrator . . . . .	206
4.4	Extract of the source code causing the violation of the requirement . . . . .	207
4.5	Excerpt of QSDN code relevant to the case study . . .	210
A.1	OnOff PLC program . . . . .	259
B.1	NuSMV model for the OnOff UNICOS object . . . . .	277
B.2	Abstract model $AM'_1$ , where $p$ is verified . . . . .	320
B.3	Abstract model $AM'_1$ , where $p$ is verified . . . . .	322
B.4	Abstract model $AM'_2$ , where $p$ is verified . . . . .	323

# Bibliography

- Abdellatif, T., Bensalem, S., Combaz, J., de Silva, L., and Ingrand, F. (2012). Rigorous design of robot software: A formal component-based approach. *Robotics and Autonomous Systems*, 60(12):1563–1578.
- Accelerator and Operations Technology website (2014). <http://www.lanl.gov/orgs/aot/>.
- Advanced Photon Source website (2014). <http://www.aps.anl.gov/>.
- ALBA website (2014). <https://www.cells.es/>.
- Amnell, T. et al. (2001). UPPAAL – now, next, and future. In *Modeling and Verification of Parallel Processes*. Springer.
- ARCADE.PLC website (2014). <http://arcade.embedded.rwth-aachen.de/doku.php?id=arcade.plc>.
- Bartha, T., Vörös, A., Jámbor, A., and Darvas, D. (2012). Verification of an industrial safety function using coloured Petri nets and model checking. In *Proc. of the 14th Int. Conf. on MITIP 2012*, pages 472–485. Hungarian Academy of Sciences, Computer and Automation Research Inst.
- Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T.-H., and Sifakis, J. (2011). Rigorous component-based system design using the BIP framework. *IEEE Sw.*, 28:41–48.
- Basu, A., Bensalem, S., Bozga, M., Delahaye, B., and Legay, A. (2012). Statistical abstraction and model-checking of large heterogeneous systems. *International Journal on Software Tools for Technology Transfer*, 14:53–72.

- Bauer, N., Engell, S., Huuck, R., Lohmann, S., Lukoschus, B., Remelhe, M., and Stursberg, O. (2004). Verification of PLC programs given as sequential function charts. In *Integration of Software Specification Techniques for Applications in Engineering*, pages 517–540. Springer.
- Behm, P., Benoit, P., Faivre, A., and Meynadier, J.-M. (1999). Météor: A successful application of B in a large project. In *Formal methods world congress*.
- Behrmann, G., David, A., and Larsen, K. G. (2004). A tutorial on UPPAAL. In *International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004. Revised Lectures*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–237. Springer.
- Bender, D. F., Combemale, B., Crégut, X., Farines, J.-M., Berthomieu, B., and Vernadat, F. (2008). Ladder metamodeling and PLC program validation through time Petri nets. In Schieferdecker, I. and Hartman, A., editors, *ECMDA-FA*, volume 5095 of *Lecture Notes in Computer Science*, pages 121–136. Springer.
- Bensalem, S., Griesmayer, A., Legay, A., Nguyen, T.-H., Sifakis, J., and Yan, R. (2011). D-finder 2: Towards efficient correctness of incremental design. In Bobaru, M., Havelund, K., Holzmann, G., and Joshi, R., editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 453–458. Springer Berlin Heidelberg.
- Bérard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., Schnoebelen, P., and McKenzie, P. (2001). *Systems and Software Verification. Model Checking Techniques and Tools*. Springer.
- Besalem, S., Bozga, M., Nguyen, T.-H., and Sifakis, J. (2010). Compositional verification for component-based systems and application. *IET Software*, 4:181–193.
- Biallas, S., Brauer, J., and Kowalewski, S. (2010). Counterexample-guided abstraction refinement for PLCs. In *Proc. of SSV'10*. USENIX Association.

- Biallas, S., Giacobbe, M., and Kowalewski, S. (2013). Predicate abstraction for programmable logic controllers. In Pecheur, C. and Dierkes, M., editors, *18th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2013)*, volume 8187 of *Lecture Notes in Computer Science*, pages 123–138. Springer Berlin Heidelberg.
- Biallas, S., Kowalewski, S., Stattelmann, S., and Schlich, B. (2014). Efficient handling of states in abstract interpretation of industrial programmable logic controller code. In *Proceedings of the 12th International Workshop on Discrete Event Systems*, pages 400–405, Cachan, France. IFAC.
- Biere, A., Cimatti, A., Clarke, E. M., Strichman, O., and Zhu, Y. (2003). Bounded model checking. *Advances in computers*, 58.
- BIP website (2014). <http://www-verimag.imag.fr/Rigorous-Design-of-Component-Based.html>.
- Blanco Viñuela, E., Merezhin, A., Bradu, B., Fernández Adiego, B., Willeman, D., Rochez, J., Beckers, J., Ortola Vidal, J., Durand, P., and Izquierdo Rosas, S. (2011). UNICOS evolution: CPC version 6. In *Proc. of 12th ICALEPCS*.
- Blech, J. O. and Biha, S. O. (2011). Verification of PLC properties based on formal semantics in Coq. In *Proc. of Int. Conf. on Software Engineering and Formal Methods*, volume 7041 of *Lecture Notes in Computer Science*, pages 58–73. Springer.
- Blech, J. O. and Grégoire, B. (2011). Certifying compilers using higher-order theorem provers as certificate checkers. *Formal Methods in System Design*, 38(1):33–61.
- Blech, J. O., Hattendorf, A., and Huang, J. (2011). An invariant preserving transformation for PLC models. In *Proc. of the 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW)*, pages 63–71.
- Blech, J. O. and Ould Biha, S. (2011). Verification of PLC properties based on formal semantics in Coq. *Software Engineering and Formal Methods*, pages 58–73.

- Bliudze, S. and Sifakis, J. (2007). The algebra of connectors — Structuring interaction in BIP. In *7th International Conference on Embedded Software (EMSOFT)*.
- Bowen, J. and Hinchey, M. (1995). Seven more myths of formal methods. *IEEE Software*, 12:34 – 41.
- Bozga, M., Graf, S., Mounier, L., and Ober, I. (2008). Modeling and verification of real time systems using the IF toolbox. In *Real Time Systems 1: Modeling and verification techniques*, volume 1 of *Traité IC2, série Informatique et systèmes d'information*, chapter 9. Hermes, Lavoisier.
- Brat, G., Drusinsky, D., Giannakopoulou, D., Goldberg, A., Havelund, K., Lowry, M., Pasareanu, C., Venet, A., Visser, W., and Washington, R. (2004). Experimental evaluation of verification and validation tools on martian rover software. *Formal Methods in System Design*, pages 167–198.
- Bruch, J., Clarke, E., McMillan, K., Dill, D., and Hwang, L. (1992). Symbolic model checking:  $10^{20}$  states and beyond. *Information and Computation*, pages 142–170.
- Campos, J. C. et al. (2008). Property patterns for the formal verification of automated production systems. In *Proc. of 17th IFAC World Congress*, pages 5107–5112. IFAC.
- Campos, J. C. and Machado, J. (2013). A specification patterns system for discrete event systems analysis. *International Journal of Advanced Robotic Systems*.
- Canet, G., Couffin, S., Lesage, J.-J., Petit, A., and Schnoebelen, P. (2000a). Towards the automatic verification of PLC programs written in Instruction List. In *IEEE Int. Conf. on Systems, Man, and Cybernetics*, volume 4, pages 2449–2454. IEEE.
- Canet, G., Couffin, S., Lesage, J.-J., Petit, A., and Schnoebelen, Ph. (2000b). Towards the automatic verification of PLC programs written in Instruction List. In *Proc. of the IEEE Int. Conf. on Systems, Man and Cybernetics*, pages 2449–2454. Argos Press.

- Casas-Cubillos, J., Gayet, P., Gomes, P., Sicard, C., Pezzeti, M., and Varas, F. (2002). Application of object-based industrial controls for cryogenics. In *European Accelerator Conference (EPAC)*.
- Cavada, R., Cimatti, A., Dorigatti, M., Mariotti, A., Micheli, A., Mover, S., Griggio, A., Roveri, M., and Tonetta, S. (2014). The nuXmv symbolic model checker. Technical report, Fondazione Bruno Kessler.
- Cavada, R., Cimatti, A., Jochim, C. A., Keighren, G., Olivetti, E., Pistore, M., Roveri, M., and Tchaltsev, A. (2011). *NuSMV 2.5 User Manual*. FBK-irst.
- CERN BE/CO group website (2014). <https://espace.cern.ch/be-dep/CO/default.aspx>.
- CERN EN/ICE group website (2014). <http://www.cern.ch/enice/>.
- Chaize, J.-M., Götz, A., Klotz, W.-D., Meyer, J., Perez, M., and Taurel, E. (1999). TANGO - an object oriented control system based on CORBA. In *Proc. of 8th ICALEPCS*.
- Chen, G., Song, X., and Gu, M. (2010). PLC program verification and analysis using the Coq theorem prover. *Peking University Journal*, 46(1):30–34.
- Clarke, E. (2008). The birth of model checking. In *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 1 – 26. Springer-Verlag.
- Clarke, E. and Emerson, E. (1982). Design and synthesis of synchronization skeletons using branching time temporal logic. In Kozen, D., editor, *Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer Berlin Heidelberg.
- Clarke, E., Grumberg, O., Jha, S., Lu, Y., and Veith, H. (2000). Counterexample-guided abstraction refinement. In *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer.
- Clarke, E. M., Grumberg, O., and Peled, D. A. (1999). *Model Checking*. The MIT Press.

- Clarke, E. M. and Wing, J. M. (1996). Formal methods: State of the art and future directions. *ACM Comput. Surv.*, 28(4):626–643.
- Cooper, K. D. and Torczon, L. (2012). *Engineering a Compiler*. Morgan Kaufmann Publishers Inc., second edition.
- Copy, B., Blanco Viñuela, E., Fernández Adiego, B., Nogueira Fernandes, R., and Barreiro, P. (2011). Model oriented application generation for industrial control systems. In *Proc. of 12th ICALEPCS*.
- Coq website (2014). <http://coq.inria.fr/>.
- CORBA website (2014). <http://www.corba.org/>.
- Coupat, R., Meslay, M., Burette, M.-A., Riera, B., A., P., and Anebicque, D. (2014). Standardization and safety control generation for SNCF systems engineer. In *Proc. of IFAC World Congress*.
- Dalesio, L., Kraimer, M., and Kozubal, A. (1991). EPICS architecture. In *Proc. of 4th ICALEPCS*.
- Darvas, D., Fernández Adiego, B., and Blanco Viñuela, E. (2013). Transforming PLC programs into formal models for verification purposes. Internal Note CERN-ACC-NOTE-2013-0040, CERN.
- Darvas, D., Fernández Adiego, B., Vörös, A., Bartha, T., Blanco Viñuela, E., and González Suárez, V. M. (2014). Formal verification of complex properties on PLC programs. In Ábrahám, E. and Palamidessi, C., editors, *Formal Techniques for Distributed Objects, Components, and Systems*, volume 8461 of *Lecture Notes in Computer Science*, pages 284–299. Springer.
- DESY website (2014). <http://www.desy.de/>.
- EPICS website (2014). <http://www.aps.anl.gov/epics/>.
- ESRF website (2014). <http://www.esrf.eu/>.
- European Committee for Electrotechnical Standardization website (2014). <http://www.cenelec.eu/>.

- Eysholdt, M. and Behrens, H. (2010). Xtext: Implement your language faster than the quick and dirty way. In *SPLASH/OOPSLA Companion*, pages 307–309. ACM.
- Faber, J. and Meyer, R. (2006). Model checking data-dependent real-time properties of the european train control system. In *Proc. of Formal Methods in Computer Aided Design*, pages 76 – 77. IEEE.
- Fernández Adiego, B., Blanco Viñuela, E., and Barreiro, P. (2011). UNICOS CPC6: Automated code generation for process control applications. In *Proc. of 12th ICALEPCS*.
- Fernández Adiego, B., Blanco Viñuela, E., and Merezhin, A. (2013a). Testing & verification of PLC code for process control. In *Proc. of 13th ICALEPCS*.
- Fernández Adiego, B., Blanco Viñuela, E., Tournier, J.-C., González Suárez, V. M., and Bliudze, S. (2013b). Model-based automated testing of critical PLC programs. In *11th IEEE Int. Conf. on Industrial Informatics*, pages 722–727.
- Fernández Adiego, B., Darvas, D., Blanco Viñuela, E., Tournier, J.-C., González Suárez, V. M., and Blech, J. O. (2014a). Modelling and formal verification of timing aspects in large PLC programs. In *Proc. of IFAC World Congress*.
- Fernández Adiego, B., Darvas, D., Tournier, J.-C., Blanco Viñuela, E., Blech, J. O., and González Suárez, V. M. (2014b). Automated generation of formal models from ST control programs for verification purposes. Internal Note CERN-ACC-NOTE-2014-0037, CERN.
- Fernández Adiego, B., Darvas, D., Tournier, J.-C., Blanco Viñuela, E., and González Suárez, V. M. (2014c). Bringing automated model checking to PLC program development – A CERN case study. In *Proc. of the 12th IFAC–IEEE International Workshop on Discrete Event Systems*.
- Flake, S., Müller, W., Pape, U., and Ruf, J. (2004). Specification and formal verification of temporal properties of production automation systems. In Ehrig, H., Damm, W., Desel, J., Große-Rhode, M., Reif, W., Schnieder, E., and Westkämper, E., editors, *Integration of*

- Software Specification Techniques for Applications in Engineering*, volume 3147 of *Lecture Notes in Computer Science*, pages 206–226. Springer Berlin Heidelberg.
- Formal Methods Europe website (2014). <http://www.fmeurope.org/>.
- Frey, G. and Litz, L. (2000). Formal methods in PLC programming. In *IEEE Int. Conf. on Systems, Man, and Cybernetics*, volume 4, pages 2431–2436. IEEE.
- Fukumoto, A., Hayashi, T., Nishikawa, H., Sakamoto, H., Tomizawa, T., and Yokomura, T. (1998). A verification and validation method and its application to digital safety systems in ABWR nuclear plants. *Nuclear Engineering and Design*, 183:117–132.
- Gayet, P. and Barillère, R. (2005). UNICOS a framework to build industry like control systems: Principles & methodology. In *10th ICALEPCS*.
- Gourcuff, V., de Smet, O., and Faure, J.-M. (2006). Efficient representation for formal verification of PLC programs. In *Proc. of 8th International Workshop on Discrete Event Systems*, pages 182–187.
- Gourcuff, V., de Smet, O., and Faure, J.-M. (2008). Improving large-sized PLC programs verification using abstractions. In *Proc. of 17th IFAC World Congress*.
- GSI website (2014). <https://www.gsi.de/>.
- Hall, A. (1990). Seven myths of formal methods. *IEEE Software*, 7:11–19.
- Hall, A. and Isaac, D. (1992). Formal methods in a real air traffic control project. In *IEEE Colloquium on Software in Air Traffic Control Systems - The Future*, pages 7/1–7/4.
- Huuch, R. (2003). *Software Verification for Programmable Logic Controllers*. PhD thesis, der Technischen Fakultät der Christian-Albrechts-Universität zu Kiel.

- IAEA (1999). Verification and validation of software related to nuclear power plant instrumentation and control. Technical report, International Atomic Energy agency.
- IEC 61131 (2013). IEC 61131: Programming languages for programmable logic controllers.
- IEC 61499 (2013). IEC 61499: Function blocks for embedded and distributed control systems design.
- IEC 61508 (2010). IEC 61508: Functional safety of electrical / electronic / programmable electronic safety-related systems.
- IEC 61511 (2003). IEC 61511: Functional safety - safety instrumented systems for the process industry sector.
- IEC 61512 (2009). IEC 61512: Batch control.
- IEC 62061 (2012). IEC 62061: Safety of machinery: Functional safety of electrical, electronic and programmable electronic control systems.
- IEEE 830 (1998). IEEE std 830: Recommended practice for software requirements specifications.
- International Electrotechnical Commission website (2014). <http://www.iec.ch/>.
- International Organization for Standardization website (2014). <https://www.iso.org/>.
- International Society of Automation website (2014). <https://www.isa.org/>.
- ISA 62381 (2010). Factory acceptance test (FAT), site acceptance test (SAT), and site integration test (SIT), automation systems in the process industry.
- ISA 88 (2010). ISA 88: Batch control.

- James, P. and Roggenbach, M. (2010). Automatically verifying railway interlockings using SAT-based model checking. In *Proc. of the 10th International Workshop on Automated Verification of Critical Systems(AVoCS)*, volume 35.
- Jee, E., Kim, S., Cha, S., and Lee, I. (2010). Automated test coverage measurement for reactor protection system software implemented in function block diagram. In Schoitsch, E., editor, *Computer Safety, Reliability, and Security*, volume 6351 of *Lecture Notes in Computer Science*, pages 223–236. Springer Berlin Heidelberg.
- Jee, E., Yoo, J., and Cha, S. (2005). Control and data flow testing on function block diagrams. In Winther, R., Gran, B., and Dahll, G., editors, *Computer Safety, Reliability, and Security*, volume 3688 of *Lecture Notes in Computer Science*, pages 67–80. Springer Berlin Heidelberg.
- Kramer, U. (2001). Continuous testing as a strategy of improving the PLC software development cycles. In *Proc. of IEEE/ASME International Conference on Advanced Intelligent Mechatronics*, volume 2, pages 781 – 786. IEEE.
- KRONOS website (2014). <http://www-verimag.imag.fr/DIST-TOOLS/TEMPO/kronos/>.
- Lange, T., Neuhäuser, M., and Noll, T. (2013). Speeding up the safety verification of programmable logic controller code. In *Hardware and Software: Verification and Testing*, volume 8244 of *Lecture Notes in Computer Science*, pages 44–60. Springer.
- Laplante, P. A. (2013). *Requirements Engineering for Software and Systems. Second edition*. CRC Press.
- Legay, A., Delahaye, B., and Bensalem, S. (2010). Statistical model checking: an overview. In *Runtime Verification*, pages 122 – 135. Springer-Verlag.
- Loiseaux, C., Graf, S., Sifakis, J., Bouajjani, A., Bensalem, S., and Probst, D. (1995). Property preserving abstractions for the verification of concurrent systems. *Formal methods in system design*, 6(1):11–44.

- Lopez, B. (2006). Non-commercial frameworks for distributed control systems. Technical Report EGO-NOT-OPE-92, European Gravitational Observatory.
- Machado, J. (2006). *Influence de la prise en compte d'un modèle de processus en vérification formelle des Systèmes à Événements Discrets*. PhD thesis, École Normale Supérieure de Cachan.
- Machado, J. and Eurico, S. (2013). HiL simulation workbench for testing and validating PLC programs. In *11th IEEE Int. Conf. on Industrial Informatics*, pages 230–235.
- Machado, J. M., Denis, B., Lesage, J.-J., Faure, J.-M., and Ferreira da Silva, J. C. L. (2003a). Increasing the efficiency of PLC program verification using a plant model. In *Proc. of International Conference on Industrial Engineering and Production Management (IEPM)*.
- Machado, J. M., Denis, B., Lesage, J.-J., Faure, J.-M., and Ferreira da Silva, J. C. L. (2003b). Model of mechanism behavior for verification of PLC programs. In *Proc. of the 17th International Congress of Mechanical Engineering (COBEM)*.
- Mader, A., Brinksma, H., Wupper, H., and Bauer, N. (2001). Design of a PLC control program for a batch plant - VHS case study 1. *European Journal of Control*, 7(4):416–439. Imported from DIES.
- Mader, A. and Wupper, H. (1999). Timed automaton models for simple programmable logic controllers. In *Proc. of 11th Euromicro Conference on Real-Time Systems*, pages 106–113. IEEE Comp. Soc. Press.
- McMillan, K. L. (1993). *Symbolic Model Checking*. Kluwer Academic Publishers, first edition.
- Meenakshi, B., Das Barman, K., Babu, K., and Sehgal, K. (2007). Formal safety analysis of mode transitions in aircraft flight control system. In *IEEE/AIAA 26th Digital Avionics Systems Conference*, pages 21–25.
- Modbus website (2014). <http://www.modbus.org/>.

- Mokadem, H. B., Bérard, B., Gourcuff, V., De Smet, O., and Rousel, J.-M. (2010). Verification of a timed multitask system with UPPAAL. *IEEE Trans. Autom. Sci. Eng.*, 7:921–932.
- NASA (1977). NASA: Formal Methods Specification and Analysis. Guidebook for the Verification of Software and Computer Systems. vol. 2: A Practioner’s Companion. Technical Report NASA-GB-001-97, NASA Office of Safety and Mission Assurance, Washington D.C.
- NASA Formal Methods Symposium website (2014). <http://nasaformalmethods.org/>.
- NASA Langley Formal Methods website (2014). <http://shemesh.larc.nasa.gov/fm/>.
- NuSMV website (2014). <http://nusmv.fbk.eu/>.
- Pavlović, O. and Ehrich, H.-D. (2010). Model checking PLC software written in function block diagram. In *International Conference on Software Testing*, pages 439–448.
- Perin, M. and Faure, J.-M. (2013). Building meaningful timed models of closed-loop DES for verification purposes. *Control Engineering Practice*, 21(11):1620–1639.
- Peryt, M. and Martín Marquez, M. (2009). FESA 3.0: Overcoming the XML/RDBMS impedance mismatch. Internal Note CERN-ATS-2009-102, CERN.
- PLC Checker website (2014). <http://www.plcchecker.com/>.
- Plotkin, G. D. (2004). A structural approach to operational semantics. *The Journal of Logic and Algebraic Programming*, 60-61:17–139.
- Profibus - Profinet website (2014). <http://www.profibus.com/>.
- PVS website (2014). <http://pvs.cs1.sri.com/>.
- Queille, J.-P. and Sifakis, J. (1982). Specification and verification of concurrent systems in CESAR. In *International Symposium on programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337 – 351. Springer Berlin Heidelberg.

- Radek, P. (2006). *Reduction and Abstraction techniques for model checking*. PhD thesis, Masaryk University.
- Remenska, D., Willemse, T. A. C., Templon, J., Verstoep, K., and Bal, H. E. (2014). Property specification made easy: Harnessing the power of model checking in UML designs. In *Formal Techniques for Distributed Objects, Components, and Systems*, pages 17–32.
- Sadolewski, J. (2011). Conversion of ST control programs to ANSI C for verification purposes. *e-Informatica*, 5(1):65–76.
- Sarmiento, C. A., Silva, J. R., Miyagi, P. E., and Filho, D. J. S. (2008). Modeling of programs and its verification for programmable logic controllers. In *17th IFAC World Congress*.
- Schneider Electric's automation website (2014). <http://www2.schneider-electric.com/sites/corporate/en/products-services/automation-control/automation-control.page/>.
- Siemens (1998a). *SIMATIC Statement List (STL) for S7-300 and S7-400 Programming – Reference manual*.
- Siemens (1998b). *Structured Control Language (SCL) for S7-300/S7-400 Programming*. Siemens.
- Siemens AG (2010). *SIMATIC Programming with STEP7 – Manual*.
- Siemens automation website (2014). <http://www.automation.siemens.com/>.
- Smet, O. D., Couffin, S., Rossi, O., Canet, G., Lesage, J.-J., Schnoeblen, P., Papini, H., Suprieure, E. N., and Fabrications, C. D. (2000). Safe programming of PLC using formal verification methods.
- Soliman, D. and Frey, G. (2011). Verification and validation of safety applications based on PLCopen safety function blocks. *Control Engineering Practice*, 19:929–946.
- Soliman, D., Thramboulidis, K., and Frey, G. (2012). Transformation of function block diagrams to UPPAAL timed automata for the verification of safety applications. *Annual Reviews in Control, Elsevier*, 36:338–345.

- SPIN website (2014). <http://spinroot.com/spin/whatispin.html>.
- Stattelmann, S., Biallas, S., Schlich, B., and Kowalewski, S. (2014). Applying static code analysis on industrial controller code. In *19th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE.
- Steinberg, D., Budinsky, F., et al. (2009). *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional.
- Sülflow, A. and Drechsler, R. (2008). Verification of PLC programs using formal proof techniques. In *FORMS/FORMAT 2008*, pages 43–50.
- TANGO website (2014). <http://www.tango-controls.org/>.
- TÜV Rheinland website (2014). <http://www.tuv.com/en/corporate/home.jsp>.
- UNICOS CPC Team (2013). UNICOS CPC documentation. <http://www.cern.ch/enice/UNICOS>.
- UPPAAL website (2014). <http://www.uppaal.org/>.
- Virtual Library On Formal Methods (2014). <http://formalmethods.wikia.com/wiki/VL>.
- Völker, N. and Krämer, B. J. (1999). Automated verification of function block based industrial control systems. *Electronic Notes in Theoretical Computer Science*, 25:97–110.
- Vörös, A., Darvas, D., and Bartha, T. (2011). Bounded saturation based CTL model checking. In *Proc. of the 12th Symposium on Programming Languages and Software Tools*, pages 149–160.
- Wang, R., Guan, Y., Liming, L., Li, X., and Zhang, J. (2013). Component-based formal modeling of PLC systems. *Journal of Applied Mathematics*, 2013.
- Willeman, D., Blanco Viñuela, E., Bradu, B., and Ortola Vidal, J. (2011). UNICOS CPC new domains of application: vacuum and cooling & ventilation. In *Proc. of 12th ICALEPCS*.

- Xiaoa, L., Li, M., Gu, M., and Sun, J. (2012). The modelling and verification of PLC program based on interactive theorem proving tool Coq. In *International Conference on Computer Science and Information Technology (ICCSIT)*.
- Xtend website (2014). <http://www.eclipse.org/xtend/>.
- Xtext website (2014). <http://www.eclipse.org/Xtext/>.
- Yoo, J., Cha, S., and Jee, E. (2008). A verification framework for FBD based software in nuclear power plants. In *Proc. of Software Engineering Conference (APSEC)*, pages 385–392. IEEE.
- Yoo, J., Cha, S., Kim, C., and Song, D. (2005). Synthesis of FBD-based PLC design from NuSCR formal specification. *Reliability Engineering and System Safety*, 87:287–294.
- Zhang, P. (2010). *Advanced Industrial Control Technology*. Elsevier Inc.