DAS Departamento de Automação e Sistemas
CTC Centro Tecnológico
UFSC Universidade Federal de Santa Catarina

Properties of LD Programs: Expression and Verification

Monografia submetida à Universidade Federal de Santa Catarina
como requisito para a aprovação da disciplina:
DAS 5511: Projeto de Fim de Curso

Ana Maria Mainhardt Carpes

Florianópolis, Março 2012

Properties of LD Programs: Expression and Verification

Ana Maria Mainhardt Carpes

Esta monografia foi julgada no contexto da disciplina
DAS 5511: Projeto de Fim de Curso
e aprovada na sua forma final pelo
Curso de Engenharia de Controle e Automação Industrial

ACKNOWLEDGMENTS

I would like to thank:

First of all, to my beloved mother, Agnes, for the great effort dedicated to my education and the invaluable love invested in me, which I treasure with all my heart, and to my friend and partner, Germano, for all the affection and encouragement he has shown me through every moment of our journey.

To my sister, Maria, an example of determination and trust, who has become my inspiration.

To my dear godmother and adviser, Irmgard, for being by my side whenever I needed.

To my father, Valter, for all his teachings and for showing me the world.

To the FLAGRAVA family, who have made what I am today and have shown me the true meaning of friendship.

To professors Jean-Marie and Xavier Crégut, for the great opportunity, ideas and advisement, and who have made possible the development of this project.

To everyone else who has somehow contributed to my growth.

ABSTRACT

Programmable Logic Computers, or PLC, are industry dedicated computers built with the aim to control and automate machines and processes. These processes are usually critical ones, where the occurrence of any error endangers the equipments, as well as human and environmental safety. Thereby, it is of extreme importance the use of formal techniques to ensure safety, quality and efficiency to the controlled systems. The Ladder Diagram language, or LD, is the most used PLC programming language. It has a graphical structure which greatly difficults error detection and code modification, which increases the necessity of formal methods.

In this work we aim to develop a tool for performing the formal verification of Ladder Diagram Programs, through the Reinterpretation approach and following the Model-Driven Engineering methodology. Two chains have been built. The first one translates LD programs into formal models. The FIACRE language has been chosen as formal representation, due to its better data manipulation characteristic. The second chain allows to write the system specifications and, with that, to obtain formal properties, which can be verified in the FIACRE models resulting from the first chain. The verification is performed by the Model-checking technique. A case study is presented for all the steps of both tool chains.

Keywords: Ladder Diagram, Formal Verification, MDE, FIACRE, Model-checking.

RESUMO ESTENDIDO

Computadores Lógicos Programáveis, ou CLP, são computadores dedicados à indústria, construídos especialmente para resistir a um ambiente inóspito e realizar o controle e a sincronização de máquinas e processos. Processos estes que são, grande parte das vezes, críticos, cujos erros podem levar a falhas e comprometer equipamentos, produtos e, nos piores casos, a saúde do meio ambiente e do homem. O uso de métodos formais de verificação é, portanto, de extrema importância para garantir segurança, qualidade e correção aos sistemas controlados.

A linguagem de programação de CLP mais usada até hoje é o *Ladder Diagram*, ou LD. Embasada na lógica de relés, ela apresenta uma estrutura gráfica que dificulta muito o processo de detecção de erros, fator que agrava a necessidade de métodos formais para garantir que o controle corresponda às expectativas.

O presente trabalho tem como objetivo construir uma ferramenta que permita aplicar a abordagem de Reinterpretação para proceder à Verificação em programas LD. A Reinterpretação considera a dificuldade em escrever uma lógica de controle diretamente em uma linguagem formal, uma vez que os programadores necessitariam ter domínio em modelagem e técnicas de verificação formal. Ela pressupõe a transformação do modelo informal – neste caso, do programa LD – em um formal.

Apresentamos aqui duas cadeias de transformação. Para o desenvolvimento de ambas, seguiu-se a metodologia da Engenharia orientada a Modelos, ou MDE – *Model-Driven Engineering*.

A primeira cadeia trata-se de um aperfeiçoamento de um trabalho de Iniciação Científica PIBIC-CNPQ, realizado pela mesma estudante envolvida neste projeto. Ela permite a translação automática de programas *Ladder Diagram* em modelos FIACRE – uma linguagem usada para representação formal de sistemas que, devido à sua estrutura, permite uma melhor manipulação de dados. Mais especificamente, essa primeira cadeia translada um programa LD em dois modelos FIACRE. Um deles representa um sistema onde o programa LD controla uma planta genérica – cujas variáveis interagem entre si somente através do CLP – e propriedades a serem verificadas sobre ele. Estas propriedades se referem à boa escrita do programa LD. O segundo modelo FIACRE gerado representa também um sistema controlado pelo programa de CLP; porém, desta vez, falta a modelagem dos processos da planta – tarefa que ainda necessita ser feita "manualmente" – bem como a escrita de propriedades – estas, por sua vez, referentes ao sistema real controlado, i.e., às especificações do sistema – que são geradas pela segunda cadeia elaborada.

A segunda cadeia construída auxilia a escrita de especificações de sistemas através da elaboração de uma Linguagem de Domínio Específico, ou DSL, chamada BPL – *Business Property Language* – e permite sua transcrição automática para propriedades formais, usando uma extensão da linguagem FIACRE. Essas propriedades obtidas automaticamente devem ser unidas ao modelo do sistema real – resultante da primeira cadeia.

Utilizando as duas cadeias de translação elaboradas neste projeto, pode-se obter modelos dos sistemas controlados e as propriedades a serem verificadas, ambos escritos de uma maneira formalizada, a fim de proceder a técnica de *Model-Checking* através de ferramentas preexistentes, como o TINA. Neste documento, um caso de uso é abordado, permitindo detalhar os passos e os resultados das duas cadeias de translação. Por fim, são apresentadas conclusões acerca do projeto e as perpectivas de trabalhos futuros.

CONTENTS

LIST OF FIGURES

1 INTRODUCTION

This document introduces and describes the work developed in the context of the discipline DAS5511 – Final Project – of the Control and Automation Engineering Course, between September 2011 and February 2012. This project has been realized at the IRIT laboratory, in Toulouse, France.

The aim of this discipline is to put in practice the concepts learned in class, giving the student the chance to realize what a project is and of which phases it consists. Therefore, this is also an opportunity to improve the students' skills, stimulating their autonomy as well as their teamwork ability.

The project presented here consists of a chain to allow the formal verification of Ladder Diagram programs, including the system modeling and the writing of its properties. It has been built in the Eclipse Environment[1], using the Model-Driven Methodology, MDE. This document is organized as follows:

- Chapter 2: Project – presents the project's context, its motivation and objectives, the laboratory where the work has been done and the followed methodology.
- Chapter 3: Business Domain – this chapter has as goal to introduce the concepts used in this work, but from the point of view of technicians that do not have a ground in formal methods. It presents what are PLCs, the IEC 61311-3 Standard and the Ladder Diagram language, as well as what are the desired properties of a system that is controlled by a PLC. A case study and its specifications are introduced.
- Chapter 4: Formal Domain – here are shown the formal concepts that have been used in this project. There is a brief foundation about the FIACRE language and its extension, and the TINA verification tool. The case study presented in the previous chapter is used now as an example for FIACRE properties.
- Chapter 5: Development and Preliminary Results – the longest chapter, it treats about the project's implementations. A previous work is presented, the new architecture is shown and its parts are described.
- Chapter 6: Final Results – the developed work is reviewed and its results are given through the case study cited above.

1 http://www.eclipse.org/

- Chapter 7: Conclusion Work – here, we give the final comments about this project, and what are its perspectives.

2 PROJECT

2.1 CONTEXT

Programmable Logic Computers, or PLC, are industry dedicated computers, built with the aim to control and automate machines and processes. Since their advent they have been essential for the Automation Industry and have promoted the development of new fields of research. Due to this wide use, a necessity of a norm emerged, resulting in the IEC 61131-3 Standard [5]. This Standand describes four PLC programming languages, among which is the Ladder Diagram, or LD, still the most largely used in the industry.

The Ladder Diagram language has a graphical structure which greatly difficults error detection and code modification, since the logical sequence of the programs is masked by the relay diagrams. Therefore, error detection in such programs used to be performed through simulation and tests. Nevertheless, apart from its high cost, the simulation of all possible states of a system is very time-demanding and, in most cases, infeasible, depending on the level of complexity. In other words, simulation is not an exhaustive process and, being so, does not assure the total absence of errors.

Industrial processes that make use of PLCs are usually critical processes, where the occurence of any error endangers the equipments, as well as human and environmental safety. It is of extreme importance that the programs which control such processes are completely absent of errors, so as to avoid financial loss, and what's more, respect the imposed restrictions so that safety, quality and efficiency are assured. It is to assure that these specifications are met that formal verification methods are required.

In general, there are two ways to proceed with formal verification: Formalization and Reinterpretation [9]. Formalization consists of conceiving the program directly in a formal language and converting it automatically to PLC code. In this case, two approaches are possible: Synthesis of a Supervisory Control – obtained from modelling the system and it's restrictions – or Verification – performed in the already controlled system's model, so that it can be checked if the system complies with the desired properties.

The second way to validate PLC programs, Reinterpretation – which is going to be used in this project – takes into account the difficulty of making technicians and engineers change their programming paradigm, i.e., learn formal methods so that they can write programs directly in formal languages. This would be impracticable for many programmers, and would make the reuse of the thousands of existing PLC programs impossible. Thus, in order to avoid this dramatic change, this method is based on the translation of such existing programs into formal languages.

Once this Reinterpretation is performed automatically, it is possible to resort to the Verification approach, mentioned above. In this approach, the operational description of the system, control included, is made based on the obtained formal models – usually Transition Systems, such as Petri Nets, for instance – which represent its behavior. Concomitantly, the desired properties for this controlled system are also formally modelled, in most cases by Temporal Logics. This way, through appropriate techniques and algorithms, and pre-existing tools, it is possible to decide whether or not the system fulfills the desired properties.

2.2 MOTIVATION

This work is an extension of that developed during the Scientific Initiation Scholarship, by the same student engaged in this Coursework Final Project, which was based on [12] and [2]. In this previous work, referred on [8, 7], a transformation chain has been built, where, from Ladder Diagram (LD) programs, formal models in the FIACRE language[1] were obtained.

Nevertheless, the difficulty of making technicians and engineers change their programming paradigm – mentioned in the above topic – remains, because the PLC programmer still needs to have knowledge about the formal verification techniques. After all, what the user gets with the existing translating tool are FIACRE models correspondent to the LD programs. The writing of the properties to be verified on this FIACRE model still remains to be done, a task which a programmer is normally not capable of doing.

This is the reason why we propose here the formulation of a business language – one that enables the PLC programmer to write properties he wants to check about his program – and its translation to Temporal Logic. Thereby, parallel to the transformation from LD to FIACRE, we shall have the transformation of properties written in this new business language to Temporal Logic. Finally, the temporal formulas obtained are going to be verified over the obtained FIACRE models.

2.3 IRIT LABORATORY

This project has been developed at the IRIT[2] – Informatics Research Institute of Toulouse – a mixed unit of research that has about 600 members – among them engineers, teachers, researchers, PhD students

1 See Section 4.1
2 http://www.irit.fr/

and administrative staff – and represents one of the biggest potentials on its area in France. There are 19 teams that work with seven main topics. One of these teams, named ACADIE – french acronym to Assistance to the Certification of Distributed and Embedded Applications – deals with the *Safety of Software Development* topic. This project is inserted in this group and the safety criterion is applied to PLC programs, more specifically, Ladder Diagrams programs.

2.4 OBJECTIVES

Based on the Motivation topic, given above in Section 2.2, the project objectives can be listed as below:

- Make a study of the desired properties of PLC controlled systems, with the aim to capture their most usual form and, thereby, to elaborate a syntax for their writing. This syntax must be accessible to the technicians that have no ground in formal verification techniques.
- Implement a "business language" with this syntax, i.e., implement a Domain Specific Language, or DSL, whose goal is to write the system specifications.
- Build an automatic chain to connect the system properties written with this DSL to properties in a formal representation, which can be used as entries in a preexisting model-checking verification software.
- Reformulate the previous translating tool – mentioned above in Section 2.2 – to generate, from Ladder Diagram programs, formal models containing not just the PLC representation, but also a system modeling. Actually, the aim is to obtain two kinds of models. The first one will contain a generic system modeling – where the comportment of the input and output variables are not dependent, as simple buttonholes and LEDs – to perform the verification of properties related to the LD program writing. The second model will contain the real system representation to verify if the controlled plant has the desired behavior.
- Use a case study of a simple real system to test and present the developed tool.

The methodology used for reaching these goals is given in the continuity of this document, as well as the project phases.

2.5 METHODOLOGY

The project is going to be based on the Model-Driven Methodology, or MDE [10]. At first, a DSML (Domain Specific Modeling Language) is going to be defined to express general and specific properties of PLC programs. This language is going to be a business language, on a PLC level, defined through the Xtext[3] tool.

From this language, a model transformation chain is going to be constructed, now with aid from the ATL[4] (Atlas Transformation Language, still in the MDE context) tool, so that the behavioral properties in the LTL (Propositional Linear Temporal Logic) format are obtained.

Finally, behavioral properties of PLC programs can be verified in Ladder Diagram programs, by using the other tool previously mentioned[5], developed during a Scientific Initiation Project PIBIC-CNPQ by the same student (entitled Conception of Complex Systems with Guarantee of Quality of Service). This *Translation Tool*, as it is called in this document, has been improved using, once again, the ATL toolkit, and it allows us to obtain formal FIACRE models from Ladder Diagram programs. The verification approach to be utilized is the Model-Checking (an exhaustive technique that allows to check whether or not a given property is true and, in the negative case, provide a counter-example, i.e., a sequence of states, starting from the initial state, that contradicts the property), through already existing tools, such as TINA[6], for instance.

3 http://www.eclipse.org/Xtext/
4 http://www.eclipse.org/atl/
5 See Motivation, at Section 2.2, and for further details, Section 5.1.
6 http://homepages.laas.fr/bernard/tina/

3 BUSINESS DOMAIN

In this chapter we will present the concepts used in this work, but from the point of view of technicians that do not have a ground in formal methods. It presents what are PLCs, the IEC 61311-3 Standard and the Ladder Diagram language, as well as what are the desired properties of a system that is controlled by a PLC. A case study and its specifications are introduced.

3.1 LADDER DIAGRAM LANGUAGE – LD

3.1.1 *IEC 61131-3 Standard*

The increase of the use and relevance of PLC in the field of industry has led to a necessity of an international norm that allows the reuse of existent PLC programming logics and the ease of learning and adaptation for the users. For this reason, the IEC 61311-3 Standard [5] has been created and accepted internationally. Indeed, this norm represents a third part of a whole standard for describing PLC features and defines one structuring and four programming languages (two textual and two graphical languages).

Besides the International Standard, other efforts have arised to help with software reusability and exchangeability, such as, for example, the XML TC6 Open Format, by the PLCOpen Organization[1]. This allows to save all the data concerning the PLC programs which follow the IEC Pattern – including graphical informations – in XML files and to exchange between different editors and compilers. As a partnership, the Beremiz Organization[2] has built an open source environment, called PLCOpen Editor, that allows the edition of PLC programs as TC6 XML files. Both of these – the TC6 XML format and the PLCOpen Editor – have been used in this project, as will be shown later in this document.

3.1.2 *LD Description*

The Ladder Diagram Language is one of the two graphical languages described by the IEC 61131-3 Standard, and it is based in relay logic. This feature is what puts LD in advantage over the other PLC languages, because it facilitates adapting already existing relay programs

1 http://plcopen.org/
2 http://www.beremiz.org/

to PLC programs and, besides, avoids the necessity of changing the programmers' paradigm.

However, it is exactly the relay logic based characteristic that confers the intricate aspect of the LD programs, making it difficult to debug, modify and analyze the code. Due to this fact we have chosen, among the existing PLC languages, the Ladder Diagram for our verification study.

A LD program consists of two vertical rails and a series of horizontal rungs between them, and can hold simple and complex elements. Each one of these rungs contains input and output instructions, which allow to manipulate data according to the configuration. Among the simple elements we can find the contacts, or relays, and the coils. The relays, represented graphically by two bars | |, are input instructions and do not modify the value of their associated program variables, but form the function to calculate the new value of their rung outputs, by "and" and "or" boolean operations. The coils, represented by two parentheses ( ), are these outputs, and they, unlike the relays, do modify the value of the associated variables.

Besides the simple elements – which can have different configurations, as normally open, normally closed, set, reset, etc. – we can also have complex ones, like Functional Blocks – timers and counters, for example – and jumpers, comments, non-boolean variables and others, which can be found in the complete documentation of the Standard [5].

The execution of the Ladder Program is made in consecutive cycles, called Execution Cycle or just *Scan*, and they can be separated in three steps. First, the reading of input variables from the sensors connected with the PLC. After, the calculating of intern memory and output variables of the program, according to the rung input function and taking into account the values of input variables stored in the previous step. This computation is made following the rungs from top to bottom, and from left to right. At the end, the third step is the writing of output variables – that have been calculated before – in PLC actuators. In the next chapter, we will see how to model this PLC behavior.

3.2 DESIRED PROPERTIES OF LD PROGRAMS

When engineers or technicians write PLC programs, they have in mind a desired behavior for the system to be controlled, which means that they want the program to restrict its comportment. Thereby, the plant will never exhibit an inadequate or unsafe behavior, and will do the necessary to reach its goal – i.e., the goal it was constructed for.

Nevertheless, this does not always happen. The programs generally contain mistakes, bad structures, or even its control logic is inappropriate. These problems are really hard to see, and in a LD code it is even worse, because the logical sequence of the program is masked by the relay diagrams. It is necessary to use techniques to guarantee the correctness of the program – as we will see in Chapter 4.

However, before using any verification process, we need to stipulate which properties we want the program, or the controlled system, to have. These properties need to reflect the aforementioned "desired behavior", and can be classified into *generic* and *specific* ones. The generic properties are those that do not depend on the plant to be controlled, but can be applied to every – or to a group of – systems. For example, many times we do not want the plant to reach an end state, where no progress is possible to happen – this is known as *deadlock free property*. For LD programs, we can enunciate as general property that the reading state of the PLC will be reached infinitely often – or analogously for the writing state, and for rung computation states, for example – these are known as *liveness properties*.

Another generic property we want to check is the *absence of race conditions* in the program [1]. Race conditions occur when operations that should be sequential in order to reach the expected results are simultaneous and, as the name tells, race to influence the result of the process. In a Ladder program, it occurs when, for stable entries – i.e., for fixed input variables and fixed functional block states – the output variables do not stabilize, they change at each program scan or at n consecutive scans – where n is a positive integer – forever. We want the outputs to depend only on inputs and functional block states, nothing else. For example, in a traffic lights program, where the outputs change periodically (the lights), the inputs are fixed but the timer states are not, so it is not a race condition case. The absence of race conditions is an important property to be verified because it can cause actuator problems, due to the high frequency of output change, or other malfunction, since the outputs are not obeying the input signals. Besides that, race conditions are really difficult to detect.

The specific properties are those that reflect the specification of the plant, and therefore depend on each system. In the next topic, we will see some examples for a case study system.

3.3 CASE STUDY – APS PROBLEM

We will present as a case study in this document an Automatic Pneumatic System, or APS, built by LASHIP-UFSC[3] (Laboratory of Hydraulic and Pneumatic Systems of the Federal University of Santa Catarina) as a test bed. In fact, this is the third module of a whole Pneumatic System, called MOD3, which is connected with the other modules by two variables – MOD3INI and MOD3FIM. These variables indicate, respectively, the beginning and the end of each operation cycle of the plant.

3 http://www.laship.ufsc.br/

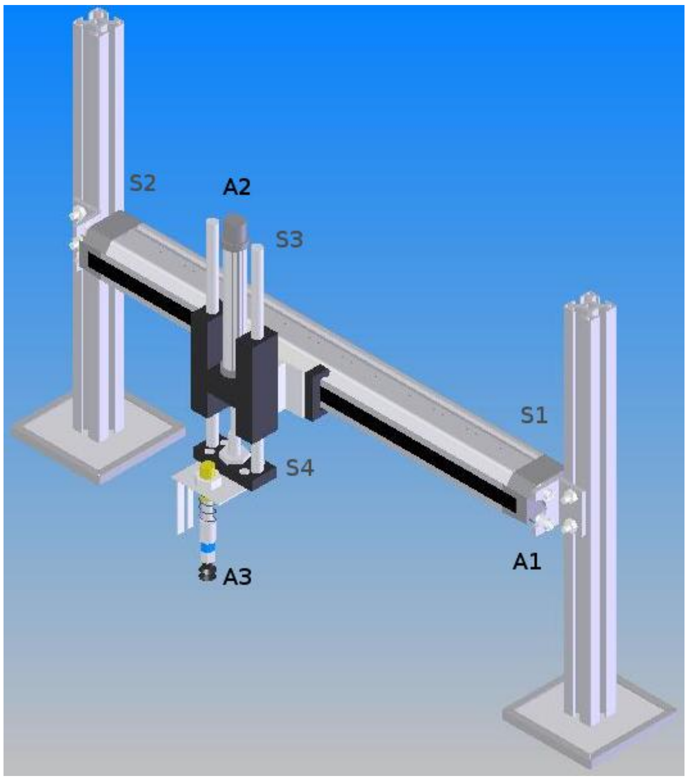


Figure 1: Schematic Draft for the APS

The MOD3 is composed by two perpendicular cylinders, A1 and A2 – each of them equipped with a beginning and an end sensor – and one suction cup, A3 – fixed in cylinder A2. This module has been designed for transporting boxes between the limits of cylinder A1, and its schematic draft can be seen below – Figure 1.

The initial state of the module is the configuration where the two cylinders are retracted and the cup is off, and the operation cycle is described by:

- With both cylinders retracted, and the cup turned off, cylinder A2 starts to extend
- When cylinder A2 is completely extended (down position), the suction cup is activated
- After 5 seconds – time needed for the box to be correctly fixed – cylinder A2 starts to retract
- When cylinder A2 is completely retracted (up position), cylinder A1 starts to extend
- When cylinder A1 is completely extended, the suction cup is turned off (note: in this moment, cylinder A2 is retracted)

- After 5 seconds – time needed for the box to be correctly dropped – cylinder A2 starts to retract, returning to the initial state of the system.

The end and beginning sensors for the cylinders are, respectively: S1 and S2 for A1 and S3 and S4 for A2. Therefore, note that:

- When A1 is retracted, S1 is on and S2 is off
- When A1 is extended, S1 is off and S2 is on
- When A1 is extending or retracting, S1 and S2 are off

The same holds for cylinder A2 and its sensors, S2 and S3. The suction cup does not have any sensor. The Ladder Diagram Program for the control of MOD3 can be found in Appendix A.

3.3.1 *Examples of Properties – APS Specification*

Besides the generic properties mentioned above, the APS problem has other properties, specific ones, that reflect the desired behavior of the plant. Here we are going to cite some of these properties, that will be used throughout the document.

For the safety of the plant and the blocks that are intended to be transported, we want the two cylinders never to move together, which means:

- Cylinder A1 will never be extending or retracting at the same time as cylinder A2 is extending or retracting.

Other safety property we desire in order to prevent the blocks from falling is:

- Whenever cylinder A1 is extending or cylinder A2 is retracting, suction cup A3 must be activated.

For preventing the useless activation of the suction cup, we also want that:

- Whenever cylinder A1 is retracting or cylinder A2 is extending, suction cup A3 must not be activated.

The main goal of this APS module is transporting boxes, thus we want the boxes always to be caught and always to be dropped in the right place. This can be enunciated as two properties:

- Infinitely often, cylinder A1 will be retracted, cylinder A2 will be extended and cup A3 will be turned on at the same time.
- Infinitely often, cylinder A1 will be extended, A2 will be retracted and cup A3 will be turned off at the same time.

Finally, we want the module to follow the operational cycle that we have planned, which means:

- Whenever the system is in the *n*-th state, we want it to remain in this state until the next desired state, i.e., the (*n+1*)-th state.

Where, by "state", we mean any state taken into account in the operational cycle, namely:

- State 0: A1 retracted, A2 retracted, A3 turned off
- State 1: A1 retracted, A2 extending, A3 turned off
- State 2: A1 retracted, A2 extended, A3 turned off
- State 3: A1 retracted, A2 extended, A3 turned on
- State 4: A1 retracted, A2 retracting, A3 turned on
- State 5: A1 retracted, A2 retracted, A3 turned on
- State 6: A1 extending, A2 retracted, A3 turned on
- State 7: A1 extended, A2 retracted, A3 turned on
- State 8: A1 extended, A2 retracted, A3 turned off
- State 9: A1 retracting, A2 retracted, A3 turned off
- State 10: State 0

4 FORMAL DOMAIN

In this chapter we will present the context over which this project is grounded. A formal modeling language that has been used to represent the LD programs and the system to be controlled is introduced. However, we have considered that the reader has a basic knowledge about Discrete Events Systems and Temporal Logic – for a detailed approach, refer respectively to [3] and [4].

The goal of this project is to facilitate to the programmers the verification of Ladder Programs. Usually, the technique applied is an informal one, the simulation. However, as we have already mentioned, this is not an exhaustive process, and for this reason we have used here formal verification techniques, which guarantee the desired comportment of the systems. In fact, this "guarantee" is not completely irrefutable, even with this formal approach. This can happen – and indeed happens – due to badly stated properties or due to a formal model that does not represent the system in a very precise manner. In this case, the verification is not valid and the system behavior can diverge from the expected.

In this project we have chosen Reinterpretation[1] as the technique for proceeding with the verification of Ladder Diagram programs. It consists of:

- Translating the LD programs representation into a formal model
- Writing the properties that we want to check in an informal way
- Translating these informal properties into formal ones
- Verifying if the formal representation of the program satisfies these formal properties by a Model-Checking technique.

The formal representation for LD programs that has been used along this project is the FIACRE language, that allows a better data manipulation, reducing the problem of states explosion and thus allowing the verification of a wider range of Ladder programs, as those with functional blocks, for instance. The formal properties have been written in Linear Temporal Logic. The FIACRE Language will be presented below.

4.1 FIACRE LANGUAGE

FIACRE [6], an acronym for Intermediate Format for the Architectures of Embedded Distributed Components, is a formal modeling language,

1 See Section 2.1

for purposes of verification and simulation, developed for representing both behavioral and timing aspects of embedded distributed systems. It is called "intermediate" because it works as source and target in model transformations – as will be shown later. This language was developed in the context of the Project TOPCASED[2] – Open Source Engineering Workshop – gathering many partners, both from the industrial and academic fields.

4.1.1 *Description*

A FIACRE model is structured in process and components notions.

A FIACRE process describes the behavior of sequential components, and is declared as a set of control states and parameters, both associated with a set of complex transitions. These complex transitions are nothing else than statement sequences for specifying how its parameters must be changed and which transition must be fired. These statements can be formed by deterministic constructs – as assignments, if-then-else and while conditionals, sequential compositions – non deterministic ones – for the choice of simple state transitions and assignments – and communication events on ports – for synchronization only or also for data exchange between different processes and components.

The FIACRE components describe a system as a parallel composition of processes or other components – via instantiation – possibly in a hierarchical manner, and communicating with each other through ports and shared variables. These components allow to restrict the access mode and the visibility of shared variables and ports, to associate timing constraints with communications, and to define priority between communication events.

FIACRE is considered as an intermediate language since it can be seen as a target model in transformations from other higher level languages and, at same time, as a compilers' entry for verification tools.

Among FIACRE compilers there is FRAC[3], developed for the verification tool named TINA, which has been used in this work. FRAC allows the generation, from a FIACRE model, of a *Time Transition System*, or TTS – a Time Petri Nets extension which provides the possibility to handle data and priorities, and is used for internal representation by the TINA tool, introduced in the next section.

In the topic below, we will present an example of FIACRE model for representing the controlled system introduced in the case study of the previous chapter, Section 3.3.

2 http://www.topcased.org/
3 http://homepages.laas.fr/bernard/fiacre/

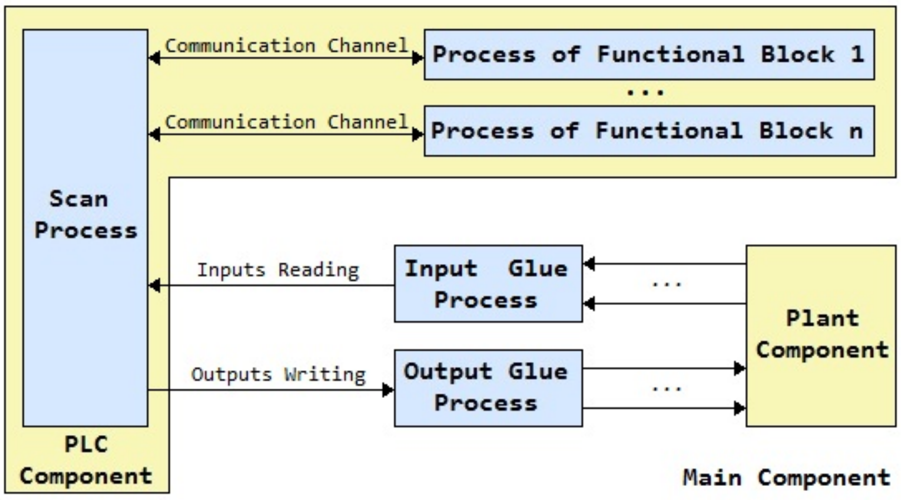


Figure 2: Communication schema in a plant controlled by a PLC

4.1.2 *Fiacre Example – APS Model*

We will now explain the modeling of the APS case study, which has been introduced in the previous chapter. This model has been written in FIACRE language and is provided here as an example of its use. Here, we show relevant parts of the Main and PLC components and of the Execution Cycle and Timer processes – which form the PLC. The complete model can be found at Appendix C – where the FIACRE elements that are indicated but "hidden" can be seen in Appendix B – and the LD program at Appendix A.

First of all, we show the overview of the system, represented by the Main component of the FIACRE model. The whole system is formed by the Plant component – which means the system to be controlled – the PLC component and the processes of input and output "glues". The necessity of these two processes is to work as a link between the plant and the PLC, which are asynchronous. They are those responsible for reading and writing the input and output variables, and to send and receive them to/from the PLC, allowing the new data value exchange. In Figure 2, we can see the flow of these update values. The PLC reads all the input variables together, from the glue process, as an array, under its reading *scan* step[4] and sends all its update output values together under its writing step, for the output glue. The plant, on the other hand, reads one PLC output at a time, and also sends the input values individually.

In this LD program, we have considered as inputs the variables s3S1, s3S2, s3S3 and s3S4; as outputs a3A1, a3A2 and a3A3. Indeed, the variable MOD2INI is an input, and MOD3FIM an output, but they have been treated here as memory variables. This happens because these variables are the connection between this particular LD program and

4 As mentioned in Section 3.1.2

those that refer to the other APS modules. But, as we are dealing only with this particular program that controls the third APS module, we can neglect this fact. Thus, the FIACRE code for the APS component is as follows:

```
component APS is p
port
        /* ports with timing constraints [0,0] */
par * in
        PLC [portInputs, portOutputs]
        || Plant
                [port_s3S, port_s3S2, port_s3S3, port_s3S4,
                        port_a3A1, port_a3A2, port_a3A3]
        || Input_glue
                [port_s3S1, port_s3S2, port_s3S3, port_s3S4,
                        portInputs, portOutputs]
        || Output_glue
                [port_a3A1, port_a3A2, port_a3A3,
                        portOutputs,portInputs]
end
```

The PLC component, in its turn, is the composition of the Scan process, that represents the execution cycle of the LD program, and the two Functional Block processes it uses – in this case, two timers-on, called TON:

```
component PLC [portInputs: in type_in, portOutputs: out type_out]
     is
        port
        /* internal communication ports -- between the scan and
            the functional blocks -- with timing constraints
            [0,0] */
        /* functional blocks dummy ports -- in this case, TON1
            and TON2 -- with time constraints referring to the
            time variable in their entries */
par * in
        par
                /* Timers of the LD program */
                TON[TON1_1, TON1_2, TON1]
                || TON[TON2_1, TON2_2, TON2]
        end
        || Scan [portInputs, portOutputs,
                                TON1_1, TON2_1, TON1_2, TON2_2]
end
```

The Scan process follows the three sequential steps of reading inputs, calculating new values, and writing outputs. The initial state of this process is the reading one, and the final is the state for the reinitialization of the cycle, after the output writing. For the calculating step, there is one state and one transition for each rung of the LD program. Besides that, there is also one extra state and transition for each rung that has a Functional Block: these kinds of rungs are split in two *semi rungs*, one before and other after the block. The reason is that only one communication is allowed in each transition, and in these rungs it is needed to send the entry value of the block to its process and to receive from it the new output value. That is why we use a Semi-rung class in the LD modeling – Figure 6 – as is treated in Section 5.1.3. The Scan process for the APS Ladder program is as follows:

```
process Scan [ /* ports */ ] is
        states
                initial, rung_1, /* ... */, writing, final
        var
                /* variables declaration */
        init to initial

/* Reading transition */
from initial
        /* reading inputs from input_glue */
        portInputs? vars_in;
        /* updating the input variables values */
        s3S1:= vars_in[0];
        s3S2:= vars_in[1];
        s3S3:= vars_in[2];
        s3S4:= vars_in[3];
to rung_1

/* Rung transitions without Functional Blocks... */
from rung_n
        wait [0,0];
        coil_variable:= input_function;
to rung_n+1
/* or << to writing >> if it is the last rung of the program */

/* Rung transitions with Functional Blocks... */
from rung_m
        IN_TONi:= input_function;
        TONi_1! IN_TONi;
to rung_m_1

from rung_m_1
        TONi_2? Q_TONi;
        Ti:= Q_TONi;
to rung_m+1
/* ... */

from writing
        /* sending outputs to glue */
        portOutputs! [a3A1, a3A2, a3A3];
to final
```

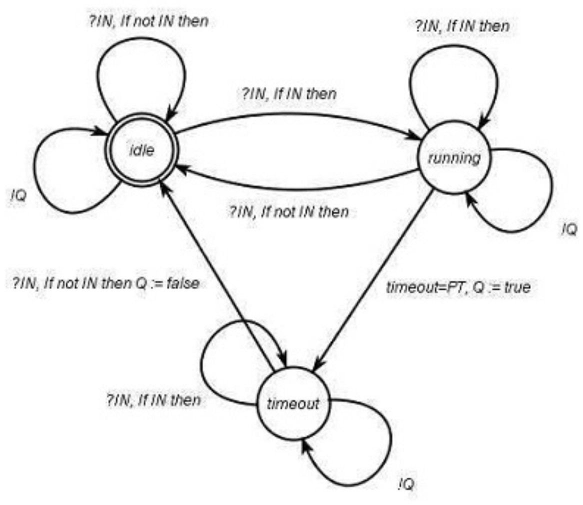


Figure 3: Automata representation for the TON process

```
from final
       wait [1,1];
to initial
```

Notice the "wait" statements in this process. This is a new statement, included in the new version of FIACRE, which do not significantly impacts its semantics. It is used to impose timing constraints to transitions that do not have communications – because until the moment the only way to constrain these transitions was by making a dummy communication, i.e., a synchronization with no element. The only transition whose constraint is not "instantaneous" is the last one, between the final and the initial state of the program scan, for representing the cycle time duration.

This Scan process communicates with the TON process, instantiated in the PLC component – two instances of the same declared process, because both are of timers-on kind. Nevertheless, each block has a dummy port with the time constraint referring to its entry time variable – in this case, both TON blocks have an input time of 5 seconds, i.e., a time constraint of [5, 5]. The motive for using a dummy synchronization port and not the "wait" statement is because the LD program could have more than one block of the same type – as the TON – but with different times – in contrast with the APS example. Thereby – as "wait" cannot handle variables, just fixed values – instead of having one process declaration for each kind of block, we will have one for each block in the LD program, even when the blocks are of the same type. Now, regarding the TON process, it has three states – idle, running and timeout. Its behavior can be found in the LD specification [5], the FIACRE model at Appendix B, and its automata transition system can be seen at Figure 3.

Finally, for describing the system that the PLC controls we have used a Plant component, which instantiates one process for each part of this system: two cylinders – modeled in the same manner, i.e., they are different instances of the same process – and one suction cup. The Cylinder process has four states – retracted, extending, extended, retracting – and the Cup has also four – opened, closing, closed, opening – corresponding to descriptions given in Section 3.3.[5]

4.2 TINA VERIFICATION TOOL

TINA[6] is a toolbox for edition and analysis of Petri Nets and Time Petri nets, developed by the OCL group of LAAS/CNRS[7], Toulouse, France. Among its available tools, we have:

- *nd* (Net Draw): graphical or textual editor for (Time) Petri Nets and Automata, including a simulator.
- *tina*: this tool, with the same name of the toolbox, allows the construction of reachability graphs and Kripke transitions systems – useful for the verification by model-checking – from Petri Nets, for example.
- *selt*: allows the user to provide LTL (Linear Time Logic) formulas and verify if the Kripke transitions system – generate by *tina* – satisfies them. When a property is not verified, the tool returns a counterexample – which can be simulated by *tina*.

Thereby, this set of tools provides a powerful verification software, which is employed in this project. From our FIACRE models and their respective LTL properties that we want to check, the TINA toolbox has been used for obtaining the Kripke Structures – after the models' compilation by FRAC, which generates the Time Transition System (TTS) – and then verifying the models' behavior.

4.3 FIACRE REAL TIME EXTENSIONS – RT-FIACRE

During the evolution of the FIACRE language, some extensions have been proposed for including the real time aspects, changing its name to RT- FIACRE, i.e., Real Time FIACRE [11]. There are two kinds of extensions: the behavioral – which aims to increase the expressiveness of the language – and the properties extensions, but here we will consider only the second one.

4.3.1 *Patterns*

The properties extensions for FIACRE are proposed to provide a high level verification pattern, comprehensive enough for expressing the

5 For more details about the FIACRE code of the Plant processes, see Appendix C.
6 http://homepages.laas.fr/bernard/tina/
7 http://www.laas.fr/

most common user requirements and thus hide the details from the user – which is pretty much desirable. Before the FIACRE properties[8], the user needed to write them using the *selt* module of TINA – the model-checking verification tool that we are using in this work – for writing the LTL formules. This formules must be written accessing directly the states in the TTS[9] representation, which means that the user must know about LTL expressions and understand the TTS models.

These patterns have been classified in:

- General: for expressing properties like Absence of Deadlock, propositions that are true infinitely often and propositions that cease to be true from a moment forever on – mortal ones.
- Presence: for properties that refer to propositions which are true at least once.
- Absence: for propositions that are always false – to check if an undesirable behavior never happens, for example.
- Response: for expressing cause-effect properties – one fact that leads to another in the future.
- Universality: for propositions that are always true – to check if a necessary requirement is always met, for example.
- Precedence: for specifying when one requirement must be fulfilled before another.
- Composition: allows to nest all other patterns through the negation, conjunction and disjunction operations – *not*, *and*, *or*, respectively.

There is already a version of the FRAC compiler that accepts the declaration of properties patterns and also of LTL properties. The most desirable is to use only the patterns, but for more complex properties, like the aforementioned Absence of Race Condition[10] – which needs nested patterns behind the existing compositions – we use the LTL properties.

4.3.2 *Example of Patterns Use – APS properties*

We present in this document, as an example of use for the FIACRE patterns, the specific properties of the APS problem. The informal specifications of this case study have been explained in Section 3.3.1,

8 The complete pattern list can be found at `http://homepages.laas.fr/bernard/fiacre/properties.html`.
9 Time Transition System
10 See Section 3.2

and now they have been translated to the FIACRE properties declarations – which are shown in Appendix E related to the FIACRE model at Appendix C. There, we can see that the FIACRE property declaration is written as:

```
property propertyName is
        /* property */
```

Where a *property* can be of LTL or pattern type, both using the states of the processes instances as atomic propositions. These states are written in a specific way, as shown below, where *mainName* is the name of the main component of the FIACRE model – in this case it is *APS* – *nat* is a natural number – specifying the position of the desired instance inside the previous one, i.e., as a path for this final process instance – and, finally, *stateName* is the name of the desired state in the process declaration:

```
mainName/ (nat/)* state stateName
```

As an use example of FIACRE properties, we can see below how one of the informal properties of the APS case study is expressed:

Informal Property	FIACRE expression
Infinitely often, cylinder A1 will be retracted, cylinder A2 will be extended and cup A3 will be turned on at the same time.	property example is infinitelyoften ((APS/2/1/state retracted) and (APS/2/2/state extended) and (APS/2/3/state closed))

Where *APS* is the main component in the FIACRE model – which you can see in Appendix C – the number 2 in *APS/2/*$\cdots$ is the Plant component – because it is the second instance inside the APS component – the number *i* in *APS/2/i/*$\cdots$ where *i* can be 1, 2 or 3 is, respectively, the first, the second and the third instances inside the Plant component, i.e., if you look to the FIACRE model, the two cylinders and the cup.

To find the relation between the informal specifications and the FIACRE properties, we wrote the first ones in LTL formulas, and then found the matched patterns – see the table in Appendix F.

5 DEVELOPMENT AND PRELIMINARY RESULTS

In this chapter, we will briefly present the previous work in which this project is inspired. The proposed architecture to fulfill the requirements given in the introductory chapter is also introduced, followed by the presentation of the developed tools.

5.1 PREVIOUS WORK

This work is an extension of that developed during the Scientific Initiation Scholarship, by the same student engaged in this Coursework Final Project. In the referred work, the goal was to construct a verification chain where Ladder Diagram programs could be translated to formal models, and then verified by model-checking. The formal model used was FIACRE, and the verification tool was TINA, both already introduced in this document. Now, we will briefly present this previous work, so as to understand the general purpose and the context of this project.

5.1.1 *MDE Methodology*

The methodology used in the previous work, as well as in the current project, is the Model-Driven Engineering, or MDE [10]. The MDE is a software development methodology, model based – as the name suggests – which came to improve, and not to replace, the object-oriented paradigm.

Models are system representations built in a simplified fashion to deal just with the relevant aspects for a certain goal – they are abstractions to deal with the desired features, unprovided of unnecessary details – which does not mean that these models are simple ones, just that they are designed for a particular purpose. In MDE, these models are not only used to facilitate the system understanding, but also for software generation. Thereby, everything is seen as a model in this approach, which presupposes technologies for their generation and tools for manipulating and transforming them.

In the next section, the ATL language and toolkit will be presented, which allow model transformations so as to manipulate these models. Further, in this chapter, we will also introduce what are Domain Specific Languages and Xtext, a framework with which it is possible to define them and generate an abstract model.

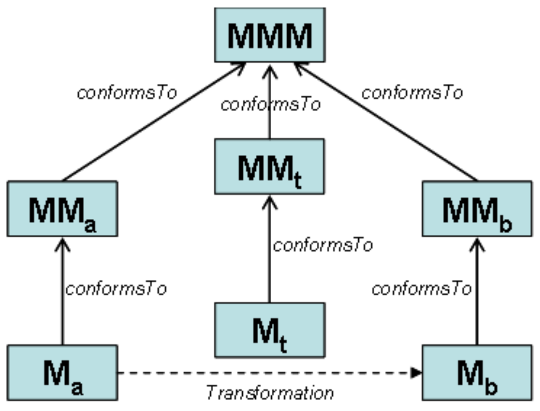


Figure 4: Relation between models, meta-models and their meta-meta-model

5.1.2 *ATL – Toolkit and Models Transformation Language*

ATL[1] – Atlas Transformation Language – is a model transformation language and toolkit, built for Eclipse environment, and it is part of a subproject of Eclipse Modeling Project[2]. ATL is grounded and also supports the MDE methodology, since it is model based and supports model operations. The ATL allows to generate a set of target models from a set of input models, through binding rules.

These rules, which constitute an ATL program, make a link between the elements of the input and the target models, and can be referent to two kinds of programming paradigms: the declarative and the imperative ones. Therefore, ATL language is a mixed of these two kinds of paradigms – the declarative is the predominant mode, used for simple mappings between the source and the target models, and the imperative is used for complex mappings, allowing the call of other rules or helpers. These helpers can be functional, like the methods in Java language, or of attribute kind, as program variables.

Now, looking at the ATL architecture, inside of the MDE concepts, we can actually consider the program with the binding rules as a model. So, we have the entry models, the target models and, finally, the transformation model. But this is not the end: all these models need to be defined in a pattern way. As for natural languages, it is necessary to have a syntax to adopt. In this case, the syntax is defined in what is called meta-model. We need one meta-model for each kind of system that we have, i.e., one for each specific domain. As an example, we can look at the ATL: each program, i.e., each model, has specific rules, but all of them are written in accord with the same syntax – the ATL language syntax, which is defined in the ATL meta- model. Thereby, a meta-model can be seen, indeed, as a model for a model, or even we can say that the model is an instance of its meta-model.

1 http://www.eclipse.org/atl/
2 http://www.eclipse.org/modeling/

The models and meta-models' relationship is shown in Figure 4 above, where M means model, MM meta-model, and the indices *a*, *b* and *t* indicate the entries, the targets, and the ATL transformation, respectively. Note that there is also a MMM represented: this refers to the meta-meta-model. Analogously to the models syntax, all the meta-models need to be built following a syntax defined in the meta-meta-model. It seems to be confuse, but actually it is a simple hierarchy: there is the meta-meta-model, that is the "father" model conforming with itself and working like a syntax for writing meta-models; then, we have the meta-models, each one built to describe a certain domain of systems; and, at last, there are the models describing specific systems, each one within a certain domain and conforming with the appropriate meta-model for this domain.

Therefore, for making an ATL transformation, we need to have:

- A meta-meta-model – the Eclipse Project provides two options, one of them called Ecore, which has been chosen in this project. For this reason, the meta-models are also called *Ecore models*, and they are similar to UML[3] Class Diagrams – so, the elements defined in meta-models can be called classes.
- One or more entry models and their meta-models – i.e., their instances and their respective Ecore models.
- One or more target meta-models.
- The ATL meta-model – which is already provided by the ATL toolkit.
- The ATL program/model – which has the rules to map the entry classes into the target classes.

The result of the transformation is one or more target models, conforming to their meta-models – i.e., containing instances of their meta-model classes. These instances, in turn, were generated from the entry classes instances by the ATL model rules.

This facet of ATL, which allows to translate between two or more models, is called M2M, i.e., Model to Model transformation, and the ATL program/model is an *ATL module*. But ATL can be used differently because it also enables to make a program for getting any instance of primitive data type, like Strings, from an abstract model. In this case, the program is an *ATL query* and it does not contain the same rules as in the ATL module, but an expression instantiation and helpers to build a value type from the classes' instances in the abstract model. Thereby, you keep the entry models and meta-models, but instead of obtaining a target model conforming with some meta-model, you will have only a simple output file with the desired value type. In the particular case where the type is String this process is called *unparser*, and in the ATL case, it is a M2T, i.e., Model to Text transformation.

3 Unified Modeling Language – `http://www.uml.org/`.

5.1.3 *Translation Tool Architecture*

The aim of the previous work introduced here was to build a tool for obtaining a formal model from a Ladder Diagram program, allowing the formal verification by model-checking. This technique is called Reinterpretation, as we saw in the introductory chapter.

The formal representation chosen was the FIACRE one, and for the Ladder Diagram programs, we have used the PLCOpen Editor, which conforms with the International Standard IEC 61311-3 and uses a particular XML pattern, the TC6 XML, that allows the interchange of PLC programs and configurations.[4] We have already presented the FIACRE language and the PLCOpen Editor in the previous chapters, as well as the reason for their choice. Now we will explain the tool architecture.

The FIACRE language is a domain specific language, and has its own meta-model to describe it, conforming with the Ecore meta-meta-model – see above Section 5.1.2. An instance of this meta-model is a FIACRE abstract model: an XML[5] file that can be transformed – for example by the ATL M2T transformation – in a textual one to obtain the familiar FIACRE model, presented in Section 4.1. On the other hand, the output from the PLCOpen Editor is a TC6 XML file that contains the entire LD program edited. This XML file conforms with its Schema, the TC6 XSD[6], which can be easily transformed in a meta-model – more precisely, in an Ecore model – by the MDE tools. Thus, it seems very natural and obvious to choose the MDE methodology in the Reinterpretation process: the input and the target of the desired tool were already born within this approach, and it provides the necessary technology to support the transformations between the models.

We can see in Figure 5 the schema of the translation tool. These are its components:

- The input meta-model – obtained from the TC6 Schema, and so, called *TC6 Ecore model*;
- The target meta-model – representing the FIACRE structured, thus called *FIACRE Ecore model*;
- An intermediate meta-model – called *Ladder Ecore model*, because it also represents the LD program, but differently from the TC6 one, as we will explain below;
- The first transformation – an ATL M2M one, called *TC62Ladder* since it maps the TC6 to the Ladder Ecore model;

4 See Section 3.1.1
5 Extensible Markup Language
6 XML Schema Definition

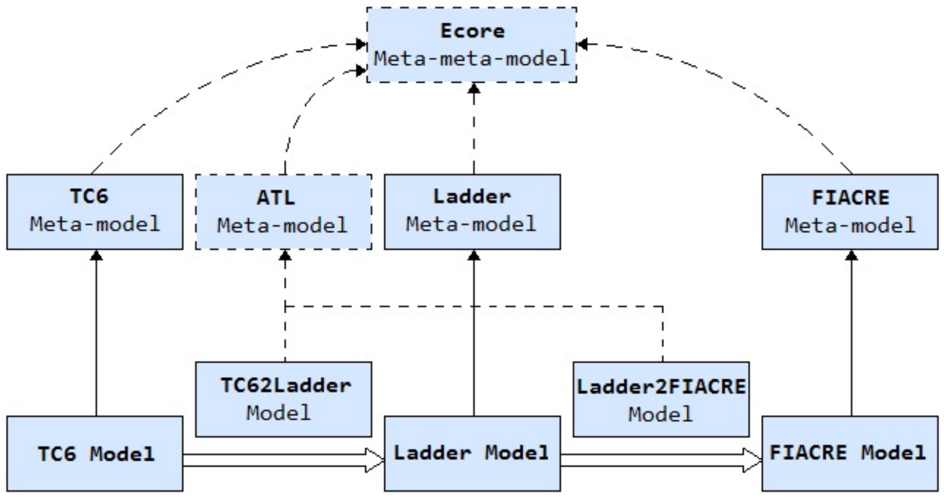


Figure 5: Schema of the Previous Translation Tool

- The second transformation – also an ATL M2M one, but now called *Ladder2FIACRE*, analogous to the preceding case;
- The tool's input – the TC6 model that is the PLCOpen Editor output and contains the edited LD program, i.e., the TC6 XML file.

Besides that, we also have the ATL meta-model and the Ecore meta-meta-model, but they are in dashed lines because they are already included in the Modeling Eclipse environment. All the conformation relations between the models and meta-models are shown in Figure 5.

Given all the necessary components cited above, the two ATL transformations, running in sequence, make the translation between the TC6 XML file from the PLCopen Editor and the FIACRE abstract model. This abstract model can be unparsed[7] to result in a FIACRE textual file. Maybe you are wondering why there are two transformations instead of one, which, besides all, requires an extra meta-model.

Indeed, it would be possible to make a single transformation, directly between the input and the target models. However, the TC6 XML Schema was designed to hold all the information about the PLC configurations and programs, not just the LD programs, neither just the logical data, but all the details, including the graphical ones. Besides that, the LD program logic control is not evident in the TC6 structure: it is needed to be reconstructed from the graphical information about what is connected with what, i.e., which are the LD elements connected to one another. That is why the intermediate *Ladder* meta-model was introduced – Figure 6 – facilitating the ATL rules since its structure is well-organized specifically for the LD program representation. Besides that, this intermediate meta-model allows other LD editors to use the same Translation Tool – it is only

7 As explained before, in Section 5.1.2

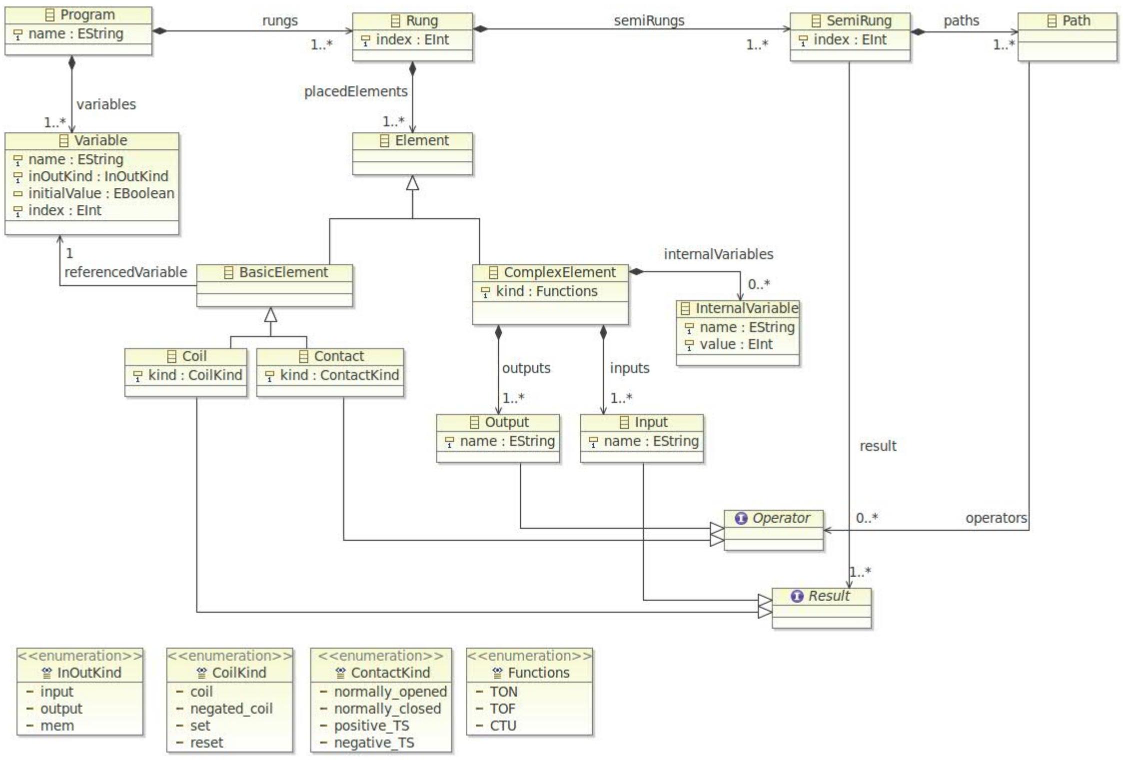


Figure 6: UML Class Diagram to represent a Ladder Program

necessary to make another transformation, using ATL or other MDE appropriate language, for getting the *Ladder* model from the output of your desired LD software editor. Therefore, the *Ladder* meta-model is a kind of convergence point: once you have obtained its instance that represents your LD program, you can use the Tool for getting the FIACRE model.

The FIACRE model generated at the end of this translation chain contains the PLC component, with the Scan Process and the Functional Blocks Processes. These elements have been explained in Section 4.1.2. There it has also the reason for the Semi-rung Class in the *Ladder* meta-model, introduced to deal with LD programs that use Functional Blocks.

5.2 PROJECT ARCHICTETURE

In this section we will present the desired architecture for this project, and in the next ones, the developed work. As seen above, in the previous work a chain between the Ladder Diagram programs, written in PLCOpen Editor, and the FIACRE abstract models was constructed. Nonetheless, there are two problems left, because the PLC programmers still need to have knowledge about the formal verification techniques for:

- writing the formal properties to be verified, i.e., translating the informal specification into a formal one;
- writing the formal model to represent the plant – the system to be controlled – since the FIACRE model obtained by the previous chain refers only to the PLC – the Scan process and the Functional Blocks process.

For the first problem, we propose here the Business Property Language, or BPL, that enables the PLC programmer to write properties he wants to check about his system and to automatically translate it to Temporal Logic. For the second issue, there is a partial solution via an improvement in the previous translation tool, which will now generate two FIACRE models, one where the PLC component is composed with a generic Plant without restrictions – analogously to buttonholes for the inputs and LEDs for the outputs, to check the good writing of the LD program (the *Fiacre Complete Model for Generic Properties* at the figure) – and a second model where the PLC is composed with an unfinished Plant Component (*Fiacre Incomplete Model for Specific Properties Verification* at the figure), which needs the manual formal modeling for the processes that compose it – that is why we said "partial solution". Both approaches are shown in Figure 7, and explained in the next sections.

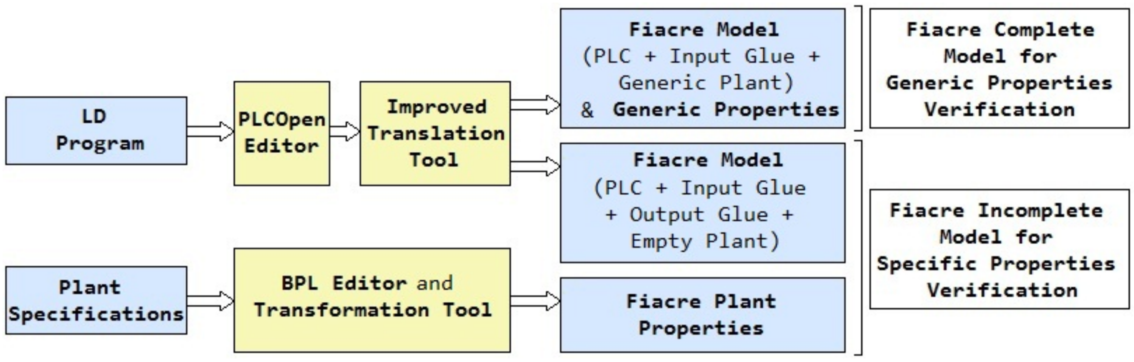

LD Program
PLCOpen Editor
Improved Translation Tool
Fiacre Model (PLC + Input Glue + Generic Plant) & Generic Properties
Fiacre Model (PLC + Input Glue + Output Glue + Empty Plant)
Fiacre Complete Model for Generic Properties Verification
Plant Specifications
BPL Editor and Transformation Tool
Fiacre Plant Properties
Fiacre Incomplete Model for Specific Properties Verification

Figure 7: Simplified Schema of the Translation Chains

5.3 TRANSLATION TOOL IMPROVEMENT

In this section we will present what was done in the previous Translation Tool[8], for trying to solve the Plant modeling problem. This tool, as we can see at Figure 5, is formed by two M2M ATL transformations, which allow to pass from an edited LD program – a TC6 model – to the *Ladder* model, and then to the FIACRE abstract model (conforming to the FIACRE Ecore meta-model).

There were two problems. One referent to the FIACRE abstract model, which only contains the PLC component (the Scan process

8 Introduced above in Section 5.1.3

composed with the Functional Blocks processes), lacking the Plant modeling and the glues processes to bind it with the PLC. The second issue is the missing of an unparser to obtain FIACRE textual models from the abstract ones, generated by the translator. They are discussed below.

5.3.1 *FIACRE target model complement*

To address the lacking of a Plant component mentioned above, this project uses only a partial solution. We say "partial" because it does not solve the problem of modeling the Plant system without having formal knowledge. This means that there is not a language appropriated for the technicians that have not a foundation to use the FIACRE model, for example, and that serves to model the plant.

The new architecture of the translation tool is shown in Figure 8, where we can see which part has been changed. Our goal was to reach two target models. The first one, called *A* in the picture, would be the general verification of the LD program, i.e., to analyze if the LD program is well-written, if it has any race condition in any variable, if there is deadlock, and so on. The second model, *B*, was to verify the complete controlled system, which means the PLC with the Plant model.

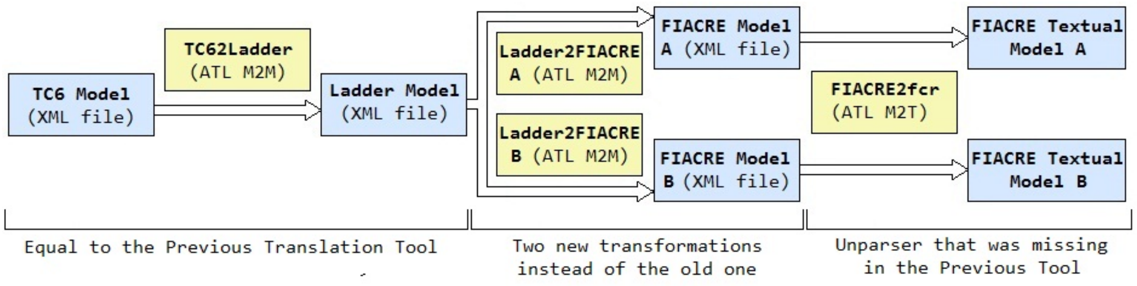

TC6 Model (XML file)
TC62Ladder (ATL M2M)
Ladder Model (XML file)
Ladder2FIACRE A (ATL M2M)
Ladder2FIACRE B (ATL M2M)
FIACRE Model A (XML file)
FIACRE Model B (XML file)
FIACRE2fcr (ATL M2T)
FIACRE Textual Model A
FIACRE Textual Model B
Equal to the Previous Translation Tool
Two new transformations instead of the old one
Unparser that was missing in the Previous Tool

Figure 8: Schema of the Improved Translation Tool

With this purpose, we use the second ATL transformation in the previous chain as a basis to be incremented. This program already allowed to go from the *Ladder* model to the FIACRE model with the PLC component. What was done was the addition of rules for obtaining the model of the rest of the system, as we explained above. Thereby, the second ATL program was split in two – *Ladder2FIACREgeneric*, resulting in the model *A*, and *Ladder2FIACREspecific*, resulting in *B*. The *TC62Ladder* transformation was not changed, because it works for the data organization, and not really for its processing.[9]

To the first model, *A*, we added a generic Plant Component. It is generic because it represents the input and output variables of the LD program, changing independently. Thereby, there are no real plant

9 See Section 5.1.3.

restrictions on the variables' behavior. The only limitations on the entire system's comportment are imposed by the PLC control. In the real system we can say that it never happens, because generally the system without control already has some kind of restrictions in its variables – since they are somehow connected. That is why this model is used specifically for the LD program verification, despite of the plant that it controls. In it, the PLC is composed with the Plant through the aforementioned "glue" process.[10] The plant, in turn, is a composition of instances of input and output variable processes – one for each of these variables present in the LD program. The input and the output processes declarations have two states: one for the true and other for the false value. In Section 6.1 and in Appendix B we will see this model for the APS problem.

The second model, *B*, has also the PLC and the Plant connected by the glue processes. However, the Plant is not complete. It is just a structure of a component, with the appropriate ports – related, once more, to the input and output variables of the LD program – but without any process instance. This is where someone with formal verification skills must use the FIACRE language for modeling the plant processes and its bindings. One example of this model is also given in the Final Results chapter and in Appendix C – where the part that was automatically engendered and the one that was manually written are indicated.

5.3.2 *FIACRE model to FIACRE text*

In the topic above we treat the improvement of the previous Translation Tool as regards to the FIACRE model complement that is obtained as target. Nevertheless, this "FIACRE model" is actually an abstract model, or in other words, an instance of the FIACRE Ecore model – i.e., an XML file. The formal model that we want to obtain, however, is the FIACRE textual model, which holds as entry for the model-checking tool – in this project, the aforementioned TINA toolkit.

Thereby, it is necessary to construct the textual model file from the the abstract model information. As we have seen before the ATL language and toolkit provides a kind of transformation, called M2T, which works as an *unparser* – see Section 5.1.2. Therefore, to address this problem an ATL M2T transformation has been implemented, which binds the Classes and Relations in the abstract FIACRE model to the textual constructions of the FIACRE syntax, respecting the language semantic. This ATL program is called *fiacre2fcr* due to the files extensions – *fiacre* for the abstract model, and *fcr* for the textual one. In this way, the first planned chain of the project is complete and allows to obtain models of the format presented in Section 4.1.2. Two

10 See Section 4.1.2

examples of this target file can be seen in Appendix B and Appendix C, for the APS Case Study.

In these appendices we can also see that, in the case of a FIACRE abstract model containing a generic Plant – i.e., with the input and output process declarations and their instances in the Plant component – some FIACRE properties are generated. These properties are generic ones, and refer to the LD program writing. Therefore, in the case of the generic FIACRE abstract model – target of the *Ladder2FIACREgeneric* transformation – the *fiacre2fcr* transformation returns a FIACRE textual model ready to be verified in the model-checking tool TINA, since it has a complete model and the properties to be checked. The target of the *Ladder2FiacreSpecific* transformation, on the other hand, needs to be edited to add the process of the plant – see previous topic – and the specific properties – which will be treated in the next topic.

5.4 PROPOSED LANGUAGE – BPL

The BPL – Business Property Language – is part of this project's developed work, and its aim is to make it easy for the technicians to translate the informal specification into the formal one. It is a textual natural language, dedicated to the properties writing, which can be automatically transformed in FIACRE formal ones – seen at Section 4.3.

In fact, the best solution would be to use a preexisting business language, that the technicians were familiar with. For example, in hydraulic systems, it is usual to write a route-step diagram – like the one shown in Figure 9, used in [8] for APS – from which we can extract some desired properties. But this procedure is not formalized in a norm or pattern, nor has it a software editor, and it is used just for specific kinds of systems. Therefore, in the absence of a general standard for the system informal properties, the proposed solution was this textual language, introduced below. After its syntax presentation, we will see how it has been implemented and how the transformation to FIACRE properties has been done. We will also see an example of use, for the APS case study.

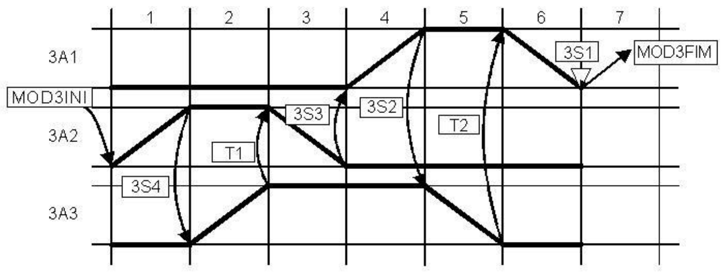


Figure 9: An example of Route-step Diagram

5.4.1 *Syntax*

The Syntax of BPL is given in Appendix G. We have used a variant of the Extended Backus-Naur Form for describing it – the same as the one used in the FIACRE syntax description at [6].

As mentioned before, BPL is a kind of textual natural language for expressing system properties. The basic body of a BPL text is shown below:

```
/* BPL text */
modelName model properties
        /* properties statements */
        using
        /* declarations */
        /* propositions */
        atomic propositions {
                /* atomic propositions declaration */
        }
```

There we can see properties statements, declarations, propositions and atomic propositions. The atomic propositions refer to the states that we want to observe in the plant model, like the one where a cylinder is retracted, for example. As we explained before, we have just a partial solution for the plant modeling problem, so, after the translation chain, the processes that compose the plant must be modeled, "by hand", inside the FIACRE model obtained automatically. Each one of these processes will have a name – *processName* – and it will be instantiated inside the Plant component. So, the atomic propositions must be written as below, where *int* refers to the position of the process' instance inside the Plant component, and *stateName* is the name used in the FIACRE process declaration for its state that we want to observe:

```
processName_int_stateName
```

The propositions are facultative, used when we want to make a construction built by one or more atomic propositions through the conjunction, disjunction and negation operations (*and*, *or* and *not*, respectively). Their syntax is:

```
proposition propName:
                /* atomic propositions constructions */
```

The declarations are also facultative, this time used for making it easy to write properties. They are, in fact, simple properties – which we explained above – but that will be verified only inside the properties statements. They are declared as:

```
declaration declName:
                /* simple properties */
```

Finally, we have the properties statements, written as:

```
property propertyName:
                /* general, simple or complex properties */
```

The general property is used for testing if the entire system is free of deadlock. The simple properties are written by using the propositions and atomic propositions declarations given at the end of the BPL text – which must not be declared again, just mentioned by their names. The complex properties, in turn, are built by declaring simple properties or just by mentioning the name of the ones that were already stated in the declarations, explained before. The complete BPL syntax with the properties declarations possibilities are given in Appendix G.

5.4.2 *Domain-Specific Language and Xtext*

The BPL language presented above is an example of a Domain-Specific Language, or DSL. A DSL, as the name says, is constructed to provide a way to express basically anything, in a particular domain or point of view. In this case, BPL is used to write the desired systems' behaviors, i.e., the systems' properties. There are some tools for implementing DSLs, and in this project we have chosen the Xtext.

Xtext[11] is a language development framework, inserted in the MDE context, used to define a textual syntax, like a DSL or a general purpose programming language. It also allows the generation of a meta-model[12] or it can be based on a preexisting one; actually, Xtext can hold with these two approaches mixed to improve an already defined meta-model and build a syntax language.

The Xtext grammar is a domain-specific language itself, designed for the description of textual languages. Once a concrete syntax is defined, it is mapped into a semantic model representation, which is the meta-model that we were talking about. Each textual file that is written using the defined syntax can be transformed in a model, conforming to the meta-model, through the *parser* procedure – the inverse of the unparser, introduced in Section 5.1.2. The parser algorithm that Xtext implements is the Left Recursion one, or LL(*), which analyzes the content of the textual file from left to right. To write, for example, the propositions, mentioned in the previous topic – which have operations between one or more atomic propositions, like conjunction, disjunction and negation – this characteristic needs to be carefully considered for obtaining an appropriate and free-of-errors model.

After the conceptual definition of the BPL syntax, it was implemented using Xtext so as to obtain:

11 http://www.eclipse.org/Xtext/
12 Meta-model concept introduced in Section 5.1.1 and Section 5.1.2.

- its meta-model – named BPL – the necessary plug-ins to create its instance models – with the extension "bpl" – and an editor to modify them;
- a simple editor for the textual BPL files – which have the extension "bpltext".

The textual BPL files are the ones which have the model properties that we want to check, written under the syntax defined with the Xtext. The BPL instance models are XML files that contain the same properties, but arranged conforming to their semantic expressed in their meta-model structure. Next, we will give an example of BPL textual files, for the APS case study, and after we show what is done for transforming the BPL properties in FIACRE ones.

5.4.3 *BPL Example – APS Properties*

In Section 3.3 we saw the APS case study and also its desired properties, written in an informal fashion. Now, with the BPL definition, we can write them according to its syntax. Therefore, we have built a table with each APS informal specification mentioned before and their respective BPL properties. In this same table, we can also see their formal meaning as LTL expressions and the FIACRE properties that are obtained after the translation – which is explained below in the next topic. The table can be found at Appendix F. We can also see the complete BPL text for the APS properties in Appendix D.

5.4.4 *BPL Properties to RT-FIACRE properties*

After we have implemented the BPL syntax with Xtext, it became possible to write the properties in a text file, using the simple editor engendered in the process. The appropriate links between the syntax elements – in the textual file – and the semantic ones – defined in the meta-model built by Xtext through the grammar definition – were also generated. With these links, it was easy to construct a *parser* to obtain BPL models from the BPL textual file – see Section 5.4.2 above.

With the BPL meta-model, named *BPL Ecore model* – because its meta-meta-model is an Ecore one – and its models – whose extension is "bpl" – it was possible to make an ATL transformation, named *BPL2FIACRE*, of the kind M2T[13], to directly obtain the FIACRE properties textual model. Otherwise, we could have done an M2M[14] ATL transformation, to obtain a FIACRE properties abstract model (a XML file); however, the RT-FIACRE meta-model is still in the process of development and so it is always changing. For this reason, we have chosen to

13 Model to Text transformation
14 Model to Model transformation

use a model-to-text translation to avoid the use of the RT-FIACRE meta-model.

For constructing the transformation, we needed to establish the semantic meaning of the BLP properties. Therefore, we have written the LTL expression for each possible BPL property structure, and then we have put them in FIACRE declarations. For the general and simple properties – mentioned in Section 5.4.1 – and for some of the complex ones, there is a table in Appendix H showing the relation between BPL, LTL and FIACRE. For the other complex properties that can be built with the simple ones, there is no FIACRE pattern, and we use the LTL formula instead[15]. There are two kinds of complex statements, the *if-then* and the *whenever-then*, each one of them using two simple properties and having the basic LTL expression below:

- if *simple_prop1* then *simple_prop2*:

```
(simple_prop1 => simple_prop2)
```

- whenever *simple_prop1* then *simple_prop2*:

```
[] (simple_prop1 => simple_prop2)
```

Where *simple_prop1* and *2* are the LTL expressions given in the second column of the table mentioned above for each one of the simple properties defined by BPL.

In Section 6.1 we will see an example of this transformation target, for the APS case study. This result can be found at Appendix E. Here, just as an example, you can see how one informal specification of the APS case study is written:

Informal Property	BPL expression
Infinitely often, cylinder A1 will be retracted, cylinder A2 will be extended and cup A3 will be turned on at the same time.	property example: (Cylinder.1.retracted and Cylinder.2.extended and Cup.3.closed) will be true an infinite number of times

Where the numbers *1*, *2* and *3* refer to the process instance position inside the Plant composition – if you see the FIACRE Plant declaration, at Appendix C, the Plant is:

```
component Plant [···] is
par * in
Cylinder [a3A1Port, s3S1Port, s3S2Port](true)
|| Cylinder [a3A2Port, s3S3Port, s3S4Port](true)
|| Cup [a3A3Port](false)
end par
```

15 Remembering that FIACRE properties can be written as patterns or as LTL expressions – see Section 4.3.

6 FINAL RESULTS

In this chapter, we will present a brief overview of the entire work, showing what was done and how we must proceed to verify an LD program. We will also see the complete verification chain for the APS example.

At Figure 10, there is a schematic picture for the constructed tool. There are two chains, one for the system modeling, and the other for its properties writing. We will start by the first one, which corresponds to the *Improved Translation Tool* box at Figure 7. Initially, we must have an edited LD program as a TC6 XML file. It can be achieved through the PLCOpen Editor of Beremiz Project, which has a graphical LD editor[1]. Using this file as an entry, in the Eclipse environment, we run the first ATL transformation, *TC62Ladder*, to obtain the corresponding *Ladder* model. This transformation must have, as an entry meta-model, the TC6 Ecore, and as a target meta-model, the *Ladder* Ecore. After that, we use the generated *Ladder* model in the two next ATL transformations – *Ladder2FIACREgeneric* and *Ladder2FIACREspecific*, or, respectively, *A* and *B* in the picture – this time having the *Ladder* Ecore as the entry meta-model and the FIACRE Ecore as the target one. Thereby we obtain, respectively, a FIACRE abstract model containing the PLC component bound by the "glue" process to a Plant, which is formed by input and output variables processes, and other FIACRE model with the same PLC but with an incomplete Plant model.[2]

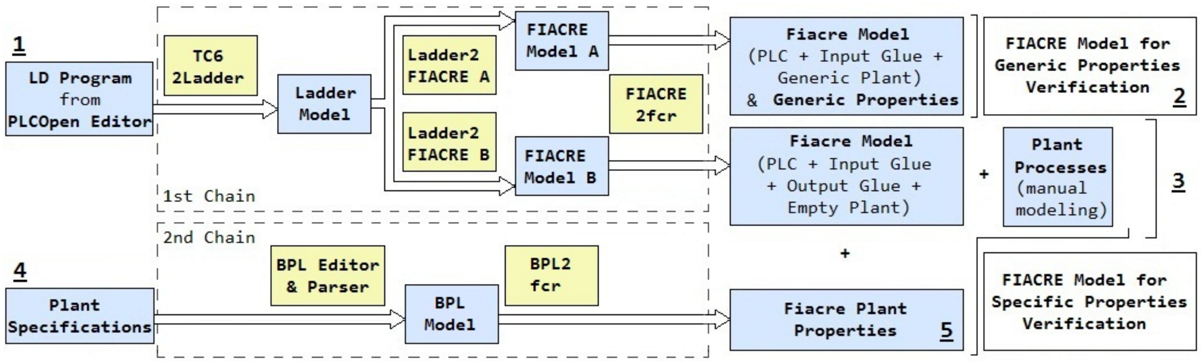

1
LD Program from PLCOpen Editor
TC6 2Ladder
Ladder Model
Ladder2 FIACRE A
Ladder2 FIACRE B
FIACRE Model A
FIACRE Model B
FIACRE 2fcr
1st Chain
Fiacre Model (PLC + Input Glue + Generic Plant) & Generic Properties
FIACRE Model for Generic Properties Verification
2
Fiacre Model (PLC + Input Glue + Output Glue + Empty Plant)
+
Plant Processes (manual modeling)
3
2nd Chain
4
Plant Specifications
BPL Editor & Parser
BPL Model
BPL2 fcr
+
Fiacre Plant Properties
5
FIACRE Model for Specific Properties Verification

Figure 10: Developed Tool Architecture

After this, each one of these FIACRE abstract models must be unparsed, by the ATL M2T transformation *fiacre2fcr* – named like this because the abstract models have the extension "fiacre" and the textual ones have the extension "fcr". In the first textual model some

1 See Section 3.1.1.
2 For futher details about the first chain, see Section 5.1.3 and Section 5.3.1.

general properties to be verified on the LD program will be already engendered. This does not happen with the second one, which must be completed with processes' declarations – referring to the plant system – and their instantiations inside the Plant component.[3]

Now we will see the second chain, still in Figure 10 – which corresponds to the *BPL Editor and Transformation Tool* at Figure 7. In parallel with the steps explained above, we can build the system's desired properties to be verified. First of all, we write the system specification using the BPL syntax, also in the Eclipse environment. Thus, we use the ATL M2T transformation *BPL2FIACRE*[4] to obtain these specifications as FIACRE properties, which must be joined to the end of the FIACRE textual model – the one that has the plant processes edited manually.

Therefore, we will have two FIACRE textual models with a modeled system and its properties. At this moment, we use the TINA toolbox[5] – specifically the modules *tina* and *selt* – to perform the model-checking. In the next topic, we will talk about the APS example and its results.

6.1 APS CASE STUDY RESULTS

Along this entire document, we have been dealing with the APS case study. It was introduced in Chapter 3, and its modeling procedure and properties writing have been treated in Chapter 4. For organizing the ideas, Figure 10 above have some numbers to indicate what is shown in this report's attachments. In the next table, we indicate exactly which appendix refers to each item.

Element Number	Appendix
1	Appendix A
2	Appendix B
3	Appendix C
4	Appendix D
5	Appendix E

Table 1: Connections between the elements in Figure 10 and the Appendices

The results of the model-checking of the APS properties on the given models are shown in Appendix F. There, we can find a table, already mentioned in this document, in which are the APS specifications, their respective BPL, LTL and FIACRE expressions and, in the last column, the results and the verification time (starting at the FIACRE models) using TINA toolbox. Actually, for these properties, all results are positive, because the system controlled by its LD program has the

3 See Section 5.3.2.
4 See Section 5.4.4.
5 Introduced in Section 4.2.

desired behavior. We have also tested some properties that should not to be true, just as a test, but we have not put them here to avoid misleading results.

7 CONCLUSION AND FUTURE WORK

In this document we have seen the project motivation as well as the development phases. The implemented tool is based in the MDE methodology and allows to perform the Reinterpretation approach for the formal verification of the Ladder Diagram programs. The formal model used is the FIACRE language, which allows to represent the system behavior and also its properties – as patterns or LTL formulas. A Translation Tool's preliminary version – implemented by the same student involved in this work – has been now improved, resulting in a first chain that, from an LD program, leads to a FIACRE model. Besides that, a business language, called BPL, has been constructed, which enables to write in an informal manner the plant specifications. Finally, a second translation chain was constructed, which allows to generate formal FIACRE properties from the BPL ones. With these two automatic translation chains, it is possible to proceed with the model-checking in a preexisting tool, as TINA, using as entries the FIACRE models and the FIACRE properties obtained as target before.

The developed tool can already be used for LD programs' verification. However, there are several perspectives of future work. First, the BPL language proposed for dealing with the informal specification can be improved or changed. As an example of improvement, the way of writing the atomic propositions can be changed, removing the integer numbers that express the instance positions inside the Plant component and, instead of that, using a name more intuitive for the user, which addresses directly to the process instance. The BPL language was elaborated as a first step to allow the writing of the system properties by the technicians. Nevertheless, a better solution would be to find a preexisting and standardized language with which the programmers are already familiar.

Other proposal for future work is about the plant modeling problem, explained at Section 5.3.1. It still needs to be done manually, because its characteristics cannot be inferred from the LD program. Remembering, the translation tool has two target FIACRE models. The first has a system modeling included, but it does not refer to the real plant, because there are no restrictions in the variables change – it is just for the verification of the LD program writing. The second model, in turn, does not have the plant representation included. To address this issue a Domain Specific Language, as BPL, must be formulated, and also a chain to go from the model written in this language to the FIACRE model. The elaboration of this kind of DSL is not at all trivial, because it must be capable of representing any behavior that the system can have, and besides that, it must not require formal models skills.

Finally, a last proposal cited here will be about the model-checking result. The model-checking verification results true or false for each property, and in the case of a false one, it gives a counterexample. The problem is that this counterexample is given in a formal representation. Thereby, the proposal is to build a translation chain in the inverse path: from the formal models to informal ones, which the technicians are able to understand, and, by doing so, to fix the LD program where it falls.

BIBLIOGRAPHY

[1] A. Aiken, M. Fähndrich, and Z. Su. Detecting Races in Relay Ladder Logic Programs. *Lecture Notes in Computer Science*, 1384: 184–200, June/July 1998.

[2] Darlam Bender, Benoit Combemale, Xavier Crégut, Jean-Marie Farines, and Francois Vernadat. Ladder Metamodeling & Plc Program Validation through Time Petri Nets. *Fourth European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA)*, 5095:121–136, June 2008.

[3] C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Kluwer Academic Publishers, Massachussets, USA, 2nd edition, 1999.

[4] E. M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

[5] International Electrotechnical Commission. *IEC 61131-3: Programmable Controllers – Part 3*. Geneva, Switzerland, 2nd edition, 2003.

[6] B. Berthomieu et al. The Syntax and Semantics of Fiacre. January 2011.

[7] J.M. Farines, M. H. de Queiroz, M. F. de Souza, A. M. M. Carpes, and F. Vernadat. Modeling and Verification of Plc Programs by using Fiacre Tool Chain. *TOPCASED DAYS 2011*, 2011.

[8] J.M. Farines, M.H. de Queiroz, V.G. da Rocha, A.M.M. Carpes, F. Vernadat, and X. Cregut. A Model-Driven Engineering Approach to Formal Verification of Plc Programs. *Emerging Technologies Factory Automation (ETFA), 2011 IEEE 16th Conference on*, pages 1–8, September 2011.

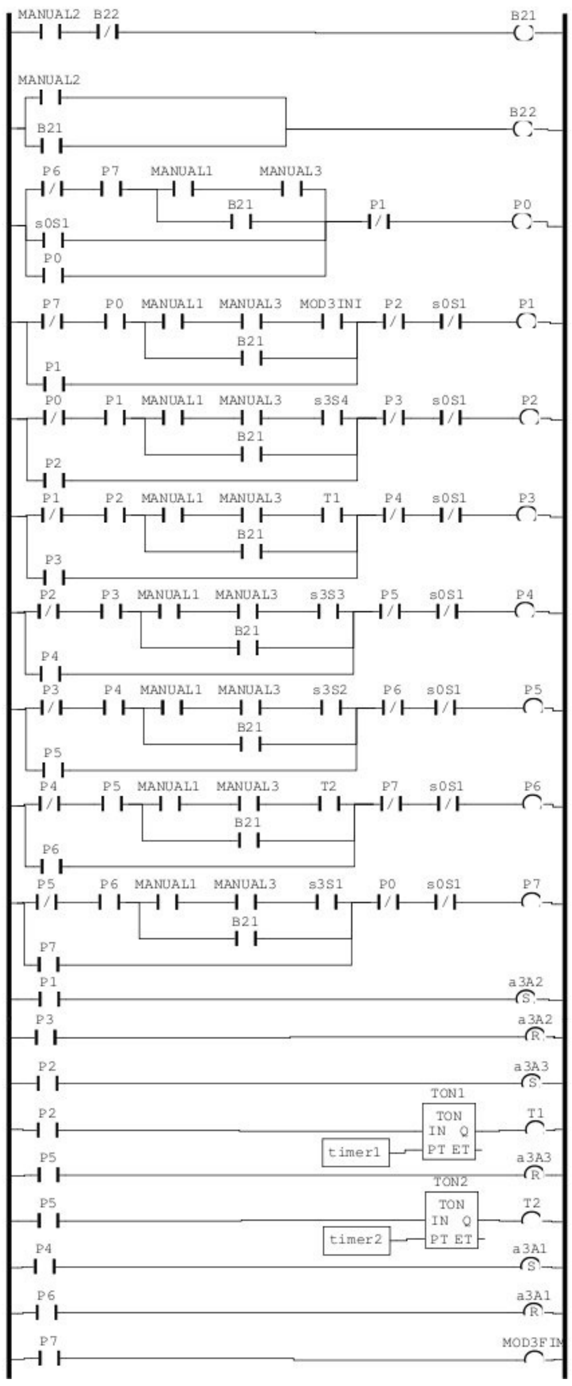
[9] G. Frey and L. Litz. Formal Methods in Plc Programming. *Proceedings of the IEEE SMC 2000*, 4:2431–2436, October 2000.

[10] Stuart Kent. *Model Driven Engineering*. Springer, 2002.

[11] A. Nouha and S. Dal Zilio. Real-time Extensions for the Fiacre Modeling Language. *International Summer School on Modelling and Verifying Parallel Processes (MOVEP' 2010)*, pages 1–6, June/July 2010.

[12] Mateus F. Souza. Modelagem e Verificação de Programas de Clp escritos em Diagrama Ladder. *Dissertação de Mestrado (Master), PPGEAS-UFSC*, Outubro 2010.

A
APS LD PROGRAM


MANUAL2 B22 B21
MANUAL2
B21 B22
P6 P7 MANUAL1 MANUAL3
B21 P1 P0
s0S1
P0
P7 P0 MANUAL1 MANUAL3 MOD3INI P2 s0S1 P1
B21
P1
P0 P1 MANUAL1 MANUAL3 s3S4 P3 s0S1 P2
B21
P2
P1 P2 MANUAL1 MANUAL3 T1 P4 s0S1 P3
B21
P3
P2 P3 MANUAL1 MANUAL3 s3S3 P5 s0S1 P4
B21
P4
P3 P4 MANUAL1 MANUAL3 s3S2 P6 s0S1 P5
B21
P5
P4 P5 MANUAL1 MANUAL3 T2 P7 s0S1 P6
B21
P6
P5 P6 MANUAL1 MANUAL3 s3S1 P0 s0S1 P7
B21
P7
P1 a3A2 S
P3 a3A2 R
P2 a3A3 S
TON1
P2 TON IN Q PT ET T1
timer1
P5 a3A3 R
TON2
P5 TON IN Q PT ET T2
timer2
P4 a3A1 S
P6 a3A1 R
P7 MOD3FIN

B APS FIACRE MODEL WITH GENERIC PLANT

```
// Fiacre Textual Model
// It refers to the APS controlled system

type indexOut is 0 .. 2
type arrayIn is array 4 of bool
type arrayOut is array 3 of bool

process Input [sendVar: out bool] is
        states varTrue, varFalse
        init to varFalse
        from varFalse
                select
                        sendVar! false; loop
                        []sendVar! true; to varTrue
                end select
        from varTrue
                select
                        sendVar! true; loop
                        []sendVar! false; to varFalse
                end select

process InputGlue [writeInputs: out arrayIn, syncPort: in
    arrayOut, s3S4Port: in bool, s3S3Port: in bool, s3S2Port: in
    bool, s3S1Port: in bool] is
        states writing, synchronizing, getInput_s3S4, getInput_s3
            S3, getInput_s3S2, getInput_s3S1
        var irrelevantArray: arrayOut, s3S4: bool, s3S3: bool, s3
            S2: bool, s3S1: bool
        init to getInput_s3S4
        from writing
                writeInputs! [s3S4, s3S3, s3S2, s3S1]; to
                    synchronizing
        from synchronizing
                syncPort? irrelevantArray; to getInput_s3S4
        from getInput_s3S4
                s3S4Port? s3S4; to getInput_s3S3
        from getInput_s3S3
                s3S3Port? s3S3; to getInput_s3S2
        from getInput_s3S2
                s3S2Port? s3S2; to getInput_s3S1
        from getInput_s3S1
                s3S1Port? s3S1; to writing

process Output [receiveVar: in arrayOut] (arrayIndexVar: indexOut
    ) is
        states varTrue, varFalse
        var outputsVarsArray: arrayOut
        init to varFalse
        from varFalse
                receiveVar? outputsVarsArray; if outputsVarsArray
                    [arrayIndexVar] then to varTrue else loop end
        from varTrue
                receiveVar? outputsVarsArray; if outputsVarsArray
                    [arrayIndexVar] then loop else to varFalse
                    end

process Scan
        [portInputs: in arrayIn, portOutputs: out arrayOut,
            portTON1_IN: out bool, portTON1_Q: in bool, portTON2_
            IN: out bool, portTON2_Q: in bool]
        is
        states initial, writing, final, rung_1, rung_2, rung_3,
            rung_4, rung_5, rung_6, rung_7, rung_8, rung_9, rung
            _10, rung_11, rung_12, rung_13, rung_15, rung_17,
            rung_18, rung_19, rung_14, rung_141, rung_16, rung
            _161
        var
                varsIn: arrayIn, s3S4: bool := false, s3S3: bool
                    := false, s3S2: bool := false, s3S1: bool :=
                    false, MANUAL1: bool := false, MANUAL2: bool
                    := false, MANUAL3: bool := false, B22: bool
                    := false, B21: bool := false, P1: bool :=
                    false, P2: bool := false, P3: bool := false,
                    P4: bool := false, P5: bool := false, P6:
                    bool := false, P7: bool := false, P0: bool :=
                     false, s0S1: bool := false, MOD3INI: bool :=
                     false, MOD3FIM: bool := false, a3A3: bool :=
                     false, a3A2: bool := false, a3A1: bool :=
                    false, T1: bool := false, T2: bool := false,
                    TON1_IN: bool := false, TON2_IN: bool :=
                    false, TON1_Q: bool := false, TON2_Q: bool :=
                     false
        init to initial
        from initial
                portInputs? varsIn; s3S4 := varsIn[0]; s3S3 :=
                    varsIn[1]; s3S2 := varsIn[2]; s3S1 := varsIn
                    [3]; to rung_1
        from writing
                portOutputs! [a3A3, a3A2, a3A1]; to final
        from final
                wait[1, 1]; to initial
        from rung_1
                wait[0, 0]; B21 := ((not B22) and MANUAL2); to
                    rung_2
        from rung_2
                wait[0, 0]; B22 := (MANUAL2 or B21); to rung_3
        from rung_3
                wait[0, 0]; P0 := ((P0 and (not P1)) or (((P7 and
                     (B21 and (not P6))) and (not P1)) or ((s0S1
                    and (not P1)) or ((P7 and ((not P6) and (
                    MANUAL1 and MANUAL3))) and (not P1))))); to
                    rung_4
        from rung_4
                wait[0, 0]; P1 := (((not P7) and (P0 and (MANUAL
                    1 and (MOD3INI and ((not P2) and MANUAL3)))))
                     and (not s0S1)) or (((P1 and (not P2)) and (
                    not s0S1)) or ((B21 and ((not P7) and (P0 and
                     (not P2)))) and (not s0S1)))); to rung_5
        from rung_5
                wait[0, 0]; P2 := (((P2 and (not P3)) and (not s0
                    S1)) or (((P1 and ((not P3) and (B21 and (not
                     s0S1)))) and (not P0)) or ((s3S4 and (P1 and
                     ((not P3) and (MANUAL1 and ((not s0S1) and
                    MANUAL3))))) and (not P0)))); to rung_6

        from rung_6
                wait[0, 0]; P3 := (((B21 and (P2 and ((not P1)
                    and (not P4)))) and (not s0S1)) or (((P3 and
                    (not P4)) and (not s0S1)) or ((T1 and (P2 and
                     (MANUAL1 and ((not P1) and ((not P4) and (
                    not s0S1)))))) and MANUAL3))); to rung_7
        from rung_7
                wait[0, 0]; P4 := ((((not s0S1) and (P3 and (s3S3
                     and ((not P5) and (MANUAL1 and (not P2))))))
                     and MANUAL3) or ((((not s0S1) and (not P5))
                    and P4) or (((not s0S1) and (B21 and (P3 and
                    (not P5)))) and (not P2)))); to rung_8
        from rung_8
                wait[0, 0]; P5 := ((((not P6) and P5) and (not s0
                    S1)) or (((B21 and ((not P6) and ((not P3)
                    and P4))) and (not s0S1)) or (((not P6) and
                    ((not P3) and (MANUAL3 and (P4 and (s3S2 and
                    MANUAL1))))) and (not s0S1)))); to rung_9
        from rung_9
                wait[0, 0]; P6 := ((((not P7) and (not s0S1)) and
                     P6) or ((((not P4) and ((not P7) and ((not s
                    0S1) and (P5 and (MANUAL1 and MANUAL3)))))
                    and T2) or (((not P4) and (B21 and ((not P7)
                    and (not s0S1)))) and P5))); to rung_10
        from rung_10
                wait[0, 0]; P7 := ((((not s0S1) and ((not P5) and
                     (P6 and (MANUAL1 and (MANUAL3 and s3S1)))))
                    and (not P0)) or ((((not s0S1) and ((not P5)
                    and (P6 and B21))) and (not P0)) or (((not s0
                    S1) and P7) and (not P0)))); to rung_11
        from rung_11
                wait[0, 0]; a3A2 := (a3A2 or P1); to rung_12
        from rung_12
```

```
                wait[0, 0]; a3A2 := (a3A2 and (not P3)); to rung
                    _13
        from rung_13
                wait[0, 0]; a3A3 := (a3A3 or P2); to rung_14
        from rung_15
                wait[0, 0]; a3A3 := (a3A3 and (not P5)); to rung
                    _16
        from rung_17
                wait[0, 0]; a3A1 := (a3A1 or P4); to rung_18
        from rung_18
                wait[0, 0]; a3A1 := (a3A1 and (not P6)); to rung
                    _19
        from rung_19
                wait[0, 0]; MOD3FIM := P7; to writing
        from rung_14
                TON1_IN := P2; portTON1_IN! TON1_IN; to rung_141
        from rung_141
                portTON1_Q? TON1_Q; T1 := TON1_Q; to rung_15
        from rung_16
                TON2_IN := P5; portTON2_IN! TON2_IN; to rung_161
        from rung_161
                portTON2_Q? TON2_Q; T2 := TON2_Q; to rung_17

process TON [portIN: in bool, portQ: out bool, portTimer: none]
    is
        states idle, running, elapsed
        var IN: bool := false, Q: bool := false
        init to idle
        from idle
                select
                        portIN? IN; if IN then to running else
                            loop end
                        []portQ! Q; loop
                end select
        from running
                select
                        portIN? IN; if (not IN) then to idle else
                             loop end
                        []portTimer; Q := true; to elapsed
                        []portQ! Q; loop
                end select
        from elapsed
                select
                        portIN? IN; if (not IN) then Q := false;
                            to idle else loop end
                        []portQ! Q; loop
                end select

component PLC [portInputs: in arrayIn, portOutputs: out arrayOut]
     is
        port portTON1_IN: bool in [0,0], portTON1_Q: bool in
            [0,0], portTON1_Timer: none in [5,5], portTON2_IN:
```

```
            bool in [0,0], portTON2_Q: bool in [0,0], portTON2_
            Timer: none in [5,5]
        par * in
                Scan [portInputs, portOutputs, portTON1_IN,
                    portTON1_Q, portTON2_IN, portTON2_Q]
                || TON [portTON1_IN, portTON1_Q, portTON1_Timer]
                || TON [portTON2_IN, portTON2_Q, portTON2_Timer]
        end par

component Inputs [writeInputs: out arrayIn, readOutputs: in
    arrayOut] is
        port s3S4Port: bool in [0,0], s3S3Port: bool in [0,0], s3
            S2Port: bool in [0,0], s3S1Port: bool in [0,0]
        par * in
                InputGlue [writeInputs, readOutputs, s3S4Port, s3
                    S3Port, s3S2Port, s3S1Port]
                || Input [s3S4Port]
                || Input [s3S3Port]
                || Input [s3S2Port]
                || Input [s3S1Port]
        end par

component Outputs [readOutputs: in arrayOut] is
        par * in
                Output [readOutputs] (0)
                || Output [readOutputs] (1)
                || Output [readOutputs] (2)
        end par

component Plant [writeInputs: out arrayIn, readOutputs: in
    arrayOut] is
        par * in
                Inputs [writeInputs, readOutputs]
                || Outputs [readOutputs]
        end par


component APS is
        port portInputs: arrayIn in [0,0], portOutputs: arrayOut
            in [0,0]
        par * in
                PLC [portInputs, portOutputs]
                || Plant [portInputs, portOutputs]
        end par

APS

/* ----------------------------------------- */
/* Generic Properties about the Ladder Program */
/* ----------------------------------------- */

property notdead is deadlockfree
property plcReads is infinitelyoften APS/1/1/state initial
property plcWrites is infinitelyoften APS/1/1/state writing
property plcRestarts is infinitelyoften APS/1/1/state final
property absenceRaceCondition_a3A3 is
        ltl [] ((
                (([] APS/2/1/2/state varTrue) or ([] APS/2/1/2/
                    state varFalse))
                        and (([] APS/2/1/3/state varTrue) or ([]
                            APS/2/1/3/state varFalse))
                        and (([] APS/2/1/4/state varTrue) or ([]
                            APS/2/1/4/state varFalse))
                        and (([] APS/2/1/5/state varTrue) or ([]
                            APS/2/1/5/state varFalse))
                        and (([] APS/1/2/state idle) or ([] APS
                            /1/2/state running) or ([] APS/1/2/
                            state elapsed))
                        and (([] APS/1/3/state idle) or ([] APS
                            /1/3/state running) or ([] APS/1/3/
                            state elapsed)))
                => <> (([] APS/2/2/1/state varTrue) or ([] APS
                    /2/2/1/state varFalse)))
property absenceRaceCondition_a3A2 is
        ltl [] ((
                (([] APS/2/1/2/state varTrue) or ([] APS/2/1/2/
                    state varFalse))
                        and (([] APS/2/1/3/state varTrue) or ([]
                            APS/2/1/3/state varFalse))
                        and (([] APS/2/1/4/state varTrue) or ([]
                            APS/2/1/4/state varFalse))
                        and (([] APS/2/1/5/state varTrue) or ([]
                            APS/2/1/5/state varFalse))
                        and (([] APS/1/2/state idle) or ([] APS
                            /1/2/state running) or ([] APS/1/2/
                            state elapsed))
                        and (([] APS/1/3/state idle) or ([] APS
                            /1/3/state running) or ([] APS/1/3/
                            state elapsed)))
                => <> (([] APS/2/2/2/state varTrue) or ([] APS
                    /2/2/2/state varFalse)))
property absenceRaceCondition_a3A1 is
        ltl [] ((
                (([] APS/2/1/2/state varTrue) or ([] APS/2/1/2/
                    state varFalse))
                        and (([] APS/2/1/3/state varTrue) or ([]
                            APS/2/1/3/state varFalse))
                        and (([] APS/2/1/4/state varTrue) or ([]
                            APS/2/1/4/state varFalse))
                        and (([] APS/2/1/5/state varTrue) or ([]
                            APS/2/1/5/state varFalse))
                        and (([] APS/1/2/state idle) or ([] APS
                            /1/2/state running) or ([] APS/1/2/
                            state elapsed))
                        and (([] APS/1/3/state idle) or ([] APS
                            /1/3/state running) or ([] APS/1/3/
                            state elapsed)))
                => <> (([] APS/2/2/3/state varTrue) or ([] APS
                    /2/2/3/state varFalse)))

assert notdead
assert plcReads
assert plcWrites
assert plcRestarts
assert absenceRaceCondition_a3A3
assert absenceRaceCondition_a3A2
assert absenceRaceCondition_a3A1
```

C APS FIACRE MODEL WITH PLANT MODELED MANUALLY

```
// Fiacre Textual Model
// It refers to the APS controlled system

type arrayIn /∗ ... See Appendix B ... ∗/
type arrayOut /∗ ... ∗/
process InputGlue /∗ ... ∗/

process OutputGlue [readOutputs: in arrayOut, syncPort: in
    arrayIn, a3A3Port: out bool, a3A2Port: out bool, a3A1Port:
    out bool] is
        states reading, synchronizing, sendOutput_a3A3,
            sendOutput_a3A2, sendOutput_a3A1
        var varsOut: arrayOut, irrelevantArray: arrayIn, a3A3:
            bool, a3A2: bool, a3A1: bool
        init to synchronizing
        from reading
                readOutputs? varsOut; a3A3 := varsOut[0]; a3A2 :=
                     varsOut[1]; a3A1 := varsOut[2]; to
                    sendOutput_a3A3
        from synchronizing
                syncPort? irrelevantArray; to reading
        from sendOutput_a3A3
                a3A3Port! a3A3; to sendOutput_a3A2
        from sendOutput_a3A2
                a3A2Port! a3A2; to sendOutput_a3A1
        from sendOutput_a3A1
                a3A1Port! a3A1; to synchronizing

process Scan /∗ ... ∗/
process TON /∗ ... ∗/
component PLC /∗ ... ∗/

/∗ -------------------------------------- ∗/
/∗ The Cylinder and the Cup processes were modeled manually and
    refers to the Plant Processes ∗/

process Cylinder [pA: in bool, pRetractedS, pExtendedS: out bool]
     (bolRetracted : bool) is
        states  retracted, extending, extended, retracting
        var A: bool:= false
        init if bolRetracted then to retracted else to extended
            end
        from retracted
                select
                        pA? A; if A then to extending else loop
                            end
                        [] pRetractedS! true; loop
                        [] pExtendedS! false; loop
                end
        from extending
                select
                        pA? A; if A then loop else to retracting
                            end
                        [] wait[2,2]; to extended
                        [] pRetractedS! false; loop
                        [] pExtendedS! false; loop
                end
        from extended
                select
                        pA? A; if A then loop else to retracting
                            end
                        [] pRetractedS! false; loop
                        [] pExtendedS! true;loop
                end
        from retracting
                select
                        pA? A; if A then to extending else loop
                            end
                        [] wait[2,2]; to retracted
                        [] pRetractedS! false; loop
                        [] pExtendedS! false; loop
                end

process Cup[pA3: in bool] (On: bool) is
        states opened, closing, closed, opening
        var A3: bool := false
        init if On then to closed else to opened end
        from opened
                pA3? A3; if A3 then to closing else loop end
        from closing
                select
                        wait[4,4]; to closed
                        []pA3? A3; if A3 then loop else to
                            opening end
                end
        from closed
                pA3? A3; if A3 then loop else to opening end
        from opening
                select
                        wait[4,4]; to opened
                        []pA3? A3; if A3 then to closing else
                            loop end
                end
/* --------------------------------------- */

component Plant [s3S4Port: out bool, s3S3Port: out bool, s3S2Port
    : out bool, s3S1Port: out bool, a3A3Port: in bool, a3A2Port:
    in bool, a3A1Port: in bool] is
        par * in
```

```
                /* Instantiate the Plant processes here */
                /* ----------------------------------------------
                     */
                /* Cylinders and Cup Instantiations made manually
                     */
                Cylinder [a3A1Port, s3S1Port, s3S2Port](true)
                || Cylinder [a3A2Port, s3S3Port, s3S4Port](true)
                || Cup [a3A3Port](false)
                /* ----------------------------------------------
                     */
        end par

component APS is
        port portInputs: arrayIn in [0,0], portOutputs: arrayOut
            in [0,0], s3S4Port: bool in [0,0], s3S3Port: bool in
            [0,0], s3S2Port: bool in [0,0], s3S1Port: bool in
            [0,0], a3A3Port: bool in [0,0], a3A2Port: bool in
            [0,0], a3A1Port: bool in [0,0]
        par * in
                PLC [portInputs, portOutputs]
                || Plant [s3S4Port, s3S3Port, s3S2Port, s3S1Port,
                     a3A3Port, a3A2Port, a3A1Port]
                || InputGlue [portInputs, portOutputs, s3S4Port,
                    s3S3Port, s3S2Port, s3S1Port]
                || OutputGlue [portOutputs, portInputs, a3A3Port,
                     a3A2Port, a3A1Port]
        end par

APS

// Please insert here the model properties - which are obtained
    from the BPL2FIACRE transformation
```

D APS BPL PROPERTIES

```
  APS model properties:
      property general: dead lock free program

      property cylindersNotMoveTogheter: bothCylindersMoving is never true

      property closedCupWhileC1extends: whenever C1extends then CupClosed

      property closedCupWhileC2retracts: whenever C2retracts then CupClosed

      property openedCupWhileC2extends: whenever C2extends then CupOpened

      property openedCupWhileC1retracts: whenever C1retracts then CupOpened

      property catchBoxInfinitelyOften:
            step3 will be true an infinite number of times
      property dropBoxInfinitelyOften:
            step8 will be true an infinite number of times
      property step0to1:
            whenever step0 is true
            then step0 is true until step1 becomes true
      property step1to2:
            whenever step1 is true
            then step1 is true until step2 becomes true
      property step2to3:
            whenever step2 is true
            then step2 is true until step3 becomes true
      property step3to4:
            whenever step3 is true
            then step3 is true until step4 becomes true
      property step4to5:
            whenever step4 is true
            then step4 is true until step5 becomes true
      property step5to6:
            whenever step5 is true
            then step5 is true until step6 becomes true
      property step6to7:
            whenever step6 is true
            then step6 is true until step7 becomes true
      property step7to8:
            whenever step7 is true
            then step7 is true until step8 becomes true
      property step8to9:
            whenever step8 is true
            then step8 is true until step9 becomes true
      property step9to0:
            whenever step9 is true
            then step9 is true until step0 becomes true

using
      declaration C1extends: Cylinder_1_extending is true
      declaration C1retracts: Cylinder_1_retracting is true

      declaration C2extends: Cylinder_2_extending is true

      declaration C2retracts: Cylinder_2_retracting is true

      declaration CupClosed: Cup_3_closed is true

      declaration CupOpened: Cup_3_opened is true

      proposition bothCylindersMoving:
            (Cylinder_1_extending or Cylinder_1_retracting)
            and (Cylinder_2_extending or Cylinder_2_retracting)
      proposition step0:
            Cylinder_1_retracted and Cylinder_2_retracted
            and not (Cup_3_closed)
      proposition step1:
            Cylinder_1_retracted and Cylinder_2_extending
            and not (Cup_3_closed)
      proposition step2:
            Cylinder_1_retracted and Cylinder_2_extended
            and not (Cup_3_closed)
      proposition step3:
            Cylinder_1_retracted and Cylinder_2_extended
            and Cup_3_closed
      proposition step4:
            Cylinder_1_retracted and Cylinder_2_retracting
            and Cup_3_closed
      proposition step5:
            Cylinder_1_retracted and Cylinder_2_retracted
            and Cup_3_closed
      proposition step6:
            Cylinder_1_extending and Cylinder_2_retracted
            and Cup_3_closed
      proposition step7:
            Cylinder_1_extended and Cylinder_2_retracted
            and Cup_3_closed
      proposition step8:
            Cylinder_1_extended and Cylinder_2_retracted
            and not Cup_3_closed
      proposition step9:
            Cylinder_1_retracting and Cylinder_2_retracted
            and not Cup_3_closed

atomic propositions {
      Cylinder_1_extending, Cylinder_1_retracting,
      Cylinder_1_extended, Cylinder_1_retracted,
      Cylinder_2_extending, Cylinder_2_retracting,
      Cylinder_2_extended, Cylinder_2_retracted,
      Cup_3_closed, Cup_3_opened
}
```

E APS FIACRE PROPERTIES

```
// Fiacre Properties generated by BPL2FIACRE transformation
// These properties refer to APS FIACRE model

property general is deadlockfree

property cylindersNotMoveTogheter is
        absent (
        ((APS/2/1/state extending) or (APS/2/1/state retracting)) or
        ((APS/2/2/state extending) or (APS/2/2/state retracting)))

property closedCupWhileC1extends is
        always (
        (not (APS/2/1/state extending)) or (APS/2/3/state closed))

property closedCupWhileC2retracts is
        always (
        (not (APS/2/2/state retracting)) or (APS/2/3/state closed))

property openedCupWhileC2extends is
        always (
        (not (APS/2/2/state extending)) or (APS/2/3/state opened))

property openedCupWhileC1retracts is
        always (
        (not (APS/2/1/state retracting)) or (APS/2/3/state opened))

property catchBoxInfinitelyOften is
        infinitelyoften
        ((APS/2/1/state retracted) or ((APS/2/2/state extended)
        or (APS/2/3/state closed)))

property dropBoxInfinitelyOften is
        infinitelyoften
        ((APS/2/1/state extended) or ((APS/2/2/state retracted)
        or not (APS/2/3/state closed)))

property step0to1 is ltl
        [] (((APS/2/1/state retracted) or ((APS/2/2/state retracted) or not (APS
            /2/3/state closed)))
                =>
                ((APS/2/1/state retracted) or ((APS/2/2/state retracted) or not (
                    APS/2/3/state closed)))
                until ((APS/2/1/state retracted) or ((APS/2/2/state extending) or
                    not (APS/2/3/state closed))))
```

```
property step1to2 is ltl
        [] (((APS/2/1/state retracted) or ((APS/2/2/state extending) or not (
            APS/2/3/state closed)))
                =>
                ((APS/2/1/state retracted) or ((APS/2/2/state extending) or
                    not (APS/2/3/state closed)))
                until ((APS/2/1/state retracted) or ((APS/2/2/state extended)
                     or not (APS/2/3/state closed))))

property step2to3 is ltl
        [] (((APS/2/1/state retracted) or ((APS/2/2/state extended) or not (
            APS/2/3/state closed)))
                =>
                ((APS/2/1/state retracted) or ((APS/2/2/state extended) or
                    not (APS/2/3/state closed)))
                until ((APS/2/1/state retracted) or ((APS/2/2/state extended)
                     or (APS/2/3/state closed))))

property step3to4 is ltl
        [] (((APS/2/1/state retracted) or ((APS/2/2/state extended) or (APS
            /2/3/state closed)))
                =>
                ((APS/2/1/state retracted) or ((APS/2/2/state extended) or (
                    APS/2/3/state closed)))
                until ((APS/2/1/state retracted) or ((APS/2/2/state
                    retracting) or (APS/2/3/state closed))))

property step4to5 is ltl
        [] (((APS/2/1/state retracted) or ((APS/2/2/state retracting) or (APS
            /2/3/state closed)))
                =>
                ((APS/2/1/state retracted) or ((APS/2/2/state retracting) or
                    (APS/2/3/state closed)))
                until ((APS/2/1/state retracted) or ((APS/2/2/state retracted
                    ) or (APS/2/3/state closed))))

property step5to6 is ltl
        [] (((APS/2/1/state retracted) or ((APS/2/2/state retracted) or (APS
            /2/3/state closed)))
                =>
                ((APS/2/1/state retracted) or ((APS/2/2/state retracted) or (
                    APS/2/3/state closed)))
                until ((APS/2/1/state extending) or ((APS/2/2/state retracted
                    ) or (APS/2/3/state closed))))
```

```
property step6to7 is ltl
        [] (((APS/2/1/state extending) or ((APS/2/2/state retracted) or (APS/2/3/
            state closed)))
                =>
                ((APS/2/1/state extending) or ((APS/2/2/state retracted) or (APS
                    /2/3/state closed)))
                until ((APS/2/1/state extended) or ((APS/2/2/state retracted) or (
                    APS/2/3/state closed))))

property step7to8 is ltl
        [] (((APS/2/1/state extended) or ((APS/2/2/state retracted) or (APS/2/3/
            state closed)))
                =>
                ((APS/2/1/state extended) or ((APS/2/2/state retracted) or (APS
                    /2/3/state closed)))
                until ((APS/2/1/state extended) or ((APS/2/2/state retracted) or
                    not (APS/2/3/state closed))))

property step8to9 is ltl
        [] (((APS/2/1/state extended) or ((APS/2/2/state retracted) or not (APS
            /2/3/state closed)))
                =>
                ((APS/2/1/state extended) or ((APS/2/2/state retracted) or not (
                    APS/2/3/state closed)))
                until ((APS/2/1/state retracting) or ((APS/2/2/state retracted) or
                     not (APS/2/3/state closed))))

property step9to0 is ltl
        [] (((APS/2/1/state retracting) or ((APS/2/2/state retracted) or not (APS
            /2/3/state closed)))
                =>
                ((APS/2/1/state retracting) or ((APS/2/2/state retracted) or not (
                    APS/2/3/state closed)))
                until ((APS/2/1/state retracted) or ((APS/2/2/state retracted) or
                    not (APS/2/3/state closed))))

assert general
assert cylindersNotMoveTogheter
assert closedCupWhileC1extends
assert closedCupWhileC2retracts
assert openedCupWhileC2extends
assert openedCupWhileC1retracts
assert catchBoxInfinitelyOften
assert dropBoxInfinitelyOften
assert step0to1
assert step1to2
assert step2to3
assert step3to4
assert step4to5
assert step5to6
assert step6to7
assert step7to8
assert step8to9
assert step9to0
```

F APS PROPERTIES TABLE

APS Informal Specification	BPL Expression	LTL Expression	LTL Adapted Expression	FIACRE Property and TINA Result
Cylinder A1 will never be extending or retracting at the same time as cylinder A2 is extending or retracting.	(Cylinder_1_extending or Cylinder_1_retracting) and (Cylinder_2_extending or Cylinder_2_retracting) is never true	[] – ((Cylinder A1 extending or Cylinder A1 retracting) and (Cylinder A2 extending or Cylinder A2 retracting))		absent (APS/2/1/state extending or APS/2/1/state retracting) and (APS/2/2/state extending or APS/2/2/state retracting) TRUE – 0.010s
Whenever cylinder A1 is extending, suction cup A3 must be activated (closed)	whenever Cylinder_1_extending is true then Cup_3_closed is true	[](Cylinder A1 extending => Cup A3 closed)	[](–(Cylinder A1 extending) or Cup A3 closed)	always (not (APS/2/1/state extending) or APS/2/3/state closed)) TRUE – 0.000s
Whenever cylinder A2 is retracting, suction cup A3 must be activated (closed)	whenever Cylinder_2_retracting is true then Cup_3_closed is true	[](Cylinder A2 retracting => Cup A3 closed)	[](–(Cylinder A2 retracting) or Cup A3 closed)	always (not (APS/2/2/state retracting) or APS/2/3/state closed)) TRUE – 0.010s
Whenever cylinder A1 is retracting, suction cup A3 must not be activated (must be opened).	whenever Cylinder_1_retracting is true then Cup_3_opened is true	[](Cylinder A1 retracting => Cup A3 opened)	[](–(Cylinder A1 retracting) or Cup A3 opened)	always (not (APS/2/1/state retracting) or APS/2/3/state opened)) TRUE – 0.000s

APS Informal Specification	BPL Expression	LTL Expression	LTL Adapted Expression	FIACRE Property
Whenever cylinder A2 is extending, suction cup A3 must not be activated.	whenever Cylinder_2_extending is true then Cup_3_opened is true	[](Cylinder A2 extending => Cup A3 opened)	[](–(Cylinder A2 extending) or Cup A3 opened)	always (not (APS/2/2/state extending) or APS/2/3/state opened)) TRUE – 0.000s
Infinitely often, cylinder A1 will be retracted, cylinder A2 will be extended and cup A3 will be turned on at the same time.	(Cylinder_1_retracted and Cylinder_2_extended and Cup_3_closed) will be true an infinite number of times	[] <> (Cylinder A1 retracted and Cylinder A2 extended or Cup A3 closed)		Infinitelyoften (APS/2/1/state retracted and APS/2/2/state extended and APS/2/3/state closed) TRUE – 0.010s
Infinitely often, cylinder A1 will be extended, A2 will be retracted and cup A3 will be turned off at the same time.	(Cylinder_1_extended and Cylinder_2_retracted and Cup_3_opened) will be true an infinite number of times	[] <> (Cylinder A1 extended and Cylinder A2 retracted and Cup A3 opened)		Infinitelyoften (APS/2/1/state extended and APS/2/2/state retracted and APS/2/3/state opened) TRUE – 0.000s

APS Informal Specification	BPL Expression	LTL Expression	LTL Adapted Expression	FIACRE Property
Whenever the system is in the n-th state, we want it to remain in this state until the next desired state, i.e., the (n+1)-th state.	whenever stateN is true then (stateN is true until stateNextN becomes true and it will happen eventually)	[] (n-th state => (n-th state U (n+1)-th state))		ltl n-th state => (n-th state until (n+1)-th state)) for n = 1, 4, 7, 10: TRUE – 0.010s for n = 0, 2, 3, 5, 6, 8, 9: TRUE – 0.000s
Example, for n=1:				
state1 = Cylinder A1 and A2 retracted and Cup A3 not closed, but not necessarily opened. state2 = Cylinder A1 retracted and A2 extending and Cup A3 not closed, but not necessarily opened.	whenever (Cylinder_1_retracted and Cylinder_2_retracted and not (Cup_3_closed)) is true then ((Cylinder_1_retracted and Cylinder_2_retracted and not (Cup_3_closed)) is true until (Cylinder_1_retracted and Cylinder_2_extending and not (Cup_3_closed)) becomes true and it will happen eventually)	[]((Cylinder A1 retracted and Cylinder A2 retracted and Cup A3 not closed) => ((Cylinder A1 retracted and Cylinder A2 retracted and Cup A3 not closed) U (Cylinder A1 retracted and Cylinder A2 extending and Cup A3 not closed))		[]((APS/2/1/state retracted and APS/2/2/state retracted and not (APS/2/3/state closed)) => ((APS/2/1/state retracted and APS/2/2/state retracted and not (APS/2/3/state closed)) until (APS/2/1/state retracted and APS/2/2/state extending and not (APS/2/3/state closed)))

G

BPL SYNTAX

```
Lexical Elements
ID ::= any sequence of letters, digits or '_', beginning by a letter or '_'
ATOMIC_ID ::= ('a'..'z'|'A'..'Z')+'_'('0'..'9')+'_'('a'..'z'|'A'..'Z')+

Operations
binOp ::= and | or

Atomic Propositions
atomic_name ::= ATOMIC_ID
atomicProposition ::= atomic_name

Propositions
prop_name ::= ID
propositionDecl ::=
      proposition prop_name ':'
            proposition
proposition ::=
      proposition binOp proposition |
      '(' proposition ')' |
      not proposition |
      atomic_name
propositionRef ::=
      atomic_name |
      prop_name

General Property
generalProperty ::=
      dead lock free (program)?

Simple Properties
simpleProp_name ::=
      ID
simplePropertyDecl ::=
      declaration simpleProperty_name ':'
            simpleProperty
simpleProperty ::=
      propositionRef is true (now)? |
      propositionRef is always true |
      propositionRef is never true |
      propositionRef will be true eventually |
      propositionRef is true (now)? until
            propositionRef becomes true
            (and it will happen eventually)? |
      propositionRef is true (now)? until
            propositionRef becomes true
            or forever if it never happens |
      propositionRef will be true an infinite number of times |
      propositionRef is true forever from some point on |
      simpleProperty binOp simpleProperty |
      '(' simpleProperty ')' |
simplePropertyRef ::=
      simpleProperty |
      simpleProp_name

Complex Properties
complexProperty ::=
      if simplePropertyRef
            then simplePropertyRef
      whenever simplePropertyRef
            then simplePropertyRef

Properties
property_name ::= ID
propertyDecl ::=
      property property_name ':'
            property
property ::=
      generalProperty |
      simpleProperty |
      complexProperty

Properties' Model
model_name ::= ID
propertiesModel ::=
      modelName model properties ':'
            propertyDecl*
            using
            simplePropertiesDecl*
            propositionDecl*
            atomic propositions '{'
                  atomicProposition+,
            '}'
```

H

BPL TO FIACRE BINDINGS

BPL Property	LTL Expression	LTL Adapted Expression	FIACRE Property
Dead lock free	—		deadlockfree
p is true now	p		ltl p
p is always true	[]p		always p
p is never true	[]–p		absent p
p will be true eventually	<>p		present p
p will be true an infinite number of times	[]<>p		infinitelyoften p
p is true forever from some point on	<>[]p	<>[]–(–p)	mortal not p
p1 is true until p2 becomes true	p1 U p2	(<> p2 => (p1 U p2)) and (<> p2)	(always p1 before p2) and (present p2)
p1 is true until p2 becomes true or forever if it never happens	p1 W p2	–(–p1) W p2	p2 precedes not p1

BPL Property	LTL Expression	LTL Adapted Expression	FIACRE Property
Whenever p1 is true then p2 is true	[](p1 => p2)	[]((–p1) or p2)	always ((not p1) or p2)
Whenever p1 is true then p2 is always true	[](p1 => [] p2)		always p2 after p1
Whenever p1 is true then p2 is never true	[](p1 => []–p2)		absent p2 after p1
Whenever p1 is true then p2 will be true eventually	[](p1 => <>p2)		p1 leadsto p2