Formal verification of PLC programs using the B Method

Haniel Moreira Barbosa

Haniel Moreira Barbosa

Formal verification of PLC programs using the B Method

Dissertação apresentada ao Programa de Pós-Graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada do Centro de Ciências Exatas e da Terra da Universidade Federal do Rio Grande do Norte como requisito parcial para a obtenção do grau de Mestre em Sistemas e Computação.

Orientador

Prof. Dr. David Déharbe

Universidade Federal do Rio Grande do Norte — UFRN
Programa de Pós-Graduação em Sistemas e Computação — PPgSC

Natal-RN

Outubro, 2012

Dissertação sob o título *Formal verification of PLC programs using the B Method* apresentada por Haniel Moreira Barbosa e aceito pelo Programa de Pós-Graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada do Centro de Ciências Exatas e da Terra da Universidade Federal do Rio Grande do Norte, sendo aprovado por todos os membros da banca examinadora abaixo especificada:

Prof. Dr. David Déharbe
Orientador
Departamento de Informática e Matemática Aplicada
Universidade Federal do Rio Grande Norte

Prof. Dr. Marcel de Oliveira
Departamento de Informática e Matemática Aplicada
Universidade Federal do Rio Grande do Norte

Prof. Dr. Alexandre Mota
Centro de Informática
Universidade Federal de Pernambuco

Natal-RN, primeiro de Novembro de dois mil e doze.

Àqueles que amo. E às derrotas, porque nada me ensinou tanto quanto elas.

Agradecimentos

Agradeço aos meus pais, que tanto me apoiaram, que tanto me elevaram... e que por tanto tempo não tiveram o reconhecimento devido.

Aos meus professores, que tanto me ensinaram, positiva ou negativamente.

A David, que me aguentou. A Thierry, que me deu uma oportunidade que tentei aproveitar ao máximo.

Aos meus amigos, pela força, pelos puxões de orelha, pelo necessário escapismo, pela esperança.

E a uma certa pixototinha, que sabe muito bem o porquê, mas principalmente pelo sorriso.

Muad'Dib learned rapidly because his first training was in how to learn. And the first lesson of all was the basic trust that he could learn. It's shocking to find how many people do not believe they can learn, and how many more believe learning to be difficult. Muad'Dib knew that every experience carries its lesson.

Dune, by Frank Herbert

Contents

List of Figures

Chapter D: CDC extra information

List of Tables

List of Codes

Chapter 4: Method   p. 79

Chapter 5: Case Study   p. 137

Chapter B: Textual declarations in IEC 61131-3   p. 165

Formal verification of PLC programs using the B Method

Autor: Haniel Barbosa

Orientador: Prof. Dr. David Déharbe

Resumo

Controladores Lógico Programáveis (PLCs — *Programmable Logic Controllers*, em inglês) desempenham funções de controle, recebendo informações do ambiente, processando-as e modificando este ambiente de acordo com os resultados obtidos. São comumente utilizados na indústria nas mais diversas aplicações, do transporte de massa à indústria do petróleo, gás e energias renováveis. Com o crescente aumento da complexidade dessas aplicações e do seu uso em sistemas críticos, faz-se necessária uma forma de verificação que propicie mais confiança do que testes e simulação, padrões mais utilizados na indústria, mas que podem deixar falhas não tratadas. Métodos formais podem prover maior segurança a este tipo de sistema, uma vez que permitem a sua verificação matemática. Neste trabalho fazemos uso do Método B, que é usado com sucesso na indústria para a verificação de sistemas críticos, possui amplo apoio ferramental e suporte à decomposição, refinamento e verificação de corretude em relação à especificação através de obrigações de prova. O método desenvolvido e apresentado aqui consiste em gerar automaticamente modelos B a partir de programas para PLCs e verificá-los formalmente em relação a propriedades de segurança, estas derivadas manualmente a partir dos requisitos do sistema. O escopo do trabalho são as linguagens de programação para PLCs do padrão IEC 61131-3, mas sistemas com linguagens que apresentem modificações em relação ao padrão também são suportados. Esta abordagem visa facilitar a integração de métodos formais na indústria através da diminuição do esforço para realizar a verificação formal de PLCs.

Palavras-chave: PLC, IEC 61131-3, Método B, métodos formais, sistemas críticos.

Formal verification of PLC programs using the B Method

Author: Haniel Moreira Barbosa

Advisor: Prof. Dr. David Déharbe

Abstract

PLCs (acronym for Programmable Logic Controllers) perform control operations, receiving information from the environment, processing it and modifying this same environment according to the results produced. They are commonly used in industry in several applications, from mass transport to petroleum industry. As the complexity of these applications increase, and as various are safety critical, a necessity for ensuring that they are reliable arouses. Testing and simulation are the *de-facto* methods used in the industry to do so, but they can leave flaws undiscovered. Formal methods can provide more confidence in an application's safety, once they permit their mathematical verification. We make use of the B Method, which has been successfully applied in the formal verification of industrial systems, is supported by several tools and can handle decomposition, refinement, and verification of correctness according to the specification. The method we developed and present in this work automatically generates B models from PLC programs and verify them in terms of safety constraints, manually derived from the system requirements. The scope of our method is the PLC programming languages presented in the IEC 61131-3 standard, although we are also able to verify programs not fully compliant with the standard. Our approach aims to ease the integration of formal methods in the industry through the abbreviation of the effort to perform formal verification in PLCs.

Keywords: PLC, IEC 61131-3, b method, formal methods, safety critical systems

1   Introduction

Programmable Logic Controllers (from now on, PLCs) perform control operations in a system, running in *execution cycles*: they receive information from the environment as *inputs*, process them and affect this environment with the resulting *outputs*, **controlling** some aspects of it. This work deals with discrete-time PLCs, which receive and process signals with fixed values, so they are always defined given a particular time instant, in opposite to a continuous-time signal, a function defined at every time in an interval.

In many fields (Lahiri and Bryant, 2003), such as mass transport and petroleum industry, it is very common to use PLCs to handle control applications. Those are mostly programmed according to IEC 61131-3 (IEC, 2003), an international standard that specifies the five standard PLC programming languages, namely: LD (Ladder Diagram) and FBD (Function Block Diagram) as graphical languages; IL (Instruction List) and ST (Structured Text) as textual languages; and SFC (Sequential Function Chart), that shows the structure and internal organization of a PLC. It is not rare that a variation of such languages is employed too.

As the complexity of the PLC applications increases, and as various are critical, it is important to ensure their safety (KRON, 2003). Since testing and simulation, the *de-facto* methods in many industries to perform verification, can leave flaws undiscovered, which may be intolerable given the level of risk acceptance of a system, another strategy is necessary. A mean to fulfill this requirement is with formal methods, mathematical approaches to software and system development which support the rigorous specification, design and verification of computer systems (FME, 2012). However, they are difficult to integrate with the industrial process (AMEY, 2004), since most control engineers are not familiarized with formal verification (PARNAS, 2010).

In this work we use the B Method (ABRIAL, 1996) as the formal method to characterize our proposal, since it is successfully applied in safety-critical applications, e.g. in the railway industry (LECOMTE et al., 2007), besides having a strong support of tools

such as development environment, theorem provers, model checkers and animators, etc. Another positive point is the B language itself, which is very expressive, including first order logic, integers and set theory, and on top of that it can handle decomposition, refinement, verification of correctness according to the specification and generation of certified code (LECOMTE et al., 2007). Nonetheless, the difference in expressivity between B and the PLC languages, like the lack of float numbers in the former, lead to restrictions on how much of the PLC programs we can cover.

Our approach consists of developing a tool that receives a PLC program based on the IEC 61131-3 standard and automatically builds an intermediary object from it. From this object a formal model in the B notation is generated, also automatically. The correctness of this translation, between the PLC program and the B model representing it, is not mathematically verified, its confidence relying on expertise and simulations over the model's behaviour. *Safety constraints* are then manually inserted into the B model and verified using theorem proving, which allows us to cover the full behavior set of the problem in unlimited depth. We can verify structural properties, such as *deadlock freedom*, performing *model checking* in the model, although some adaptations may be necessary in the model, like restricting its state space, due to the different nature of the verification techniques. We use the ProB (LEUSCHEL and BUTLER, 2003) Model Checker to analyze the model, which also supports the definition and verification of new properties using Linear Temporal Logic (BLACKBURN et al., 2001), such as *liveness*; animation of the model is available as well, providing visual validation. Our approach is thus able to verify that the PLC is presenting the expected behaviour in its *execution cycle.*

Automation is a key aspect of our work since it is essential for the success of a verification technique to be used by engineers, in our case PLC developers, that it insures viability, allowing the formalism to be mostly hidden from them. Although the safety constraints have to be manually inserted in the model, given that they were already available one would have only to run the tool implementing the method over the PLC programs to perform formal verification on them, allowing error management. In order to keep the formalism hidden from them also during the error management phase the results of the verification should be presented in a readable way for the engineers, but that is still between the limitations of our approach.

In order to include all the IEC 61131-1 languages, we based our intermediary structure (from now on called "PLC Object") in the PLCopen (PLCOPEN, 2009) standard. The standard provides an interface representing all such languages in an XML-based format,

working also as *documentation* and sometimes being required by Stakeholders. This PLC Object stands between the PLC programs and the formal models to be generated, hence reducing the semantic gap between PLCs and B, and defining a unique representation for different PLC languages (FARINES et al., 2011). The process also involves the development of a customizable parser, so we can treat PLC programs that do not *strictly* follow the IEC standard; as numerous legacy systems deviate from the standard, our approach would thus be able to handle them.

Being able to deal with legacy systems is a very useful feature once there are numerous existing PLC programs not amenable to be formally treated except with formal methods (FREY and LITZ, 2000). Reverse engineering can be performed in these programs so a PLCopen documentation can be generated, a desirable feature as well, as an equivalent program, easier to understand and maintain, is acquired.

We also present a case study in a real safety-critical railway system: the Central Door Controller (from now on, CDC), part of the doors subsystem of a train. We show the step by step automatic generation of the formal specification from its PLC program and, after defining the safety constraints, perform a full formal verification in the application, at the end exhibiting the results.

Thus, the main contributions in our work are:

1. The automatic generation of formal models from PLC programs, facilitating the insertion of formal methods in the industrial process.

2. To have correctness, according to the specification, as a realistic and achievable goal in PLC development, with the verification of safety constraints by theorem proving.

3. Allowing the structural and behavioral verification of the PLC programs through model checking, LTL checking and animation over the formal model.

4. Evaluation of the proposed approach in a case study.

Despite the importance of these contributions, our approach has relevant restrictions: as previously said, the translation from the PLC programs to the B models is not mathematically verified — its validity has been attested solely by simulation and expertise; since our method covers only the PLC programs we do not check the whole system, just its control software — the environment in which the PLC is inserted is not considered. These issues compose part of our future work, as well as other subjects, such as dealing with *time*, treating the yet uncovered IL language and overcoming restrictions concerning

the covered elements of the PLC languages due to restrictions of the B language. A full account of the limitations of our approach may be found in Chapter 4, at Section 4.4.

1.1 Goals

In a nutshell, our goals in this work are:

1. Automatically build an intermediary structure instance representing the information in the PLC programs.

2. Reduce the semantic gap between the PLC programs and the formal models through an intermediary structure.

3. Provide documentation to PLC programs using PLCopen.

4. Automatically generate B models from the intermediary structure.

5. Insert safety constraints, extracted from the requirements of the PLC system, in the B models generated.

6. Perform formal verification through theorem proving and model checking over the B models generated.

7. Evaluate the method developed to accomplish the previous goals through a case study in a real industrial setting.

1.2 Structure of the dissertation

In Chapter 2 we discuss some related work in the area of PLC verification. Chapter 3 presents in more detail PLCs, PLCopen and the B method. In Chapter 4 we have the description of the different phases of our method, how they are performed and were implemented. Chapter 5 contains the case study we performed, with the detailed application of our method and the results. Finally, Chapter 6 concludes the dissertation, also presenting the future work that lies ahead of us.

2   Related work

As stated in (FREY and LITZ, 2000), there are several reasons for the application of formal methods in PLC programming:

- "The growing complexity of the control problems, the demand for reduced development time[1], and the possible reuse of existing software modules result in the need for a formal approach in PLC programming."

- "The demand for high quality solutions and especially the application of PLC in safety-critical process need verification and validation procedures, i.e. formal methods to prove specific static and dynamic properties of the programs, as for example liveness, unambiguity or response times."

In this chapter we present some works that have been trying to tackle these issues, categorizing them, as in (FREY and LITZ, 2000), based on the method applied to do so:

- MODEL CHECKING: "It is an automatic technique for verifying finite-state reactive systems, such as sequential circuit designs and communication protocols. Specifications are expressed in temporal logic, and the reactive system is modeled as a state-transition graph. An efficient search procedure is used to determine whether or not the state-transition graph satisfies the specifications" (CLARKE et al., 1999). While testing and simulation explore only some of the possible behaviours of systems, model checking makes an exhaustive exploration of all possible behaviours, checking the validity of a given property and providing counter examples when the system does not satisfy it. This exhaustive search, however, leads to one of the main disadvantages of model checking, the state-explosion problem — since the number os states grows **exponentially** in the number of system's variables, the procedure

---

[1]Relative to tasks that would take much longer if not done with formal methods, like checking the validity of hundreds of boolean properties in a given system.

may be prevented from terminating timely. Then it is usually limited to some bound, in order to avoid this issue, reducing its coverage.

- THEOREM PROVING: It proves that an implementation satisfies a specification by mathematical reasoning, without the need of searching all possible states in a model, as expected properties are formalized using some mathematical logic. A Theorem Prover assists the user in formulating the proof. One of its greatest advantages is the avoidance of the state-explosion problem, as this approach allows one to deal with the full behavior set of a problem in unlimited depth. Between its disadvantages are the considerable technical expertise and understanding of the system required, as well as the fact that if the proof of a property fails, the prover will not tell whether the property is indeed unprovable or whether it is lacking information to complete the proof, relying on the user the task of verifying what is the scenario.

After presenting the approaches to formal verification we show some *semi-formal* approaches, tackling the verification effort with *simulation* techniques, where the validation is made against an informal specification. Its problem is that it is never complete and it takes considerable time and efforts (FREY and LITZ, 2000).

These works deal with PLC programs according to the IEC 61131-3 standard, but most are focused only in a subset of the standard five languages (SFC, LD, FBD, ST, IL): to the best of our knowledge, only one related work, on formal verification, considers the whole standard, and it also uses PLCopen, like our proposed approach.

2.1 Approaches using theorem proving

(SÜFLOW and DRECHSLER, 2008)

The approach presented in (SÜFLOW and DRECHSLER, 2008) proposes the automatic translation of an IL program into a *SystemC* model, so a high level representation of the former is obtained, which is then compared, through *Equivalence Checking* using SAT solvers, with a reference model specification, showing that they are functionally equivalent.

SystemC is a C++ class library that provides cycle accurate simulation and different levels of abstraction in the system's description — from software down to the hardwire description level; decomposition can also be handled, since each module can be partitioned

into several modules. Through a series of rules a transformation takes place from IL programs to a semantic equivalent *SystemC* model, which can then be formally verified.

SAT solvers evaluate a given boolean formula and determine if it is *satisfiable*[2]. In this approach it is used to perform equivalence checking between the *SystemC* resulting translation of the IL program and a given reference specification.

The variables, basic statements and control flow statements of IL are covered by this approach, as well as call statements. It is a wide-ranging one in terms of IL. Nonetheless, as *properties checking* is not performed, only the validation of the translation process with a reference model — which has to be manually constructed —, the contribution of the work has still to be increased.

(CAVALCANTI et al., 2011)

A strategy to translate Simulink diagrams into *Circus* and to prove that a parallel ADA implementation refines this *Circus* specification is presented in (CAVALCANTI et al., 2011). Simulink is a block-based language for control applications, similar to FBD (Function Block Diagram); although it is not used in PLCs their principals are much alike. *Circus* is a notation that combines Z, CSP, and a refinement calculus, thus capable of formal modeling and analyzing the data structures (Z) and communication and concurrency features (CSP) of a system.

The approach consists of validating and extending existing industrial tools that translate discrete-time Simulink diagrams into Ada implementations. This is done by *proving* the correctness of the translation by verifying that a *Circus* model for the Simulink diagram is *refined* by another *Circus* model for the generated Ada implementation. The approach makes use of the *ClawZ* tool, that generates bits of Z and CSP models from Simulink diagrams and, by a series of refinements, obtain a *Circus* model capable of being verified by the theorem prover *ProofPower*, proving whether this model refines the one for its respective Simulink diagram.

Time-related issues are not covered in this work, but parallelism is easily handled by CSP, as well as other features such as the dependency between the order in which the inputs are received by a block and its resulting outputs. Simulink has a library of standard blocks to whom a corresponding library of translated specifications is available in *ClawZ*.

---

[2]"f is valid" — always true, a theorem — *if and only if* "not f is unsatisfiable" *if and only if* not "f is satisfiable".

A setback in the work is the lack of automation, but the authors state they are going to carry this out as future work — the steps of the approach have already been proven correct and are automatable. Efforts to produce a toolset are being made, so a powerful resource in the analysis of control diagrams and their implementations can be obtained, aiming error management as well.

(VÖLKER and KRÄMER, 1999)

In (VÖLKER and KRÄMER, 1999) the proposed approach deals with PLC function blocks implemented using FBD, SFC or ST. It aims to prove the correctness of the components independently, so the applications who use them may have their verification process considerably simplified, since only the component's integration has to be checked.

Higher order logic (HOL) is used for verification purposes. It provides abstraction and quantification, which makes it a very expressive mechanism, suited for the description of complex theories. HOL serves as a logical glue that connects different programming and specification formalisms and allows their integration and analysis within the framework. The Isabelle tool[3] is used for theorem proving automation.

The process described in the paper embodies a subset of the ST language in HOL, thus representing its syntax and semantics. Subsets of SFC and FBD are interpreted in terms of ST and therefore also taken into account. Requirements derived from the controller specification are inserted into the model in Linear Temporal Logic.

Results claim that small examples are easily handled by the verification toolchain, however larger specifications need alterations in the translation tactics for the generation of the HOL theories, so the verification can be eased. How much of the SFC, FBD and ST languages are present in the subsets covered by the approach is not specified, neither how much of the process of the HOL theories derivation is automatic.

2.2 Approaches using model checking

(RAUSCH and KROGH, 1998)

The approach presented in (RAUSCH and KROGH, 1998) aims to perform formal

---

[3]http://www.cl.cam.ac.uk/research/hvg/isabelle/

verification over PLC programs by converting them to SMV, state transition systems, once the latter allows the engineer to verify the behavior of the control program over all possible operating conditions.

SMV is a language used to describe Mealy automata, synchronous or asynchronous networks and deterministic or non-deterministic process. As it provides modular descriptions it is suitable to retain the structure of the original control program, thus keeping traceability between the formal verification and the original PLC program. The verification of the SMV modules is made through model checking over its Computational Tree Logic (CTL) (BLACKBURN et al., 2001) specification, evaluating if this CTL specification is true and providing a counter-example otherwise.

Each state transition in the SMV model corresponds to one scan of the PLC program, so the transient behavior of the PLC during its execution is represented in the model. The language in the scope of this approach was only LD. Although the process is efficient to perform formal verification of PLC programs it is not automatic, despiste the fact that integration with other tools was also intended.

(SOLIMAN and FREY, 2009)

In (SOLIMAN and FREY, 2009) a method is proposed to verify applications using Safety Function Blocks with timed-automata through model-checking and simulation.

Safety Function Blocks (SFBs) are a special kind of function block whose presentation is divided in three parts: firstly, a graphical description of its internal state and behavior; then a list of its properties in natural language, that, with the other part being composed of timing diagrams describing its temporal behavior, for some specific scenarios, may be used to verify its safety.

Timed automata (TA) model timed systems as state-transition systems, with explicit variables representing time and being incremented according to the transitions performed, as in a discrete-event simulation. A TA in the Uppall tool language is generated from the SFB graphical description. The temporal behavior is validated through *simulation* over the model and comparison with the given timing diagrams. The textual properties are formalized in LTL and verified through *model checking* in the TA built. The process of building the TAs is manual, made through the graphical interface of the Uppall tool, based on the specification of SFBs. The formalization of the LTL properties is also manually executed.

All the twenty SFBs defined by PLCopen were successfully formalized following this approach. The challenge faced by the method was with networks, several SFBs interconnected, assuring that the overall system would be compliant to the safety conditions as well as the independent components. The approach presented is useful, however the automation is still in their future works, and the scope is limited only to the SFBs.

(FARINES et al., 2011)

A model-driven engineering approach is used in (FARINES et al., 2011) to generate models in a FIACRE (Intermediate Format for the Embedded Distributed Component Architectures) language from LD programs. The work makes use of an intermediary model to reduce the semantic gap between the LD programs and the Timed Transition System (TTS) models that are verified through the toolkit TINA (TImed petri Net Analyzer); it is inserted in the context of the TOPCASED project[4], which aims the development of an environment for embedded systems, based on Model-Driven Engineering (MDE).

The accuracy of the model transformation in the approach is grounded on the use of metamodel hierarchy in the representation of Domain Specific Modeling Languages. The FIACRE intermediary model is generated from the LD programs in the XML format of PLCopen. Then, a *Frac* compiler, of the TOPCASED project, is used to generate the TTS model for verification. LTL formulas, written by the PLC program designer, will be used to represent the system properties to be validated through model checking.

The strength of the approach is in the use of the FIACRE intermediary language to model the PLC, as a PLC model is automatically generated from the LD programs and FIACRE eases the inclusion of verification tools such as TTS. Extension to the other languages of IEC 61131-3 standard is intended for this project.

2.3 Approaches using simulation

(PARK et al., 2008)

In (PARK et al., 2008) an approach to perform visual verification of PLC programs through the modeling of a 3D plant where they run is performed. Not only the control

---

[4]http://www.topcased.org/index.php?idd_projet_pere=77

software is analyzed, but the whole system where it is inserted, with the environment being taken into account, a very interest feature that helps the validation of PLCs.

A PLC programming environment is presented along with a methodology for the construction of a plant model based on the DEVS (Discrete Event Systems Specification) formalism. This plant model is used for *visual validation* of PLC programs through simulation, taking into account also how the program interacts with its environment and if the actuation performed in it is as expected. The use of the DEVS formalism in the modeling of an environment adaptable to different configurations allows the generation of state transition systems representing the different tasks executable in the plant, which are also considered when simulating the PLC programs.

(WANG et al., 2012)

The application of a technique to validation of PLCs through simulation is described in (WANG et al., 2012) through a case study in a PLC-controlled system. The Behavior-Interaction-Priority (BIP) component framework is used to model a gate control system in a real industry application, also taking into account real-time constraints and the system with whom the control software interacts, making the validation through simulation.

The requirements of the system where modeled as *monitors* that during system component's simulation checked if any transition to a not allowed behaviour occurred. Each property was checked independently through random and controlled set behaviours for the system model, eventually finding erros that helped the correction of the original PLC application.

Major setback, as well as in the previous approach, is the dependency of confidence in the modeling and simulation processes, as *only* visual validation is performed in the system. Nonetheless the decomposition support is of great help for mastering complexity; the ability to deal with time constraints and the approach's overall expressivity also contribute to make it a powerful resource.

2.4   Considerations

We make use of several mechanisms also used in these works, such as an intermediary structure to reduce the semantic gap, like in (FARINES et al., 2011), although ours is entirely based on the PLCopen standard and implemented in C++; the decomposition sup-

port through B mechanisms, helping us to master complexity, as in (WANG et al., 2012); the construction of a verification environment based on theorem proving but that supports other techniques, like model checking, as intended by (VÖLKER and KRÄMER, 1999) and its HOL use; visual validation through animation provides simular results as the simulation approaches in (WANG et al., 2012) and (PARK et al., 2008), although without the expressivity claimed by them.

Our approach begins intended to tackle the verification of PLC programs in *all* the languages of the standard, through PLCopen, something intended only by (FARINES et al., 2011). We also automatically generate the formal models to be verified, something only a few of the presented approaches make, to the best of our knowledge.

Number	Related work
[1]	Our work
[2]	(SÜFLOW and DRECHSLER, 2008)
[3]	(VÖLKER and KRÄMER, 1999)
[4]	(RAUSCH and KROGH, 1998)
[5]	(SOLIMAN and FREY, 2009)
[6]	(FARINES et al., 2011)
[7]	(CAVALCANTI et al., 2011)
[8]	(WANG et al., 2012)
[9]	(PARK et al., 2008)

Table 1: List of related works

In Table 1 a list with the covered related works is presented, with Venn Diagrams below, in Figures 1 and 2, relating the verification techniques and the level of automation of them with our work, whose reach is depicted in light gray.
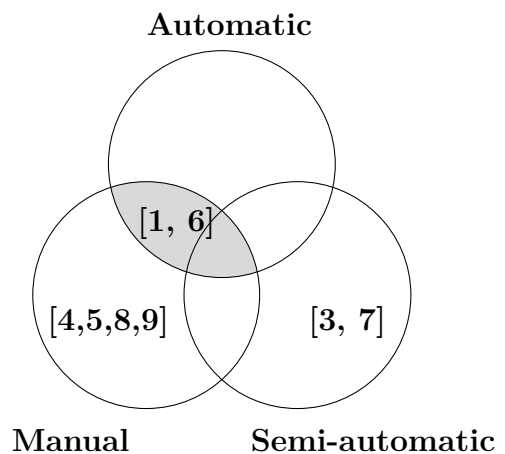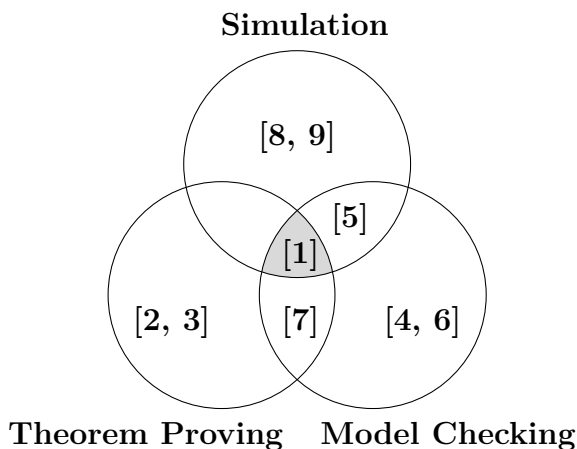
Figure 1: Verification techniques

Figure 2: Approach's automation

We chose to focus on these criteria because they show that regardless that three

different verification techniques have been used by the distinct approaches, automation, a **key** attribute to the adoption of there works by industry, is mostly neglected. The study of these works, nonetheless, has helped us in the overall design for our approach, identifying the good mechanisms to use and which obstacles to consider first.

3   Context and techniques

In this chapter we present the basic concepts of Programmable Logic Controllers (PLCs), of the PLCopen standard and of the B Method, the basic elements of our work.

For the PLCs we introduce how they are organized and *programmed*, according to the international standards. We present the PLCopen standard in detail, with its XML based format. For the B method we make a whole presentation of its notation and features.

3.1   Programmable Logic Controllers

Programmable Logic Controllers are digitally operating electronic systems designed for use in an industrial environment. They use a programmable memory for the internal storage of user-oriented instructions in order to implement specific functions such as logic, sequencing, timing, counting and arithmetic, to control, through digital or analogue inputs and outputs, various types of machines or processes. Both the PLC and its associated peripherals are designed so that they can be easily integrated into an industrial control system and easily used in all their intended functions (IEC, 2003).

PLCs have become very common in control applications throughout the industry, from the mass transport field to the beverage factories. Being applied in many different fields and situations, however, led to the existence of several differences in the way they were constructed, as in I/O addressing, memory organization and instruction sets, for instance. Aiming a standardization, the IEC (International Electrotechnical Commission), a worldwide organization for standardization comprising all national electrotechnical committees, defined the 61131 standard, which establishes the definitions and identifies the main characteristics relevant to the selection and application of programmable logic controllers and their associated peripherals (IEC, 2003).

In Section 3.1.1 we introduce the 61131 standard and which aspects of it will be pertinent to our work. Then, in Section 3.1.2, we show the languages defined in the

standard for PLC programming.

3.1.1   IEC 61131

The IEC 61131 embodies a series of standards over PLCs, from their general information and equipment requirements to their programming and use, as seen below:

1. General information

2. Equipment requirements and tests

3. Programming languages

4. User guidelines

5. Communications

6. Reserved

7. Fuzzy-control programming

8. Guidelines for the application and implementation of programming languages for programmable controllers

In this section we present in detail the standards IEC 61131-1 and IEC 61131-3 (IEC, 2003), which present the general information and the programming languages for PLCs, respectively. The general structure of a PLC, as an embedded system, is depicted in Figure 3.

The main part is the one responsible for signal processing, the **CPU** of the PLC. It consists of the *application program storage*, the *data storage*, the *operating system*, and the execution of the *application program* functions. The CPU function processes signals obtained from sensors as well as internal data storage and generates signals to actuators as well as internal data storage in accordance with the application program.

The *operating system* is responsible for the management of internal PLC interdependent functions, like configuration control, diagnostics and memory management. The *application program storage* supplies memory locations for storing the instructions of the PLC program, as well as initial values for the program data. The *application data storage* provides memory locations to store the I/O image table, where information about
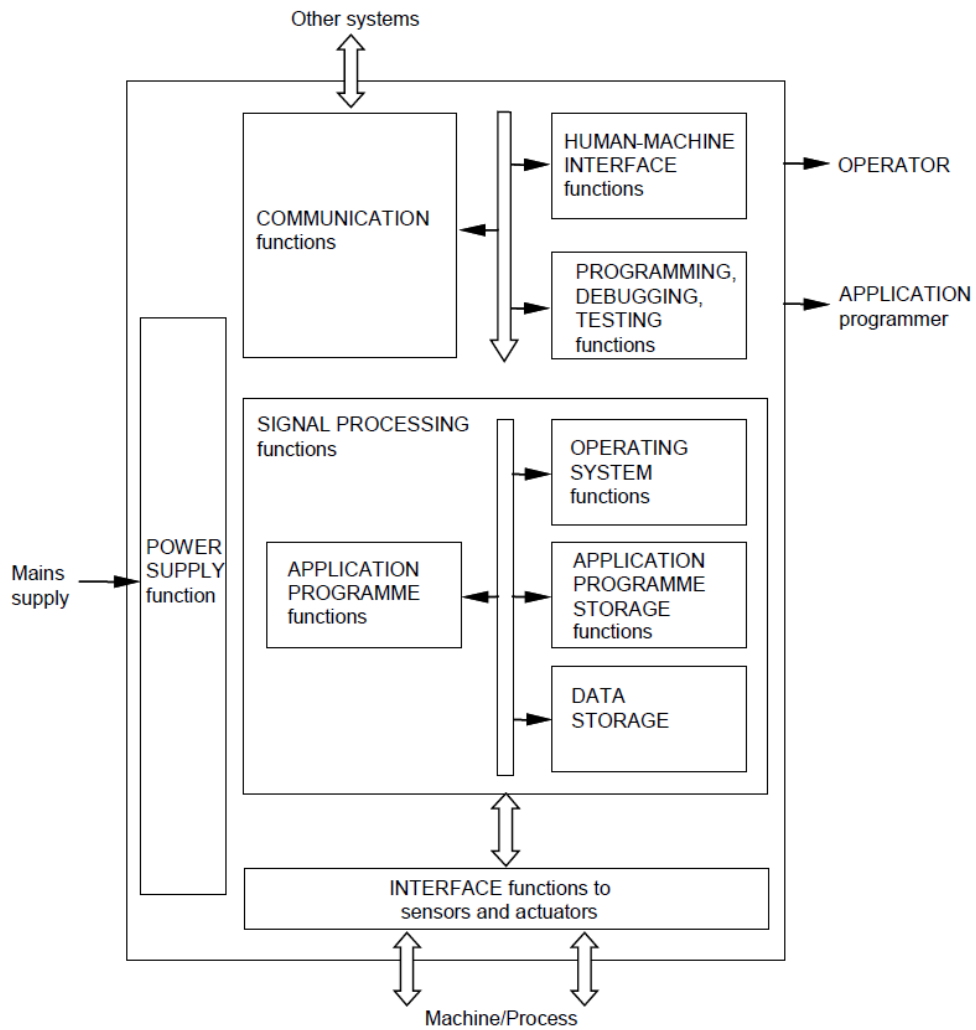
Figure 3: A PLC system.

inputs and outputs is temporarily stored and updated until the end of execution of the application program, as well as data required by it.

A PLC program is generally executed repeatedly as long as the controlled system is running. An application program may consist of a number of tasks, while the execution of each task is accomplished sequentially, one programmable function at a time until the end of the task. The initiation of a task, periodically or upon the detection of an event (interrupt condition), is under the control of the operating system.

The interface with sensors and actuators provides information and data from the machines/processes that will be transmitted to the I/O system of the PLC, as decisions and results determined by its processing function are transmitted to these very machines/processes.

The *communication function* provides data exchange with other systems, such as other PLCs, computers, etc. The *human-machine interface* (HMI) makes the interaction between the operator, the signal processing function and the machine/process. The *programming*, *debugging*, *testing* and *documentation* functions are responsible for supplying auxiliary features to the PLC; the *programming* functionality is specially important to us, since it is the one used to build the **key** element of the CPU: the application program, a logical assembly of all the programming language elements and constructs necessary for the intended signal processing required by a PLC. The programming languages defined in the IEC 61131-3 standard are presented in the next section.

3.1.2   PLC programming

The Part 3 of the IEC 61131 standard, IEC 61131-3, defines five languages for PLC programming, which are classified as shown in Figure 4. ST and IL are textual languages, while FBD and LD are graphical languages. SFC, besides its own elements, can be used in conjunction with any of the textual and graphical languages. They are described in detail below, with the languages' elements summarized, examples presented and applications hinted.
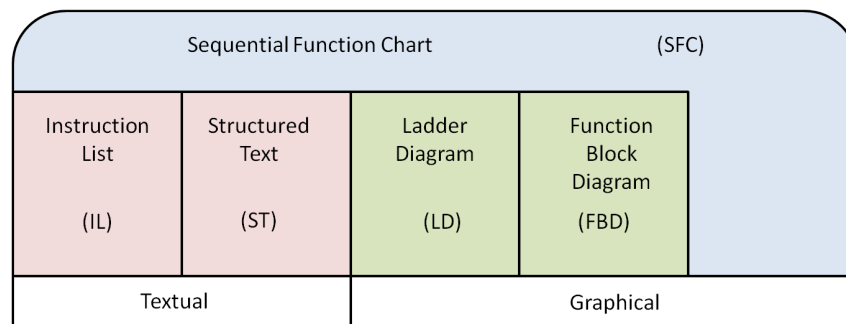
Figure 4: The five languages of the IEC 61131-3 standard for PLC programming.

3.1.2.1   Program Organization Units

The PLC programs are composed of POUs — *Program Organization Units*. These elements are implemented using the languages of the standard and are divided in three categories:

- The POU *functions*, when executed, produce exactly *one* value, typed according to the possible data types in the standard, the function result, and arbitrarily many additional outputs. These POUs are stateless: they contain no state information,

i.e., invocation of a function with the *same arguments* shall always produce the *same result*.

- The POU *function blocks* produce *one or more* values as result. The state of a function block persists from one execution to the next — they are stateful — , therefore invocation with the *same arguments* may produce *different results*.

- The POU *programs* are defined as a "logical assembly of all the programming language elements and constructs necessary for the intended signal processing required for the control of a machine or process by a programmable controller system" (IEC, 2003). Their declaration and usage is equivalent to the *function blocks*. It also may use the previous two POU types as auxiliary elements.

3.1.2.2 Variables and Data Types

Each POU has an I/O table associated, in which the variables declared for its use are summarized. They can be of the following categories:

- LOCAL: Internal variables of the POU.

- INPUTS: Received from the environment; they cannot be modified within the POU.

- OUTPUTS: Supplied by the POU to the environment.

- INOUTS: Both received and supplied; they can be modified within the POU.

- EXTERNAL: Makes reference to a global variable of the PLC, which can be modified within the POU.

- TEMPORARY: Temporary storage for variables in POU *function blocks* and *programs*.

	Keyword	Data type
	BOOL	Boolean
	BYTE	Bit string of length 8
Bit string	WORD	Bit string of length 16
	DWORD	Bit string of length 32
	LWORD	Bit string of length 64
	SINT	Short Integer
	INT	Integer
Integer	DINT	Double Integer
	LINT	Long Integer

	Keyword	Data Type
Unsigned Integer	USINT	Unsigned Short Integer
	UINT	Unsigned Integer
	UDINT	Unsigned Double Integer
	ULINT	Unsigned Long Integer
Floating point	REAL	Real numbers
	LREAL	Long reals
Strings	STRING	Variable-length single-byte character string
	WSTRING	Variable-length double-byte character string

Table 2: Elementary types.

Those variables are typed according to the PLC *Data Types*. They are either *elementary types* or *derived types*. The elementary types are summarized in Table 2; the derived types are:

- ARRAY: it has *base type* (any data type) and a set of *dimensions* (each a *subrange*).

- ENUMERATION: a list of enumerated identifiers; the data element with this type can have only one of them as value at a time.

- DERIVED: References types that can be defined. The POU *functions* and *function blocks* are included in this category.

- STRUCT: the data elements of this type shall contain sub-elements of specified types which can be accessed by their specified names.

- SUBRANGE: the value of any data element of this type can only take on values between and including the specified *lower* and *upper* limits. It is either *signed* or *unsigned*.

3.1.2.3   IL

Instruction List is a textual programming language that uses *instructions* for representing an application program for a PLC. It is a low level language, with syntax resembling *assembly*, being recommended for small applications or code optimization. The language basic elements, *instructions*, are composed of an *operator* with optional *modifiers* and, if necessary, *operands* separated by commas. A summary of the IL commands is presented in appendix A.

```
1        LD    Y1
2        SUB   Y2      (* Substract Y2 from Y1 *)
3        ST    Temp    (* Store Y1-Y2 in Temp *)
4        MUL   Temp    (* Multiply by Temp to square *)
5        ADD(  X1      (* Defer ADD *)
6        SUB   X2      (* Substract X1 from X2 *)
7        ST    Temp    (* Store X1-X2 in Temp *)
8        MUL   Temp    (* Multiply by Temp to square *)
9        )
10       SQRT          (* Call Square root fun *)
11       ST    ILTest  (* Setup function result *)
12       GT    TMax    (* Greater than TMax ? *)
13       JMPC  ERR     (* Yes, Jump to Error *)
14       S     ERROR   (* Set ERROR *)
15       RET           (* Normal return *)
16 ERR:  RET           (* Error return, ENO not set *)
```

Code 1: Example of a IL program.

In Code 1 we have an example of a program in the IL language, implementing a POU *function*, to perform a simple mathematic operation: verify if the distance between two given points X and Y — with the coordinates (X_1, X_2) and (Y_1, Y_2) — is greater than another given value — $TMax$. If it is the code will jump to its end with the ERROR variable not set, representing that an error occurred in the execution. Otherwise it will set to **1** the $ERROR$ output, then executing the normal return. In this case $ERROR$ works as a **release output ENO** (enable output) signal, that needs to have the signal state **1** to represent that the function is carried out as expected. The function result is the calculated distance, of the $REAL$ type.

The I/O table of the IL program can be seen in Table 3, summarizing its inputs, outputs and local variables with their respective types and initial values (if any).

Name	Type	Class	Initial Value
X1	REAL	Input	
X2	REAL	Input	
Y1	REAL	Input	
Y2	REAL	Input	
TMax	REAL	Input	
Temp	REAL	Local	
ERROR	BOOL	Output	

Table 3: I/O Table of the IL program.

3.1.2.4 ST

Structured text is a textual programming language that uses assignment, subprogramme control, selection and iteration statements to represent an application program for a PLC. It is a high level language, more flexible and expressive than IL, with a syntax similar to Pascal.

A summary of the language operators and statements can be viewed in Tables 17 and 18 in Appendix A, respectively. The operators are, from top to bottom, in order of precedence, from higher to lowest, with the elements in the same row presenting the same precedence. The definition of the statements is recursive: each construction may present the others or itself inside itself if it makes use of general statements.

Name	Type	Class	Initial Value
base	INT	Input	
exponent	INT	Input	
auxBase	INT	Local	
i	INT	Local	
j	INT	Local	
mult	INT	Local	
exponentiation	INT	Output	
ERROR	BOOL	Output	FALSE

Table 4: I/O Table of the ST program.

An example of a complete ST program, implementing a POU *function block*, can be viewed in Code 2, and its correspondent I/O table can be seen in Table 4. The program receives a base and an exponent value, returning the result of the respective exponentiation calculated through sum operations. The execution is easier to follow than the IL program since ST is, as it name hints, a structured programming language.

3.1.2.5 Graphical languages

In the IEC 61131-3 standard there are elements that apply to both the graphical languages, LD and FBD, as well as to the graphic parts of SFC. These common elements are depicted in Table 5.

Lines are used to draw the graphical elements and to represent the *connections* between them, while a *block* is a graphical object that represents a call statement, and

```
1  IF exponent > 0 THEN          (* Tests if the exponent if positive *)
2    auxBase := base;            (* Sets the value to be multiplicated *)
3    FOR i := 2 TO exponent DO:  (* Iterates from 2 to exponent's value *)
4      mult := 0;                (* Sets the accumulator to 0 *)
5      FOR j := 1 TO base DO:    (* Sums the current value 'base' times *)
6        mult := mult + auxBase; (* Accumulates the multiplication's value*)
7      END_FOR;                  (* Ends the current multiplication *)
8      auxBase := mult;          (* Updates the value to be multiplicated *)
9    END_FOR;                    (* Ends the exponentiation *)
10   exponentiation := auxBase;  (* Sets the output *)
11 ELSIF exponent = 0 THEN       (* Tests whether the exponent is zero *)
12   exponentiation := 1;        (* Expontiation always result in 1 *)
13 ELSE                          (* Exponent is negative *)
14   ERROR := TRUE;              (* Sets output controlling ERROR *)
15 END_IF;                       (* End of execution *)
```

Code 2: Example of a ST program.

the *connectors* are graphical objects which represent a variable, literal or the side of an expression (their counterpart are the *continuations*).

The set of interconnected graphic elements will form a *network*. Graphical languages are used to represent the execution flow through one or more networks. These flows are conceptually different, depending on the language implementing them:

- **Power flow**: analogous to the flow of electric power in an electromechanical system. This is the default flow in a Ladder Diagram (LD), going from left to right.

- **Signal flow**: analogous to the flow of signals between elements of a signal processing system. This is the default flow in the Function Block Diagrams (FBD), that will be from the output (right-hand) side of a function or function block to the input (left-hand) side of the function or function block so connected.

- **Activity flow**: analogous to the flow of control between elements of an organization, or between the steps of an electromechanical sequencer. This is the default flow in a Sequential Function Chart (SFC), going from the bottom of a step through the appropriate transition to the top of the corresponding sucessor step(s).
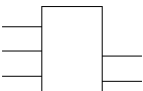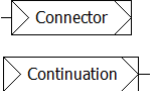
Lines and connections	Block	Connector
		Connector
		Continuation

Table 5: Common elements to IEC 61131 graphical languages

Besides the restrictions of flow direction, the evaluation of a network must not start before the states of the inputs have been read and shall not be completed until the state of the outputs have been defined.

The graphical languages and SFC are shown in detail in the next sections.

3.1.2.6 FBD

FBD is a graphical programming language that uses function block diagrams for representing an application program for a PLC. It is based on circuit diagrams, a conventional graphical representation of electric circuits, and the language basic elements are the *blocks*.

The *blocks* are a graphical representation of a call statement, encapsulating the algorithm and putting in evidence the information flow through the inputs and outputs: the block is activated as its input signals are received in the left-side, processes them and emits the results as outputs in the right-side.

A block may be defined from a set of predefined function blocks — such as timers, counters, etc. — or from POUs, *functions* or *function blocks*, defined in the PLC, using any of the other standard languages. For instance, we can see the blocks corresponding to the IL function and ST function block, presented in the previous sections, in Figures 5 and 6.
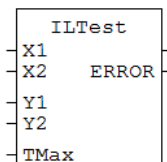
Figure 5: Block corresponding to the IL function seen in Section 3.1.2.3.
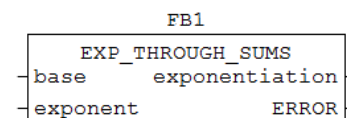
Figure 6: Block corresponding to the ST function seen in Section 3.1.2.4.

Blocks may also be *instantiated* from the original definition of POU *function blocks*, since they are stateful objects. The blocks instantiated from the same type will have different names and independent states, being, indeed, *variables* of the program they are inserted into. A whole POU defined in FBD may as well be seen as a function block to be instantiated as a block in other programs.

Another feature of blocks is the parametrization of the number of inputs and outputs that some types may present. For example, a block of a function **ADD** could have from two up to an arbitrary, but allowable, number of inputs and as output the result of the summation of all the inputs.

Besides the elements common to graphical languages, FBD introduces some new elements of its own. They can be seen in Table 6. An example of a complete FBD program is depicted in Figure 7.
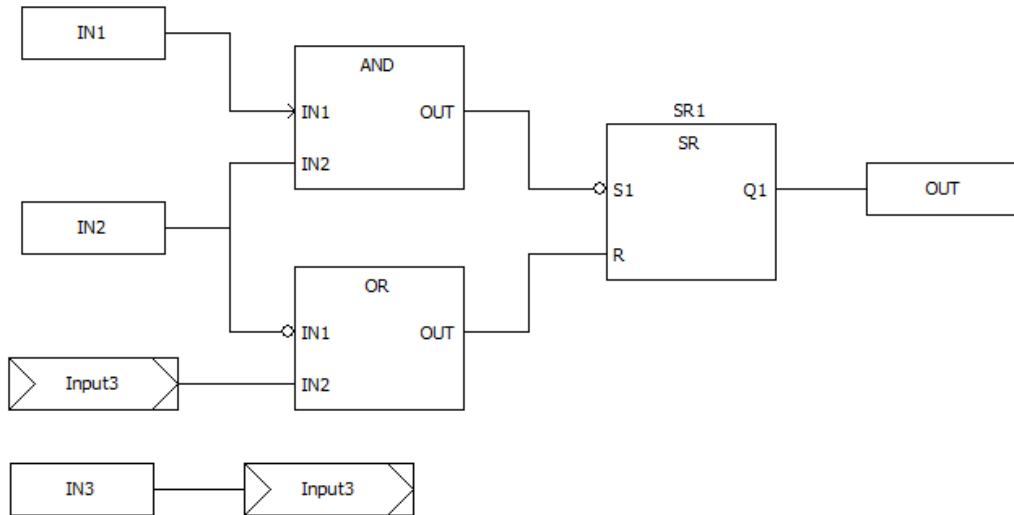
Figure 7: FBD program example

In the example, a FBD program determines the value of the output variable OUT from the output produced by a *Set_Reset Flip Flop* (SR) block, whose inputs are the evaluation results of blocks AND and OR, who receives in turn the FBD program inputs *IN1*, *IN2* and *IN3* to be evaluated. A connector/continuation is used for the *IN3* input, *Input3*.

3.1.2.7 LD

LD is a graphical programming language using ladder diagrams for representing an application program for a PLC. It is based on relay logic, with horizontal *rungs* between vertical *power rails* executing sequentially.

The rungs represent the sentences of the language, with the inputs in the left and the outputs in the right; the variables are seen as contacts to be energized, evaluation going left to right, top to bottom. Thus a ladder diagram emulates the energy flow between the power rails through the relays. The graphical elements introduced by the language can be seen in Table 7.

An example of a LD program is shown in Figure 8. The diagram is equivalent to the sentence $LDTest := (\neg IN_1 \lor (IN_3 \land \neg IN_4)) \land IN_2$. The execution goes as follows: firstly, the *contact* correspondent to IN_1 is *closed* if this input is *not* energized; then the state
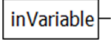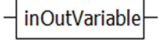
FBD element	Description
inVariable	Variable, expression or literal used as producer
outVariable	Variable, expression or literal used as consumer
inOutVariable	Variable, expression or literal used as producer and consumer
$\twoheadrightarrow LABEL$	A graphical object representing a jump statement
$LABEL:$	A graphical object representing a jump label
$\langle RETURN \rangle$	A graphical object representing a return label

Table 6: Graphical elements of the FBD language

LD element	Description
	Left power rail
	Right power rail
⊣ ⊢	A contact, closed whenever its corresponding coil or an input which controls it is energized.
⊣ / ⊢	A "not" contact, open whenever its corresponding coil or an input which controls it is energized.
–( )–	A coil, energized whenever its rung is closed. Represents outputs or local variables.
–(/)–	A "not" coil, energized whenever its rung is open.

Table 7: Graphical elements of the LD language

of the *contact* corresponding to IN_3 is determined, next analyzing the *contact* of IN_4. If IN_1's contact is closed **or** the contacts of IN_3 **and** $\neg IN_4$ are closed, the rung will be closed if IN_2's contact is also closed. *LDTest*'s *coil* will be energized only if the rung is closed.
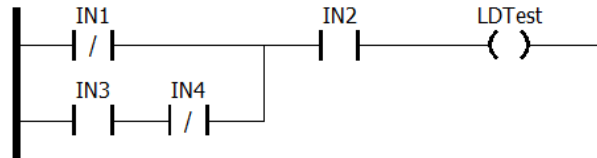
Figure 8: Example of a program in the LD language.

LD also has support for the elements of FBD, so a LD program can make use of *producers* and *consumers*, as well as *blocks*. This way the complexity and expressivity of the language can be increased. In Figure 9 we have an example of a more complex ladder diagram, where whether the rungs are closed or not will also depend on the evaluation realized in a FBD block, with inputs (*var3* and *var4*) received independently of the rung's execution flow.
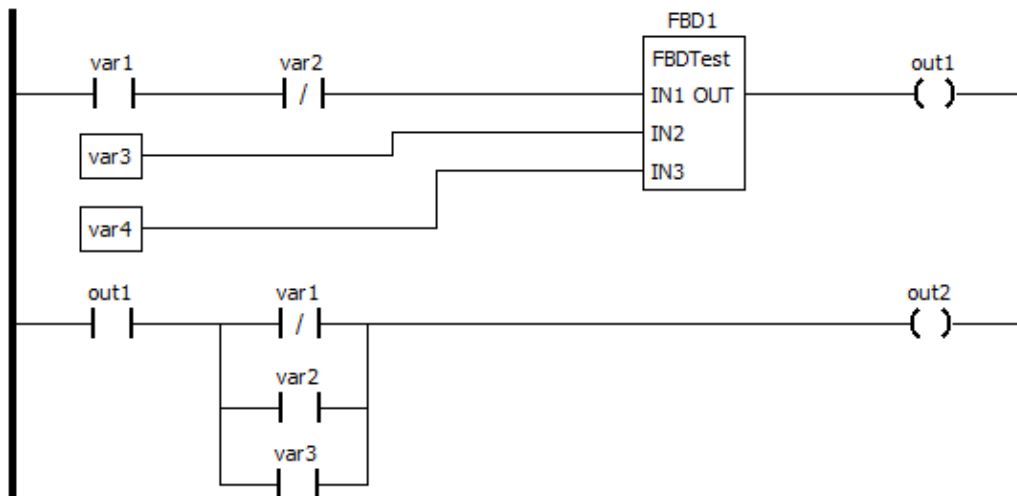
Figure 9: LD diagram making use of FBD elements.

3.1.2.8 SFC

Sequential function chart is a graphical and textual notation for the representation of the structure of a POU *program* or *function block* for a PLC. It is based on three main elements: *steps*, *transitions* and *actions*. Execution goes from step to step according to the validation of the transition between them; at each step, actions may be performed, which are implemented in the other languages of the standard. In Table 8 the SFC elements are summarized.
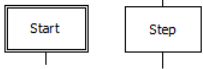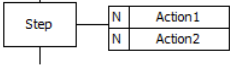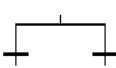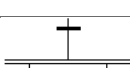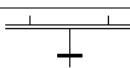
SFC element	Description
Start Step	A single **step** in a SFC sequence, with actions associated. The initial step is marked by a double border.
Step N Action1 N Action2	An **action block** is a set of **actions** associated to a step, whose implementations may be in IL, ST, FBD or LD.
MacroStep	A **macro step** is a step that embodies another SFC program.
╂	A **transition** between SFC elements, with a boolean condition associated.
↓Target	A **jump** to a step, macro step or simultaneous divergence. Acts like a step, as its predecessor must be a transition.
┬ ┿ ┿	A **selection divergence** is used to select one of several partial sequences depending on as many conditions, each contained in a transition.
┿ ┴ ┿	**Selection convergences** of sequences continue several partial sequences in a common one. Each of the alternate sequences must have a transition to the first step of the common one.
╤	A **simultaneous divergence** is used to activate several partial sequences after transitions.
╧	**Simultaenous convergences** combine over a single transition sequences from several steps.

Table 8: Graphical elements of the SFC language

A sequence of steps can be seen in Figure 10. We have *S1* as the initial step, and thenceforth the transitions and steps must be organized in such a way that any two steps are connected by one single transition, as well as two transitions must always be separated by one step. As shown in Table 8 there are special elements to aggregate more than one transition, either **to** (*convergences*) or **from** (*divergences*) a step, which is either of a *selection* or *simultaneous* nature, with the former selecting one of several execution paths and the latter activating or combining multiple execution paths. Thus, considering these *convergence* and *divergence* elements, one has that the valid predecessor elements of a step are the ones presented in Figure 11, while the valid predecessor elements of a transition can be seen in Figure 12.

When the activity flow of the SFC program reaches a step, this one is set as *active*, being reset as *not active* when its successor transition is discharged. Its time of activation is also controlled. These variables, controlling activation and time of activation, are implicit to each step, and can only be read.
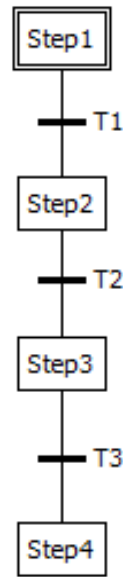
Figure 10: A sequence of SFC steps connected by transitions.
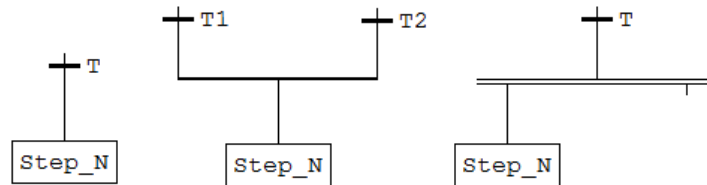
Figure 11: The valid circumstances in which a step can be reached in a SFC program.
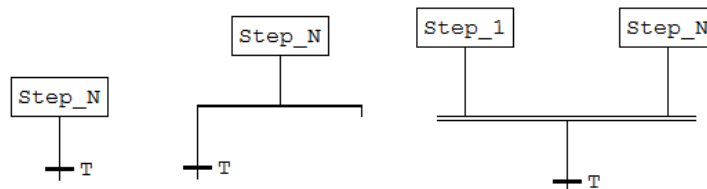
Figure 12: The valid circumstances in which a transition can be reached in a SFC program.

Transitions

Associated to each transition there is a logic *condition*, which will control the deactivation of its predecessor steps and the activation of the successors. The transition is cleared when its condition is equal to **TRUE**, so this condition has to be a boolean expression or a function with a boolean result — statements in **IL** or **ST**, or networks in **FBD** or **LD**, as long as their evaluation is equal to a boolean expression.

Flow control

Using the convergence or divergence elements it is possible for the SFC program to alter how the activity flow runs over the sequences of steps. With the *selection* structures the program may choose between several partial sequences from a step, depending on which of the transitions will be validated — divergence; or represent the alternate sequences that may lead to a step — convergence. With the *simultaneous* structures, the activity flow may be split in several partial sequences simultaneously or they can combine to a single transition the flow coming from numerous steps — divergence and convergence, respectively.
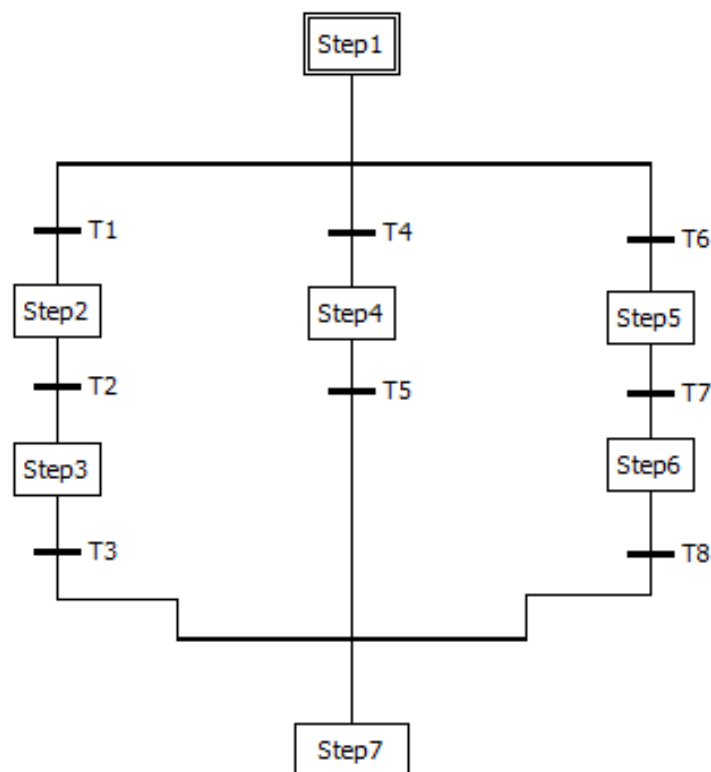
Figure 13: A SFC program using selection convergence and selection divergence.

We can see the use of the *selection* structures in Figure 13. From *Step1* the execution will go to one, and only one, of the possible three sequences depending on which of the transitions's conditions is valid. The default sequence of evaluation is from left to right, but it is possible to associate a **priority** to a transition, in which case the evaluation order will be from the highest to the lowest priority, being the lowest numbered branch the one with higher precedence; the first transition to be valid will guide the activity flow to the step it is connected to. The *selection convergence* will simply mark that *Step7* can be reached after the validation of *T3, T5* or *T8*.
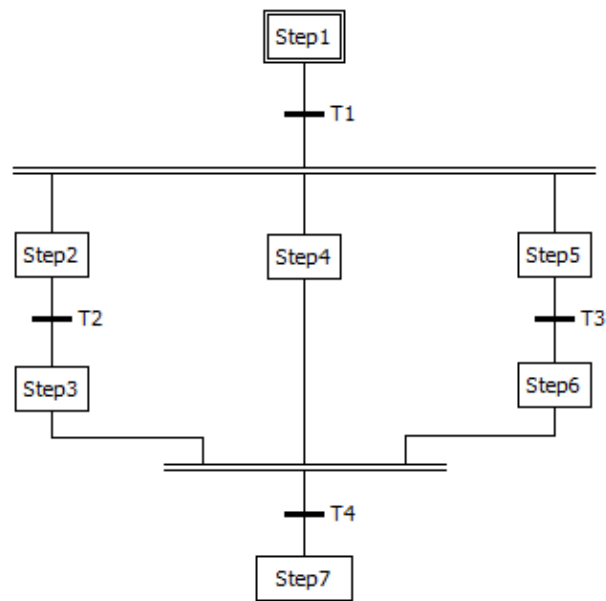
Figure 14: A SFC program using simultaneous convergence and simultaneous divergence.

In Figure 14, however, where we have a similar network but with the *simultaneous* structures, from *Step1*, after *T1* is validated, the three partial sequences will be executed, simultaneously. Then, *only* when all the predecessors steps of the *simultaneous convergence* are active, the activity flow will converge to the evaluation of transition *T4*.
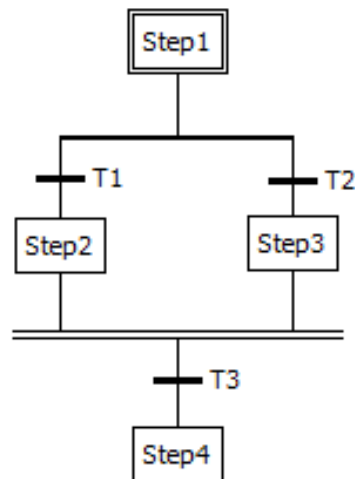
Figure 15: A SFC program wrongly using the simultaneous convergence element.

This necessity of all the predecessors steps' activation so the execution of the POU may continue can lead to errors. Figure 15 presents a case where the execution can never go further than the *simultaneous convergence*, since always only *Step2* or only *Step3* will be active, never both, due to the *selection divergence* after *Step1*. Therefore a **deadlock** situation is present in the SFC program.

Actions

They can contain either a boolean variable, statements in **IL** or **ST**, or networks in **FBD** or **LD**, like the transitions' conditions. Associated with each action there is a *qualifier*: an attribute that may alter the way in which the action is performed — if repeatedly while the respective step is active, if only once, if during some time, and so on; its default value is *N*, where no modification occurs.

Example

Name	Type	Class	Initial Value
n1	INT	Input	
n2	INT	Input	
in1	BOOL	Input	
answers	ANSWER	Local	[YES, NO]
higher	RESULT	Output	UNDETERMINED

Table 9: I/O Table of the SFC program.

An example in which actions are associated to steps can be seen in Figure 16; its I/O table is presented in Table 9, with these variables being used in the SFC transitions and actions. *ANSWER* and *RESULT* are new types declared for use in the SFC program, the former derived from the **Array** type and the latter from the **Enumeration** type.
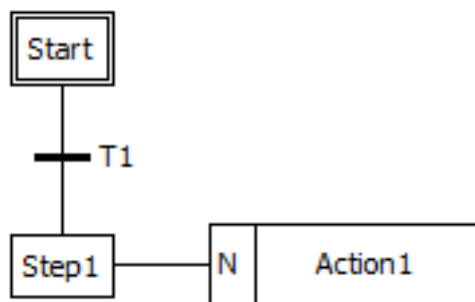
Figure 16: SFC program presenting the discussed elements

```
1  TRANSITION FROM Start TO Step1
2      := in1;
3  END_TRANSITION
4
5  ACTION Action1 :
6      IF isHigher(n1, n2) THEN
7          higher := answers[1];
8      ELSE
9          higher := answers[2];
10     END_IF;
11 END_ACTION
```

Code 3: Action1, in ST

In Code 3 the ST implementations for the transition and the action present in the SFC program are exhibited. The **TRANSITION** construct marks the connection between the steps *Start* and *Step1*, and the activity flow may go from the former to the latter when

its logic condition, the value of the boolean input *in1*, is equals to **TRUE**. The action *Action1* is associated to *Step1*, so it will be executed once the activity flows reaches this step. It evaluates whether, from two integer inputs, the first is higher than the second, producing as output the answer *YES* or *NO*. A calling to the POU *function* **isHigher** is made in line **6**; its implementation will be seen in the next Chapter, when this example will be retrieved.

3.2 PLCopen

The PLCopen standard (PLCOPEN, 2009) is an effort of the PLCopen Technical Committee to gather all the information of the five different languages of the IEC 61131-3 standard and provide an interface with their supporting tools, as well as the ability to transfer information between different platforms. It is an XML-based standard able to store not just the textual, but also the graphical information of a PLC project, allowing complete translation from a representation to another.

The PLCopen standard structures PLCs in three specific parts: the *Project* structure, with headers containing general information about the project; the *Instance* specific part, representing the configurations of the environment in which the PLC may operate; and the *Type* specific part, where we have the *Program Organization Units* (POUs) and the *Data Type Elements*.

In this section we present *Types* and other elements of the PLCopen's XML format in detail.

3.2.1 Variables and Data types

In PLCopen a *variable* is an element whose main attributes, besides its name, are its **type** and the **values** it might receive. The XML element of a variable in PLCopen can be seen in Code 4:

From lines **9** to **11** one can see the attributes of the variable element: its *name*, its *address* — optional information about the *direct representation* of an element — and its *globalId* — optional id of the variable in the project. Between lines **3** to **8** are the items of higher complexity in a variable — the previous mentioned type and initial value, as well as additional information (covered in detail in Section 3.2.3) and documentation.

```
1  <xsd:element name="variable" minOccurs="0" maxOccurs="unbounded">
2    <xsd:complexType>
3      <xsd:sequence>
4        <xsd:element name="type" type="ppx:dataType"/>
5        <xsd:element name="initialValue" type="ppx:value" minOccurs="0"/>
6        <xsd:element name="addData" type="ppx:addData" minOccurs="0"/>
7        <xsd:element name="documentation" type="ppx:formattedText"
              minOccurs="0"/>
8      </xsd:sequence>
9      <xsd:attribute name="name" type="xsd:string" use="required"/>
10     <xsd:attribute name="address" type="xsd:string" use="optional"/>
11     <xsd:attribute name="globalId" type="xsd:ID" use="optional"/>
12   </xsd:complexType>
13 </xsd:element>
```

Code 4: XML element to a Variable

3.2.1.1 Data Type

It is represented as an XML *complex type*, so it can be assigned to elements, like the *type* element in the variable, for instance. Three different groups of possible data types are embodied:

- ELEMENTARY TYPES: with the exception of the *string* types, which have the **length** information as an attribute, the other types contain in the XML element only the name of the type: *BOOL*, *INT*, *UINT*, etc.

- DERIVED TYPES: In this category are the types of higher complexity, the same ones mentioned in Section 3.1.2.2, corresponding to the Derived types for PLC programming.

- EXTENDED: In addition to the IEC 61131-3 standard, a datatype can be of the type *POINTER*. A pointer is defined by its required base type.

Elementary types

```
1  <variable name="boolVar">
2    <type>
3      <BOOL/>
4    </type>
5  </variable>
```

Code 5: XML representation of a boolean variable

A boolean variable *boolVar*, according to the previous code, Code 4, showing an XML variable element, would be of the form presented in Code 5.

Data type elements

New types can be created based on the standard ones. These elements have a name (the name of the new type), the type it is based on (a **Data Type**) and the initial value associated to the type (a **Value**). Variables can be typed according to this new type.

A new type, *SIDE*, based on the data type *Enumeration*, can be seen in Code 6. In line **1** the name of the new data type is stated, while from lines **2** to **10** the type itself is constructed. In line **3** it is defined its derivation form the Enumeration type, with the definition of the possible values an enumeration of the *SIDE* type may assume made in lines **5**-7.

```xml
1  <dataType name="SIDE">
2    <baseType>
3      <enum>
4        <values>
5          <value name="right_side"/>
6          <value name="no_side"/>
7          <value name="left_side"/>
8        </values>
9      </enum>
10   </baseType>
11 </dataType>
```

Code 6: XML representation of a new type based on the *Enumeration* data type.

Derived types

In the interest of space we show in detail only the information about the derived type **Array**, so the logic behind the data types representation can be understood. The complete specification for all the PLCopen elements is available in the standard.

The Array type is depicted in Code 7. The **baseType** element defines the data type of the array items, while the **dimension** element determines the size of each dimension

in terms of a range.

```xml
1  <xsd:element name="array">
2    <xsd:complexType>
3      <xsd:sequence>
4        <xsd:element name="dimension" type="ppx:rangeSigned" maxOccurs="
              unbounded"/>
5        <xsd:element name="baseType" type="ppx:dataType"/>
6      </xsd:sequence>
7    </xsd:complexType>
8  </xsd:element>
```

Code 7: XML element for the Array Type

Thus, an array declared as *arrayVar : ARRAY [1..10] OF INT;* would be represented in PLCopen as:

```xml
1  <variable name="arrayVar">
2    <type>
3      <array>
4        <dimension lower="1" upper="10"/>
5          <baseType>
6            <INT/>
7          </baseType>
8      </array>
9    </type>
10 </variable>
```

Code 8: XML representation of an array variable

3.2.1.2 Value

The **Value** element is organized similarly to **Data Type**, as it embodies the three different kinds of values that can be assigned to a variable:

- SIMPLE VALUE: the values that can be represented as a single attribute: the value itself as a string. It is used for the variables with elementary types.

- ARRAY VALUE: values as a list of $\{repetitionValue \Rightarrow value\}$ objects, so that repeated sequential values in the array do not need to be presented redundantly; the default value of *repetitionValue* is 1. The *values* are represented as **Value** elements.

- STRUCT VALUE: values as a list of $\{member \Rightarrow value\}$ objects, so that each value is associated to the name of the element it belongs to in the *struct*. These *values* are represented as **Value** elements.

The declaration

$$arrayVar : ARRAY \ [1..10] \ OF \ INT := \{1, 2, \underline{5}(3), \underline{3}(4)\}; \ [1]$$

would result in the value $\{1, 2, 3, 3, 3, 3, 3, 4, 4, 4\}$ to *arrayVar*. The XML whole representation of *arrayVar* can be seen in Code 9:

```
1  <variable name="arrayVar">
2    <type>
3      <array>
4        <dimension lower="1" upper=
            "10"/>
5          <baseType>
6            <INT/>
7          </baseType>
8      </array>
9    </type>
10   <initialValue>
11     <arrayValue>
12       <value>
13         <simpleValue value="1"/>
14       </value>
15       <value>
16         <simpleValue value="2"/>
17       </value>
18       <value repetitionValue="5">
19         <simpleValue value="3"/>
20       </value>
21       <value repetitionValue="3">
22         <simpleValue value="4"/>
23       </value>
24     </arrayValue>
25   </initialValue>
26 </variable>
```

Code 9: XML representation of an array variable with an initial value associated.

3.2.2 POUs

A *POU* is an element whose main attributes, besides its name, are its **POU type**, its **interface** and its **body**. The first defines if it is a POU *program*, *function block* or *function*; the second represents the **I/O** table of the POU, a list of the variables that are used in it; the last is the "implementation" of the POU, in one of the five languages of the IEC standard.

In additional to those elements, there are *action* and *transition* elements associated to a POU: they represent the key components of SFC; each has a body associated.

---

[1]The underlined numbers are the *repetition values*: the number inside the parentheses is repeated the specified times in the array.

3.2.2.1 Interface

The **Interface** element of the POU is composed of lists of variable elements. Each list represents one of the variable categories summarized in Section 3.1.2.2 — Locals, Inputs, etc. Besides these lists there is an element *returnType*, which is required by the POU *functions*, and represented as a **Data Type** element.

We can see the XML format of an **Interface** looking to how the **I/O** table of the IL example in Section 3.1.2.3 is represented in PLCopen, depicted in Code 10.

```xml
1  <interface>
2    <returnType>
3      <REAL/>
4    </returnType>
5    <inputVars>
6      <variable name="X1">
7        <type>
8          <REAL/>
9        </type>
10     </variable>
11     <variable name="X2">
12       <type>
13         <REAL/>
14       </type>
15     </variable>
16     <variable name="Y1">
17       <type>
18         <REAL/>
19       </type>
20     </variable>
21     <variable name="Y2">
22       <type>
23         <REAL/>
24       </type>
25     </variable>
26     <variable name="TMax">
27       <type>
28         <REAL/>
29       </type>
30     </variable>
31   </inputVars>
32   <localVars>
33     <variable name="Temp">
34       <type>
35         <REAL/>
36       </type>
37     </variable>
38   </localVars>
39   <outputVars>
40     <variable name="ERROR">
41       <type>
42         <BOOL/>
43       </type>
44     </variable>
45   </outputVars>
46 </interface>
```

Code 10: Interface of a POU in PLCopen.

3.2.2.2 Body

The **Body** element of a POU may be of one in the following four types, depending of the language it corresponds to:

- TEXTUAL: The **IL** and **ST** implementations are represented as text, with no special XML modularization.

- FBD: The **FBD** implementations, represented as a collection of common graphical objects and FBD objects.

- LD: The **LD** implementations, represented as a collection of common graphical objects, FBD objects, and LD objects.

- SFC: The **SFC** implementations, represented as a collection of common graphical objects, FBD objects, LD objects and SFC objects.

Common features

The following items are common to all the graphical objects, providing their identification and graphical information:

- LOCAL ID: Attribute that uniquely identifies the graphical object in the context of that POU.

- HEIGHT: Attribute representing the height of the graphical object.

- WIDTH: Attribute representing the width of the graphical object.

- POSITION: It presents the attributes x and y, defining a graphical position with (X, Y) coordinates. This position is where the top-left corner element of the object is situated: the other points of the object are implied in conjunction with the *width* and *height* attributes.

Regarding the interaction between these graphical objects, the following elements may be used:

- CONNECTION POINT OUT: It represents an *output* pin, marking, through a **Position** element, the relative position[2] of this pin to the graphical object it refers to. The extension **Connection Point Out Formal Parameter** has the attribute *formalParameter*, which identifies the output pin, useful when an object has more than one.

- CONNECTION POINT IN: It represents an *input* pin, being the counterpart of the **Connection Point Out** element. However, it has as subelement a *collection* of **Connection** elements, describing the graphical couplings between the data consuming element, with the input pin, and another elements.

---

[2]These positions are relative to the top-left corner of the graphical object, the origin (0, 0).

- CONNECTION: It describes a connection between two elements and may have a collection of **Position** elements to mark the path of this connection. The attribute *refLocalId* identifies the element where the connection starts, with *formalParameter*, if any, indicating from which output pin.

```xml
1 <block localId="10" width="76" height="120" typeName="ILTest">
2   <position x="107" y="109"/>
3   <inputVariables>
4     <variable formalParameter="X1">
5       <connectionPointIn>
6         <relPosition x="0" y="30"/>
7       </connectionPointIn>
8     </variable>
9     <variable formalParameter="X2">
10      <connectionPointIn>
11        <relPosition x="0" y="50"/>
12      </connectionPointIn>
13    </variable>
14    <variable formalParameter="Y1">
15      <connectionPointIn>
16        <relPosition x="0" y="70"/>
17      </connectionPointIn>
18    </variable>
19    <variable formalParameter="Y2">
20      <connectionPointIn>
21        <relPosition x="0" y="90"/>
22      </connectionPointIn>
23    </variable>
24    <variable formalParameter="TMax">
25      <connectionPointIn>
26        <relPosition x="0" y="110"/>
27      </connectionPointIn>
28    </variable>
29  </inputVariables>
30  <outputVariables>
31    <variable formalParameter="">
32      <connectionPointOut>
33        <relPosition x="76" y="30"/>
34      </connectionPointOut>
35    </variable>
36    <variable formalParameter="ERROR">
37      <connectionPointOut>
38        <relPosition x="76" y="50"/>
39      </connectionPointOut>
40    </variable>
41  </outputVariables>
42 </block>
```

Code 11: Block in PLCopen

FBD objects

The **BodyFBD** element is composed of a collection of the FBD objects. These objects are the same as the FBD language elements in Table 6, at page 44, besides the **Block** element, defined as part of BodyFBD.

The XML representation of the block corresponding to the IL example from Section 3.1.2.3, shown in Figure 5, at page 42, is depicted in Code 11. The *typeName* attribute identifies the type of the block, in this case the name of the POU, so the block implementation can be retrieved. As it is a POU *function* there is no instantiation, what would be represented by the addition of the attribute *instanceName*. The variable elements in a block correspond to its *input*, *inOut* or *output* variables; thus, each possess either a **Connection Point In**, a **Connection Point Out** or both, respectively. Also, each variable element has a *formalParameter* attribute, defining the name of the respective variable in the corresponding POU's interface; the exception is at line **31**, whose variable element without a *formalParameter* marks the *return* of the POU *function* being represented.

LD objects

The **BodyLD** element is composed of a collection of LD objects. These objects are the same as the LD language elements in Table 7, at page 44. The objects present in **BodyFBD** can also be used as elements of BodyLD.

The *power rail* elements are composed, besides their identification and graphical information elements, of collections of *connection point out* or *connection point in* elements — the left and right ones, respectively.

The *contact* and *coil* elements present *input* and *output* pins, as well as a *variable* element to indicate to which variable the graphical object is associated. Attributes such as *negated* establishes whether it is a *not contact* or a *not coil*.

SFC objects

The **BodySFC** element is the more complex one in PLCopen. It contains the objects presented in Table 8, at page 46, with the SFC language elements, as well as the elements in **BodyFBD** and **BodyLD**.

The **Step** element has as attributes its *name* and whether it is an *initial step*. It may present an input pin — mandatory if it is not an initial step —, an output pin — mandatory if its not a final step – and/or an output *action* pin, which will connect the step to its **Action Block**.

The **Action Block** elements are collections of **Action** elements, each one containing the implementation of an action through a **Body** element, as well as a string attribute to identify the action's qualifier.

Regarding the **Transition** elements, their main items are the attribute *priority* — optional feature that may be used in case of the transition being part of a *selection divergence* —, the input and output pins (both mandatory) and the **Condition** element. This element represents the logic condition of the transition; its attributes are whether the condition is *negated* and the implementation of the condition itself through a **Body** element.

```
1  <pou name="sfcSimple" pouType="program">
2    <interface>
3      <inputVars>
4        <variable name="n1"> ... </variable>
5        <variable name="n2"> ... </variable>
6        <variable name="in1"> ... </variable>
7      </inputVars>
8      <localVars>
9        <variable name="answers"> ... </variable>
10     </localVars>
11     <outputVars>
12       <variable name="higher">
13     </outputVars>
14   </interface>
15   <actions>
16     <action name="Action1"> ... </action>
17   </actions>
18   <transitions>
19     <transition name="T1"> ... </transition>
20   </transitions>
```

Code 12: SFC POU in PLCopen — Interface , Actions and Transitions

In terms of the XML representation, there is little difference between the convergence and divergence elements, either *simultaneous* or *selection*. The convergences present a collection of input pins and an output pin, while the divergences an input pin and a collection of output pins. The **Simultaneous Divergence** element presents a *name* attribute, since it may be a target of a **Jump Step** element.

For the SFC example shown in the end of the SFC section (3.1.2.8), depicted in

Figure 16, at page 50, we have the PLC representation in Codes 12 and 13:

Each of the POU's variable listed in Table 9 is represented as a variable element in the interface element. They are divided in lists corresponding to the variable categories — *inputVars* for inputs (line **3**), *localVars* for locals (line **8**) and *outputVars* for outputs (line **11**). Only the names of the variables are listed here, for clarity; the structure of each variable element is according to the PLCopen representation of variables shown in the previous sections. Each action and transition in a SFC POU is listed in a corresponding element; the action element for *Action1* (line **16**) will contain its ST code, as well as the transition element for *T1* (line **19**) will contain its ST logic condition.

```
21   <body>
22     <SFC>
23       <step localId="1" height="27" width="38" name="Start"
                         initialStep="true"> ... </step>
24       <transition localId="2" height="2" width="20">
25         ...
26         <condition>
27           <reference name="T1"/>
28         </condition>
29       </transition>
30       <step localId="3" height="23" width="38" name="Step1"> ... </step>
31       <actionBlock localId="4" width="100" height="30">
32         <connectionPointIn>
33           <connection refLocalId="3"> ... </connection>
34         </connectionPointIn>
35         <action localId="0">
36           <reference name="Action1"/>
37         </action>
38       </actionBlock>
39     </SFC>
40   </body>
41 </pou>
```

Code 13: SFC POU in PLCopen — Body

Code 13 has the *body* of the SFC program in the example. In lines **23** and **30** the common features of the steps are exhibited, with **24** showing them for the transition. Line **33** shows the link of the action block to its respective step, marked by the *refLocalId* being the same as the *localId* of *Step1*; the reference of the implementation of the action inside the action block (line **36**) is the same as for the transition's condition code (line **27**): they indicate the name of the *action* and *trasition* elements in lines **16** and **19** from the previous Code, respectively.

3.2.3 Additional Data

```
1 <addData>
2   <data name="http://www.anyVendor.org/Vendor_AddData"    handleUnknown=
        "implementation">
3     <AddDataRoot>
4       <FirstElement ExampleAttribute1="PLCopen_extention" />
5       <SecondAttribute/>
6     </AddDataRoot>
7   </data>
8 </addData>
```

Code 14: Example of use of the Add Data feature

The **AddData** elements represent the additional data of an application that does not fit the defined structure of the XML schema. This feature may be used to represent extensions to the IEC 61131-3 standard, or information from legacy systems deviating from the standard.

The element is structured as a collection of **Data** elements, each one consisting of a collection of free XML elements, that can be constructed based on the application needs, in attributes, content and sub-elements.

An example of the application of this technique can be seen in Code 14.

3.3 B Method

The B Method (ABRIAL, 1996) is a formal approach for the specification and development of software. It includes a first-order logic with integers and sets, substitutions and refinement rules. It is based on the Abstract Machine Notation (AMN), which provides a unique language for the construction of machines, refinements and implementations, thus representing the different levels of abstraction that a specification of a system may take. Besides, the language supports decomposition, since it is based around the concept of layered development and the construction of larger components from collections of smaller ones.

Despite the features provided by its language there are some elements of the PLC programs that cannot be represented with B by default, like *real numbers*. In Chapter 4, where the translation method is described, these limitations are discussed in detail.

Nevertheless, the B Method provides a unified pragmatic and usable development

methodology based on the concept of refinement, requiring the consistency verification for each transformation of the specification from the abstract level towards the concrete one. This, along with the generation and verification of proof obligations to guarantee the consistency of the initial model, makes correctness according to the specification a realistic and achievable goal throughout system development.

3.3.1 AMN notation

As previously said, the AMN notation provides three different elements of abstraction to represent the formal specification, containing pieces of information describing the model. Their structure is detailed below.

MACHINE

```
1  MACHINE n
2  SETS S
3  CONSTANTS C
4  PROPERTIES P
5  VARIABLES V
6  INVARIANT I
7  INITIALISATION T
8  OPERATIONS
9    outputs <-- op (inputs) =
10         PRE preconditions
11         THEN substitutions
12         END ;
13   ...
14 END
```

Code 15: Basic structure of a B machine

The machines are the most *abstract* component of the B specifications, being the base for the refinements towards a more concrete level. They are suitable for describing *what* behaviour is performed in a system, with relationships between states. The basic structure of a B machine is presented in Code 15.

The clause **MACHINE** states the name of the machine — n —, while **SETS** and **CONSTANTS** define the sets and constants to be used in it, with **PROPERTIES** declaring the conditions that must hold on them; they define the static information in a machine. A little example can be seen below,

```
SETS NAMES = {pedro, paulo, roberto, turin, rafael, antonino}
CONSTANTS capacity
PROPERTIES
    capacity : NAT1 &
    capacity = 5 &
    card(NAMES) > capacity
```

where a **set**, *NAMES*, is defined and has the elements that compose it declared, whose quantity is bounded by the **constant** *capacity* through the **property** `card(`NAMES`)>` `capacity`. This property states that *NAMES* must have *more* elements than the value of *capacity*, which is the case, since the set has *six* elements and the constant is defined as a natural number of value *five*.

As for the state information in a machine, **VARIABLES** defines the set of variables *V* that represents it, with the **INVARIANTS** clause presenting constraints over this state, hence over the variables; these constraints must *always* hold. Using the previous example, we could define the following information

```
VARIABLES members
INVARIANT
  members <: NAMES & card(members) <= capacity
```

where a **variable** *members* would compose the state of the machine, which must always be compliant with the **invariants**, stating that the values *member* may take are a subset of *NAMES* and its number of elements is never higher than *capacity*.

The **INITIALISATION** clause specifies the initial state of the machine — the initial values that its variables take —, and the **OPERATIONS** define the interface of the machine with the environment it is interacting with: they may alter the state of the machine through *substitutions* (line **11**), which specifies the behaviour of the operation — what it does; an operation can also take inputs and produce outputs (line **9**), according to its statements; *preconditions* (line **10**), if defined, establish the restrictions that must be fulfilled by the received parameters and the current state of the machine so the operation may be performed, otherwise its execution being forbidden. Extending the previous example we can define the following,

```
INITIALISATION members := {paulo, turin}
OPERATIONS
   remove_paulo =
   PRE paulo : members
   THEN
      members := members − {paulo}
   END
```

where a set, `{paulo, turin}`, is defined as the **initial value** of *members* and an **operation**, `remove_paulo`, which can remove the element "paulo", in a scenario where it is part of *members*, from this variable. One must notice that these actions have to be compliant with the invariant, which is the case, since the initial value assigned in the initialisation is indeed a subset of *NAMES* and the operation performed does not change the type of *members* or makes it exceed the *capacity* bound.

At the machine level the substitutions are executed only in parallel faction, so they alter the state of the machine only once: since in this level of abstraction one is not concerned in how the behaviour of the system is achieved more complex constructs are hardly necessary.

REFINEMENT

```
1 REFINEMENT n_r
2 REFINES n
3 SETS S_r
4 CONSTANTS C_r
5 PROPERTIES P_r
6 VARIABLES V_r
7 INVARIANT I_r
8 INITIALISATION T_r
9 OPERATIONS
10    outputs <-- op (inputs) =
11         PRE preconditions_r
12         THEN substitutions_r
13         END ;
14    ...
15 END
```

Code 16: Basic structure of a B refinement

A refinement takes the specification to a more concrete level, so one may be concerned in *how* the behaviour of a system is performed, not simply *what* it is. Thus some aspects of the specification, such as the nondeterministic statements, may be addressed differently:

providing deterministic choices between the possible outcomes of these statements, for instance. Also, a refinement may enhance the specification providing alternative ways to deal with aspects of it, which means that the information in the abstract level that is not directly accessed in the refinement does not need to be represented.

As seen in Code 16 the basic structure of a B refinement is the same of a machine, differing in the kinds of substitutions that may be used inside them, since a refinement has additional constructs besides the ones present at machine level, like *sequential composition*: performing a substitution over the resulting state of a previous substitution in the same operation — the execution of a statement after another, desirable when modeling the operations in order to obtain a more concrete specification.

In the clause **REFINEMENT** the name of the refinement is declared, which is usually defined as the name of the machine (or refinement) it refines, stated in the clause **REFINES**, plus the suffix "_r". The static information, and their properties, defined must be different from that in the refined machine, but otherwise its declaration follows the same guidelines; this restriction is due to the static part of the refined machine being available for use in the refinement. The state information (the variables) of the abstract machine is also available for use in the refinement, but it is not visible by the refinement operations, so it can only be accessed in the **INVARIANTS** (the so called *linking invariants*[3]), **INITIALISATION** or operations' *preconditions*, connecting the refining with the refined state. The variables in the refinement's **VARIABLES** clause may be used freely, obeying the same procedures for variables declaration within a machine.

In the **INITIALISATION** clause the initial state of the variables declared in the refinement must be given, and it can also use the initial values for the variables of the refined machine — first the initialisation of the machine is performed, with the substitutions in the refinement being performed *over the resulting state*. The refining operations must have the same signature of the refined ones, while new preconditions may be defined for an operation, but they have to be compliant with the preconditions already stated at machine level — the same goes for the substitutions composing the body of the refining operation: the possible behaviours must always be within the behaviour set of the refined operation.

---

[3]Invariants bounding the abstract and the refining state, relating a variable in the former with one defined in the latter.

IMPLEMENTATION

An implementation is a special kind of refinement, from which code can be generated. It is the maximum of the B method in terms of concreteness, so some restrictions must be fulfilled in order to achieve this level, like the absence of nondeterminism. Additional constructs are available in this level as well, like *loops*.

We do not present this level in depth, since we do not make use of it in this work, but further details, as well as for the other elements presented in this Chapter, may be seen in (SCHNEIDER, 2001).

3.3.2 Substitutions

One of the key elements for understanding the B method is the concept of *substitutions*, since they are the instrument for changing the state of a machine in a specification.

A substitution is the *replacement* of a free variable (not in the scope of a quantifier) for an expression inside a predicate, resulting in an equivalent predicate. Formally: $P[E/x]$, which is read as "P with E for x". Thus:

$$(jo\tilde{a}o \in HUNGRY)[maria/jo\tilde{a}o] \equiv (maria \in HUNGRY)$$

A summary of the main rules of substitution available for B specifications is given in Table 10:

N^o	Name	Substitution rule
1	**SKIP**	$[\textbf{skip}]\,Q \equiv Q$
2	**BEGIN**	$[\textbf{BEGIN}\ S\ \textbf{END}]\,Q \equiv [S]Q$
3	**PRE**	$[\textbf{PRE}\ P\ \textbf{THEN}\ S\ \textbf{END}]\,Q \equiv (P \wedge [S]Q)$
4	**SIMPLE**	$[x := E]Q \equiv Q[E/x]$
5	**ARBITRARY**	$[x :: S]Q \equiv \forall z.(z \in S \Rightarrow Q[z/x])$
6	**PARALLEL**	$[x := E \parallel y := F]Q \equiv Q[E/x, F/y]$
7	**SEQUENCING**	$[x := E;\ y := F]Q \equiv Q[E/x;\ F/y]$
8	**IF**	$[\textbf{IF}\ E\ \textbf{THEN}\ S\ \textbf{ELSE}\ T\ \textbf{END}]\,Q \equiv (E \wedge [S]Q) \vee (\neg E \wedge [T]Q)$
9	**ANY**	$[\textbf{ANY}\ x\ \textbf{WHERE}\ P\ \textbf{THEN}\ T\ \textbf{END}]\,Q \equiv \forall x\,.\,(P \Rightarrow [T]Q)$
10	**VAR**	$[\textbf{VAR}\ v\ \textbf{IN}\ S\ \textbf{END}]\,Q \equiv \forall v\,.\,([S]Q)$

Table 10: Rules of Substitution in B

The first rule is the "do nothing" substitution, where the predicate is not altered in any way: it is equivalent to itself. One of its possible uses for the body of an operation at abstraction level whose behaviour will be defined only in a following refinement.

BEGIN is a rule used to bracket statements in a block, for clarity — the substitutions S will be applied in the predicate Q with no restrictions. The **PRE** rule, however, states restrictions to the execution of the substitutions S in Q: the validity of P is mandatory for the performance of S.

The **SIMPLE** rule states the simple substitution, according to the definition of substitutions: the replacement of an expression for a free variable in a predicate; the symbol ":=" represents it. Alternatively, with the **ARBITRARY** rule, a nondeterministic assignment may be made to a free variable, in which any element belonging to a set replaces it in a predicate; this assignment is denoted by the symbol "::".

A **PARALLEL** substitution is the *simultaneous* application of simple substitutions in a predicate, represented by the symbol "∥"; an important remark is that in a parallel substitution each variable may be substituted only once, since all the replacements are simultaneous. Rule number **7** presents the **SEQUENCING** substitution, which allows a replacement to be made *after* another, thus the same variable could be used in different sequential substitutions, having different states in each of them.

Conditional substitution is defined by the **IF** rule, which states that either you have condition E valid and the performance of substitutions S in Q <u>or</u> you have that the condition E is not valid and the T substitutions are performed in Q. Hence it presents a deterministic behaviour.

The behaviour of the **ANY** rule, however, is nondeterministic, as it states that <u>any</u> value that satisfies a given predicate, P, is used in the substitutions T applied to Q. Thus, "local variables" are created, with a type and any other arbitrary constraining information established in predicate P, to be used in the statements in T, which may refer to these local variables, making its execution dependent on the valuation they received. An interesting feature is that as new **ANY** statements may be defined in the T substitutions the local variables may "propagate", simulating sequential substitutions even without the rule of **SEQUENCING**.

The **VAR** rule however defines *local variables* truly, so they can be used is some context without the need to attach them to the whole state space, as they are discarded once that context is left — it is very similar to the **ANY** rule, although deterministic and

not so restricted. Some v variables are declared and may be used in S; the restriction is only that the values of the v variables must be assigned *prior* to its use in other statements of S, where they can be freely modified again; therefore, this construct in available only in refinement level, as sequential compositions are mandatory.

3.3.3   Proof Obligations

Once a B specification is built one has to verify its correctness. This task is carried out using *proof obligations* — logic expressions generated from a specification and that must be validated to guarantee its consistency.

Proof obligations are produced to guarantee that the model never reaches an invalid state, based on clauses such as **INVARIANTS**, that defines constraints over the state of the of the model, asserting that the system always stays within some allowed region. The key elements for the generation of proof obligations are the *substitutions*, as they may change the state of the system: it is necessary to guarantee that an invalid state is not reached by the substitutions performed in a valid one. Recursively, the basic step is in the clause **INITIALISATION**, where the state the system starts in is defined: it must be compliant to the invariant one. Thus, the proof obligations concerning invariant preservation are generated for the initialisation and operations executed in the system according to the following guidelines:

Consistency of INVARIANT

$$\exists v.I$$

This requires that there is a valid valuation for the variables that satisfy the invariant clause I in the machine — there is *at least one* valid state in the machine, so the invariant is consistent.

Proof obligations for INITIALISATION

$$[T]I$$

It is necessary to guarantee that the state resulting from the initialisation satisfies the invariant — the initial state obtained by the substitutions T establishes I.

Proof obligations for OPERATIONS

$$(P \wedge I) \Rightarrow [S]I$$

For each operation it is mandatory that its execution does not lead the machine to an invalid state. Thus, once its preconditions P (if any) are satisfied, establishing the conditions over the machine's state in which the operation may be performed, <u>and</u> the invariant I is valid, specifying that the state *before* the operation's performance is valid, the substitutions S carried out must lead the machine to a state where I is also valid.

The validity of the preconditions before the invocation of an operation must be guaranteed by the one who is making the invocation — the operation is concerned in not leading the machine to an invalid state only when P holds; its behaviour with P invalid is not guaranteed in any way. Another characteristic of the operation's preconditions is that they are *local*: they are taking into account only in the proof obligations of the respective operation, hence their consideration in the proof obligations of other machine is not possible; only the invariants are globally visible by all the proof obligations.

If all the operations in a machine are proven to not lead it to an invalid state from a valid one, and if its initial state is also valid, then the invariant must indeed hold for every reachable state in the machine's execution, thus allowing one to claim the validity of the full behaviour set of the specification in unlimited depth.

Proof obligations for refinements

Proof obligations are also generated to attest the correctness of a refinement step in a B specification. These proof obligations will concern the initialisation and operations of a refinement, as well as the static information (sets and constants) introduced. The refinement must not initiate in a state that prevents the refined machine of having any initial state satisfying the invariants that hold between them (the linking invariants). Also, the substitutions performed in the operations must not lead the refinement to a state that can not be matched by any execution of the operation in the abstract level. The operations in the refinement consider the preconditions defined in the abstract level (if any) to restrict the states in which it can be executed, but new preconditions can also be defined; in this case these preconditions must hold whenever the operation would hold within its abstract precondition.

3.3.4 Example

To a better understanding of the proof obligations generation and of the previous features of the B method discussed we present an overviewing example.

The B machine seen in Code 17 models a car transporter: a big truck capable of carrying fifteen different cars; the truck is initially empty, but can be loaded with cars, one at a time. The state of the machine is defined by the quantity of cars that are already loaded in the truck — the variable *loadedCars* (line **3**); this variable needs to be typed, so in line **2** we have the definition of a set, *CARS*, a static construct to whom the elements in *loadedCars* will belong — hence the variable is a subset of *CARS* (line **4**). Since the truck transports at most 15 cars the number of loaded cars must be limited, which is represented by the second conjunct in the invariant in line **4** — the cardinality of the subset has to be no greater than fifteen. The act of loading a car into the truck is modeled by the operation in lines **7-11**, which receives a variable *car* as input and adds it to *loadedCars*; the precondition of the operation restricts its execution to the states in which the variable *car* is a member of *CARS* and it is not already in the truck (line **8**). As the truck is initially empty its initial state is stated as without any loaded car (line **5**).

```
1 MACHINE CarTransporter
2 SETS CARS
3 VARIABLES loadedCars
4 INVARIANT loadedCars <: CARS &
      card(loadedCars) <= 15
5 INITIALISATION loadedCars := {}
6 OPERATIONS
7     loadCar (car) =
8     PRE car : CARS & car /: loadedCars
9     THEN
10        loadedCars :=
              loadedCars \/ {car}
11    END
12 END
```

Code 17: Example of a B *machine.*

```
1 loadCar (car) =
2 PRE car : CARS &
      car /: loadedCars &
      card(loadedCars) < 15
3 THEN
4     loadedCars :=
          loadedCars \/ {car}
5 END
```

Code 18: Necessary alteration for *invariant compliance*

Three proof obligations are generated for this model: two for the INITIALISATION and one for the **loadCar** operation. They are summarized below:

1: $\{\} \overset{?}{<:} CARS$ Verify if the initial state of *loadedCars* is compliant with the invariant (type)

2: $card(\{\}) \overset{?}{<=} 15$ Verify if the initial state of *loadedCars* is compliant with the invariant (cardinality)

3: $card(loadedCars \setminus/ \{car\}) \overset{?}{<=} 15$ Verify if the substitution performed respects the cardinality constraint over *loadedCars* in the invariant.

The first two are proven correct, as the empty set is a subset of any set and its cardinality is zero. The third, however, is not; its full structure is:

$$
\begin{aligned}
&(\\
&\quad (car : CARS \And car/ : loadedCars)\\
&\quad \wedge (loadedCars <: CARS \And card(loadedCars) <= 15)\\
&)\\
&\quad\quad \overset{?}{\implies} (card(loadedCars \setminus/ \{car\}) <= 15)
\end{aligned}
$$

There is not enough information in the precedents (precondition and invariant) to assure that the resulting substitution will guarantee the invariant. Thence one possible solution is strengthening the preconditions of the operation, as seen in Code 18, where the restriction $card(locatedCars) < 15$ is added, ensuring that $card(loadedCars \setminus/ \{car\})$ is at most equals to 15, thus satisfying the proof obligation.

Refining

A refinement may be performed over the abstract model to add information, like *how* the cars are allocated in the truck. The refinement in Code 19 introduces the variable *allocatedCars* (line **4**), typed in line **5** as a partial function from *SPOTS*, a new set defined in line **3**, to *CARS*, thus biding the loaded cars to the spot they were allocated in the truck; the restriction that the range of the function must be equal to the *loadedCars* variable in the refined machine links the abstract and the refinement state, assuring that the same behaviour is represented.

The initialisation of the refinement (line **6**) states that the no spot has a car allocated, which represents the same behaviour as in the abstract component — as both variables are valuated with the empty set the invariant biding them is satisfied. In the refinement the operation **loadCar** will nondeterministically cast a spot not yet allocated in the truck (line **10**) and associate it with the car being loaded; since the same preconditions as in the abstract level are applied this car is not already in the truck and the total

```
1 REFINEMENT CarTransporter_r
2 REFINES CarTransporter
3 SETS SPOTS
4 VARIABLES allocatedCars
5 INVARIANT
      allocatedCars : SPOTS +-> CARS & ran(allocatedCars) = loadedCars
6 INITIALISATION allocatedCars := {}
7 OPERATIONS
8     loadCar (car) =
9     BEGIN
10        ANY ss WHERE ss : SPOTS & ss /: dom(allocatedCars) THEN
11            allocatedCars(ss) := car
12        END
13    END
14 END
```

Code 19: Example of a B *refinement*

number of cars is at most 15 after the performance of the operation, thus the reachable states are matchable with the abstract execution of **loadCar** and the proof obligation $ran(allocatedCars<+\{ss \mapsto car\}) \overset{?}{=} (loadedCars \setminus \{car\})$ is satisfied.

3.3.5 Decomposition

The B method provides support for decomposition through the **INCLUDES**, **SEES** and **USES** clauses, which allow a machine to be expressed by the combination of subsidiary ones, so the complexity of the specification is split among different components. State information may be separated into a number of different machines, each responsible for the behaviour performed on that part of the state; this also provides the independent verification of the *internal consistency* of these machines, hence reducing the proof effort and also permitting the re-use of these components without the need of new unit verifications; these machines are defined **independently** of the composing components, they have access to no information in them, which permits this independent verification.

The main difference between the kinds of possible relationships is regarding the way they deal with the *state information* and its access. These differences are better explained below. A characteristic common to all the structuring relationships is that the target of the relation, the included, seen or used machine, must be at the most abstract level, without any refinement; this is so because the composing component, may it be a machine or a refinement, is not interested in *how* the behaviour is performed inside the composed component, only in *what* it is.

INCLUDES

In the Includes relationship the static information of the included machine is seen as if declared in the including component itself, while the restrictions over them, defined in the **PROPERTIES** clause, may be increased of constraints defined in the including component.

The included state is part of the including state — a read-write access is provided, with direct reading being possible in invariants, preconditions or substitutions; the writing access is more restricted, possible only through the operations of the included machine. This restriction is necessary because direct-assignments could lead the included machine to a state not allowed by its invariant: the included machine can only guarantee that its invariant always hold if no other component can modify its state except through its own operations. Proof obligations are generated to ensure that the invocation of these operations by the including component is performed only when satisfying the operation's preconditions.

Constraints over the state in the included machine may be defined in the Invariant of the including component, so it can be under complete control of the including component when updated; otherwise updates that send the machine to an inconsistent state (in view of the including component invariants) would be achievable (if the included machine were included by other machine that modified it, for instance) — operations of the included machine are available only for the unique component including it.

The *visibility* of an included machine's state is just "one level deep". A machine including a machine that in turn includes another does not have access to the state of the latter, either for reading the variables or performing its operations. The **PROMOTES** clause can be used in the including component to make available the operations of the included machine for a higher depth of visibility.

SEES and USES

While the Includes relationship forces the included machine to be part of and completely under control, the Sees/Uses mechanism permits a read-only access between machines. As there is no alteration of state in the relationship a machine can be related to a number of others by these relationships.

As in the Includes case, the visibility is not transitive, but the absence of more-than-one-relationship restriction permits the explicit connection of a machine that *sees* another to also *see* the latter and have complete read access to it (the same with the *uses* relation).

Uses allows requirements on the state of the two related machines, while *Sees* does not. INVARIANT and PROPERTIES clauses of a machine using another can relate the elements of this machine to the elements of the former. That is the sole difference between the two structural constructs.

As these relationships concede read-only access, variables can not be referred to in the composing components, because if their state could be modified **outside** the control of the using/seeing machine, an inconsistent state could be reached away from the considerations of this machine, as it cannot make any claim about states it does not control. The static information however can be referenced without any restriction.

A major characteristic of these constructs that must be taken into account is that, as there can be no state alterations through the Sees/Uses relations, only operations that do not modify the state of the seen/used machine can be performed in the composing component — only operations that have no assignments to its variables among the operation's substitutions.

Example

To exemplify the use of decomposition in B we return to the car transporting model. Imagine that the loading of cars in the truck was made from ships carrying cars, which unloaded in some docks. Each ship has several cars and the docks control the total number of cars already unloaded to the car transporter, which must be increased any time a car is unloaded from a ship into the truck.

The machine representing the docks with the ships is depicted in Code 20. It will **include** the machine *CarTransporter* (line **2**), whose state will be directly altered in *Docks*. Also, in its **INVARIANT** clause a binding is made between the value of *unloadedCars* and the cardinality of *loadedCars* (line **6**), thence each car loaded in the truck must be acknowledged by the dock's control. An important remark is that for *Docks* the way in which the cars are loaded in the truck is not relevant, only the fact they are loaded, thus the behaviour represented in the abstract level is enough; the information on *how* the loading is made in the truck introduced at refinement level concern only the *CarTransporter* component.

```
1  MACHINE Docks
2  INCLUDES CarTransporter
3  SEES Calculator
4  SETS SHIPS
5  VARIABLES shipsWithCars, unloadedCars
6  INVARIANT shipsWithCars : SHIPS +-> POW(CARS) & unloadedCars : INT &
       unloadedCars = card(loadedCars)
7  INITIALISATION shipsWithCars := {} || unloadedCars := 0
8  OPERATIONS
9      unloadShip(ship) =
10     PRE ship : dom(shipsWithCars) & card(loadedCars) < 15
11     THEN
12         ANY car WHERE car : shipsWithCars(ship) & car /: loadedCars THEN
13             loadCar(car) ||
14             shipsWithCars(ship) := shipsWithCars(ship) - {car} ||
15             unloadedCars <-- add1(unloadedCars)
16         END
17     END
18 END
```

Code 20: Example of a B *including* machine

The operation **unloadShip** presented from line **9** to **17** takes a *ship* that has cars to be unloaded (first conjunct in line **10**) and select *any* of these cars that is not yet in the truck (line **12**), loading it to the car transporter (line **13**) and *simultaneously* unloading it from the ship (line **14**), also acknowledging the loading in the truck (line **15**). The addition in the value of *unloadedCars* is made in the **seen** machine *Calculator* (line **3**), whose <u>stateless</u> structure is shown is Code 21.

```
1  MACHINE Calculator

2  OPERATIONS

3      res <-- add1(nn) =

4      PRE nn : INT

5      THEN

6          res := nn + 1

7      END

8  END
```

Code 21: Example of a B *seen* machine
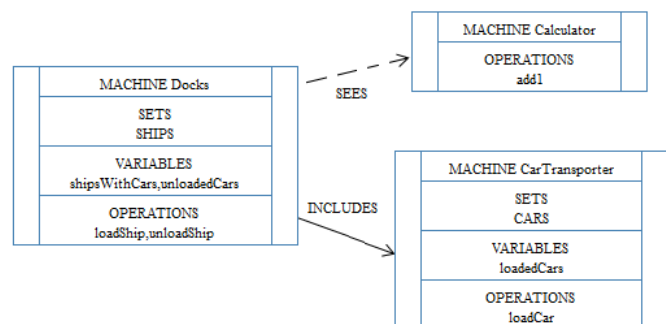
Figure 17: Architecture of a B model with decomposition

The structure of the decomposed model is depicted in Figure 17, with the includes and sees relationships between the machines.

3.3.6 Tools

A strong support of tools is provided for the B Method. We make a quick presentation of the two tools we use in our work, the AtelierB and the ProB Model Checker, but other tools are also available, like the Rodin[4] platform (for the Event-B notation), the BRILLIANT platform[5] and the BETA test generator[6].

Atelier B

The company ClearSy System Engineering[7] developed a tool for systems specification and verification through the B Method called AtelierB[8]. It provides an integrated development environment for B specifications, assisting the construction of abstract machines, refinements and specifications, project management and documentation.

A key feature of the tool is its *automatic* theorem prover, which allows the automatic verification of the proof obligations generated for a specification. If the prover cannot solve some proof then an *interactive* theorem prover can be used to perform an assisted proof over the proof obligation (if it can be done, of course.)

Currently it is a free multi-platform tool, available for Windows, Linux and OS X.

ProB

The ProB Animator and Model Checker[9] is a tool first developed for analyzing B specifications, but later extended for supporting other formalisms: Event-B, CSP-M, TLA+ and Z.

It allows one to perform *model checking* over a specification, searching for structural failures; besides, a step-by-step animation is possible, helping visual validation. The definition os LTL formulae to be checked through model checking in the model is also conceded, strengthening the verification endeavour.

Additional features are the exhibition of the specification state graphs, the presenta-

---

[4]http://www.event-b.org/install.html
[5]https://gna.org/projects/brillant/
[6]http://www.forall.ufrn.br/beta/
[7]http://www.clearsy.com
[8]http://www.atelierb.eu
[9]http://www.stups.uni-duesseldorf.de/ProB

tion of the model's architecture and the ProB Logic Calculator — a solver for predicates and expressions in the B syntax; the latter may be useful for verifying assumptions that the AtelierB prover could not handle.

The ProB tool is also free and available for Windows, Linux and OS X.

4   Method
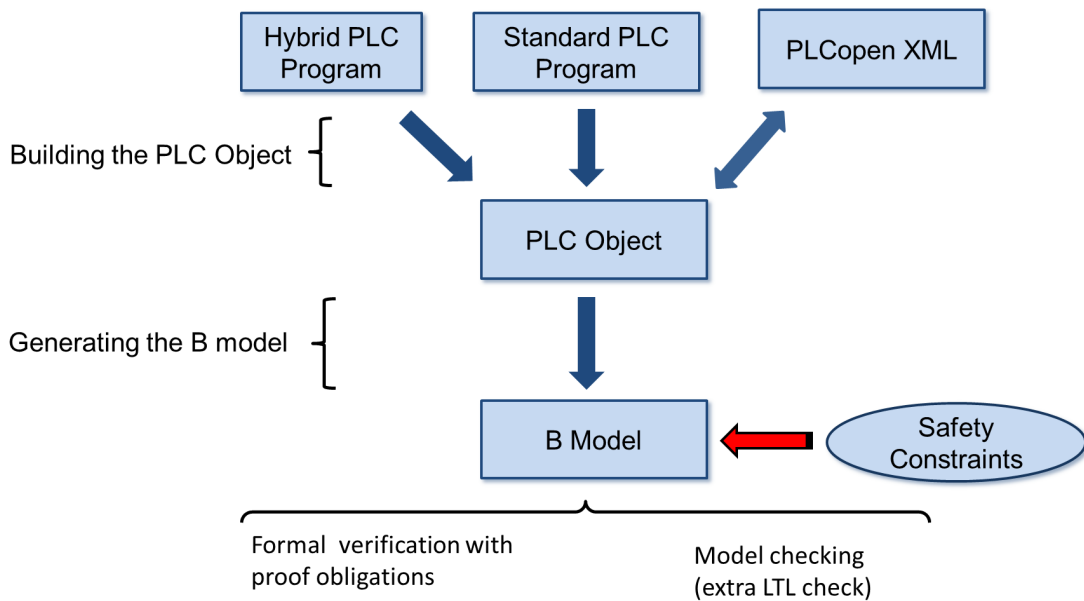
Figure 18: Illustration of the complete method.

We are proposing a method to formally verify PLC programs using the B method. This is done through the automatic generation of formal models from the POUs in the PLCs, followed by the insertion of the safety constraints in the model and its formal verification. To ease this process we make use of an intermediary structure, implemented in C++, based on the PLCopen standard, the *PLC Object*, to reduce the semantic gap between the PLC programs and the B language. Therefore, our method consists of three main phases, also illustrated in Figure 18:

1. gather the information in the PLC programs and store it in an intermediary structure, either from a standard or hybrid PLC program, or from an XML file in the PLCopen standard;

2. generate from it a B model that makes possible to check the structural and safety properties of the project;

3. and at last complete the formal model with such safety properties, derived from the project requirements (manually, for now).

As we explain our method in detail, we also show the framework we built in order to provide tool support for it. This framework embodies three main components:

- PLC READER: Module responsible for reading the information in the PLC programs and building the PLC Object. It is composed by the *IEC reader* and the *PLCopenXML reader*.

- PLC OBJECT: Structure built with the C++ language to store the information in the PLC programs as delivered by the PLC reader, standing as an intermediary object between the original programs and the formal models to be generated.

- B WRITER: Module responsible for the generation of the B models from the PLC Object.

The Figure 19 shows the structure of this framework and how the different modules interact with each other, as well as the artifacts produced by each one.
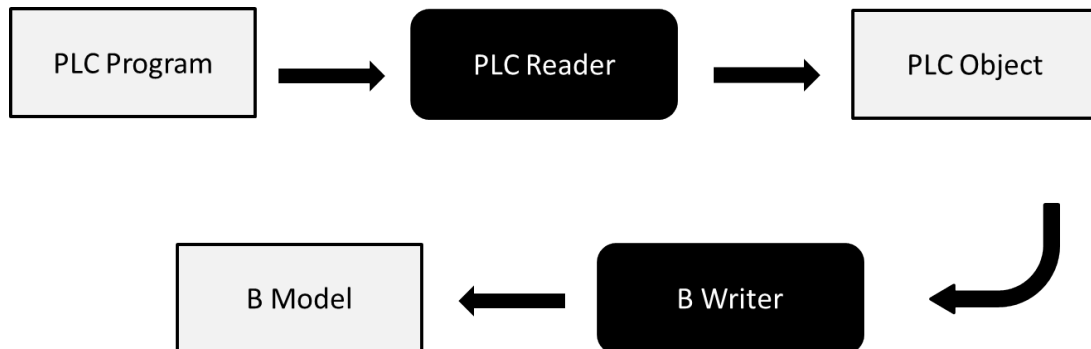
Figure 19: The framework we built to implement our method. The light gray elements represent the artifacts used and produced by the modules *PLC reader* and *B Writer*.

In the next sections we explain in detail each phase of our method, as well as the components of our framework that carry them out. Our development has covered the elements of SFC, ST, FBD and LD. The programs in IL are to be treated in future works, since their structure demands a very specific handling; nevertheless, some works, like (MEDEIROS JR, 2009), present guidelines of how to tackle this issue that we could use in our approach.

Also, some restrictions concerning the elements of the covered languages remain, such as the use of the *Struct* data type, of *loop* statements in ST or the overall absence of *time* properties in the B models. These have not been dealt with yet due to lack of time, but are to be overcome in future works, as the range of the approach is expanded.

Limitations coming from the expressiveness of the B Language itself are harder to tackle, like the impossibility to represent floating numbers as default. We either leave such issues out of our approach's reach or try to adapt the PLC program to be written without the not supported element, with no loss of generality.

In the last section of this chapter we summarize the restrictions of the approach, categorizing them by their nature.

4.1 Reading the PLC programs

The PLC Object may be built either directly from a PLCopen XML-based representation, from the textual IEC programs (ST/SFC)[1] or from programs in some hybrid textual language, not fully compliant to the IEC standard. Such languages are common, as adaptations to specific domain PLCs may be necessary.

Thus, our PLC reader embodies two components: the *PLCopen reader*, responsible for loading the PLC Object from PLCopen XML files; and the *IEC reader*, that loads the PLC Object from PLC programs in textual representation.

4.1.1 PLC Object

The intermediary structure of our approach is built based on the PLCopen standard organization, using the programming language C++ and several features of the QT framework[2].

The structure of the PLC Object consists of several classes associated with each other. The main class is *PLC*, which embodies 5 main elements:

1. Class *FileHeader*: information concerning whom is developing the project.

2. Class *ContentHeader*: information concerning the development itself of the project.

---

[1]See Appendix B for the definitions of the textual representations for SFC and POUs.
[2]http://qt.nokia.com

3. Class *Types*: Contains collections of *DataTypeElement* and *POU* objects.

  - Class *DataTypeElement*: Types defined by the user, based on the standard data types.

  - Class *POU*: Unities of PLC programs; they are the main elements of the PLC Object. Here are stored the graphic and architectural information, as well as the POUs interface and implementation.

4. Class *Instances*: Contains the configurations of the environment in which the PLC may operate (resources, tasks, access variables, etc.).

5. Besides, it has the auxiliary class *AddData*, responsible for storing additional data in the PLC Object through the class *Data*, so information not present in the IEC standard may be covered.

The most important element of *PLC* is *Types*, that encapsulates the information in the PLC programs through the data types defined, the variables and the POUs. We can see in Figure 20 an excerpt of the PLC Object class diagram, regarding the *Data Type Element* and *POU* classes. The *SFC Objects* class is shown in Appendix C, as well as the *Action Blocks* that may be associated with their *Step* objects.
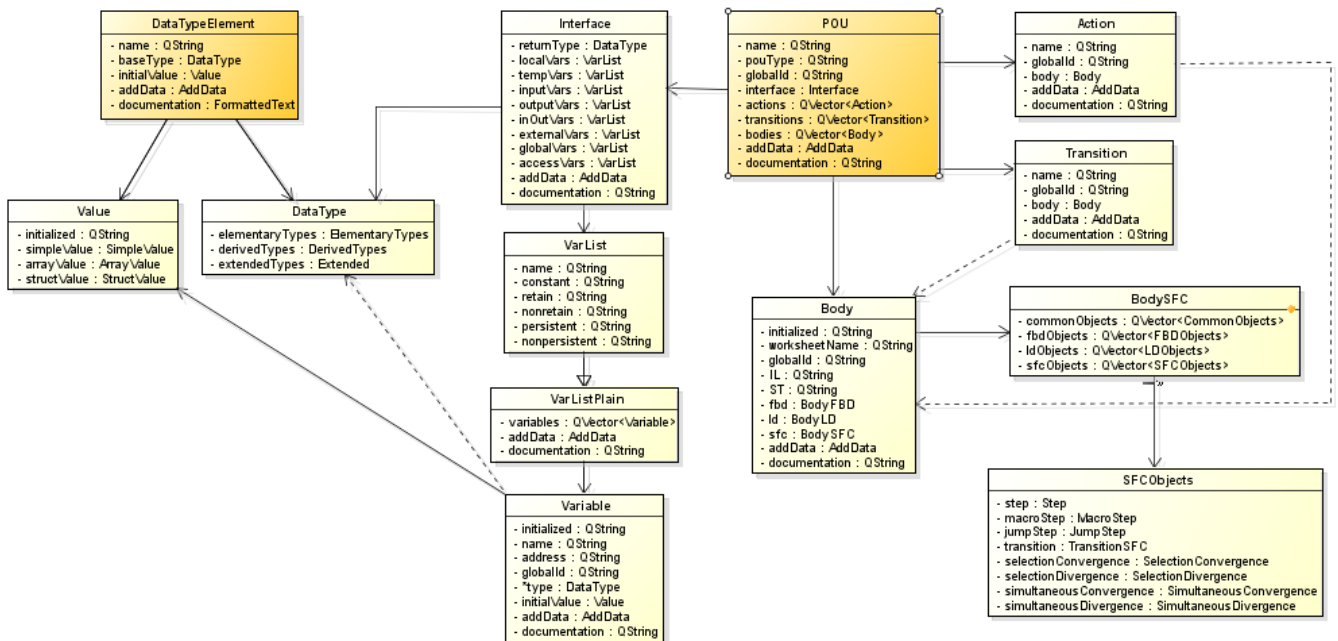
Figure 20: The main elements of *Types*

The Data Type Elements have the new types defined, so they make reference to the standard data types and the initial value that may be associated to that type.

The POUs are associated to an Interface[3], containing the variables used in it (see Section 3.1.2.2, summarizing the PLC data types). This interface may have a return type (if it is the interface of a POU *function*), so an association to DataType is present.

The VarList objects inherits from the VarListPlain objects a collection of variables, besides specifying some attributes, such as if the variables in that collection are constant, persistent, etc. Each variable, as shown in Section 3.2.2.1, describing the PLCopen Interface element, is associated to a data type and an optional initial value.

Also associated with the POU are the Body, the Actions and the Transitions. The Body is the implementation of the POU, and it can be in any of the five languages of the standard. IL and ST are represented as strings since they only have textual elements, but LD, FBD and SFC have body objects corresponding to the complex elements they may use, as shown in Section 3.2.2.2, describing the PLCopen Body element. The Action and Transition objects represent the actions and transitions in the SFC POUs, as each has an own Body association. The BodySFC object can be viewed in detail in Appendix C; it represents the same elements shown in Section 3.2.2.2.

Besides being used as an intermediary structure to the generation of the B models, the PLC Object can be used to create a PLCopen XML file, as documentation, to the PLC programs that were not in this format. The process to do so is equivalent to the one for reading the PLCopen XML files, explained in Section 4.1.3.

4.1.2 IEC reader

The PLC programs that do not have a PLCopen representation are received by our method as textual files. They will contain POU *programs*, *functions* and *function blocks* declarations, as well as new *types* definitions. The IEC Reader will build a PLC Object with its *Types* object containing all the information in these textual files — therefore, in order to analyze them, we built a parser capable of doing so.

```
1 for each PLC program textual file received:
2   call IEC parser
3   for each POU object in SFC built:
4     generate graphical information
5   build Types object with all the DataTypeElement and POU objects built
```

Code 22: Algorithm for the PLC reading

---

[3]Represents the I/O table of the POU, it does not have the meaning of an interface in an Object Oriented language like Java.

The algorithm representing the execution of the IEC reader module is presented in Code 22. For each textual file corresponding to a PLC program (line **1**), the *IEC parser* will be called to treat it (line **2**). The result from the parser performance over the file will be a series of *DataTypeElement* and *POU* objects, the latter corresponding to the POU *programs*, *function blocks* and *functions* in the PLC program. For the POUs in SFC an additional processing is executed to generate their *graphical information*, as those are not present in the textual files. We follow some proper guidelines to set the values of the common features in the graphical objects, like their **width**, **height**, **relPosition** of connection pins and the other items mentioned in Section 3.2.2.2 for the Body element in the PLCopen.

The details of the parser calling are presented in the next Section; the result of the IEC Reader module is the construction of a *Types* object with the POU and DataTypeElement objects built composing it (line **7**).

4.1.2.1   IEC parser

The parser we projected is invoked when a PLC program is received by the module. The parser deals with the elements of the standard languages and may be customized to specified differences, to accommodate new languages. This way we can deal with legacy programs that are not strictly standard compliant.

We used the tool Coco/R[4] to build this parser. Coco/R is a compiler generator, which takes an attributed grammar of a source language and generates a scanner and a parser for this language. The scanner works as a deterministic finite automaton, using recursive descent; resolution of conflicts is also available, carried out by a multi-symbol lookahead or by semantic checks.

We based the *production rules* to be used for the compiler generation on the grammars for the IEC 6113-3 languages presented in the standard (see Appendix B). The parser makes use of the scanner to do the lexical analysis of the PLC program, then performing the syntactic analysis on it, while the semantic analysis is made simultaneously, through instructions we defined in the C++ programming language. This semantic analysis is used to generate the elements of the PLC Object and, for the ST statements, make the translation to the B notation representing their elements for when the B model is generated; these translations are stored in the PLC Object using *AddData* objects.

---

[4]http://ssw.jku.at/Coco/

```
1  for each file received:
2      perform lexical analysis with Scanner
3      for each type declared:
4          build DataTypeElement object with the new type
5      for each POU function declared:
6          create POU object
7          build POU's interface
8          build POU's ST body
9      for each POU program declared:
10         create POU object
11         build POU's interface
12         if ST body then
13             build ST body
14         else
15             buildSFCBody()
16     for each POU function block:
17         create POU object
18         build POU's interface
19         for each instance:
20             build POU's instance body
```

Code 23: Algorithm for the PLC reading

Thus, the algorithm representing the execution of the IEC parser has the structure shown in Code 23. To each file received (line **1**), lexical analysis is performed (line **2**), then the *Parser* itself will process the sequence of *tokens* generated, according to the production rules defined for the SFC and ST languages, *Types* and *POUs* declarations. Each type declaration will be parsed (lines **3**-**4**) to generate a *DataTypeElement* object with the respective information: its name, the type it is based on (*baseType*) and initial value, if any. From line **5** on, the parsing process takes care of the POU declarations, building the POU objects following guidelines dependent on the POU type and implementation language: the POU *functions*, for being stateless, may have their body only in ST, since SFC, per definition, requires states to model a PLC program, while the *function blocks* and *programs* may be in ST or SFC, which requires different POU body parsing.

The POU objects are created in the same manner (differing on the *name* and *pouType* attributes, clearly) — lines **6**, **10** and **17**; a distinction will occur only for the POU *functions*, whose names will be suffixed by "_op" — this is necessary because the POU *functions* have a default return value that is not represented as a variable in its interface, which, when in the generation of the B model, will require an explicit representation, that we chose to be named after the POU, thus demanding the changing of the POU object's name to avoid naming redundancy when building the B model.

The POU objects' interfaces are the result of parsing the variables declarations for each POU, prior to its body parsing: the variables will be categorized according to their

class (input, output, etc.), with types and initial values, if any, also being used to build the *Variable* objects to be put in the lists of the POU's *Interface* object — lines **7**, **11** and **18**.

As previously said, the POU's body parsing is language dependent. If the language is ST, the procedure will consist of processing the statements (assignments, function invocations, etc.) and **generating** the equivalent B translations, organized in **sequential compositions**; this will be carried out according to the translations rules presented in the **Translation Process** Section,4.2.2, later in this chapter. A string with the resulting B statements is stored in an *AddData* object in the POU object's *Body*, while a string with the original ST statements is assigned to the *Body* itself.

Besides language issues, the *function blocks* also demand additional treatment for their instantiation feature: for each instance **declared** in the parsed POUs, a **distinct** body parsing is performed over the function block (line **20**); the necessity for this arises from the independency of state between the different instances of a POU *function block*, thus naming alterations are made in the *output* and *inOut* variables — the elements used by the POUs instantiating the function blocks: references to these variables in the resulting B translations for the POU body will be prefixed by the *instanceName*. The resulting strings with the B statements will be stored in the *AddData* object for the POU, with a *Data* object for each instance.

```
1  for each step:
2      build Step object
3      if initial step then
4        set attribute initial
5      map step's name to associated actions
6  for each action:
7      actionParsing()
8  for each transition:
9      transitionParsing()
10 build BodySFC from built objects
```

Code 24: Algorithm for the SFC Body construction

SFC parsing

The parsing of a SFC body requires more than just analyzing each statement and giving its respective B translation, since the SFC body's organization is not only a textual sequence of statements, but of several constructs corresponding to graphical elements associated to other elements. In Code 24 the guidelines for a SFC program parsing, in

order to build a *BodySFC* object, are presented.

The basic organization of a SFC body, as shown in the respective section, Section 3.1.2.8, in Chapter 3, is with *steps*, *transitions* connecting them to each other and *actions* associated with them. Thus, the construction of the POU body's object is made by the parsing of these elements: to each step declared a *Step* object is built (line **2**), with the respective name, and the actions' names declared as the ones performed when that step is activated, are acknowledged (line **5**); also, if the parsed step is declared as the initial one, the respective attribute is set in the *Step* object (line **4**).

```
1 translate action body
2 build an Action object
3 build an ActionCommonObject
4 get respective Step
5 if there is not an ActionBlock associated then
6     build ActionBlock object
7     associate ActionBlock with Step
8 else
9     get ActionBlock associated
10 put ActionCommonObject in ActionBlock
11 put translated statements in Step's AddData
12 put reference to modified variables in Step's AddData
```

Code 25: Algorithm for the Action parsing

The parsing of a SFC action declaration is carried out in Code 25. The action will consist of its name and its implementation, the action body, which in an IEC textual file can be only is ST. Once the ST statements are translated to B notation (line **1**), an *Action* object, named after the SFC action, is built with the original statements (line **2**); this *Action* is referenced in the *ActionCommonObject*, graphical element, built (line **3**) to be put into the Step's *ActionBlock* (line **10**). If the *Step* does not have an *ActionBlock* associated to it, one is created to that purpose (lines **6-7**), otherwise it is simply retrieved (line **9**). The B statements resulting from the ST translation are stored in the *Step*'s *AddData* (line **11**), as well as the variables that are **modified** in that action (line **12**), a relevant information when the generation of the B model is carried out.

For the SFC transitions the parsing process, whose structure can be seen in Code 26, is mostly concerned with the connections made with the steps, both *from* and *to*. Firstly, a *TransitionSFC* object, representing the graphical element in the SFC language for a transition, is built (line **1**), with its name being generated as "T" + *numberTransitionsAlreadyRead + 1*. If the transition is **from** more than one step, a *SimultaneousConvergence* object is built to represent the convergence of several paths into a transition (line **4**), while

```
1  build TransitionSFC object
2  if transition has more than one source step then
3    build SimultaneousConvergence object
4    associate TransitionSFC object to SimultaneousConvergence object
5    associate SimultaneousConvergence object with all source Step objects
6  else
7    associate TransitionSFC object with the source step
8  if transition has more than one destination step then
9    build SimultaneousDivergence object
10   associate SimultaneousDivergence object to TransitionSFC object
11   associate destination Step objects with SimultaneousDivergence object
12 else
13   associate destination Step object with TransitionSFC object
14   put translated condition in destination Step's AddData
15 for each destination Step object built:
16     put translated condition in Steps's AddData
17     put transition's source steps names in Step's AddData
18 build Transition object
19 associate TransitionSFC's Condition object with Transition object
```

Code 26: Algorithm for the Transition parsing

if it is **to** more than one step a *SimultaneousDivergence* object is built to represent the divergence of one path into several ones after the transition (line **10**); in either case the object created is associated with the *Step* objects, for each source step or for each destination step, respectively, as well as with the *TransitionSFC* object for the transition, being its source (line **5**) or its destination (line **12**); when there is only one source or destination step the association is done directly between the respective *Step* and *TransitionSFC* objects (lines **8** and **15**). Finally, once the structure of the step/transition relations is defined, the transition's logic condition is parsed, with the original ST statements being put in the *Transition* object, and the B translations, as well as the transition's source steps names, are put in the *AddData* object of the respective *Step* object for each destination step (lines **15**-**17**).

At last, after the performance of these procedures, a *BodySFC* object is built with all the *Step*, *ActionBlock*, *TransitionSFC*, *SimultaneousConvergence* and *SimultaneousDivergence* objects built during the parsing of the SFC elements (line **12** of Code 24).

4.1.2.2   PLC Example

The complete textual representation of the SFC program shown as an example for SFC, in Figure 16, at page 50, with only the partial textual representation presented in Code 3, at the same page, now as a POU *program*, can be seen in Code 29. The POU *function* declaration for **isHigher**, used in *sfcSimple*, is shown in Code 28, while the new

types declared for use in the program, *RESULT* and *ANSWER*, are presented in Code 27.

```
1 TYPE
2   RESULT : (YES, NO, UNDETERMINED);
3   ANSWER : ARRAY [1..2] OF RESULT;
4 END_TYPE
```

Code 27: Textual representation of some Data Type Elements declaration.

```
1 FUNCTION isHigher : BOOL
2   VAR_INPUT
3     n1 : INT;
4     n2 : INT;
5   END_VAR
6
7   isHigher := n1 > n2;
8 END_FUNCTION
```

Code 28: Textual representation of a POU *function* in ST declaration

```
1 PROGRAM sfcSimple                      16   TRANSITION FROM Start TO Step1
2   VAR_INPUT                            17     := in1;
3     n1 : INT;                          18   END_TRANSITION
4     n2 : INT;                          19
5     in1 : BOOL;                        20   STEP Step1:
6   END_VAR                              21     Action1(N);
7   VAR                                  22   END_STEP
8     answers : ANSWER := [YES,          23
9         NO];                           24   ACTION Action1 :
9   END_VAR                              25     IF isHigher(n1, n2) THEN
10  VAR_OUTPUT                           26       higher := answers[1];
11    higher : RESULT :=                 27     ELSE
12        UNDETERMINED;                  28       higher := answers[2];
12  END_VAR                              29     END_IF;
13                                       30   END_ACTION
14  INITIAL_STEP Start:                  31
15  END_STEP                             32 END_PROGRAM
```

Code 29: Textual representation of a POU *program* implemented with SFC + ST.

The **TYPE** clause embodies the new types declaration, defining the type name and the base type it is derived from. The definition of *RESULT* (line **2**) is made from the **Enumerated** type, thus its declaration consists of specifying the possible values an enumeration *RESULT* may assume: in this case either "YES", "NO" or "UNDETERMINED" (line **2**). The same principle goes for the declaration of *ANSWER*, derived from the **Array** type, where an unidimensional array with *basetype* "RESULT" and *dimension* "1..2" is declared (line **2**).

For the POU *program*, in Code 29, the variables declaration is made from lines **2** to **12**, with the inputs inside the **VAR_INPUT** clause (line **2**), the locals inside the **VAR** clause (**7**) and the outputs in **VAR_OUTPUT** clause (line **10**); each of the variables is identified by its name and type, with the local and output also having an initial value

assigned (lines **8** and **11**). The parsing of these declarations will be responsible for the *POU* object's *Interface* building, putting the *Variable* objects in their respective lists.

The two steps in the SFC program are declared using the **INITIAL_STEP** and **STEP** clauses, the former for the initial step (line **14**) and the latter for the normal step (line **20**); the body of the declaration is composed by the names of the actions associated with the respective step, thus *Start* has none and *Step1* has *Action1*, declared also with the *qualifier* definition — *N* (line **21**). The transition that connects both steps is declared in clause **TRANSITION**, with **FROM** specifying the source steps (*Start*) and **TO** the destination steps (*Step1*), which can be seen in line **16**; the condition of the transition is presented in line **17**, indicating that the execution flow may go from *Start* to *Step1* when the former is activated, its actions (none) performed and *in1* is equal to **TRUE**.

The action associated with *Step1* is declared in the **ACTION** clause (line **24**), identifying the name of the action and its ST code (lines **25** to **29**), which makes use, as the transition's condition, of the variables in the interface of the POU to execute its behaviour, defining the value of the *higher* output from the value at position 1 (line **26**) or 2 (line **28**) in the *answer* local variable, depending on the result of the *isHigher* function performance over the *n1* and *n2* inputs (line **25**).

The declaration of *isHigher* in Code 28 is made just as for *SFCSimple*, although it is declared as **FUNCTION** instead of **PROGRAM** (line **1**) and its body being composed solely of ST statements — it tests whether the *n1* input is higher than the *n2* input, storing the result in the POU *function*'s return value, named after the POU itself (line **7**).

PLC Object generated

The execution of the IEC reader over these textual files will produce the *Types* object seen in Figure 21[5], with its associated *POU* and *DataTypeElement* objects. The POU objects built are for *sfcSimple* and *isHigher*, the former with its *pouType* attribute set as "`program`" and the latter as "`function`"; for the DataTypeElement objects they will correspond to the new types declared, thus for *ANSWER* and *RESULT*.

The *POU* object built for *sfcSimple* can be viewed in detail in Figure 22. The original statements in action *Action1* and in transition *T1* are stored in the *Action* and *Transition*

---

[5]The objects are shown with most of their attributes, like the graphical information, hidden, for clarity.
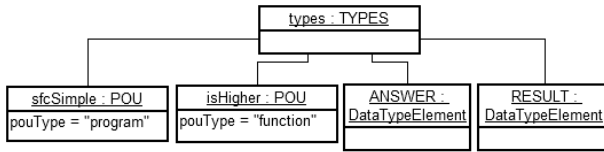
Figure 21: Types object built.
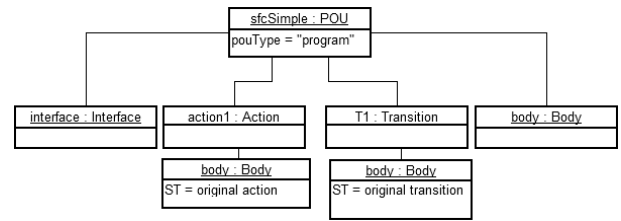
Figure 22: POU object built.
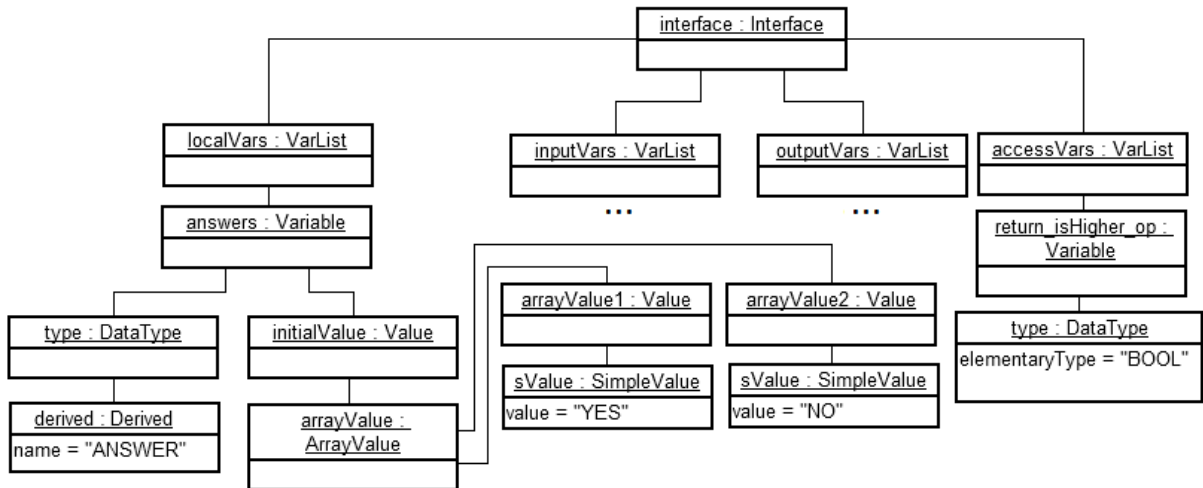
Figure 23: Interface of the POU *sfcSimple.*

objects part of the POU, as the result of the operations performed in the Codes 25 and 26, for action and transition parsing, respectively.

The *Interface* object built for *sfcSimple* is depicted in Figure 23, with each *VarList* object representing a category of variables declared in Code 29, for the *sfcSimple* POU *program*: locals, inputs and outputs; the access variables represent the ones built for the return values of the POU *function* invocations, a matter that will be discussed when we present the guidelines for the B models generation, in Section 4.2.3.

Only the local and access variables are shown in detail in the *Interface* object, for clarity. The local variable *answers* is typed with the new type *ANSWER* defined, so its type object is defined as a *Derived* object, which will make reference to the *DataTypeElement* object built for the new type — this *DataTypeElement* object for *ANSWER* is depicted in Figure 24, where its base type (array) is instantiated. The initial value of *answers* is an *ArrayValue* object, as they need to be represented as a list of values corresponding to the value in each position of the array, being every one a part of the *RESULT* enumeration, the base type of *ANSWERS*. For the `return_isHigher_op` variable the configuration is much simpler, as it has no initial value and is of an elementary type, defined as a single string — *BOOL*.
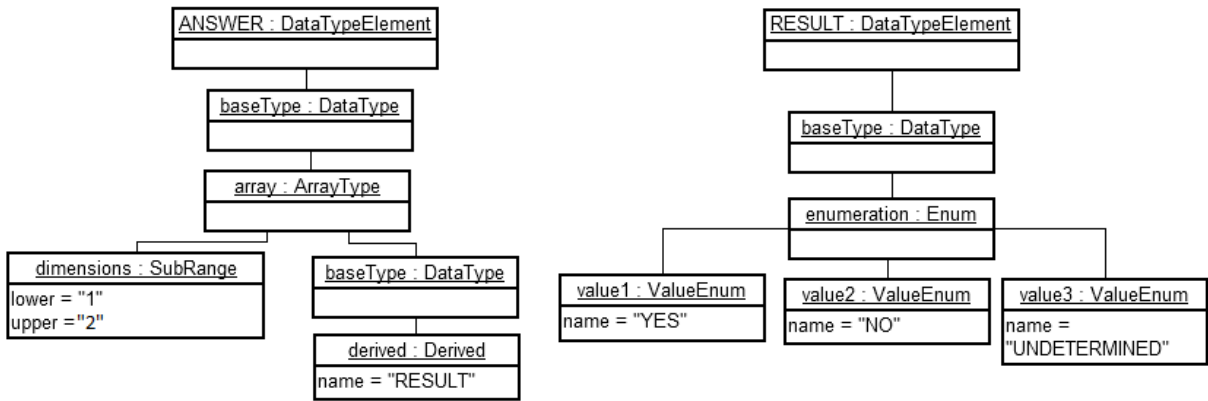
Figure 24: DataTypeElements built.

The *Body* object built for *sfcSimple* is exhibited in detail in Figure 25. It is generated from the SFC parsing procedure, as shown in Code 24, resulting in a *BodySFC* object with *Step*, *TransitionSFC* and *ActionBlock* objects. The components of a *Step* object for *Step1* are presented in detail, depicting its *ConnectionPointIn*, whose *Connection* is with the element in the *Body* that possess *localId* 2 — the *TransitionSFC* object for *t1*; it can also be seen that in the connection's *AddData* object are stored, in the *Source_Transitions Data* object, at the *transition1* element, the B translation for its predecessor transition's condition and the source steps from this transition, consequently *Step1* own source steps — *Start*. The *AddData* object of the *Step1* object itself, in the *Actions Data* object, specifically in the *action1* element, has the B translation for *Action1*'s ST statements.
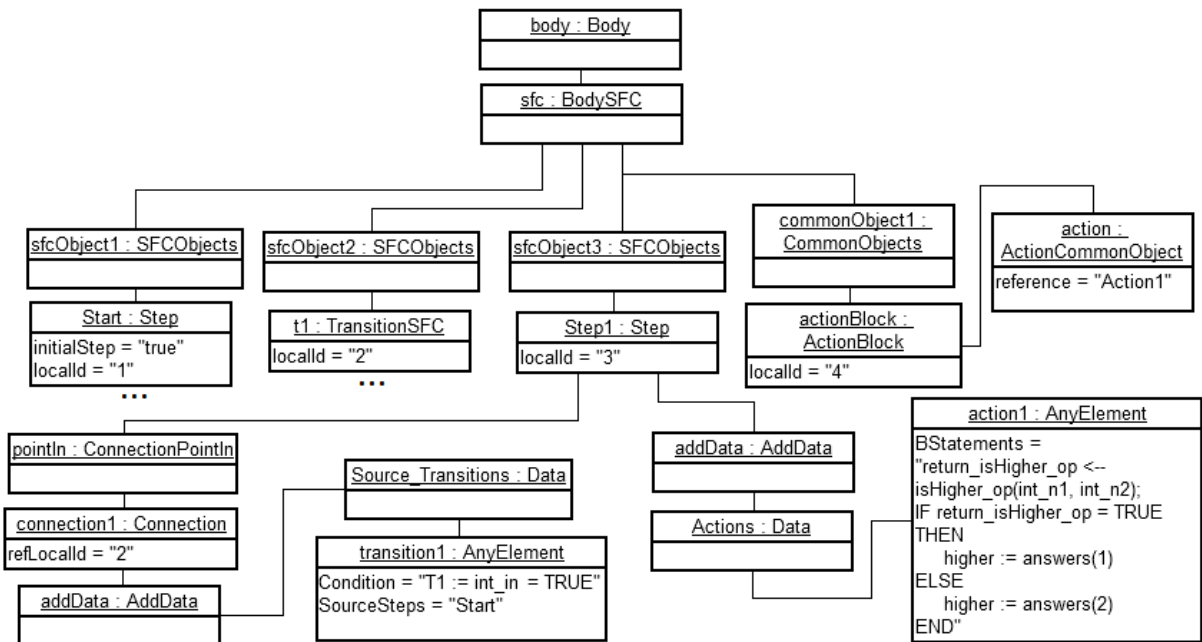
Figure 25: Body of "sfcSimple".

4.1.3 PLCopen Reader

The other option our method supports is to build the PLC Object from the PLC programs in the PLCopen XML format. As the organization of our intermediary structure is entirely based on PLCopen, the process is pretty much straightforward, once we built an XML reader using the features of the Qt Framework.

That easiness, however, is only for the elements of the graphical languages — LD, FBD and SFC; when dealing with ST we need to use a parser to process the code and generate the translated statements to be used in the B models, as well as determine auxiliary things that have to be present in the PLC Object, such as, when inside of a SFC action, its modified variables.

We make use of the parser presented in the previous section, but only as an auxiliar service for the XML reader, for processing the ST statements present in the **Body** elements of PLCopen.

For LD and FBD the parser will produce *POU* objects in the same manner as for SFC, only that their *Body* objects will be composed of *BodyLD* and *BodyFBD* objects, respectively, the former with numerous *LDObjects* and the latter with *FBDObjects*. An excerpt of the PLC Object class diagram for LD and FBD can be seen in Figure 26, while the several objects they can assume are presented in Appendix C.
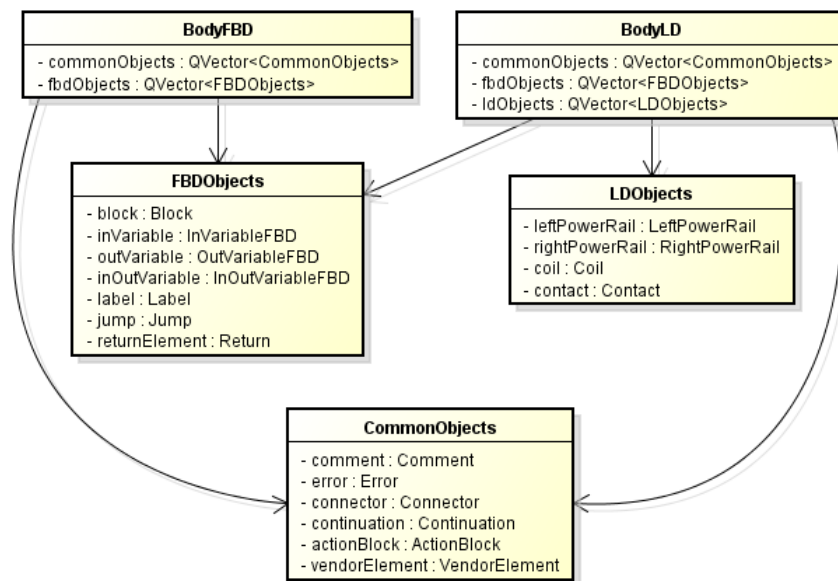
Figure 26: BodyLD and BodyLD

4.2 Generating the B Model

Once the PLC Object is constructed, our method is able to work independently from the PLC programs to generate the corresponding B models. This section explains the architecture of these models and the translation guidelines for the PLC languages; then it presents the B Writer module, which implements the generation procedure.

The result of the generation process is not a *direct formal modeling of the control algorithm*, it is rather built from the result of a *reinterpretation* procedure from the already implemented PLC code — the PLC Object (FREY and LITZ, 2000).

4.2.1 Architecture of the B model

A good architecture is essential to generate a good model, as well as to define which information from the PLC Object will be responsible for which elements of the B model, so we can keep **traceability** between the PLC programs and the corresponding B specifications.
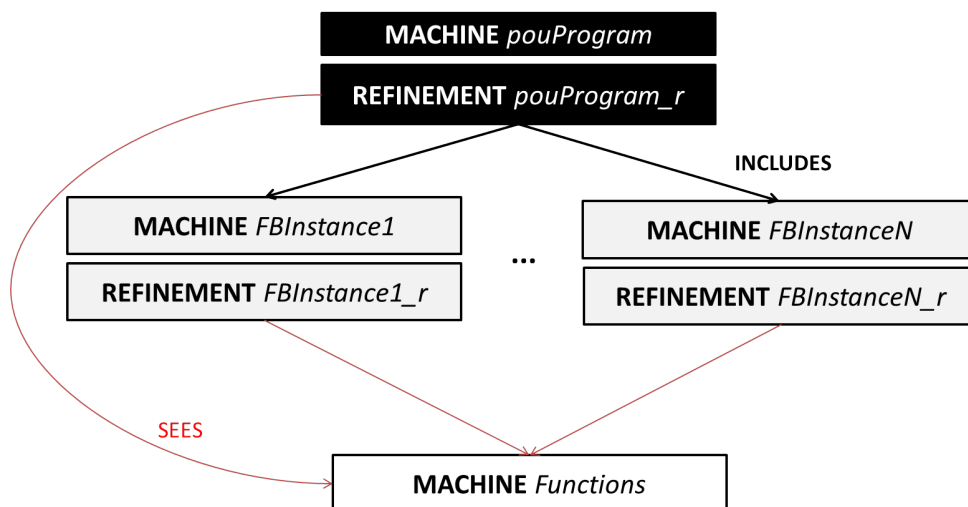
Figure 27: B model representing a POU *program* and its use of auxiliary *function* and *function block* POUs.

The architecture of this model is depicted in Figure 27. The PLC program is represented by a series of instances of this model, each one corresponding to a POU *program* and the POU *functions* and *function block* instances it may use, with every POU being modeled as a B **machine** and **refinement**, aside from the POU *functions*.

The POU *programs* and *function block* instances are modeled as a **machine + refinement** due to the gain in expressivity obtained at Refinement level, which eases the

representation of the PLC language elements in the B notation, such as the possibility of **sequential** compositions. Therefore, in the *abstract level* we only indicate which part of the machine's state will be modified in each **operation**[6], considering *any* behaviour possible, while the *actual* behaviour, derived from the POU bodies' translations, being considered solely in the *refinement level*, as the only subset of the possible behaviours established at machine level that is accounted for. Thus, the **safety constraints** will be taken into account only on this level, inserted as **invariants** in the refinements — this way the abstract level, as previously said, just indicates the elements of the problem's state space, not being considered in the verification process, which only takes place over the refinements representing the POUs' behaviours.

The decomposition in the representation of the PLC program through POUs, besides splitting the complexity for performing the intended behaviour also splits the *verification effort* to attest its correctness. Since the invariants are considered only at the refinement level, the components may be verified *independently* from the other components that access them. The **INCLUDES** relation between the POU *program* and its *function block* instances allows the free use of its state — mandatory due to the PLC semantics itself —, but as these are accessed only at the *abstract* level, where we do not exhibit the POU's behaviour, every restriction about the *correctness* of this behaviour is passed to the *function block* component — in the POU *program* the concern is that it behaves correctly whatever is the result of the *function blocks*' execution, exactly what is obtained by only considering their *abstract* state.

The POU *functions*, for being <u>stateless</u>, makes purposeless to put any constraint over its behaviour by means of **invariants** over its state, simply because they do not have any. By the same principle refinements are innocuous for them, as there will be no state to refine. This leads to the modeling of each POU *function* as a B **operation**, all collected in the B machine "Functions", where no state is kept, so these operations must make use only of the values they receive and the ones they produce (*inputs* and *outputs*); this restriction, as well as having only the *abstract* level constructs to work with, forces the use of some workarounds, to be presented in the next section, in the B language to overcome the issue of representing the POU bodies, but allows the use of the **SEES** structure when relating the "Functions" machine to the numerous other components of the model; another issue is that as no restriction over the behaviour of the POU *functions* is possible in the "Functions" machine, these must be carried out in the component invoking the

---

[6]This is represented by *nondeterministic* assignments of any possible value, given its type, to the variables.

operations, be it a POU *program* or a *function block* instance, through the results obtained after the invoked operation's execution with the given inputs.

The POU *function blocks*, for being <u>stateful</u>, are represented with a separated component for each instance used in the POU *program*, so their states may be represented independently and kept between invocations. Another feature of our B model architecture, not depicted in Figure 27, is that *function block* instances may use other *function block* instances themselves, so they may **include** the machine part of these components as well.

Once the overall architecture of the B model is known we concentrate on the internal structure of the individual components.

4.2.2 Translation Process

As important as the architecture of the model is a good set of guidelines for the translations of the PLC language elements into the B notation, so the semantics of the PLC programs may be preserved when their B representations is generated, as well as keeping the **traceability** between their elements.

The PLC language elements are the components of the POU bodies composing a PLC program, thus the following guidelines approach how the *Body* of a POU is symbolized in the B machines, which is through B **operations**, with their *preconditions* and *substitutions*.

4.2.2.1 ST

We present below the guidelines for the translation of ST code into B notation. The general aspects of this translation will be shown, not in excessive detail, for the sake of clarity.

Expressions

One of the most important steps is regarding the ST expressions, as they are used in many of the other constructs of the language. They cover the ST operations shown in Table 17, in Appendix A.

The translation of the symbols in these operations is summarized in Table 11. Since

the definition is recursive, special care is taken during the translation process so the precedence order is respected, through the insertion of parenthesis.

ST Operation	Structure	B notation
Parenthesization	(expression)	(expression)
Exponentiation	expression ** expression	expression ** expression
Multiply	expression * expression	expression * expression
Divide	expression / expression	expression / expression
Modulo	expression **MOD** expression	expression **mod** expression
Add	expression + expression	expression + expression
Subtract	expression - expression	expression - expression
Complement	**NOT** expression	**not**(expression)
Comparison	expression ($<$ | $>$ | $<=$ | $>=$) expression	expression ($<$ | $>$ | $<=$ | $>=$) expression
Equality	expression $=$ expression	expression $=$ expression
Inequality	expression $<>$ expression	expression $/=$ expression
AND	expression (**&** | **AND**) expression	expression **&** expression
OR	expression **OR** expression	expression **or** expression
Exclusive OR	exp1 **XOR** exp2	(exp1 **or not**(exp2)) **&** (**not**(exp1) **or** exp2)

Table 11: ST operations and their equivalent B notations

A remark must be made regarding the logic expressions (involving the boolean and comparison operators): in the B language, when a boolean variable is present in an expression its value cannot be directly accessed — it needs to be inside a predicate, so one needs to compare it to some other boolean value for the expression to be valid; in order to obtain the desired execution a comparison to **TRUE** is made, preserving the expression semantics. For instance,

$$boolVar1 \textbf{ AND } boolVar2$$

must be translated, for a semantic equivalent, to

(boolVar1 = **TRUE) & (**boolVar2 = **TRUE)**

Function block invocation

Other significant constructs in ST are the *function* and *function block* instance invocations. The treatment to the latter is simpler, as only a B operation, with no outputs,

is called, representing a reference to the block's body. The name of the B operation will correspond to the *instance* and depend on the language the function block body is implemented. As in the function block instances the translated statements are considered only at refinement level, the operation to be called in the abstract level is only the one receiving the parameters, which will be, for **FBD** and **ST**, *instanceName* plus the suffix "_body", since the programs for these languages are translated to a single B operation. However, since the **SFC** and **LD** programs are translated through one or more B operations, the function block invocation is made through the operation corresponding to the *initial step* or the *first rung*, respectively, where the parameters are read for the execution through the other operations — thus, the operation called is *instanceName* plus the suffix "_ + *initialStepName*" or "_rung1"[7].

```
1 fbInstance_identifier (argument list);
2 variable := fbInstance_identifeir.output;
```

Regarding the use of the resulting outputs from the function block execution, it is made through a ST assignment, as seen in line **2** of the ST structure for function block invocation; the B equivalent is almost the same, differing only on the separator from the fb instance name and the output name — instead of a ".", a "_".

An example of the translation of a ST *function block* invocation is presented in Code 30, with the above code being the one in the PLC language, while the one below is its respective B translation. An instance, "test_MEmg", of a *function block* who possesses four inputs is invoked and then has its output "authorize" assigned to a variable — the B translation is carried out and the function block instance is invoked with its body **operation** receiving the inputs, and its output, whose name is prefixed by the instance name, being assigned.

```
1 test_MEmg(stopped := speedZero, in_platform := platformSafe,
      doors_closed := TL_DOORS_CL, doors_locked := TL_DOORS_LCK);
2 TL_AUTO_EMG := test_MEmg.authorize;
```

```
1 test_MEmg_body(speedZero, platformSafe, TL_DOORS_CL, TL_DOORS_LCK);
2 TL_AUTO_EMG := test_MEmg_authorize;
```

Code 30: B generation from ST *function block* invocation

---

[7]The process of building the machines and refinements, for each POU and according to the language its body is in, is shown in detail in next section.

Function invocation

As for the *function* invocations, there are some issues to be taken care of in order to build the semantically equivalent B statement. Below the different cases where a function invocation may occur in ST are presented:

```
1 variable := function_identfier(argument list);
2 variable := expression + function_identifier + expression;
```

If a function is being invoked as in line **1**, no special treatment other than the direct translation is necessary — simply replace the ":=" symbol for "<--", which is how B marks the calling of an operation from another scope, and add the suffix "_op" to the function's name; however, if the calling is made as in line **2**, with the function invocation as part of a bigger expression, a workaround is necessary to keep the statement semantics, once in B a function can not be called as part of an expression. The process is detailed in Code 31.

```
1 calculate n: number of function invocations in these conditions
2 create returnVariable: ''return_'' + function operation name + n
3 build statement  returnVariable <-- functionInvocation (argument list)
4 for all the occurrences of that functionInvocation in the expression:
5     replace the invocation for the returnVariable
```

Code 31: B generation from ST function invocation

The other technicality is regarding the *inOut* variables that may be passed to the function. As in B there is no concept of passing arguments by reference, the variable passed as an *inOut* parameter in ST will need to be both *input* (passed as parameter) and *output* (receives return value) in the B function invocation: it is added to the list of variables receiving the returned values of the B operation invocation.

An example of the translation of a ST function is presented in Code 32. In the ST code firstly the function is being invoked alone, whose translation is pretty simple; then it is invoked as part of an expression with another elements, which requires the execution of the procedure in Code 31 for the generation of the respective B translation.

The case where an inOut variable is involved is also demonstrated: the *function* doublesHigherThanTen takes a numeric value that, if is higher than **10**, is dou-

bled, in which case its default return value is assigned as **TRUE**, otherwise being **FALSE**. The function invocation in the ST code assigns the returned value to a variable "`successfullyExecuted`" and "`n3`" is sent as an inOut parameter. The respective B translation adds `n3` as a return value for the operation invocation, clearing the issue.

```
1 n3 := getHigher(n1, n2);
2 n3 := n1 + getHigher(n1, n2);
3 successfullyExecuted := doublesHigherTen(n3);
```

```
1 n3 <-- getHigher_op(n1, n2);
2 return_getHigher_op1 <-- getHigher_op(n1, n2);
3 n3 := n1 + return_getHigher_op1
4 successfullyExecuted, n3 <-- doublesHigherThanTen_op(n3);
```

Code 32: B generation from ST *function block* invocation

Assignments

Besides the internal structure of an expression, another important factor is *how* it is used in the ST code, which may demand special treatment when the B translation is performed. In the case of the a ST assignment, whose structure is

```
variable := expression
```

the resulting B statement will be among four different outcomes, depending on three conditions over the expression being assigned:

- it is a logic expression (*isLogicExpression*).

- it is **solely** a function invocation (*isSolelyAFunctionInvocation*).

- it is an expression with a function invocation as one of its operands (*hasInExpressionFunction*). This one is mutually exclusive with the previous condition.

The possible outcomes will be determined according to the combinations of these conditions, following the algorithm in Code 33. The **outcome 1** represents the situation when none of the previous conditions was the case, so the process goes straightforwardly. If the *isSolelyAFunctionInvocation* condition was satisfied, given the other two were not, the B assignment, when a function is being invoked, requires the use of the "<--" symbol

```
1  if (!isLogicExpression & !hasInExpressionFunction) then
2      if !isSolelyAFunctionInvocation then
3          translateTo variable := expression (outcome 1)
4      else
5          translateTo variable <-- functionInvocation (outcome 2)
6  else if hasInExpressionFunction then
7      create auxiliar function assingment statement
8      if !isLogicExpression then
9          translateTo variable := expression (outcome 3)
10     else
11         translateTo variable := bool(expression) (outcome 4)
12 else
13     translateTo variable := bool(expression) (outcome 4)
```

Code 33: B generation from ST assignments

instead of "`:=`"; otherwise the translation has no other peculiarities. In lines **6**-**11** the situations where *hasInExpressionFunction* is valid are treated; firstly, a B statement is created, previously to the one being translated, for the assignment of the function result to the respective auxiliar return variable (line **7**), then the main assignment, translation of the expression with the function invocation replaced by the respective auxiliar return variable, is built depending on *isLogicExpression*: if it is false, **outcome 3**, otherwise **outcome 4**, where the expression is put inside the predicate **bool()**, once the expression's boolean result can only be assigned if inside a predicate. In line **13** the case in which *isLogicExpression* is valid and there is no function invocation is treated, also as **outcome 4**.

An example of a series of ST assignments translation is presented in Code 34. In line **1** of the ST code is the assignment that will result in **outcome 1** for the B translation, the assignment with no special conditions, whose straightforward translation is in line **1** at the B notation part. In the next line the exhibited case is for **outcome 2**, when the assignment is solely of the function invocation's result, whose B translation has the changing of the assignment symbol for the function invocation one, besides the presence of the `_op` suffix in the function's name. The case where a function is invoked as part of a *not* logic expression (assignment for `numVar3`) fits into the **outcome 3** scenario, where the creation of an assignment of the function invocation for a temporary return variable (line **3** at the B notation part) which is then used instead of the invocation in the original expression translation (line **4** at the B notation part). It is a very similar scenario as the ones in lines **4**-**5** in the ST code, where **outcome 4** is reached, differing from the previous case for being a logic expression, resulting in the B expressions inside the **bool()** predicate shown in lines **5**-**8** at the B notation part. **Outcome 4** is also applied

for the last assignment in the ST code, only that no processing for a function invocation is necessary.

```
1 numVar1 := n1;
2 numVar2 := getHigher(n1, n2);
3 numVar3 := n1 + getHigher(n1, n2);
4 boolVar1 := TRUE AND isHigher(n1, n2);
5 boolVar2 := boolVar1 OR isHigher(n1, n2);
6 boolVar3 := boolVar1 XOR boolVar2;
```

```
1 numVar1 := n1;
2 numVar2 <-- getHigher_op(n1, n2);
3 return_getHigher_op1 <-- getHigher_op(n1, n2);
4 numVar3 := n1 + return_getHigher_op1;
5 return_isHigher_op1 <-- isHigher_op(n1, n2);
6 boolVar1 := bool((TRUE = TRUE) & (return_isHigher_op1 = TRUE));
7 return_isHigher_op2 <-- isHigher_op(n1, n2);
8 boolVar2 := bool((boolVar1 = TRUE) or (return_isHigher_op2 = TRUE));
9 boolVar3 := bool( ((boolVar1 = TRUE) & not(boolVar2 = TRUE))
      or (not(boolVar1 = TRUE) & (boolVar2 = TRUE)) )
```

Code 34: B generation from ST assignments

IF

Given the previous definitions, the translation of the ST **IF** statement goes almost directly, as depicted in Code 35, only with the **END_IF** token being translated as **END**, besides the expressions, function invocations and assignments in the statements and in the **IF** conditions.

Example

We present a small example that covers part of the features presented in this section, with a piece of ST code and the respective B translation that our method generates, following these guidelines.

The line **1** of the ST code presents a function invocation that fits into the *inExpres-*

```
IF logicExpression THEN
    statements
{ELSIF logicExpression THEN
    statements
}
[ELSE
    statements
]
END_IF
```

```
IF bool(logicExpression) THEN
    statements translated
{ELSIF bool(logicExpression)
   THEN
    statements translated
}
[ELSE
    statements translated
]
END
```

Code 35: B generation from ST 'IF' statements

```
1 IF NOT isHigher(n1, n2) AND
      performOperation THEN
2    n1 := n1 + n2;
3 ELSE
4    n1 := n1 - n2;
5 END_IF
```

```
1 VAR return_isHigher_op1 IN
2    return_isHigher_op1 <--
        isHigher_op(n1, n2);
3    IF not(return_isHigher_op1 = TRUE)
       & (performOperation = TRUE)
       THEN
4       n1 := n1 + n2
5    ELSE
6       n1 := n1 - n2
7    END
8 END
```

Code 36: B generation from ST example

sionFunction case, so the presented strategy must be applied: the creation of a transient variable to receive the value of the function call and be used in the respective expression (outcome 3, following Code 33), resulting in lines **1**-**3** of the B code, with the transient variables being defined in a **VAR** clause for being used in that scope. It is interesting to notice that although the function is being invoked amidst an **AND** expression this does not satisfy *isLogicExpression*: that is so because the **IF** statement can access the boolean value of an expression without the need of the **bool()** predicate; nonetheless, the boolean variables still need to be inside a predicate(comparison with `TRUE`). The expressions are translated according to Table 11, straightforwardly.

4.2.2.2 FBD

FBD programs

To translate the FBD programs to B notation we oriented the construction of B expressions based on the organization of the elements of the program: how its *blocks*, *inVariables*, *inOutVariables* and *outVariables* are connected, interacting with each other.

In Code 37 the guidelines for this translation are shown.

```
1  for each out variable:
2      determine connected element acoording to CPointIn
3      if connnected to block:
4          build block invocation
5          if stateless block
6              build assignment: output ':=' returnVariable
7          else
8              build assignment: output ':=' fbInstance_out
9      else
10          build assignment: output ':=' variable
11 (*---------------- Building block invocation ------------*)
12 determine arguments according to the CPointIns of the block
13 for each connected block:
14     build connected block invocation
15 if staless block then
16     build function invocation statement
17 if stateful block then
18     build fb instance invocation
```

Code 37: B generation from a FBD body

As stated in line **1**, the process is for each *outVariable* (or *inOutVariable*) in the FBD program. Firstly, from the **ConnectionPointIn** in the variable element, the reference to whom it is connected is obtained (line **2**). If this element is a block then the invocation of its corresponding B operation will be necessary — the building process will correspond to the ST functions or function block invocations in the previous section (lines **16** and **18**. This procedure is recursive: each input of a block that is connected to another block will demand the construction of the invocation statement of this block (lines **12-14**). The order for building the invocation statements is from the higher to the lower pin in the block, unless the *execution order* attribute in any of the blocks is defined, in which case the blocks with higher precedence have their invocation made *before* the ones with lower precedence[8]. If the block's input is connected to a variable then it is directly passed as a parameter to its argument list. Once the chain of blocks (if any) connected to the *outVariable* in question is treated, its assignment may be constructed; if there was no block connected to it, then it had to be another variable (line **10**); if it was a block then the assignment will be either of the return variable of the corresponding function (line **6**) or from the respective output in the corresponding function block instante (line **8**).

---

[8]This is a solution for cases such as a connected block having an output as input for another connected block.

Standard blocks

Some blocks are direct representations of expressions supported in ST, in which case there is already an equivalent B translation and the building of a "B function" is not necessary to represent it. So, when these blocks are faced in the translation of a FBD program, the corresponding B expression is used instead of a function invocation.

The blocks with correspondent B operations are:

- **Arithmetic**: Adding, multiplying, etc.

- **Bitwise**: AND, OR, etc.

- **Comparison**: greater than, equal, lesser, etc.

There are also some standard function blocks defined in PLCopen that may be used in the PLC programs. A library with these blocks and their respective translation is available for use. An example is the *Set_Reset Flip Flop* function block, the **SR** block. It represents an operation where a bit may be set or reset according to the inputs received: the *state* of the block corresponds to the memory address of the bit being set or reset, so several instantiations of the block may cover several bits in different memory addresses.

Example

To exemplify our translation procedure for FBD programs we use an example similar to the one presented in the previous Chapter, in Figure 7, at page 43, only that now different blocks are used, representing POU *functions*, not standard blocks (such as were the **AND** and **OR** blocks); the SR function block is maintained. The program we translate is depicted in Figure 28.

The value of *OUT*, an *outVariable*, is determined by the output *Q1* of the *SR1* instance for the *SR* function block, whose set and reset inputs will be, respectively, from: the equality comparison of the *inputVariable IN1* with the result of the *getHigher* block, which has as output the higher value between the inputs received, thus the higher between of *IN1* and *IN2*; and whether the higher value between *IN1* and *IN2* is higher than *IN3* — the output of the block *isHigher*. The numbers just below the blocks mark their *execution order*, with the lesser number representing the higher precedence.
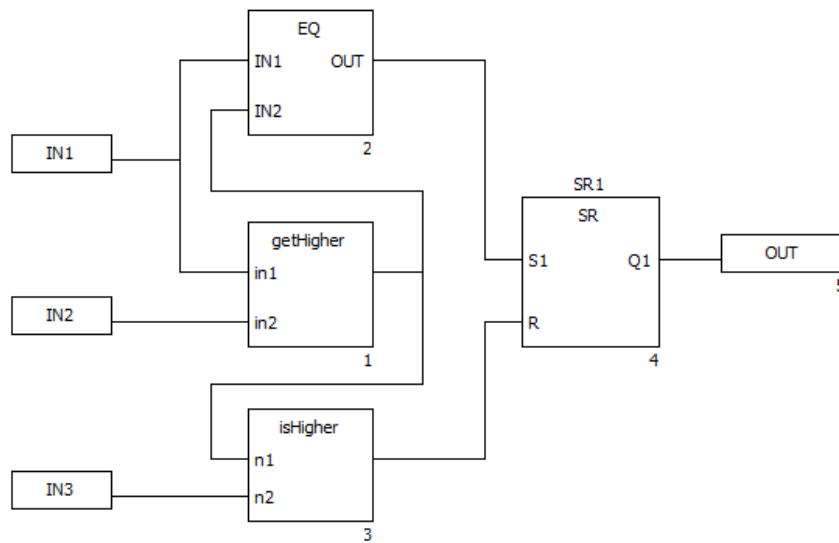
Figure 28: FBD program to be translated to B notation

```
1 VAR return_getHigher_op1, return_isHigher_op1 IN
2     return_getHigher_op1 <-- getHigher_op(IN1, IN2);
3     return_isHigher_op1 <-- isHigher_op(return_getHigher_op1, IN3);
4     SR1_body(bool(IN1 = return_getHigher_op1), return_isHigher_op1);
5     TestFBD_OUT := SR1_Q1
6 END
```

Code 38: B generation from FBDTest example

The B statements with the equivalent behaviour is presented in Code 38. In lines **2** and **3** the invocation of the stateless blocks is performed, with the necessary return variables defined in line **1**, with the **VAR** clause. The function block instance call for *SR1* is made in line **4**, with its first argument, the equality comparison block, being automatically translated to the logic expression for equality comparison in B — thus, the *EQ* block is converted in the expression **bool(**IN1 = return_getHigher_op1**)**, since the input *IN2* receives the resulting value from the *getHigher* block evaluation. In line **5**, once the function block instance has defined its output value, the assignment is made to *OUT*, which, for being the output of a instance of a *FBDTest* function block itself, *TestFBD*, has the instance name as its prefix.

4.2.2.3   LD

The B translation of a LD program, as in FBD, will also be oriented by the organization of its graphical elements: *how* the contacts, blocks and variables are disposed so that a coil is energized once the rung is closed. In Code 39 the generation procedure is

presented.

```
1  for each CPointIn in the right power rail:
2      set connected element (coil) as l-value
3      for each connected element from coil's CPointIn
4          buildExpressionFromElement
5      if more than one resulting expression
6          build an OR expression from resulting expressions
7          build assignement l-value := OR expression
8      else
9          build assignement l-value := sole resulting expression
10 (*---------------- Buildi Expression From Element -----------*)
11 take connected elements from CPointIn
12 if connected to left power rail or without connections
13     build expression solely with received element
14 else
15     for each connected element
16         buildExpressionFromElement
17     if more than one resulting expression
18         build an OR expression from resulting expressions
19         build an AND expression with received element and OR expression
20     else
21         build an AND expression with received element and sole resulting
                expression
```

Code 39: B generation from a LD body

As stated in line **1**, the process is for each coil connected to the right power rail, therefore for each rung in the LD program. A logic expression will be built corresponding to the rung organization (lines **3**-**6**), establishing that when the rung is closed its value is **TRUE**, being **FALSE** otherwise; this expression will be assigned to the coil (which represents an output, inOut or local variable).

The structure of the expression representing the rung evaluation is defined mainly from lines **11** to **21**. If, for a given element, there is more than one element connected to it, an **OR** expression must be built for each of the expressions resulting from the translation of these elements (lines **15**-**19**); then if the given element is energized **and** the path leading to it also is, that part of the rung will be closed (line **19** or **21**). The base case in this recursive procedure is when the element hit is connected to the left power rail (first element in the rung) or when it is not connected to anything (outside the scope of the LD language, like a FBD variable).

An important remark is that each of the elements evaluated in the elaboration of the rung expression are inserted into the B translation with the correspondent translation from previous language elements: blocks will be invoked as functions or function blocks in the same manner as in FBD programs, so the same goes to the variables, treated as in

a ST expression.

Example
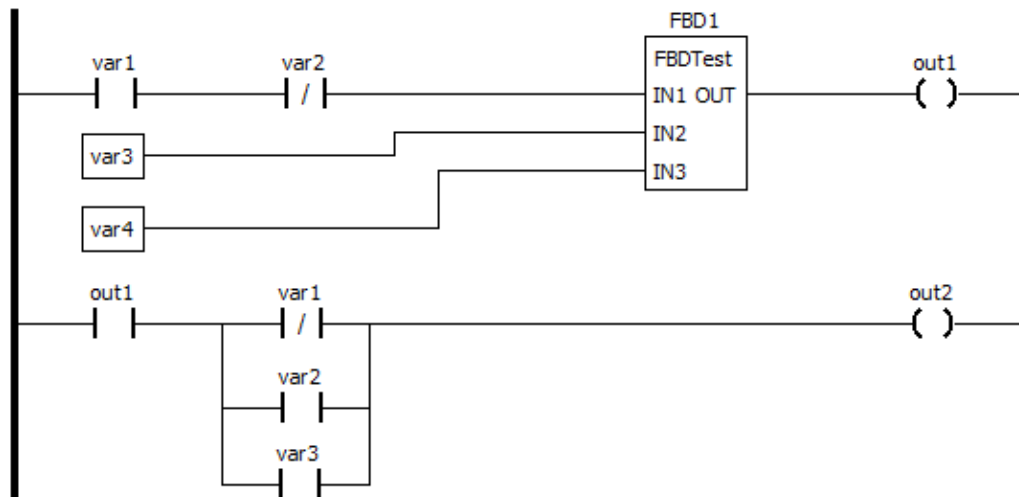
Figure 29: LD program to be translated to B notation

To illustrate the translation from LD to B we use the LD program depicted in Figure 29, where a block is used in the middle of a rung, also receiving values from FBD *inputVariables*, independent from the rung evaluation. The closing of the rung and consequent energizing of the coil *out1* in the first rung will depend on the function processing. Whether the second rung is closed, and *out2* energized, will depend on the value attributed to *out1* in the previous rung and if *var1* is **not** energized, **or** *var2* **or** *var3* is energized.

```
1 FBD1_body(bool( not(var2 = TRUE) & (var1 = TRUE)), var3, var4);
2 out1 := FBD1_OUT;
3 out2 := bool((var3 = TRUE) or (var2 = TRUE) or not(var1 = TRUE)) & (out1
    = TRUE))
```

Code 40: B generation from LD example

The resulting B translation to this LD program is presented in Code 40. The translation of the first rung into a B expression results in lines **1** and **2**: in **2** the coil (*out1*) receives the block output (FBD1_OUT), which is determined after the function block instance invocation in line **1**, whose first parameter is the resulting expression from the rest of the rung and the others are the FBD *inputVariables*. The second rung's expression is generated straightforwardly through the guidelines for the LD translation, composed by

an **AND** expression with an operand as the predicate for *out1* and the other for the **OR** expression with the predicate for the alternative paths to energizing the rung after it is past the first contact. The coil *out2* then is assigned the value resulting of this **AND** expression evaluation (line **3**).

4.2.2.4 SFC

The translation of the SFC programs is also dependent on its structure: *how* the **steps** are disposed, which are the **actions** associated to them, what's the language they are implemented in, and what is the organization of the **transitions** between the steps.

For the steps and the actions associated the translation process follows the guidelines shown in Code 41. Firstly, for each step in the SFC program (line **1**), all of its actions are retrieved (line **2**) and translated to the B notation depending on their language (lines **3**-**9**). The resulting translations are then composed as a set of *sequential* substitutions to be performed in the same order in which the actions were associated to the SFC Step (line **10**).

```
1 for each step:
2     for each associated action:
3         if ST body then
4             get translated statements
5         else
6             if LD body then
7                 build LD expressions
8             if FBD body then
9                 build FBD expressions
10 build step substitutions
```

Code 41: B generation from a SFC body

Considering the SFC example used in previous sections, but now with two actions associated to *Step1* — *Action1* in ST and *Action2* in LD —, whose new structure is depicted in Figure 30, with the *Action2*'s LD body in Figure 31, the resulting B translation for *Step1* would be the one exhibited in Code 42. From lines **2** to **7** are the resulting translations of *Action1*, following the guidelines for the ST to B translation procedure, while from lines **8** to **9** are the statements resulting from the LD to B translation, with line **8** corresponding to the first rung and line **9** to the second rung. The action statements will compose the *Step1* substitutions respecting the order they were declared in the SFC *Action Block*.

The other elements to be considered in a SFC *body* are the transitions and how they are
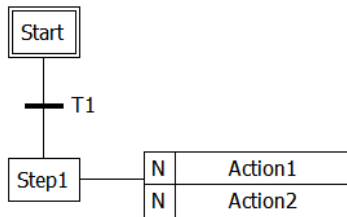
Figure 30: SFC example modified.
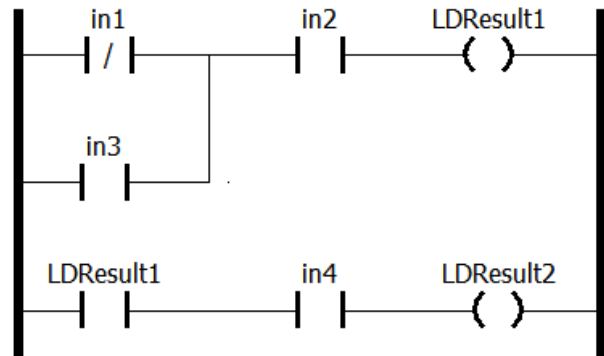
Figure 31: Action2, in LD

```
1  VAR return_isHigher_op1 IN
2      return_isHigher_op1 <-- isHigher_op(n1, n2);
3      IF return_isHigher_op1 = TRUE THEN
4          higher := answers(1)
5      ELSE
6          higher := answers(2)
7      END;
8      LDResult1 := bool( (not(in1 = TRUE) or (in3 = TRUE)) & (in2 = TRUE)
           );
9      LDResult2 := bool((LDResult1 = TRUE) & (in4 = TRUE))
10 END
```

Code 42: B generation from LD example

connecting the steps; the definition of the *reaching conditions* for a step will be determined by them, which will compose the **precondition** of the B **operation** representing each step in the SFC body. In Code 43 the guidelines for their generation are shown, which will result in a logic expression involving the transitions and steps preceding the step taken into account.

The procedure is performed for each step in the SFC body and executes according to whom the step is connected with (line **1**), from three options in a valid SFC program: a transition, a selection convergence or a simultaneous divergence. For the first two (line **2**), each transition preceding the step is analyzed (line **3**), firstly taking its *logic condition* (line **4**) and then, if this transition is preceded by a selection divergence, taking the negation of all the transitions composing the divergence (lines **5**-**7**) and building an **AND** expression with all the conditions, with only the step preceding transition's condition not negated – this represents the evaluation order in a selection divergence, where a transition is cleared only if the other transitions previously evaluated were not.

Once the logic condition for the preceding transition is defined, one needs to consider

```
1  for each connected element from step's CPointIn:
2      if transition or selectionConvergence then
3          for each preceding transition:
4              get preceding transition's condition
5              if transition's predecessor is selectionDivergence then
6                  build AND expression with all transition's conditions
                       negated
7                  set logiCondition with the preceding transition's
                       condition and the AND expression
8              else
9                  set logiCondition as the preceding transition's
                       condition
10             get transition's source steps
11             build an AND expression with source steps and logiCondition
12             set connectionCondition as the resulting AND expression
13         build step's reachingCondition as OR expression with
               connectionConditions
14     else #connected to simultaenousDivergence
15         get connected simult divergence's name -- simDiv
16         build step's reachingCondition as being a successor of the
               respective simDiv
```

Code 43: B precondition generation from Step connections

its source steps (line **10**). This is necessary because a SFC transition is evaluated only when the execution flow is coming from their predecessor steps, therefore, in order to emulate this flow, we need to allow a step to be reached only when their transitions' logic conditions are valid **and** the source steps of each of those transitions were just executed (line **11**). Thus, if the step was preceded for more than one transition, an **OR** expression is built with the respective conditions, otherwise the condition to the single transition is defined as the step's *reaching condition* (lines **12**-**13**).

The other case to consider relative to the element preceding the step is when it is a *simultaneous divergence* (line **14**). In this case the reaching condition for the step will be solely that its preceding simultaneous divergence was reached and the step has not yet been activated, since the evaluation of the transition's condition and source step's execution will be done by the simultaneous divergence itself (lines **14**-**16**).

To exemplify the generation of a step's *reaching condition* we make use of the SFC programs presented in the previous Chapter, in Section 3.1.2.8, for the SFC language. There is an example for each of the possible situations covered by the previous guidelines.

In Figure 10, at page 47, a sequence of steps connected by single transitions is depicted, which fits into the most simple case for our reaching conditions' generation procedure. Thus, for *Step2*, for instance, the resulting reaching condition would be of the form

$$\texttt{T2 \& Step1}$$

A *selection convergence* is present in the SFC program shown in Figure 13, at page 110, which has *Step7* as destination. The generation of its reaching condition must take into account all the transitions, as well as their source steps, leading to the selection convergence in question, thus the reaching condition of *Step7* has the form

$$(\texttt{T3 \& Step3})\ \textbf{or}\ (\texttt{T5 \& Step4})\ \textbf{or}\ (\texttt{T8 \& Step6})$$

For the SFC program in Figure 15, at page 49, the case where the preceding transition is in turn preceded by a *selection divergence* applies. Thus, considering that in this structure a step is reached only when its transition is cleared and the others were not the reaching conditions for, e.g., *Step2* would be of the form

$$(\texttt{T1 \& ¬T2}) \texttt{ \& Step1}$$

The last situation, when the step is preceded by a *simultaneous divergence*, is the one depicted in Figure 14, at page 49. The reaching conditions would then be thus defined:

For the *simultaneous divergence*: $\texttt{T1 \& Step1}$
For *Step2*: $\texttt{Step2} \in simDivSuccessorsYetToBeActivated$

This way the execution flow in a SFC program can be represented by its B model equivalent.

4.2.3 B Writer

The B Writer module receives a *Types* object from the PLC Object and generates a B model in the architecture shown in the previous section for <u>each</u> POU *program* in it, making use of its auxiliary POU *function* and *function block instances* objects, as well as of the *DataTypeElement* objects used.

The B machines will be composed of three main parts, each derived from a subset of the *POU* objects and responsible for an aspect in the model:

- STATIC INFORMATION: derived from *DataTypeElement* objects and flow control constructs.

- STATE: derived from *POU*'s Interface and flow control constructs.

- BEHAVIOUR: derived from *POU*'s Body.

Another component of the B machines generated will be the *composing* structures and their use: the **INCLUDES** and **SEES** clauses, the former used by the POU *programs* and *function block instances* to access the *function block instances* they use, while the latter is used by these same components to allow the invocation of the POU *functions* represented by the **operations** of the to be seen "`Functions`" machine.

The modeling of the *static information* in the machines will be made by the **SETS** clauses, while the *state* will be composed of the **VARIABLES**, their typing **INVARIANTS** and **INITIALISATION**, with the *behaviour* performed over this state being defined in the **OPERATIONS**. The guidelines for the generation of these clauses are presented below.

To emulate the *flow control* of the PLC programs, a concept non-existent in B, some workarounds are needed. The additional elements will be necessary for POU *programs* and *function blocks instances* whose body is in LD or SFC, which are modeled as a series of B operations — for the numerous rungs and steps/simulatenous divergences, respectively. The flow control will be made using a `SET` modeling the possible elements that may be *active* in a given moment, as well as the last that was *executed*; specifically for the SFC simultaneous divergences, variables are created also for flow control regarding their *successor* steps.

Once a POU runs in *execution cycles* the emulation of each cycle is made by repeating the *scan* procedure: read the inputs, process them and produce the outputs. The structure of the B **operations** representing a POU body must be compliant to this, allowing the reading of the inputs, the performance of the respective behaviour translated from this body and, finally, the definition of the output values produced after each cycle.

4.2.3.1 Static information and structuring

The static information in the B machines generated will be derived from the *DataTypeElement* objects and, if necessary, increased with the *flow control* constructs required by the respective POU implementation.

Functions

In the machine *Functions*, to be seen by all the components of the model, the SETS corresponding to the **enumeration** types declared are built. Each set is constructed following the structure[9]:

```
SET ::= enumerationTypeName '=' '{' valueName {',' valueName } '}'
```

Thus, an enumeration type of the form RESULT : (YES, NO, UNDETERMINED) would result in a B set with the declaration RESULT = {YES, NO, UNDETERMINED}.

Programs* and *Function Block Instances

If a POU *program* or *function block instance* machine is being built and its body is in LD or SFC, a set representing the language elements that are correspondent to B operations, and a value marking that it is the first execution cycle — beginning — are constructed following the guidelines:

- LD:

```
SETS ::= LD_CHAIN '=' '{' 'beginning' ','  'rung' + rungNumber {',' '
    rung' + rungNumber} '}'
```

Therefore, for the LD program shown in Figure 29, in the example for the translation guidelines for LD, the resulting flow control set would be:

$$LD\_CHAIN = \{beginning, rung1, rung2\}$$

- SFC:

```
SETS ::= SFC_CHAIN '=' '{' 'beginning' ',' stepName {',' stepName } [,
    simDivName {',' simDivName }] '}'
```

Thus, for the SFC program shown in Figure 14, at page 49, the resulting flow control set would be:

---

[9]We use a notation very similar to BNF to represent our translation rules.

```
SFC_CHAIN = {beginning, Step1, Step2, Step3, Step4, Step5, Step6, Step7,
                                simDiv1}
```

The use of these sets by the variables that will indeed make the flow control will be detailed below.

INCLUDES and SEES

The structuring mechanisms are used by the POU *programs* and *function block instances* components, **including** *function block instances* machines they use or **seeing** the *Functions* machine.

The clauses are built with the following structure:

```
INCLUDES ::=  functionBlockInstance {',' functionBlockInstance}
SEES ::= 'Functions'
```

Taking as example the LD program used in the previous section, for LD translation, shown in Figure 29, its INCLUDES and SEES clauses would be of the form:

```
INCLUDES
     FBD1
SEES
     Functions
```

4.2.3.2   State

The state information in the machines will be derived from the *Interface* objects in the respective POUs, besides the additional constructs for flow control that are demanded, if any.

The guidelines have some differences depending on the type of the POU and the language its body is implemented. These are described below.

Functions

The POU *functions* are **stateless**, then no state information needs to be modeled for them in the respective machine — *Functions*. Their *inputs* and produced *outputs*, as well as *temporary* variables eventually used for their execution will be relative only to the **operation** modeling its behaviour.

Programs

A feature common to any POU *program* being modeled is regarding the inputs received.

Variables are created for the internal representation of the POU input and inOut variables, named by the prefix "int_" plus the variable's original name. The creation of these variables is necessary because the inputs and inOuts themselves are going to be received as parameters in a B **operation**, but as they may be used in other operations (for the model of a POU in LD or SFC), that reference would be lost without the saving of the inputs' state in the B machine itself. Being part of the machine's state also allows the definition of **invariants** considering them, which will be necessary to verify the correctness of the POU's execution according to the safety constraints.

The POU *outputs* are treated as local variables; it is no loss of generality to deal with them like that since we are considering the POUs only as independent components. The safety constraints will concern mostly these *outputs*, usually relating their values to the values received as *inputs*.

For LD and SFC, which make use of numerous B **operations** to model their POU bodies, variables will be created for flow control, according to the guidelines shown for structuring program and function block instances. Only the one stating which was the last element *executed* is necessary for LD, but SFC will require more — another to mark which are the elements currently *active* and which are the successor steps not yet activated after a simultaneous divergence clearance. Thus, `executed` is typed as belonging to the set with the language operation's elements — `LD_CHAIN` or `SFC_CHAIN` —, and `active` and the `simDivNameSucs` variables, these for each simultaneous divergence, are typed as a subset of those sets.

Function Block Instances

Regarding the state representation, the only difference between the POU *programs* and *function block instances* in their respective B machines will be that, since the same function block may have multiple instances, each with its own state, the elements to access this state, the function block *inOuts* and *outputs*, must reflect that it is a unique instance.

Thus, to distinguish the different instances of a function block the *instance name* is added as a prefix to each *inOut* or *output* variable in the B component. Thus, e.g., the same function block may be included in a POU *program* as two different machines, with its variables representing the accessible state being different.

Overall structure generation

Thus, for the *programs* and *function block instances*, the basic generation process for the B clauses is presented below for the variables, their typing invariants and initial state — both at abstract and refinement level.

- **VARIABLES**

Clause with only the names of the variables used in the component.

```
VARIABLES ::= varName {‘,’ varName }
```

Considering the same LD program taken as example previously, the variables clause generated would be of the form:

```
VARIABLES
    int_var1, int_var2, int_var3, int_var4, out1, out2, executed
```

- **INVARIANT**

The invariants generated here concern only the *typing* of the variables:

```
INVARIANT ::= varName ‘:’ varType {‘&’ varName ‘:’ varType }
```

For the same example with the LD program the INVARIANT clause would be:

```
INVARIANT
    int_var1 : BOOL & int_var2 : BOOL & int_var3 : BOOL & int_var4 :
        BOOL &
    out1 : BOOL & out2 : BOOL &
    executed : LD_CHAIN
```

The internal variables receive the same type as their respective *inputs*, while the *outputs* receive the type in the POU's Interface and *executed*, the control flow variable is typed as belonging to LD_CHAIN.

- **INITIALISATION**

- **Abstract level**

    Non-deterministic assignments to the variables:

    ```
    INITIALISATION ::= varName ':::' varType {'||' varName ':::' varType }
    ```

    The INITIALISATION, in the abstract level, for the state of the LD program used as example is of the form:

    ```
    INITIALISATION
        int_var1 :: BOOL || int_var2 :: BOOL || int_var3 :: BOOL ||
            int_var4 :: BOOL ||
        out1 :: BOOL || out2 :: BOOL ||
        executed :: LD_CHAIN
    ```

- **Refinement level**

    Deterministic assignments to the variables:

    ```
    INITIALISATION ::= varName ':=' (initial_value | default_initi}) {'||'
        varName ':=' (initial_value | default_initi)}
    ```

    The *initial_value* is extracted from the respective *Variable* object in the PLC Object. If there is none, then the default_init is used. It is a value predetermined in our method for each data type, so the verification effort can be reduced as the prover will

not have to deal with *undetermined* values to the variables in the **refinement** level. For a boolean variable, the value assigned would be **FALSE**, while for an array would be **{}**, for instance. This is only adopted, though, if such a deterministic statement does not break any invariant inserted in the model after its generation, otherwise the nondeterministic assignment is maintained.

In the refinement INITIALISATION the values of the flow control variables, if any in the respective machine, are also set, but for the default state previous to the first *execution cycle*: beginning.

Thus, the refinement initialisation for the example is:

```
INITIALISATION
    int_var1 := FALSE || int_var2 := FALSE || int_var3 := FALSE ||
        int_var4 := FALSE ||
    out1 := FALSE || out2 := FALSE ||
    executed := beginning
```

4.2.3.3 Behaviour

A machine's behaviour is modeled through B **operations**, as stated before. The structure of these operations will be dependent on the *POU type* and the language its body is implemented in.

Regarding the languages, the B machines' operations will be structured as follows:

- **ST/FBD**

Only one operation is used to implement the body of the POU.

- **LD**

The number of operations will dependent on the type of the POU. As the body of a POU *function*, whose bodies are modeled in a single B **operation**, the LD program is therefore modeled in a single operation, with the resulting operations for each rung being disposed "sequentially". As the body of a POU *program* or *function block instance* the LD program is split into a B **operation** for each rung, so the evaluation of the coils is made through different steps, dividing the behaviour of

the LD program into specific parts. This structure is adopted for enhancing the expressivity of the B model for LD programs when it implements the body of a POU *program* or *function block*, stateful elements of the PLC whose verification is key and a stepwise behaviour performance (through several operations) is a better representation to achieve that.

In the case of a representation through several operations the first rung will be where the POU inputs are read — therefore where their internal representations are set. It also marks where the *execution cycle* of the POU begins, so *executed* is reset.

- **SFC**

  The SFC programs are always represented as a series of B **operations**, one for each *step* and *simultaneous divergence* in the POU body, with the former's operations being composed by the preconditions generated from the step's *reaching conditions* and the substitutions translated from the actions associated to the step; the latter's operations are used solely to organize the SFC's flow control, since they mark the *activation* of more than one step.

  The *active* variable, in a SFC program, will be used when a *simultaneous convergence* is considered, since that will require, for its clearance, that **all** its preceding steps are *active*. The operation to have its precondition considering this, be it for a step or a simultaneous divergence, will thus require that all their source steps belong to the current state of *active*.

  The *executed* variable will be used for the composition of the preconditions of all other scenarios, so that the last executed step was the source step of a step or simultaneous divergence.

Having the general structure of the machines defined according to the languages, we can approach how the generation process of the B **operations** is carried out. Below we present the process for the POU *functions* and for the *programs/function block instances*.

POU *functions* statements

Since the POU *functions* are represented in the B model as B operations, one for each *function*, in a machine that is *seen* by the other components of the model, POU *programs*

and *function block* instances, **alterations in the state of the machine can not be performed**. Thus, assignments, other than to the B operation's outputs, are not allowed.

Another issue is that, being stateless, it makes no sense to have a refinement for the POU *functions*, since there is no state to apply invariants to. The safety constraints will be addressed in the component making the function invocation, therefore the translated statements have to be represented at **abstract** level, where sequential substitutions can not be. To tackle this limitation we make use of the **ANY** statements in the B language, that may be use to simulate sequential execution even at abstract level in a B specification.

Our method supports the POU *functions* in ST, FBD or LD, however with limitations:

- **Only** blocks that have a B expression equivalent may be used in the FBD or LD program.

- For LD, the expressions representing the rungs have to be inside the same B operation, with the sequencing being emulated by means of the **ANY** statements.

The first restriction is because the machine with the POU *functions* translations can not have operations calling each other in the same machine, as well as function block instances could not be invoked, as it would require state alteration (of the included machine representing the instance), forbidden in a *seen* operation. The second is that, as the behaviour of the LD program will be considered only at abstraction level, in the *function*'s B operation being invoked, all the substitutions must be in that same operation.

The building of the **ANY** statements is oriented by the assignments in the function's program, according to the guidelines in Code 44. The assignments are the substitutions responsible for changing the state of the machine, so they are eliminated from the program, being replaced by **ANY** statements with "local variables" whose values are the same as the respective expressions being assigned to the variable it represents — lines **2**-**7**. A mapping is maintained from the variable that would be in the assignment to the number of the *local variable* representing the value it would receive, so this *local variable* can replace the *not-assigned variable* occurrences in the expressions to come in the program. (lines **4** and **8**).

An important remark is that, emulating the sequence of the original program, the **ANY** statements will propagate: each new one will be inside the scope of the last, so the *local variables* created in the previous statements may be used on it. Once the program was dealt with, the *not-assigned variables* will receive the last value that would have been

```
1  for each assignment:
2       increase expressionCounter
3       define valueIdentifier as ''value'' + expressionCounter
4       map variable being assigned to correspondent expressionCounter
5       if isLogicExpression then
6           put expression inside bool() predicate
7       translateTo "ANY " + valueIdentifier + " WHERE " + valueIdentifier
           + " = " + expression + " THEN ";
8       replace all the occurrences of the variable being assigned by the
           respective valueIdentifier
9  (*-------- After treating all assignments ------*)
10 for each variable mapped to an expressionCounter
11     if variable is output then
12         create assignment variableName := respective valueIdentifier
```

Code 44: Guidelines for generating POU *functions* translations.

assigned to them (line **12**), but only if they are *inOut* or *output* variables, which do not belong to any state the POU *function*'s B operation would be representing (lines **10-11**).

```
1  v1 := n1;
2  v2 := n2;
3  v1 := v1*45;
4  v2 := v2*45;
5  isHigher := v1 > v2;
```

```
1  ANY value1 WHERE value1 = n1 THEN
2      ANY value2 WHERE value2 = n2 THEN
3          ANY value3 WHERE value3 = value1*45 THEN
4              ANY value4 WHERE value4 = value2*12 THEN
5                  ANY value5 WHERE value5 = bool(value3 > value2) THEN
6                      v1 := value3 || v2 := value4 ||
7                      isHigher := value5
8                  END
9              END
10         END
11     END
12 END
```

Code 45: B generation from ST code in a POU *function*

The B assignments statements in line **6** would be the local variables assignments with the last values that would be assigned to them: the second assignment for *v1*, in line **3** of the ST code, and also the second assignment for *v2*, in line **4** of the ST code; the definition of the "local variables" values for these assignments are made in the **ANY** statements in lines **3** and **4**, respectively, which use the defined values for the local variables corresponding to the first ST assignments for *v1* and *v2* (lines **1** and **2** of the B statements). Nonetheless, these substitutions shall not be present in the resulting B specification for the

POU *function*, since *v1* and *v2* are **not** output variables; the only allowable assignment is the one made in line **7**, for the output *isHigher*.

POU *programs* and *Function Block Instances*

The same guidelines are followed for the generation of the B **operations** from the bodies of POU *programs* and *function block instances* — the difference between them relies on the necessity to explicit, in the machines for the function block instances, the instance that the outputs are from, through the prefixing of the instance name. However, the prefixing procedures are carried out in the PLC Object construction, thus, in the B writer module, the guidelines for the B operations generation are the same.

We present the mechanisms based on the level of abstraction, since some differences are made between the process for each case.

- **Abstract level**

As we said before the guidelines for the operations generation are language-dependent, thus we present them separately:

- ST/FBD

```
1 OPERATION ::= st_fbd_operation
2 st_fbd_operation ::= st_fbd_signature preconditions st_fbd_abstract_body
3 st_fbd_signature ::= (pouProgramName | instanceName) + '_body' [
    pou_inputs ] '='
4 pou_inputs ::= '(' (inputName | inOutName) {',' (inputName | inOutName
    } ')'
5 preconditions ::= ('PRE' inputsTyping 'THEN' | 'BEGIN')
6 st_fbd_abstract_body ::= modifiedVariablesNondeterministicAssignments
```

Line **1** illustrates that the machine will possess only one operation, while in line **3** the name of the operation, either after the POU *program*'s or the *instance*'s name, suffixed by _body, as well as, if the component has inputs, their names (line **4**). The precondition of the operation, if any, will be composed solely of the typing constraints over the inputs. The abstract operation's body will be built with

nondeterministic assignments to the modified variables in the respective ST or FBD body, obtained during the PLC Object construction.

Taking the function block instance *FBD1*, used by the LD program taken as example, we would have as resulting abstract operation the one shown in Code 46.

```
1 FBD1_body (IN1, IN2, IN3) =
2 PRE IN1 : INT & IN2 : INT & IN3 : INT
3 THEN
4     int_IN1 :: INT || int_IN2 :: INT || int_IN3 :: INT ||
5     FBD1_OUT :: BOOL
6 END
```

Code 46: Abstract operation from FBD1 body

The operation's name is prefixed by the instance name (*FBD1*), which is followed by the inputs definition (line **1**) and then their typing in the operation's precondition (line **2**). In lines **4**-**5** the nondeterministic assignments made to the variables to be modified in the refinement are presented — those are for the internal representations of the inputs, which will be assigned to the received inputs, and the instance's output, whose value will be determined also in the refinement.

- LD

```
1 OPERATION ::= rung_operation {';' rung_operation }
2 rung_operation ::= rung_signature 'PRE' preconditions 'THEN'
    rung_abstract_body
3 rung_signature ::= [instanceName + '_' + ] 'rung' + rungNumber [
    pou_inputs ] '='
4 pou_inputs ::= '(' (inputName | inOutName) {',' (inputName | inOutName
    } ')'
5 preconditions ::= [ inputsTyping '&' ] rungReachingConditions
6 rung_abstract_body ::= modifiedVariablesNondeterministicAssignments
```

One operation per rung is built (line **1**), with their numbers according to the sequence of rungs and prefixed by the instance name, if any. If the component has inputs they are received **only** in the first rung, symbolizing the beginning of the *execution cycle*; thus, the inputs typing, if any, also occurs only in the first rung operation. Nonetheless, each rung operation will have in the respective operation its *reaching conditions*, which will consist of the flow control variable *executed* having to be equal to the previous rung or to beginning, if it is the first rung.

The resulting abstract operations for the LD program example itself are depicted in Code 47.

```
1 Rung1 (var1, var2, var3, var4) =
2 PRE
3     var1 : BOOL & var2 : BOOL & var3 : BOOL & var4 : BOOL & (executed =
          beginning or executed = rung2)
4 THEN
5     int_var1 :: BOOL || int_var2 :: BOOL || int_var3 :: BOOL || int_var4
          :: BOOL ||
6     out1 :: BOOL ||
7     executed :: LD_CHAIN
8 END;
9
10 Rung2 =
11 PRE
12     executed = rung1
13 THEN
14     out2 :: BOOL ||
15     executed :: LD_CHAIN
16 END
```

Code 47: Abstract operations from LD body

The Rung1 operation (line **1**) represents the first rung's evaluation, while Rung2 (line **10**) stands for the second rung's evaluation. In Rung1, the beginning of the *execution cycle* is marked, so the inputs are received and have their acceptable types defined (line **2**). In the precondition the reaching conditions for Rung1 are also defined, specifying that either this is the first performance of the execution cycle — executed = beginning — or a previous cycle has ended — executed = rung2; for the Rung2 operation the reaching condition is that its previous rung was executed (line **12**). Besides the modified variables for inputs assignments (line **5**) and rung evaluation (lines **6** and **14**), the control flow variable executed is also marked as having its state modified in the rungs' operations (lines **7** and **15**).

- SFC

Steps

```
OPERATION ::= (step_operation | simDiv_operation) {';' (step_operation |
    simDiv_operation) }
```

The operations for SFC will be built either from a step or from a simultaneous divergence.

```
1 step_operation ::=  step_signature 'PRE' step_precondition 'THEN'
     step_abstract_body
2 step_signature ::=  [instanceName + '_' + ] stepName [ pou_inputs ] '='
3 pou_inputs ::= '(' (inputName | inOutName) {',' (inputName | inOutName
     } ')'
4 step_preconditions ::= [ inputsTyping '&' ] stepReachingConditions
5 step_abstract_body ::= modifiedVariablesNondeterministicAssignments
```

The step operations are named after the respective step's name, being prefixed by the instance name, if any (line **2**). As in the LD body, the reading of inputs will be made solely in the operation marking the beginning of the *execution cycle*, which in a SFC program is in its **initial step**. The step's reaching conditions are retrieved from the logic expression built during the PLC Object construction, only that now the source step validity is made through `executed`, `active` or `simDivNameSucs`, with the last two being used instead of the first when the step has more than one source step through a unique transition (which happens only when the transition's predecessor is a simultaneous convergence) or when it is the successor of a simultaneous divergence, respectively.

In Code 48 the resulting step operations from the example shown in Figure 30, at page 110, are depicted.

```
1 Start (n1, n2, in1, in2, in3, in4) =
2 PRE n1 : INT & n2 : INT & in1 : BOOL & in2 : BOOL & in3 : BOOL & in4 :
     BOOL & (executed = beginning or active = {step1})
3 THEN
4     active :: POW(SFC_CHAIN) || executed :: SFC_CHAIN || answers :: 1 ..
         2 +-> RESULT || higher :: RESULT || int_n1 :: INT || int_n2 ::
         INT || int_in1 :: BOOL || int_in2 :: BOOL || int_in3 :: BOOL ||
         int_in4 :: BOOL
5 END;
6
7 Step1 =
8 PRE (int_in1 = TRUE & executed = start)
9 THEN
10     active :: POW(SFC_CHAIN) || executed :: SFC_CHAIN || higher ::
         RESULT || LDResult1 :: BOOL || LDResult2 :: BOOL
11 END
```

Code 48: Abstract step operations from SFC body.

The structure differs from the LD resulting operations, shown previously, for the use of `active` to control the SFC execution flow along with `executed`. We can also see, in the third assignment an line **4** of the B code — `answers :: 1..2 +-> RESULT` — the resulting translation of the type *ANSWERS*, derived from the **Array** type. The partial function represents an array, with its *domain* being generated from the array dimensions and the *range* from its base type.

Simultaneous divergences

```
1 simDiv_operation ::= simDivName '=' 'PRE' simDivReachingConditions
     'THEN' simDiv_abstract_body
2 simDiv_abstract_body ::= modifiedVariablesNondeterministicAssignments
```

The simultaneous divergence's operations are named after the respective element's name and their preconditions regard only their reaching conditions, which are generated in the same manner as for the steps, also using `executed` or `active` to determine the source step's validity.

```
1 SimDiv1 =
2 PRE (T1 = TRUE & executed = step1)
3 THEN
4     active :: POW(SFC_CHAIN) || simDiv1Sucs :: POW(SFC_CHAIN) ||
          executed :: SFC_CHAIN
5 END
```

Code 49: Abstract simultaneous divergence operation from SFC body.

We use the example shown in Figure 14, at page 49, which presents a SFC program containing a simultaneous divergence element, to demonstrate the results of these guidelines, depicted in Code 49. The reaching condition is produced as for the steps, so the result is the one seen in line **2** — the validity of the predecessor transition's logic condition and source step's prior execution. The variables to be modified in a simultaneous divergence operation are the ones regarding general flow control and its *specific* destination steps, thus `SimDiv1Sucs` .

• **Refinement level**

In the refinement operations the signatures are maintained and the preconditions may be omitted, as they are already constraining the operation access' state by being defined in the abstract level. Therefore, the main changes are presented in the operations' bodies, where the internal representations of the inputs will be set, the B statements resulting from the PLC programs will be inserted and the flow control, if any is required, will be made.

• ST/FBD

```
1 OPERATION ::= st_fbd_operation
2 st_fbd_operation ::= st_fbd_signature 'BEGIN' st_fbd_refinement_body
3 st_fbd_refinement_body ::= internalVarsSetting initialValuesSetting
    returnVarsClause (stStatementsTranslated | fbdExpressionsGenerated)
```

The setting of the internal representation of the inputs will consist simply of each input being assigned to its internal variable, and the setting of the initial values is made for each variable that has one. If any function invocation demanded the creation of a *return* variable it will be put in a **VAR** clause, for local variables within its scope, which is where the ST's or FBD's B equivalent programs will be put.

```
 1 FBD1_body (IN1, IN2, IN3) =
 2 BEGIN
 3     int_IN1 := IN1; int_IN2 := IN2; int_IN3 := IN3;
 4     VAR return_getHigher_op1, return_isHigher_op1 IN
 5         return_getHigher_op1 <-- getHigher_op(int_IN1, int_IN2);
 6         return_isHigher_op1 <-- isHigher_op(return_getHigher_op1,
            int_IN3);
 7         SR1_body(bool(int_IN1 = return_getHigher_op1),
            return_isHigher_op1);
 8         FBD1_OUT := SR1_Q1
 9     END
10 END
```

Code 50: Refinement operation from FBD1 body.

In Code 50 we have the refinement operation generated from the same example taken in the abstract level. Here we have the actual translation of the FBD body

(lines **4** to **9**), obtained following the guidelines of the translation process for FBD, besides the internal variables setting, in line **3**, receiving the inputs values so they can be used in the translated statements.

- LD

```
1 OPERATION ::= rung_operation {';' rung_operation }
2 rung_operation ::= rung_signature 'BEGIN' rung_refinement_body
3 rung_refinement_body ::= [internalVarsSetting initialValuesSetting] [
      returnVarsClause] rungExpressionsGenerated flowControlStatements
```

The construction of the rung operations' bodies will have the internal variables' setting and initial values' assignments **only** in the first rung, symbolizing the beginning of the *execution cycle*. The **VAR** clause will be created for the rungs whose B notation equivalent required so, and the *flow control* statement will consist of `executed` receiving the current rung as value.

```
1 Rung1 (var1, var2, var3, var4) =
2 BEGIN
3     int_var1 := var1; int_var2 := var2; int_var3 := var3; int_var4 :=
          var4;
4     FBD1_body(bool( not(int_var2 = TRUE) & (int_var1 = TRUE)), int_var3,
          int_var4);
5     out1 := FBD1_OUT;
6     executed := rung1
7 END;
8
9 Rung2 =
10 BEGIN
11     out2 := bool(not(int_var3 = TRUE) or (int_var2 = TRUE) or not(
          int_var1 = TRUE)) & (out1 = TRUE));
12     executed := rung2
13 END
```

Code 51: Refinement operations from LD body

Code 51 is the refinement generated for the abstract operations in the previous LD example. It can be seen that, indeed, inputs reading and internal variables setting occurs only in the first rung (lines **1** and **3**); nonetheless, the structure of the rung operations is identical aside from that, with the expressions representing the rungs' evaluation (lines **4**-**5** and **11**) and the flow control — `executed` being valuated with the last executed rung (lines **6** and **12**).

- SFC

Steps

```
1 step_operation ::=  step_signature 'BEGIN' step_refinement_body
2 step_refinement_body ::= [internalVarsSetting initialValuesSetting] [
    returnVarsClause] actionsStatements flowControlStatements
```

At the refinement level the *initial step*'s operation will have the internal variables' setting and initial values' assignment, followed by, if needed, the **VAR** clause and, if any, the associated actions' equivalents with the B notation statements.

The control flow is made through both `executed` and `active` variables, with the former being assigned with the current step's name and the latter being *increased* also of the current step's name and *decreased* of the current step's source steps'. If the step is successor of a simultaneous divergence, when it is reached it updates the flow control variable `simDivNameSucs` by removing itself's name from the variable.

Code 52 shows the step operations at refinement level, obtained from the given example. In the *Start* operation, derived from the program's *initial step*, the inputs' reading and setting of the internal variables is made (lines **1** and **3**), as well as the initial values' assignments; the first assignment in line **4** shows the B representation for array values: a mapping from a position in the array to its respective value – an ordered pair in the partial function's scope. Since there is no action associated with the step *Start* there are no translated statements, coming after the flow control statements, regarding the last step executed — *Start* itself —, in line **6**. The `active` variable concerns the steps active, which can be more than one depending on the SFC program's organization; in *Start*, therefore, only its preceding step is deactivated, with the current step itself being put in the activation control (line **5**); although there are no connections between *Step1* and *Start*, the former is considered the predecessor of the latter due to the *execution cycles*: after the first one, a new execution cycle will begin only after the last ends — hence *Start* may execute only after *Step1*.

The operation for *Step1* presents the translated statements of its actions, from line **12** to **17** representing *Action1*, originally in ST, and lines **18**-**19** representing the rungs in the LD code for *Action2*. The flow control statements follow the same guidelines as in the *Start* operation.

```
1 Start (n1, n2, in1, in2, in3, in4) =
2 BEGIN
3     int_n1 := n1; int_n2 := n2; int_in1 := in1; int_in2 := in2; int_in3
          := in3; int_in4 := in4;
4     answers := {1 |-> YES, 2 |-> NO}; higher := UNDETERMINED;
5     active := (active - {Step1}) \/ {Start};
6     executed := start
7 END;
8
9 Step1 =
10 BEGIN
11     VAR return_isHigher_op1 IN
12         return_isHigher_op1 <-- isHigher_op(n1, n2);
13         IF return_isHigher_op1 = TRUE THEN
14             higher := answers(1)
15         ELSE
16             higher := answers(2)
17         END;
18         LDResult1 := bool( (not(in1 = TRUE) or (in3 = TRUE)))
               & (in2 = TRUE) );
19         LDResult2 := bool((LDResult1 = TRUE) & (in4 = TRUE))
20     END;
21     active := (active - {Start}) \/ {Step1};
22     executed := step1
23 END
```

Code 52: Refinement step operations from SFC body

Simultaneous Divergences

```
1 simDiv_operation ::= simDivName '=' 'BEGIN' simDiv_refinement_body
2 simDiv_refinement_body ::= flowControlStatements
```

The refinement body of the simultaneous divergences operations will be exclusively for flow control, setting `executed` with the current simultaneous divergence's name, while all its source steps are removed from `active`. Also, all its destination steps are assigned to the respective `simDivNameSucs` variable, stating all of them must be activated after its clearance.

The refinement statements for the example in the abstract level is presented in Code 53, where the flow control operations are exhibited. In line **3** the setting of *simDiv1* as last element executed is made, however it is not put into `active`, since only the **steps** are active elements in a SFC program; therefore just the removing of the predecessor steps from `active` is performed. The key operation made in the operation is the assignment of the simultaneous divergence's destination steps —

```
1 SimDiv1 =
2 BEGIN
3     executed := simDiv1; active := active - {step1};
4     simDiv1Sucs := {Step2, Step4, Step5}
5 END;
6
7 Step2 =
8 PRE step2 : simDiv1Sucs
9 THEN
10    ...
11    simDiv1Sucs := simDiv1Sucs - {step2};
12    active := active \/ {step2}; executed := step2
13 END;
```

Code 53: Refinement simultaneous divergence operation from SFC body

Step2, *Step4* and *Step5* — into the **successors yet to be activated** control variable — `simDiv1Sucs`.

How this variable is used can be seen in the flow control statements present in one of the simultaneous divergence's destination steps: when a step that is successor of such a structure is reached, besides it being set as last active and inserted into the active ones (line **12**), it has to be removed from the list of **successors yet to be activated** (line **11**), since the precondition for its reaching is uniquely being in that list (line **8**), once the transition and source steps check is made in the simultaneous divergence.

4.3   Inserting the safety constraints

The next phase is to add safety constraints. Since PLC programs do not represent such constraints explicitly, they have to be manually extracted from the project requirements and modeled to be used in the formal models[10]. This is a hard task and still an open issue in industry (ABRIAL, 2006), and we have not decided yet which methodology to adopt to tackle this problem. However, some promising approaches as (CABRAL and SAMPAIO, 2006) and (LADERNBERGER and JASTRAM, 2012) may be suited for our purpose; the latter is being used in our case study.

Once the safety constraints are defined, they are inserted into the model as **invariants** in the POU components, conditions that must always hold as the PLC actions are perfor-

---

[10]Strategies such as annotating the code with requirements labels would make it possible to automate, at some level, this process, but this is not in the scope of this work for now.

med. Tools such as Atelier B[11] can perform automatic verification of their consistency up to a certain point; an *Interactive* Theorem Prover is provided for aiding the performance of manual proofs in the Proof Obligations not discharged. Any problem found, like the impossibility of verifying a proof obligation, points out to where the inconsistence lies in the respective part of the PLC program, guiding its treatment.

To guarantee that the PLC performs the expected behavior of its *execution cycle* we may create LTL formulas over the variables representing the **operations'** execution, verifying, e.g., if a given **operation** is ever reached from a predecessor one.

We may also verify properties that cannot be modeled with regular first order logic, requiring modalities found in LTL: one example is the condition that *whenever* a given variable has a given value, there must be *some* state reached by the PLC where another certain variable receives another certain value. This can be modeled with the operators $\square$, meaning "it will *always* be the case that..."; and $\diamond$, meaning "it will *eventually* be the case that...". The ProB tool is a model checker that allows us to perform these verifications over B models, although some modifications may be necessary in a model prior to its model checking, like reducing the spectrum of values that numerical variables may take, in order to avoid the state-explosion problem. The analysis that leads to these adaptations is manual and requires expertise and understanding of the system.

4.3.1   Example

We retrieve the *Set_Reset Flip Flop* function block, presented in Section 4.2.2.2, for the FBD translation process, to show how the insertion of safety constraints into a B model, in order to verify if the PLC program is behaving accordingly to its requirement, through theorem proving verification.

The *SR* block receives two inputs, *S1* and *R*, which represent the set and reset instructions being enabled, respectively, while the output *Q* gives the state of the bit being addressed by that block. The bit is set if *S* is **1** and *R* is **0**, being reset if *S* is **0** and *R* is **1** or if both are **1**.

The graphic representation of the SR block, as well as part of the resulting B model to a *SR1* instance for such block is depicted in Figure 32, with the resulting B operation representing the POU body, with its execution and the setting of the internal representation of the inputs.

---

[11]http://www.atelierb.eu/
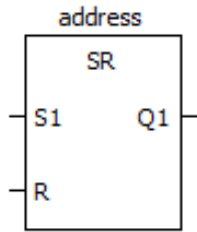
```
1 SR1_body (S1, Reset) =
2 PRE S1 : BOOL & Reset : BOOL
3 THEN
4     int_S1 := S1;
5     int_Reset := Reset;
6     IF int_S1 = TRUE & int_Reset = FALSE THEN
7         SR1_Q1 := TRUE
8     ELSIF (int_S1 = FALSE & int_Reset = TRUE)
          or (int_S1 = TRUE & int_Reset = TRUE)
          THEN
9         SR1_Q1 := FALSE
10    END
11 END
```

Figure 32: B translation for SR function block

In order to assess if the behaviour performed by the bock is compliant with the expected one we verify the following invariants, derived from the block requirements:

```
INVARIANT
    ((int_S1 = TRUE & int_Reset = TRUE) => (SR1_Q1 =  FALSE))
    &
    ((int_S1 = FALSE & int_Reset = TRUE) => (SR1_Q1 =  FALSE))
    &
    ((int_S1 = TRUE & int_Reset = FALSE) => (SR1_Q1 =  TRUE))
```

Each represents one of the allowed cases when a bit may be set (SR1_Q1 = TRUE) or reset (SR1_Q1 = FALSE).

After the insertion of these invariants in the model, four proof obligations were generated to verify their correctness according to the specification. All of them were automatically discharged by the AtelierB Prover, guaranteeing that the translation of the function block was compliant with its requirements. Regarding the translation process itself, we used the ProB Model Checker to check whether the B model's behaviour was equivalent to the execution of the SR function block, what was attested after the performance of animation over the model.

4.4   Restrictions of the approach

We summarize the restrictions of our approach in three categories, for clarity:

1. LANGUAGES:

   - For now we do not cover the elements of IL.

   - Regarding the ST statements, we do not support the *loop* statements yet.

   - The FBD elements *jump to label* and *label* have not been covered yet.

   - The SFC elements *macro steps* and *jump steps* are not yet supported, as well as most *action qualifiers*.

2. DATA TYPES:

   - Elementary types: The B language does not support floating numbers. There are workarounds to tackle this issue, as many efforts try to overcome this limitation, but they are not in the scope of this work. Thus, the elementary types *REAL* and *LREAL* are not supported by our approach.

   - Derived types: For now, our method handles *enumerations*, *arrays*, *sub ranges* and *derived* types that make reference to the data types supported. The *struct* type lies in our future work.

3. TECHNICALITIES:

   - In the SFC transition's conditions there can be no call to POU *functions*, since these conditions are represented in the B model as the preconditions of the operations and B does not support operation invocation in them.

   - The *array* elements can only be unidimensional. We have not implemented yet the translation process to multidimensional arrays.

   - The B language has several restrictions in what concerns operation invocation. Thus, the POU *functions* can not make calls between them, as operations are not allowed to call operations from the same machine.

   - We do not support time representation.

   - One feature of the PLC programs that is beyond our scope for now is the *rising edge*[12] support.

Due to these restrictions we are not able to apply our method to PLCs that use floating point arithmetic or that have time related constraints, a major setback, since many PLCs make use of them. These issues must be overcome in future works, analyzing

---

[12]Capturing the transition of a *digital signal* from *low* to *high* (**0** to **1**).

ongoing efforts into providing support for real numbers in the B language, as discussed in (BURDY et al., 2012), and time handling approaches. The translation restrictions, like the unsupported elements of the languages or the *struct* data type, also lead to a coverage loss in our approach, but they are easier to be tackled, since the required effort lies mostly in defining the translation guidelines and implementing them.

Another issue concerns the adaptation of the method to not fully standard compliant languages, or to other languages for control systems that are out of the PLC scope, since we have not yet defined *how much* different these languages may be from the ones we cover so we can treat them without the need of drastic changes in the **program reading** and **B writing** components. Future works should detail these scenarios.

5   Case Study

Our case study is the CDC (*Central Door Controller*), a part of the doors subsystem of trains in the Metro-DF project, developed by AeS[1], a small company in Brazil specialized in railway projects.

The doors subsystem became important in the railway field due to its modular architecture and a great concern with safety, an essential feature in a critical system that deals with human lives. Aiming to assure it in the control system, so to not depend only on techniques such as hardware redundancy for fault tolerance, the AeS company chose to use formal methods on the verification of their systems. They provided us with a PLC for the CDC, as well as the whole system's requirements, so we could test our method on a real industry application.

5.1   Application description

The CDC is responsible for controlling the doors' opening and closing in the train, guaranteeing that these actions are only executed under *safe* circumstances. It also controls the emergency situations that the train may be involved in, which must be taken in consideration to determine whether a given scenario is safe.

It receives, as input, information about the state of the train, such as the current speed, and commands requesting it to open or close the doors. After verifying if the conditions to execute some action are fulfilled, the CDC sends out commands, as outputs, allowing or not the required actions.

---

[1]http://www.grupo-aes.com.br/site/home/

5.1.1 Doors subsystem

In order to understand how the CDC works in the context of the train's doors subsystem we have to explain the latter in detail.
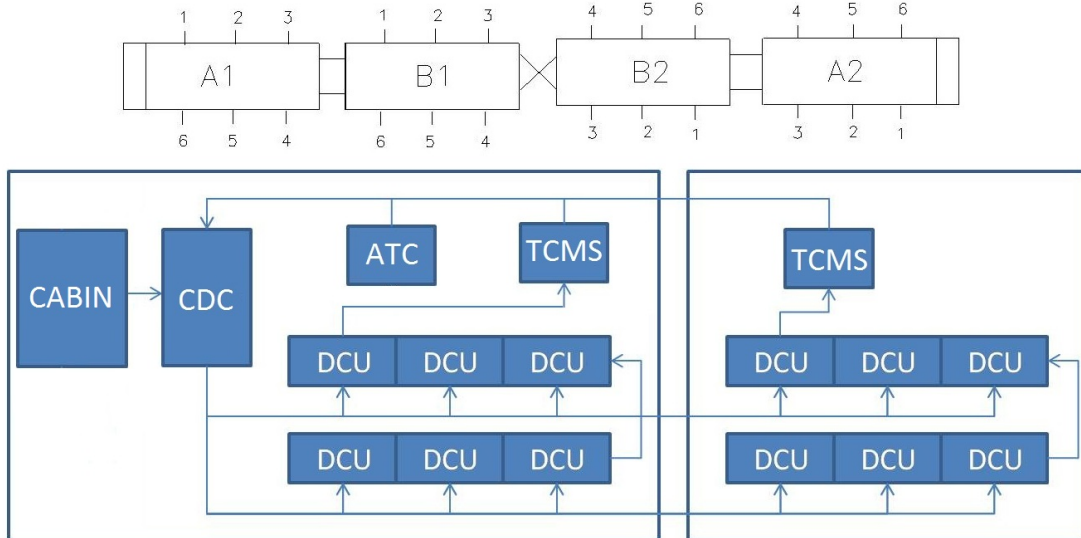
Figure 33: Overall structure of a train's door subsystem.

Each train for the Metro-DF is composed by four wagons, two of them presenting CDCs — the A type ones. Those are located in the ends of the train, as depicted in the top of Figure 33; the others, without CDCs, are in the middle, the B type ones. Each wagon has six doors, three at each side.

The bottom part of Figure 33 shows, in a simple scheme, the five components of the doors subsystem and how they are connected with each other. The components are:

- CABIN - A cabin with an operator.

- CDC - Central Door Controller.

- DCU - Door Control Unit.

- TCMS - Train Control and Monitoring System.

- ATC - Automatic Control System.

As previously said there will be two A type wagons in a train, each possessing a **Cabin**. Only one of them will be set as *leader* at each time, meaning that only the respective Cabin and CDC components in it will be allowed to operate. The former will

send commands to the latter requesting it to perform *opening* or *closing* operations, also sending information about the train — such as if it is active in *manual, semi-automatic* or *automatic* mode — so the CDC can reason about them and determine if the required operations can be performed in safe conditions.

The **DCU** components are responsible for performing control operations over a single door, therefore there is one to each door in the train. It may receive signals from the CDC demanding *opening* or *closing* operations, then it must effectively open or close the doors — it does not consider safety conditions, those are all made in the CDC. Other operations executed by the DCUs are the sound signalization to the train passengers that the door is about to be closed — operation also controlled by a signal sent by the CDC. It is also their responsibility to inform the CDC about the activation of an *emergency device*, which are of two types: the *button* emergency device, one at each door, and the *mechanic* emergency device (a switch), two for each wagon; it is sufficient that one of the emergency devices has been operated for the respective device's signal to be set and sent to the CDC.

Regarding the doors subsystem, the **TCMS** component will provide information about the current state of the train — its speed, location, etc. These will be considered by the CDC when evaluating the conditions in which opening and closing operations are allowed, as well as emergency conditions.

Each train has an **ATC** system. It will, concerning the doors subsystem, require the execution of opening or closing operations by the CDC when the train is not operating in the *manual* mode, as well as to strengthen the safety restrictions considered in it.

The *environment* where the CDC will be executed and that it will control is characterized by these components. Their disposition and interaction can be visualized in detail in Appendix D.

5.1.2 Central Door Controller

The Central Door Controller is responsible for the overall control over the doors in the train. It receives information about the train's state, through the Cabin and TCMS components, as well as from the DCUs about whether the emergency devices were performed; process them according to the restrictions for opening, closing and emergency operations; and finally emit control signals to the other components of the doors subsystem, representing the results of its processing.
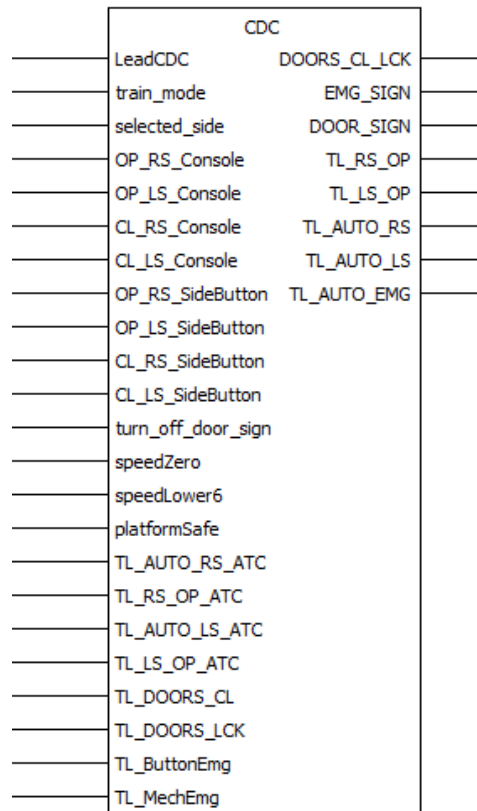
Figure 34: Central Door Controller, system view.

The CDC component is depicted in detail in Figure 34, presenting all the signals it receives and emits, whose usage will be better explained in next section, when the PLC program for CDC is shown.

The names of these signals are truncated. For a better understanding, take into account that "AUTO" stands for "authorization", "OP" for " opening", "CL" for "closing", "RS" for " right side" and "LS" for "left side"; they mark the kind and side of the operation being requested. "Console" symbolizes that the signal was sent from the buttons in the **Cabin**'s console, while "SideButton" that it was from a button in one of the **Cabin**'s side. The other abbreviation is "TL", that stands for "trainline" and marks the signals sent from the **ATC** or **DCU** components, or that it is one of the CDC's outputs.

5.1.2.1 PLC program for CDC
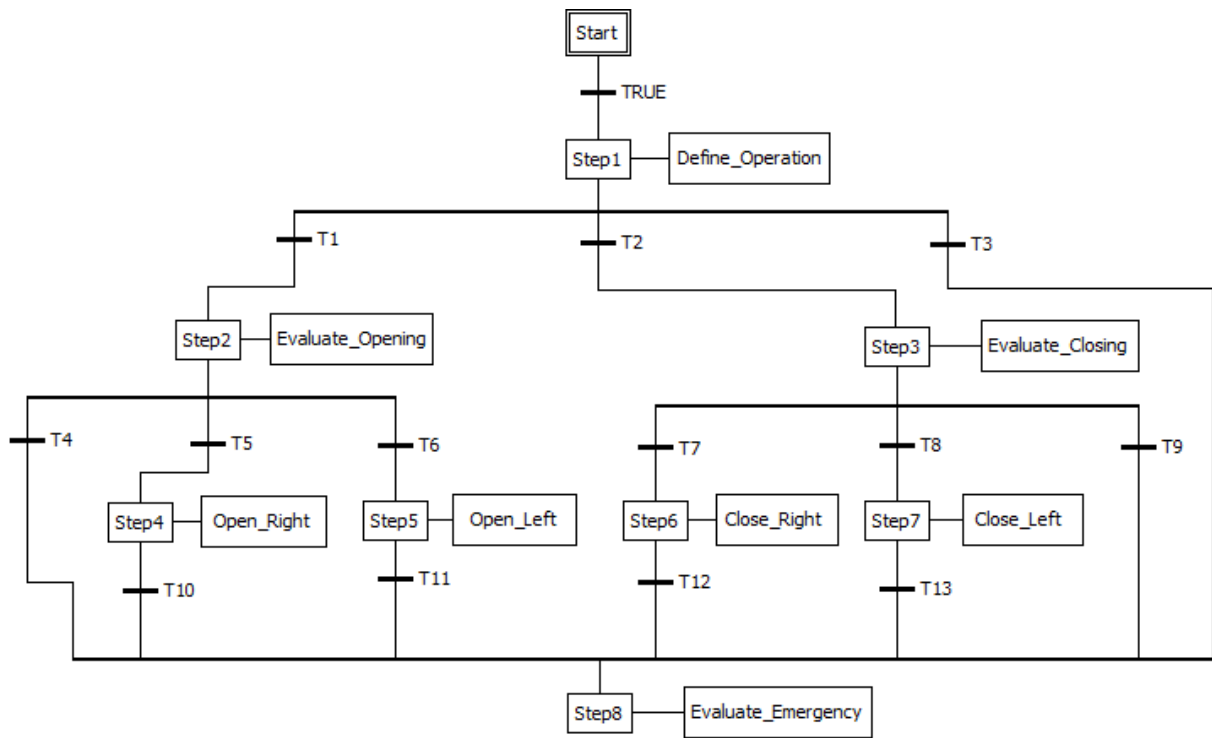
Figure 35: Central Door Controller, SFC program.

The PLC program for CDC is composed by a POU *program* in SFC as main component, depicted in Figure 35, and four auxiliar components implemented by two POU *functions* and two POU *function blocks*, all in ST. The block views for the auxiliary POUs are shown in Figures 36 and 37, while their interfaces are presented in Appendix D.

The SFC program for the POU *program* is responsible for the main computations in the CDC component, these made by the SFC *actions*, implemented in ST, associated to the SFC *steps*, which are executed according to the POU's execution flow, controlled with the SFC *transitions*, cleared according to their logic conditions', also in ST, evaluation.

The *initial step*, **Start**, has no actions associated, and the transition after it has automatic clearance — its logic condition is simply **TRUE**; this step is used solely for the PLC *scan*, reading the inputs to be used in each execution cycle. In **Step1** the action *Define_Operation* is performed, where *what* kind of operation was requested to the CDC — for opening or closing —, depending on the inputs read, also checking if the *request* itself was valid; the conditions for the operation *appliance* are checked later. The transitions **T1** and **T2** will guide the execution flow according to this result, while **T3** accounts for the case where the commands received did not qualify for a valid request.
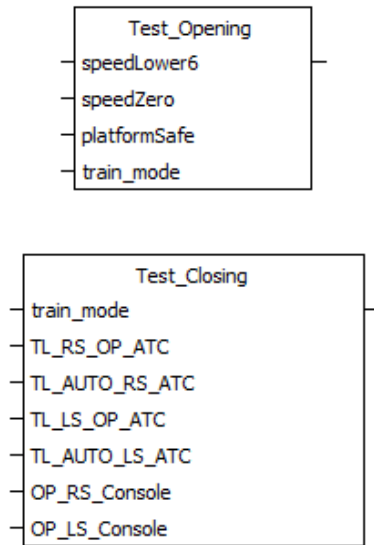
Figure 36: POU *functions* used by the CDC.
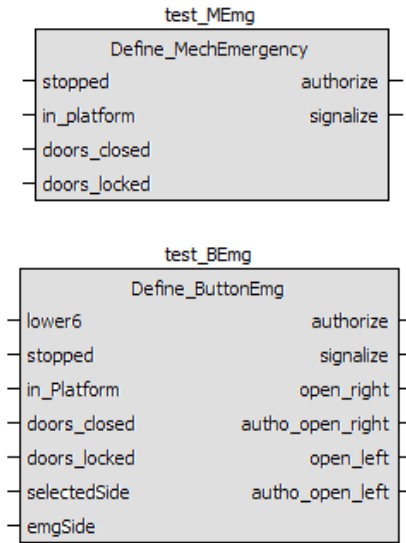
Figure 37: POU *function blocks* used by the CDC.

```
1 conditions_to_close_satisfied := Test_Closing(train_mode, TL_RS_OP_ATC,
    TL_AUTO_RS_ATC, TL_LS_OP_ATC, TL_AUTO_LS_ATC, OP_RS_Console,
    OP_LS_Console)
```

Code 54: *Action* Evaluate Closing, at Step3

In **Step2** and **Step3** the evaluation of the conditions to open or close the doors are made in actions *Evaluate_Opening* and *Evaluate_Closing*, respectively; the latter is depicted in Code 54. In these actions, the ST statements make use of the POU *functions* for splitting the complexity in more than one component — in Code 55 the body of the *Test_Closing* POU, where the testing of the closing conditions is made, is presented. Depending on the result of the actions, the transitions after their related steps will determine which, if any, of the right or left doors opening operation, or of the right or left closing operation, was allowed to occur (transitions **T5**, **T6**, **T7** and **T8**, respectively); **T4** and **T9** accounts for the scenarios where the conditions to open or close were not achieved.

```
1 Test_Closing := NOT (train_mode = ATO AND ((NOT TL_RS_OP_ATC OR NOT
    TL_LS_OP_ATC) AND (OP_RS_Console OR OP_LS_Console)));
```

Code 55: POU *function* Test Closing's ST body.

The last step in the SFC program for CDC is **Step8**, which must be be reachable in any of the possible scenarios. The action performed when the execution flow gets here is

Evaluate_Emergency (depicted in Code 56), where it is determined whether the current CDC's state applies for an emergency situation — if any passenger pressed an emergency button or moved an emergency switch at some of the doors and if emergency operations are to be performed. To inquire this the ST statements make use of instances of the POU *function blocks* that test if the conditions to such a situation are satisfied **and** determine the actions to be taken regarding emergency commands. Depending on the result, the CDC may block an opening operation that was going to occur, allow an opening that was not going to occur, ring an alarm, etc.

```
1  IF TL_MechEmg THEN
2    test_MEmg(stopped := speedZero, in_platform := platformSafe,
         doors_closed := TL_DOORS_CL, doors_locked := TL_DOORS_LCK);
3    TL_AUTO_EMG := test_MEmg.authorize; EMG_SIGN := test_MEmg.signalize;
4  ELSIF TL_ButtonEmg_RS OR TL_ButtonEmg_LS THEN
5    IF TL_ButtonEmg_RS THEN
6      emg_side := right_side;
7    ELSIF TL_ButtonEmg_LS THEN
8      emg_side := left_side;
9    END_IF;
10   test_BEmg(lower6 := speedLower6, stopped := speedZero, in_platform :=
         platformSafe, doors_closed := TL_DOORS_CL, doors_locked :=
         TL_DOORS_LCK, selectedSide := selected_side, emgSide := emg_side);
11   TL_AUTO_EMG := test_BEmg.authorize; EMG_SIGN := test_BEmg.signalize;
12   TL_RS_OP := test_BEmg.open_right;
13   TL_AUTO_RS := test_BEmg.autho_open_right;
14   TL_LS_OP := test_BEmg.open_left;
15   TL_AUTO_LS := test_BEmg.autho_open_left;
16 END_IF;
17 emergency_evaluated := TRUE;
```

Code 56: *Action* Evaluate Emergency, at Step8

In Code 56, as said, the evaluation of the emergency conditions is made. In line **1** is tested if the input indicating that an emergency switch was operated is set, in which case the current state is sent to the function block instance (line **2**) that determines whether an emergency situation is *authorized* and if it must be *signalized* (line **3**). In the case where an emergency button was pressed, either in a door in the right or the left side (line **4**), firstly the variable controlling this information about the side is set (lines **5-9**), then the current state is tested in another function block instance (line **10**), that determines if

the *authorization* or *signalizing* are made (line **11**) and also if the doors must be ordered to open or close (lines **12**-**15**).

```
1 IF stopped AND in_platform AND doors_closed AND doors_locked THEN
2     authorize := TRUE; signalize := TRUE;
3 ELSIF NOT stopped AND NOT in_platform AND doors_closed AND doors_locked
      THEN
4     authorize := FALSE; signalize := TRUE;
5 END_IF;
```

Code 57: Function block *Define_MechEmergency*.

The implementation of the POU *function block* that tests the mechanic emergency conditions and what actions must be performed depending on them can be seen in Code 57. If the train is stopped in the platform with its doors closed and locked (line **1**), then the emergency situation is authorized, and the CDC will be ordered to signalize (line **2**) that to the **Cabin** component. If the same conditions were satisfied, expect from the train being stopped, an emergency signal would still be sent, but the emergency situation would not be authorized (lines **3**-**4**). In the case of a mechanic switch actuation the authorization of an emergency scenario will oblige the respective **DCU** component to open the door.

5.2 Applying the method

We used a PLCopen XML representation of the CDC's PLC program as input to our tool, which automatically generates an instance of the PLC Object through the *PLC Reader* module that is itself used as input for the *B Writer* model, automatically generating a B model representing the CDC. This resulting model, still short of the CDC's *safety constraints*, is presented below, not in full, though, in the interest of clarity, but with representative excerpts.

5.2.1 B model

The overall architecture of the B model for CDC is depicted in Figure 38, while the ones for the *function block instances* can be seen in Appendix D.
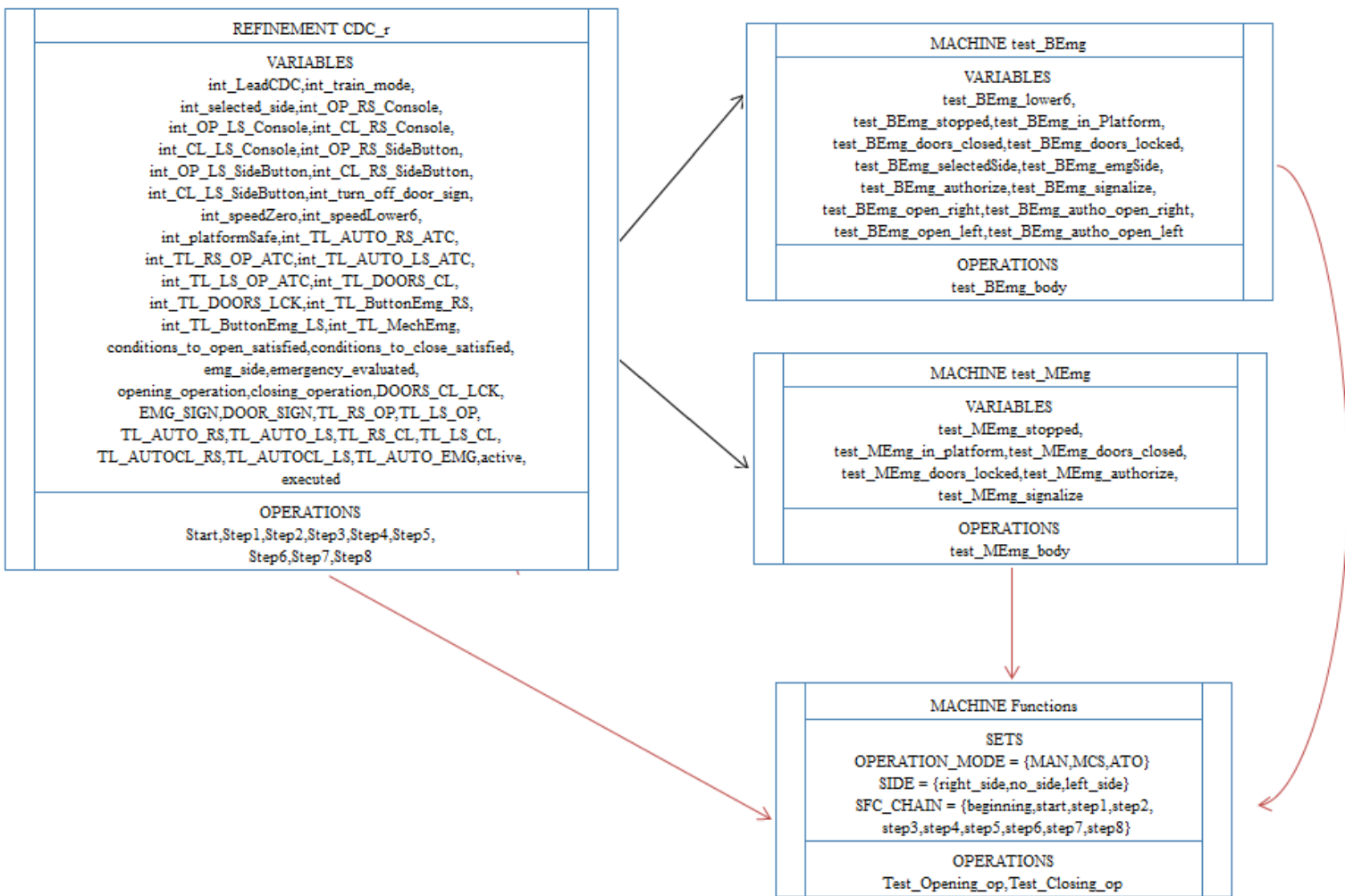
Figure 38: B model for the CDC.

The **static information** built from the declared types and for flow control are presented in the *Functions* machine — OPERATION_MODE, SIDE, enumeration types; and SFC_CHAIN, with beginning, for the first execution cycle, and the steps. The **state** information is present in the machines for the *function block instances* — test_BEmg and test_MEmg — and for the POU *program*, where besides the variables for internal representation of inputs, locals and outputs, the ones for flow control, active and executed, can also be seen.

In the model we can see as well the **structuring mechanisms** at work, with the refinement for CDC *including* the machines for the instances and all the components *seeing* the *Functions* machine, which represents the structure of the PLC program itself.

The **behaviour** part in the model is represented by the operations in each machine, who are created from the respective POUs bodies, all in ST, except for the CDC's one —

the only one that demands numerous operation, one for each of its steps.

```
1  Step8 =
2  BEGIN
3      IF int_TL_MechEmg = TRUE   THEN
4          test_MEmg_body(int_speedZero, int_platformSafe, int_TL_DOORS_CL,
               int_TL_DOORS_LCK);
5          TL_AUTO_EMG := test_MEmg_authorize;
6          EMG_SIGN := test_MEmg_signalize
7      ELSIF (int_TL_ButtonEmg_RS = TRUE or int_TL_ButtonEmg_LS = TRUE)
          THEN
8          IF int_TL_ButtonEmg_RS = TRUE   THEN
9              emg_side := right_side
10         ELSIF int_TL_ButtonEmg_LS = TRUE   THEN
11             emg_side := left_side
12         END;
13         test_BEmg_body(int_speedLower6, int_speedZero, int_platformSafe,
               int_TL_DOORS_CL, int_TL_DOORS_LCK, int_selected_side,
               emg_side);
14         TL_AUTO_EMG := test_BEmg_authorize;
15         EMG_SIGN := test_BEmg_signalize;
16         TL_RS_OP := test_BEmg_open_right;
17         TL_AUTO_RS := test_BEmg_autho_open_right;
18         TL_LS_OP := test_BEmg_open_left;
19         TL_AUTO_LS := test_BEmg_autho_open_left
20     END;
21     emergency_evaluated := TRUE;
22     active := (active − {step1, step2, step3, step4, step5, step6,
          step7}) \/ {step8};
23     executed := step8
24  END
```

Code 58: B operation for Step8

In Code 58 the B operation generated for **Step8** and its associated action, in ST, shown in Code 56, are depicted. From lines **3** to **21** the translations of the ST statements, generated according to the guidelines shown in Chapter 4, are presented. In lines **22** and **23** the *flow control* statements, for **Step8**, generated to ensure that the B machine will behave respecting the defined execution flow in the SFC program, are exhibited. At **Step8**, all its source steps must be deactivated, which happens to be all of them expect from start; and step8 itself must be activated, marking whom the execution flow has just activated, which, for the same principle, guides the assignment of step8 to executed.

At Code 59 the B operation resulting from the translation of the POU *function Test_Closing*, at Code 55, is shown. The output of the operation is named after the POU and represents the return element of *Test_Closing*, a boolean value. The operation itself has the suffix "_op" after the POU's name, which is followed by the names of the parameters passed to the *function* (line **1**). These parameters are typed in the operation's

precondition (line **2**, and, from line **4** to **6**, the translation of the POU's body, as for a POU *function*'s translation, is depicted.

```
1 Test_Closing <-- Test_Closing_op (train_mode, TL_RS_OP_ATC,
     TL_AUTO_RS_ATC, TL_LS_OP_ATC, TL_AUTO_LS_ATC, OP_RS_Console,
     OP_LS_Console) =
2 PRE train_mode : OPERATION_MODE & TL_RS_OP_ATC : BOOL & TL_AUTO_RS_ATC :
       BOOL & TL_LS_OP_ATC : BOOL & TL_AUTO_LS_ATC : BOOL & OP_RS_Console :
       BOOL & OP_LS_Console : BOOL
3 THEN
4     ANY value1 WHERE value1 = bool(not((train_mode = ATO & ((not(
         TL_RS_OP_ATC = TRUE) or not(TL_LS_OP_ATC = TRUE)) & (
         OP_RS_Console = TRUE or OP_LS_Console = TRUE))))) THEN
5         Test_Closing := value1
6     END
7 END
```

Code 59: B operation for POU *function* Test Closing

In the **ANY** statement, an element, value1, is created and defined as the resulting value of the translated ST expression in the POU's body that would be assigned to the return element of the POU (line **1**); therefore, in the scope of **ANY**, the operation's output, Test_Closing, is assigned with value1 (line **5**).

```
1 test_MEmg_body (stopped, in_platform, doors_closed, doors_locked) =
2 PRE
3   stopped : BOOL & in_platform : BOOL & doors_closed : BOOL &
       doors_locked : BOOL
4 THEN
5   int_stopped := stopped; int_in_Platform := in_platform;
       int_doors_closed := doors_closed; int_doors_locked := doors_locked;
6   IF (int_stopped = TRUE & int_in_Platform = TRUE & int_doors_closed =
       TRUE & int_doors_locked = TRUE)
7   THEN
8     test_MEmg_authorize := TRUE; test_MEmg_signalize := TRUE
9   ELSIF (not(int_stopped = TRUE) & not(int_in_Platform = TRUE) &
       int_doors_closed = TRUE & int_doors_locked = TRUE)
10  THEN
11    test_MEmg_authorize := FALSE; test_MEmg_signalize := TRUE
12  END
13 END
```

Code 60: B operation for POU *function block instance* test__MEmg

The generation procedure for the instance *test_MEmg* of the POU *function block Define_MechEmergency* results in the B operation shown in Code 60, where the POU inputs are read in an operation named after the *instance*'s name (line **1**). The statements in the operation firstly makes the setting of the inputs' internal representation (line **5**), then performing the statements resulting from the translation of the POU's body **with the outputs being prefixed of the instance's name** (lines **6-12**).

5.2.2 Safety constraints

Once the B model was generated, the next phase in our method is the insertion of safety constraints in it so we can verify the control system's requirements. This insertion had to be performed manually, as the automatic derivation of formal constraints from natural language requirements is still out of our work's scope.

To translate the safety constraints from the system's documentation into B **invariants**, we used the ProR approach (LADERNBERGER and JASTRAM, 2012), easing the process and assuring reliable traceability; the whole effort is in (BARBOSA, 2010), we present here only some examples.

To the requirement concerning the safe conditions for performing closing operations,

- In ATO mode, the Central Door Controller must not close the doors while receiving the command to open them from the driver push buttons.

we have the resulting B invariant:

```
((conditions_to_close_satisfied = FALSE) =>
    (int_train_mode = ATO & (((not(int_TL_RS_OP_ATC = TRUE) &
        int_TL_AUTO_RS_ATC = TRUE) or (not(int_TL_LS_OP_ATC = TRUE) &
        int_TL_AUTO_LS_ATC = TRUE)) & (int_OP_RS_Console = TRUE or
        int_OP_LS_Console = TRUE)))
)
```

It makes use of the internal representation of inputs (for the train mode, commands from ATC and from the Cabin), as well as of the local variable conditions_to_close_satisfied, whose result is determined by the invocation of the operation Test_Closing_op in the *Functions* machine; therefore, the invariant will concern the seen operation's execution, even though its declaration is in the refinement of CDC, that *sees Functions*.

Another requirements we show here are the ones related with what the CDC must do when a mechanic emergency switch is actuated:

1. The actuation of the mechanic emergency switch with the train stopped in the platform with doors closed and locked shall induce:

   - The central door controller is informed of the actuation of the mechanic emergency switch.

   - In the cabin a sound alarm shall be warned.

   - The central door controller must authorize the unlocking of the door through the emergency switch.

2. The actuation of the mechanic emergency switch with the train in movement in the platform with doors closed and locked shall induce:

   - The central door controller is informed of the actuation of the mechanic emergency switch.

   - In the cabin a sound alarm shall be warned.

3. The actuation of the mechanic emergency switch with the train in movement outside the platform with doors closed and locked shall induce:

   - The central door controller is informed of the actuation of the mechanic emergency switch.

   - In the cabin a sound alarm shall be warned.

   - While the train is in movement and outside the platform, the central door control must not authorize the unlocking of the door through the emergency switch.

We have the following resulting invariants from these requirements:

```
(
    ((int_stopped = TRUE & int_in_Platform = TRUE & int_doors_closed =
        TRUE & int_doors_locked = TRUE) or
    (int_stopped = FALSE & int_in_Platform = TRUE & int_doors_closed =
        TRUE & int_doors_locked = TRUE) or
    (int_stopped = FALSE & int_in_Platform = FALSE & int_doors_closed =
        TRUE & int_doors_locked = TRUE))
    => (test_MEmg_signalize = TRUE)
)
&
(
    ((int_stopped = TRUE & int_in_Platform = TRUE & int_doors_closed =
        TRUE & int_doors_locked = TRUE))
    => (test_MEmg_authorize = TRUE)
)
```

These invariants are to be put into the `test_MEmg` refinement component, representing the respective instance of the POU *function block* where the evaluation of the mechanic emergency conditions are made. The CDC does not have access to this behaviour, so all constraints over it must be accounted in the respective component.

The invariant also makes use of the internal representation of inputs, only that, regarding the current state of the train (whether stopped in the platform with its doors closed and locked), and, from specific valuations of these inputs, forces the value of the *function block instance*'s outputs.

Considering all the requirements for the CDC execution a total of thirteen invariants were inserted into the components of the model. Their distribution through scope can be seen in Table 12.

Subject	Number of invariants
Opening conditions	2
Closing conditions	1
CDC opening commands	1
CDC closing commands	1
Closing signalization	1
Mechanic emergency conditions	4
Button emergency conditions	3

Table 12: Invariants inserted into the B model for CDC.

5.3 Results

We present below the results we obtained from the formal verification of the B model for the CDC after the insertion of several invariants, concerning safety constraints, in it[2]. The results are divided by the category of verification: through *proof obligations* to be discharged with *theorem proving* in order to assess **invariant compliance**, representing that the CDC was executing correctly; and through model checking, also with the definition of LTL formulae, to check structural properties over the model, representing that the CDC was presenting the correct execution..

5.3.1 Proof obligations verification

Several proof obligations were generated to verify the invariants in the components possessing them — `CDC_r`, `test_MEmg_r` and `test_BEmg_r`. The summarization of the proof obligations, relative to the elements of the machines they are associated with, is made through Table 13.

		Proof Obligations	Proven	Unproved
	CDC	329	328	**1**
	Initialisation	1	1	0
	Start	5	5	0
	Step1	234	234	0
	Step2	11	10	0
	Step3	13	**12**	1
Operations	Step4	5	5	0
	Step5	5	5	0
	Step6	5	5	0
	Step7	5	5	0
	Step8	45	45	0
	test_MEmg_body	4	**3**	1
	test_BEmg_body	8	8	0

Table 13: Proof obligations generated to verify the B model for CDC.

In Table 13 we can also see the results for the *theorem proving* effort, made with the AtelierB Theorem Prover, which was not able to verify all the proof obligations: two of them were not discharged, one for `text_MEmg_body` and one for `Step3`. The AtelierB Prover uses four types of *force*, from 0 to 3, with higher forces using mechanisms that consume more CPU and memory. If the proof obligations are not automatically discharged using these mechanisms one may use the *Interactive Theorem Prover*[3] to verify whether the proof obligation was indeed satisfiable, which was not the case for the two unproven

[2]We run the experiments in a computer with a 3.33GHz Core 2 Duo CPU and 4GB of RAM.
[3]A tool to assist the formal proofs.

POs in our project. A proof in *force 0* requires, in average, some seconds to be proven, while more complex ones may take several minutes. Using force 0 we had, in about two seconds, almost the same results as the ones shown in Table 13, except for one proof obligation, for the `Step2` operation, that required the use of *force 3*, being discharged in about tree seconds — this proof obligation concerned the invariant for opening conditions and the `Test_Opening_op` operation, which involved several variables and had to be verified taking into account the use of temporary values, since it was the result of a POU *function* translation, which we stated previously could lead to difficulties during the verification phase.

The proof obligations not discharged were generated to attest the satisfiability of the invariants for **closing safety** and **mechanic emergency actuation**, presented in the previous section; the former through the behaviour in the `Test_Closing_op`, invoked in `Step3`, and the latter through the behaviour performed at `text_MEmg_body`.

Due to the traceability in the model's architecture with the PLC program it represents we could attest that these proof obligations were relative to the POU *function block Define_MechEmergency*'s body and the *Evaluate_Closing* action performed in *Step3*, through the use of the POU *function Test_Closing*'s body.

The impossibility to satisfy the proof obligations led us to realize that *Define_MechEmergency* was not considering the situation when the train was with its doors closed and locked, in platform, but **not stopped**, a situation constrained by the requirements, however not approached in the PLC program. Adding the statements

```
ELSIF NOT stopped AND in_platform AND doors_closed AND doors_locked THEN
    authorize := FALSE; signalize := TRUE;
```

to its body the resulting B translation was able to clear the proof obligation.

With *Test_Closing* the procedure was the same: we realized, through the proof obligation unproven, that the ST expression was not properly considering the commands sent by the ATC: a closing command from the ATC to open or close the doors in one side of the train is composed of two trainlines, one representing the direct command and the other its authorization — e.g, `TL_RS_OP_ATC` and `TL_AUTO_RS_ATC` for a right side opening; in the POU body, however, only the direct trainline — e.g., `TL_RS_OP_ATC` — was being considered, without its authorization counterpart. Changing the ST assignment seen in Code 55, previously in this Chapter, to

```
Test_Closing := NOT (train_mode = ATO AND (((NOT TL_RS_OP_ATC AND
    TL_AUTO_RS_ATC) OR (NOT TL_LS_OP_ATC AND TL_AUTO_LS_ATC)) AND (
    OP_RS_Console OR OP_LS_Console)));
```

made the resulting B translation compliant with the invariant, thus allowing the proving of the remaining proof obligation.

5.3.2 Model checking

With *model checking* we verify if the model contains structural and behavioral problems, testing if it executes as expected — properties to assert this may be defined through LTL and then verified using model checking.

The state space we deal with for the CDC model is quite big, due mostly to the high number of inputs received, so some bounds must be established when performing the model checking, like reducing the number of possible transitions generated from each operation — this, however, decreases our confidence when making claims about the results.

Nonetheless, applying the model checking approach over the already-invariant-compliant model led us to situations of incompleteness, such as the one when verifying the LTL formula

$$\square((executed = step6) \Rightarrow \diamond(executed = step8))$$

which states that the CDC must, *always*, *eventually* evaluate the emergency conditions (Step8) once it performs closing operations (Step6). The formula could not be verified with model checking, which led us to see that from the Step6 operation the execution flow would never go to the Step8 operation, due to the unsatisfiability of Step8's precondition in any scenario after Step6.

The **traceability** of the B model's architecture with the PLC programs guided us to track the error to the wrong definition of transition **T12**, whose logic condition composed part of Step8's precondition. The transition **T12**'s condition, "T12 := **NOT** TL_RS_OP **AND** TL_AUTO_RS;", was preventing the PLC for reaching the final step, **Step8**, whenever it was performing closing operations; changing the condition to "T12 := **NOT** TL_RS_OP **AND NOT** TL_AUTO_RS;" allowed the execution flow to reach **Step8** from **Step6**, what was verified by the satisfiability of the LTL formula in the model checking over the resulting CDC model.

Operation	Number of visits
Initialisation	1
Start	100
Step1	100
Step2	33
Step3	67
Step4	33
Step5	33
Step6	67
Step7	67
Step8	14330
Functions.Test__Opening__op	327144
Functions.Test__Closing__op	1363100
deadlocked	0
explored__but__not__all__transitions__computed	13631
Enabling operations	100
TOTAL__OPERATIONS	1705075
Model Checking time	∼10min

Table 14: Model checking over the CDC model with 100 enabling operations

After this correction in the model we performed model checking to verify deadlock freedom and other properties. We obtained the results shown in Table 14 for the model checking with the number of enabling operations from each previous one in 100. Although it showed no structural issues and covered all the operations, a high number of transitions was not computed, which led us to try the model checking with a higher number of enabling operations, aiming full coverage.

We incrementally increased the number of enabling operations so we could test the relation between time and resources spent and state space covered, until with a limit of 1000 we reached a much lesser number of transitions not computed, still not finding any more structural problems; however, the computer performing the procedure run out of memory due to the huge state space that had to be explored. The results are summarized in Table 15.

5.4 Conclusions

We presented a case study in a real railway application where our approach was applied with success. We were able to attest the efficacy of the automatic provers and verify the safety constraints of the project. Another feature checked was the traceability, since the errors found in the formal model corresponded, indeed, to the errors in the real system (the PLC program), which were easily traceable. Therefore, despite being a small example, the case study successfully illustrates how the issues related to the architecture

Operation	Number of vistis
Initialisation	1
Start	1000
Step1	1000
Step2	195
Step3	394
Step4	53
Step5	53
Step6	126
Step7	126
Step8	97456
Functions.Test__Opening_op	973872
Functions.Test__Closing_op	7502976
Computer run out of memory explored__but__not__all__transitions__computed	91
Enabling operations	1000
TOTAL_OPERATIONS	8541252
Model Checking time	~30min

Table 15: Model checking over the CDC model with 1000 enabling operations

of the model to represent the PLC programs work properly.

The automatic generation of PLC models had a huge impact in the work, facilitating the process; this is a very desired feature. Some works approaching the CDC, aiming to provide formal models for its modeling, have been made before and the construction of the formal model to represent it generally took several weeks, which took no time with our approach. Of course that we have the setback of lacking the safety constraints, that must be derived from the requirements and will require time, but once they could be provided by others the verification procedure can be done very rapidly — as was in this case study, since the INVARIANTS were already available, only an adaptation procedure being necessary for its use in the generated model.

6   Conclusion

We have overviewed a method to carry out formal verification of PLC programs, according to the IEC 61131-3 standard, through the automatic generation of a B specification. Safety constraints are inserted in the formal model and then verified through theorem proving; we also verify structural properties and if the PLC presents the expected behavior by performing model checking and using LTL formulae. Thus we increase the level of confidence on an application, having correctness according to the specification as a realistic and achievable goal.

Another key point of our approach is that it allows the users to generate the B models **automatically** from the PLC programs, only lacking the safety properties, which boosts the process of formal verification of such programs, significantly reducing the hard work of designing and modeling the formal model that prevents several formal approaches from being easily inserted in industrial projects.

We have also presented a case study in a real railway application where our method was applied with success. We were able to attest the efficacy of the automatic model generation, prove the safety constraints compliance and validate the PLC's execution through model checking.

The research that led to this had as initial goals to make generation of verified LD code from B specifications, but we decided to change the "direction" of the usual procedure: we would derive B specifications **from** LD code. This decision was taken aiming to **ease** the insertion of the approach in industrial developments: formal methods struggle to be adopted in industry mostly for the intrinsic difficult in the creation of their models by engineers, since the majority of them is not familiar with the necessary mathematical thinking for dealing with formal verification; therefore, taking away from them the effort of creating such models, simply having them generated automatically from the control programs they are used to develop, would increase the chances of the developed method being absorbed and applied for the verification of PLCs, although it is still necessary to

make the *results* of the formal verification more readable to them.

Once we defined the central goal of the work we also decided to expand the scope: instead of only LD we would deal with all the IEC 61131-3 standard, therefore drastically extending our approach's reach. The PLCopen standard was instrumental in this decision and on its performance, as it eased the development of a common intermediary structure for the different languages to be used in the B generation process. The automation of the approach was accounted as key since the beginning of the development, as we were experienced with the usual difficulties when trying to integrate formal methods without automation in the industrial development.

As we stated before, we successfully implemented our method to automatically generate the B models from PLC programs in SFC, ST, FBD and LD, short, as for the IEC 61131-3 standard, only of the IL programs. A big effort was devoted to the construction of our PLC Object in order for it to be able to represent the elements of all these languages; this required the study of the PLCopen standard in depth. We also faced difficulties to define the guidelines for the B generation, struggling to build the actions in the PLC readers so the organization of the information in the intermediary structure would be effective in reducing the semantic gap between it and the B notation. A careful study analysis of the IEC 61131-3 standard was determinant to provide us with the required understanding so the B model could indeed represent the behavior of the PLC programs; toy problems that helped the preliminaries evaluations of the method were also of great importance.

During the development of the research we have also published papers in formal method conferences. A first sketch, with a very different architecture for the B model, was presented in (BARBOSA and DÉHARBE, 2011), while a more developed version of the work was shown in (BARBOSA and DÉHARBE, 2012a). A mature presentation of the work was published in (BARBOSA and DÉHARBE, 2012b), with an excerpt of the CDC case study used to evaluate the method; the architecture of the B model generated presented in this paper was very close to our final version, but it yet needed some work on; the case study was exhibited with detail and, more interestingly, the definition and usage of LTL formulae was shown in depth and with several examples.

While developing our approach we have also dealt with a PLC language deviating from the IEC 61131-3 standard, employed in high-speed train's scenario, the WinGUIDEP language: it is similar to SFC, using ST in its elements, although presenting some extensions — one of them is a new element introduced to the syntax of SFC: the *subgraph*, which,

like a SFC MacroStep, encapsulates a new SFC program inside of it; however, when the subgraph is reached in the execution flow the current program **stops**, while the subgraph runs. Modifications were necessary in the parser to accommodate such differences, so they would be represented as additional data in the PLC Object generated; however, we have not yet adapted the process of the B model generation to make room for these deviations of the standard. Nonetheless, this type of effort plants the seeds for an important future work: the expansion of the method's reach to handle not only PLC languages deviating from the IEC 61331-3 standard, but languages for different control platforms, such as VERILOG, VHDL, C for microcontrollers, etc.; embedded systems control programming in general, in a nutshell. Adaptations in the parsers would be mandatory, but using the guidelines developed so far we would ease significantly such effort rather than starting from scratch.

Regarding the restrictions of the approach, the future work lies mostly in improving the way the method has to deal with some issues, such as the absence of coverage for the IL programs, how to enhance the process of specifying the safety constraints and *time handling*. The derivation of safety constraints for now is made manually by the natural language requirements of the application, but techniques such as *annotating* the PLC code with the constraints to be verified could allow us to generate the B invariants *along* with the generation of the B models themselves. Concerning time, another point of interest, besides the capacity of time handling itself in the model, is about *continuous-time* systems; since we deal only with discrete-time control programs, for PLCs, this diminishes the broadness of our approach, which led us to plan studying ways to overcome this issue, like the one presented in (ABRIAL et al., 2012), with an approach using the *Event-B* formal method, which has many similarities with the B Method itself. The capacity of modeling the environment with whom the control program interacts is also facilitated by Event-B constructs, which, if used in combination with B specifications, could notably increase our method's reach.

As for the verification performance and analysis, a methodology could be developed to guide the correction of the PLC programs verified through our approach. With traceability provided, results such as a proof obligation not being discharged or a LTL formula not satisfied, presented in a way that did not require much expertise of the formal techniques, if inserted in an easy-to-follow methodology would boost the whole approach's application. For the verification procedures themselves, the similarities between control operations could be used to reduce the verification effort by defining some default *tactics* to prove a specific structure of proof obligations, a feature provided by provers like the AtelierB's

prover — once a guideline was applied to discharge a proof obligation, the same one could be used in another unproven set of proof obligations to try their clearance. These *tactics* could be stored as a library to be used during the tool's execution by a user request, once they were applicable to the faced scenario.

Concerning the translation between the PLC programs and the B specifications, a possible way to improve the confidence on it is applying testing techniques: validate the process through the re-generation of the PLC programs back from the B models, therefore attesting the traceability between them with more strength. Full correctness of the translation process is always desirable, even more when dealing with the rigorous scenarios of critical systems, but it is often in low priority during an approach's development due to the intrinsic difficult in proving such a property, requiring large amounts of time and work; therefore, we left the formal proving of the translation for our method in a far horizon, relying for now on the visual validation of the translation procedure through animation on the model and planning to strengthen the procedure with testing techniques.

Finally, the performance and evaluation of more case studies is also of great importance to enhance the approach developed. We have performed a case study in a railway setting, but other fields with similar control applications, such as the petroleum field, can as well be targeted by our method and provide interest results.

References

ABRIAL et al., 2012 ABRIAL, J., SU, W., and ZHU, H. (2012). Formalizing hybrid systems with event-b. In *Proceedings of the ABZ 2012*.

ABRIAL, 1996 ABRIAL, J. R. (1996). *The B Book: Assigning Programs to Meanings.* Cambridge University Press.

ABRIAL, 2006 ABRIAL, J. R. (2006). Formal methods in industry: achievements, problems, future. In *Proceedings of the 28th international conference on Software Engineering.*

AMEY, 2004 AMEY, P. (2004). Dear sir, yours faithfully: an everyday story of formality. In *Proceedings of 12th Safety-Critical Systems Symposium.*

BARBOSA, 2010 BARBOSA, H. (2010). Desenvolvendo um sistema crítico através de formalização de requisitos utilizando o método b. Monografia de Graduação, DIMAp/UFRN.

BARBOSA and DÉHARBE, 2011 BARBOSA, H. and DÉHARBE, D. (2011). Towards formal verification of plc programs. In *14th Brazilian Symposium on Formal Methods: Short Papers.*

BARBOSA and DÉHARBE, 2012a BARBOSA, H. and DÉHARBE, D. (2012a). An approach using the b method to formal verification of plc programs in an industrial setting. In *15th Brazilian Symposium on Formal Methods: Short Papers.*

BARBOSA and DÉHARBE, 2012b BARBOSA, H. and DÉHARBE, D. (2012b). Formal verification of plc programs using the b method. In *Proceedings of the ABZ 2012.*

BLACKBURN et al., 2001 BLACKBURN, P., RIJKE, M., and VENEMA, Y. (2001). *Modal Logic.* Cambridge University Press, 1st edition.

BURDY et al., 2012 BURDY, L., DUFOUR, J. L., and LECOMTE, T. (2012). The b method takes up floating-point numbers. In *Proceedings of Embedded Real Time Software and Systems (ERTS).*

CABRAL and SAMPAIO, 2006 CABRAL, G. and SAMPAIO, A. (2006). Formal specification generation from requirement documents. In *Proceedings of Brazilian Symposium on Formal Methods.*

CAVALCANTI et al., 2011 CAVALCANTI, A., CLAYTON, P., and O'HALLORAN, C. (2011). From control law diagrams to ada via *Circus. Formal Aspects of Computing*, pages 465–512.

CLARKE et al., 1999 CLARKE, E. M., GRUMBERG, O., and PELED, D. (1999). *Model checking.* MIT Press, 1st edition.

FARINES et al., 2011 FARINES, J.-M., QUEIROZ, M. H. d., ROCHA, V. G., and CARPES, A. M. M. (2011). A model-driven engineering approach to formal verification of plc programs. In *Proceedings of IEEE EFTA'2011.*

FME, 2012 FME (2012). Fme.: What is formal methods. Available at: `http://www.fmeurope.org/?pageid=2`.

FREY and LITZ, 2000 FREY, G. and LITZ, L. (2000). Formal methods in plc programming. In *Proceedings of the IEEE Conference on Systems, Man and Cybernetics (SMC'2000).*

IEC, 2003 IEC (2003). *IEC 61131 standard.* International Electrotechnical Commission.

KRON, 2003 KRON, H. (2003). On the evaluation of risk acceptance principles. In *Proceedings of the 19th Dresden Conference on Traffic and Transportation Science.*

LADERNBERGER and JASTRAM, 2012 LADERNBERGER, L. and JASTRAM, M. (2012). Requirements traceability between textual requirements and formal models using pror. In *Proceedings of the Posters & Tools demos sessions, ABZ 2102.*

Lahiri and Bryant, 2003 Lahiri, S. K. and Bryant, R. E. (2003). Deductive verification of advanced out-of-order microprocessors. *Proceedings 15th International COnference on Computer-Aided Verification (CAV)*, 2725:341–354.

LECOMTE et al., 2007 LECOMTE, T., SERVAT, T., and POUZANCRE, G. (2007). Formal methods in safety-critical railway systems. In *Proceedings of Brazilian Symposium on Formal Methods.*

LEUSCHEL and BUTLER, 2003 LEUSCHEL, M. and BUTLER, M. (2003). The prob animator and model checker. Available at: `http://www.stups.uni-duesseldorf.de/ProB/`.

MEDEIROS JR, 2009 MEDEIROS JR, V. (2009). Aplicação do método b ao projeto formal de software embarcado. Master's thesis, UFRN.

PARK et al., 2008 PARK, S., PARK, C., WANG, G., KWAK, J., and YEO, S. (2008). Plcstudio: Simulation based plc code verification. In *Proceedings of the 2008 Winter Simulation Conference.*

PARNAS, 2010 PARNAS, D. (2010). Really rethinking 'formal methods'. *COMPUTER*, pages 28–34.

PLCOPEN, 2009 PLCOPEN (2009). *PLCopen: XML Formats for IEC 61131-3.* PLCopen Technical Committee 6.

RAUSCH and KROGH, 1998 RAUSCH, M. and KROGH, B. H. (1998). Formal verification of plc programs. In *Proceedings of the American Control Conference.*

SCHNEIDER, 2001 SCHNEIDER, S. (2001). *The B-method, an introduction.* Palgrave, 1st edition.

SOLIMAN and FREY, 2009 SOLIMAN, D. and FREY, G. (2009). Verification and validation of safety applications based on plcopen safety function blocks using timed automata in uppaal. In *Proceedings of the second IFAC workshop on dependable control of discrete systems (DCDS)*.

SÜFLOW and DRECHSLER, 2008 SÜFLOW, A. and DRECHSLER, R. (2008). Verification of plc programs using formal proof techniques. In *Proceedings of FORMS/FORMAT (2008)*.

VÖLKER and KRÄMER, 1999 VÖLKER, N. and KRÄMER, B. J. (1999). Modular verification of function block based industrial control systems. In *Proceedings of IFAC/IFIP (1999)*.

WANG et al., 2012 WANG, R., ZHOU, M., YIN, L., and BOZGA, M. (2012). Modeling and validation of plc-controlled systems: A case study. In *Proceedings of IEEE Sixth International Symposium on Theoretical Aspects of Software Engineering*.

APPENDIX A – Textual languages constructs

A.1 IL Constructs

Operator	Modifiers	Operand	Semantics
LD	N		Set current result equal to operand
ST	N		Store current result to operand location
S		BOOL	Set boolean operand to 1
R		BOOL	Set boolean operand to 0
AND	N, (	BOOL	Boolean AND
&	N, (	BOOL	Boolean AND
OR	N, (	BOOL	Boolean OR
XOR	N, (	BOOL	Boolean Exclusive OR
ADD	(		Addition
SUB	(		Subtraction
MUL	(		Multiplication
DIV	(		Division
GT	(		Comparison: >
GE	(		Comparison: >=
EQ	(		Comparison: =
NE	(		Comparison: <>
LE	(		Comparison: <=
LT	(		Comparison: <
JMP	C, N	LABEL	Jump to label
CAL	C, N	NAME	Call POU
RET	C, N		Return from called POU
)			Evaluate deferred operation

Table 16: Elements of the instructions in IL.

A.2 ST constructs

Operation	Structure
Parenthesization	(expression)
Exponentiation	**
Negation	-
Complement	NOT
Multiply	*
Divide	/
Modulo	MOD
Add	+
Subtract	-
Comparison	<, >, <=, >=
Equality	=
Inequality	<>
Boolean AND	&, AND
Boolean Exclusive OR	XOR
Boolean OR	OR

Table 17: ST operations and their symbols

Statement type	Structure
Assignment	variable := expression;

Function invocation	variable := function_identifier(argument list);
	
	variable1 := function_identifier(argument list) + variable2;

Function block invocation and FB output usage	fb_identifier(argument list);
	
	variable := fb_identifier.output;

RETURN	RETURN;
IF	IF logic expression THEN statements ELSIF logic expression THEN statements ELSE statements END_IF;
CASE	CASE variable OF value1 : statements; ... valueN : statemetns; ELSE statements; END_CASE;
FOR	FOR initialize counter TO upper bound DO statements END_FOR;
EXIT	EXIT;
EMPTY STATEMENT	;

Table 18: ST statements and examples of their usage.

APPENDIX B – Textual declarations in IEC 61131-3

B.1   POU declaration

A PLC program declaration in the textual form of IEC is

```
1 plc_declaration ::= data_type_declaration
2   | function_declaration | function_block_declaration
3   | program_declaration | configuration_declaration
```

Code 61: Production rules for the textual declaration of a PLC program

The declaration of POUs in the textual form of IEC is

```
1 function_name ::= standard_function_name | derived_function_name
2 standard_function_name ::= <predefined in the standard>
3 derived_function_name ::= identifier
4 function_declaration ::=
5     'FUNCTION' derived_function_name ':'
6         (elementary_type_name | derived_type_name)
7       { io_var_declarations | function_var_decls }
8       function_body
9     'END_FUNCTION'
10 io_var_declarations ::= input_declarations | output_declarations |
11     input_output_declarations
12 function_var_decls ::= 'VAR' ['CONSTANT')
13     var2_init_decl ';' {var2_init_decl ';'} 'END_VAR'
14 function_body ::= ladder_diagram | function_block_diagram
15     | instruction_list | statement_list | <other languages>
16 var2_init_decl ::= var1_init_decl | array_var_init_decl
17     | structured_var_init_decl | string_var_declaration
```

Code 62: Production rules for the textual declaration of a POU *function*

```
 1 function_block_type_name ::= standard_function_block_name
 2     | derived_function_block_name
 3 standard_function_block_name ::= <predefined in the standard>
 4 derived_function_block_name ::= identifier
 5 function_block_declaration ::=
 6     'FUNCTION_BLOCK' derived_function_block_name
 7       { io_var_declarations | other_var_declarations }
 8       function_block_body
 9     'END_FUNCTION_BLOCK'
10 other_var_declarations ::= external_var_declarations | var_declarations
11     | retentive_var_declarations | non_retentive_var_declarations
12     | temp_var_decls | incompl_located_var_declarations
13 temp_var_decls ::=
14     'VAR_TEMP'
15       temp_var_decl ';'
16       {temp_var_decl ';'}
17     'END_VAR'
18 non_retentive_var_decls ::=
19     'VAR' 'NON_RETAIN'
20       var_init_decl ';'
21       {var_init_decl ';'}
22     'END_VAR'
23 function_block_body ::= sequential_function_chart | ladder_diagram
24     | function_block_diagram | instruction_list | statement_list
25     | <other languages>
```

Code 63: Production rules for the textual declaration of a POU *function block*

```
 1 program_type_name :: = identifier
 2 program_declaration ::=
 3     'PROGRAM' program_type_name
 4       { io_var_declarations | other_var_declarations
 5         | located_var_declarations | program_access_decls }
 6       function_block_body
 7     'END_PROGRAM'
```

Code 64: Production rules for the textual declaration of a POU *program*

B.2   SFC declaration

```
1 sequential_function_chart ::= sfc_network {sfc_network}
2 sfc_network ::= initial_step {step | transition | action}
3 initial_step ::=
4     'INITIAL_STEP' step_name ':' {action_association ';'} 'END_STEP'
5 step ::= 'STEP' step_name ':' {action_association ';'} 'END_STEP'
6 step_name ::= identifier
7 action_association ::=
8     action_name '(' [action_qualifier) {',' indicator_name} ')'
9 action_name ::= identifier
10 action_qualifier ::=
11     'N' | 'R' | 'S' | 'P' | timed_qualifier ',' action_time
12 timed_qualifier ::= 'L' | 'D' | 'SD' | 'DS' | 'SL'
13 action_time ::= duration | variable_name
14 indicator_name ::= variable_name
15 transition ::= 'TRANSITION'
16     [transition_name) ['(' 'PRIORITY' ':=' integer ')')
17     'FROM' steps 'TO' steps
18      transition_condition
19     'END_TRANSITION'
20 transition_name ::= identifier
21 steps ::= step_name | '(' step_name ',' step_name {',' step_name} ')'
22 transition_condition ::= ':' simple_instruction_list | ':=' expression
23     ';' | ':' (fbd_network | rung)
24 action ::= 'ACTION' action_name ':'
25             function_block_body
26           'END_ACTION'
```

Code 65: Production rules for the textual declaration of a SFC *program*

APPENDIX C – PLC object extra information

C.1   Class diagramas to the PLC Object

A picture of the whole structure can be seen in Figure 39. It represents the totality of the PLCopen XML standard.

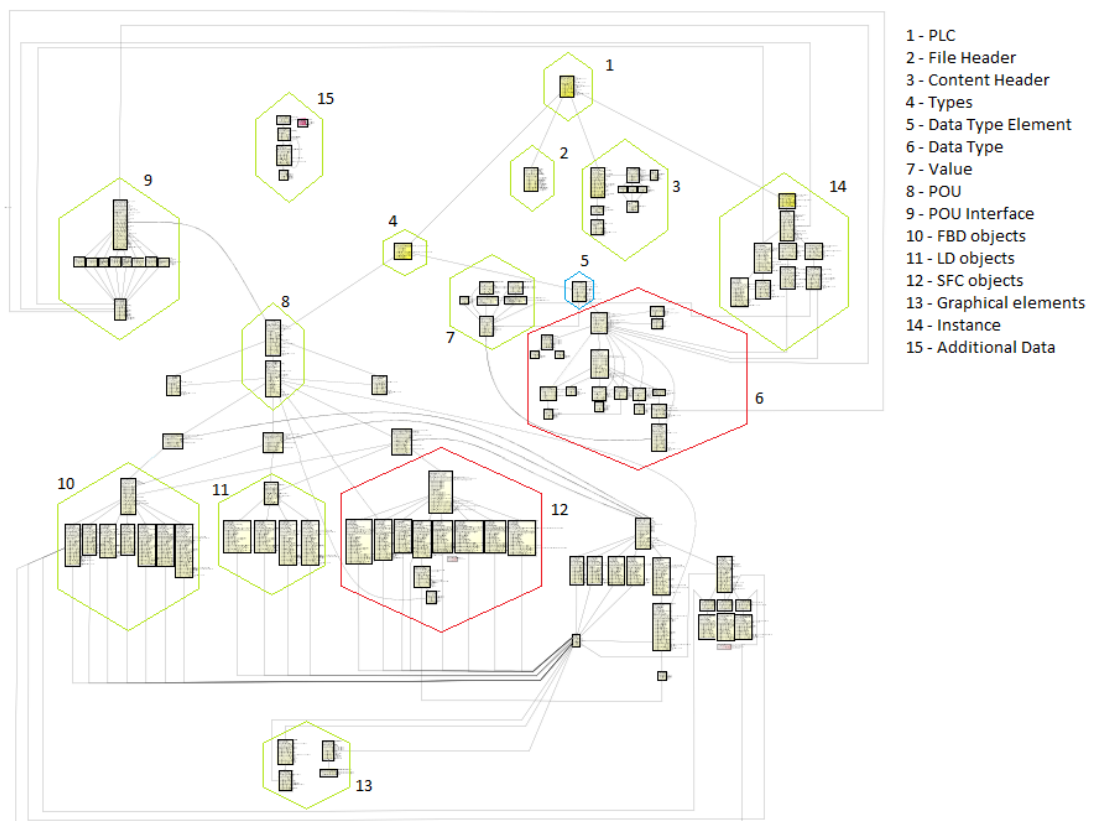

Figure 39: The whole PLC Object, representing the PLC with is header, types and instance information.

We can see in Figure 40 one part of this diagram "zoomed up", the one where the *SFCOBjects* class is presented.
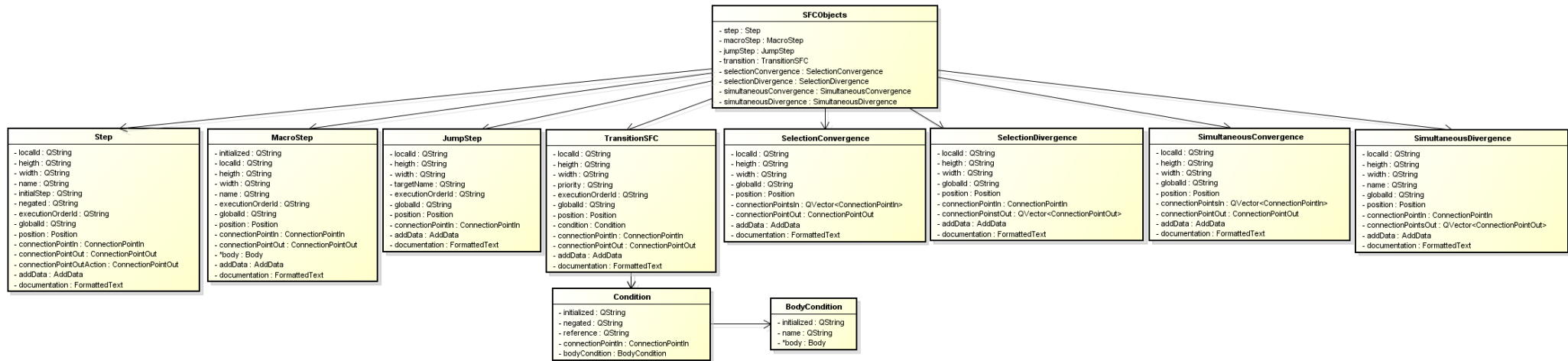
SFCObjects

- step : Step
- macroStep : MacroStep
- jumpStep : JumpStep
- transition : TransitionSFC
- selectionConvergence : SelectionConvergence
- selectionDivergence : SelectionDivergence
- simultaneousConvergence : SimultaneousConvergence
- simultaneousDivergence : SimultaneousDivergence

Step

- localId : QString
- heigth : QString
- width : QString
- name : QString
- initialStep : QString
- negated : QString
- executionOrderId : QString
- globalId : QString
- position : Position
- connectionPointIn : ConnectionPointIn
- connectionPointOut : ConnectionPointOut
- connectionPointOutAction : ConnectionPointOut
- addData : AddData
- documentation : FormattedText

MacroStep

- initialized : QString
- localId : QString
- heigth : QString
- width : QString
- name : QString
- executionOrderId : QString
- globalId : QString
- position : Position
- connectionPointIn : ConnectionPointIn
- connectionPointOut : ConnectionPointOut
- *body : Body
- addData : AddData
- documentation : FormattedText

JumpStep

- localId : QString
- heigth : QString
- width : QString
- targetName : QString
- executionOrderId : QString
- globalId : QString
- position : Position
- connectionPointIn : ConnectionPointIn
- addData : AddData
- documentation : FormattedText

TransitionSFC

- localId : QString
- heigth : QString
- width : QString
- priority : QString
- executionOrderId : QString
- globalId : QString
- condition : Condition
- connectionPointIn : ConnectionPointIn
- connectionPointOut : ConnectionPointOut
- addData : AddData
- documentation : FormattedText

SelectionConvergence

- localId : QString
- heigth : QString
- width : QString
- globalId : QString
- position : Position
- connectionPointsIn : QVector<ConnectionPointIn>
- connectionPointOut : ConnectionPointOut
- addData : AddData
- documentation : FormattedText

SelectionDivergence

- localId : QString
- heigth : QString
- width : QString
- globalId : QString
- position : Position
- connectionPointIn : ConnectionPointIn
- connectionPointOut : QVector<ConnectionPointOut>
- addData : AddData
- documentation : FormattedText

SimultaneousConvergence

- localId : QString
- heigth : QString
- width : QString
- globalId : QString
- position : Position
- connectionPointsIn : QVector<ConnectionPointIn>
- connectionPointOut : ConnectionPointOut
- addData : AddData
- documentation : FormattedText

SimultaneousDivergence

- localId : QString
- heigth : QString
- width : QString
- name : QString
- globalId : QString
- position : Position
- connectionPointIn : ConnectionPointIn
- connectionPointsOut : QVector<ConnectionPointOut>
- addData : AddData
- documentation : FormattedText

Condition

- initialized : QString
- negated : QString
- reference : QString
- connectionPointIn : ConnectionPointIn
- bodyCondition : BodyCondition

BodyCondition

- initialized : QString
- name : QString
- *body : Body

Figure 40: The class SFCObjects, representing the elements that are used in a SFC program.

ActionBlock

- localId : QString
- negated : QString
- height : QString
- width : QString
- executionOrderId : QString
- globalId : QString
- position : Position
- connectionPointIn : ConnectionPointIn
- actions : QVector<ActionCommonObject>
- addData : AddData
- documentation : FormattedText

ActionCommonObject

- initialized : QString
- localId : QString
- qualifier : QString
- height : QString
- width : QString
- duration : QString
- indicator : QString
- executionOrderId : QString
- globalId : QString
- relPosition : Position
- reference : Reference
- *inlineBody : Body
- connectionPointOut : ConnectionPointOut
- addData : AddData
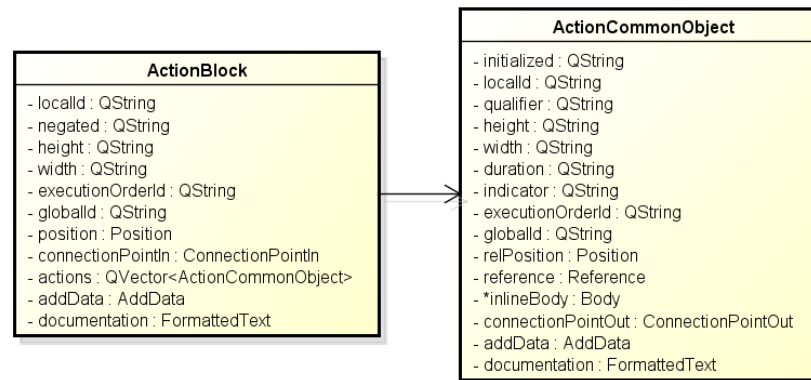- documentation : FormattedText

Figure 41: The class ActionBlock and their ActionCommonObjects associated, representing the blocks of actions that may be associated with a Step object in a SFC body.

In Figures 42 and 43 the *LDOBjects FBDOBjects* classes are presented in more detail.
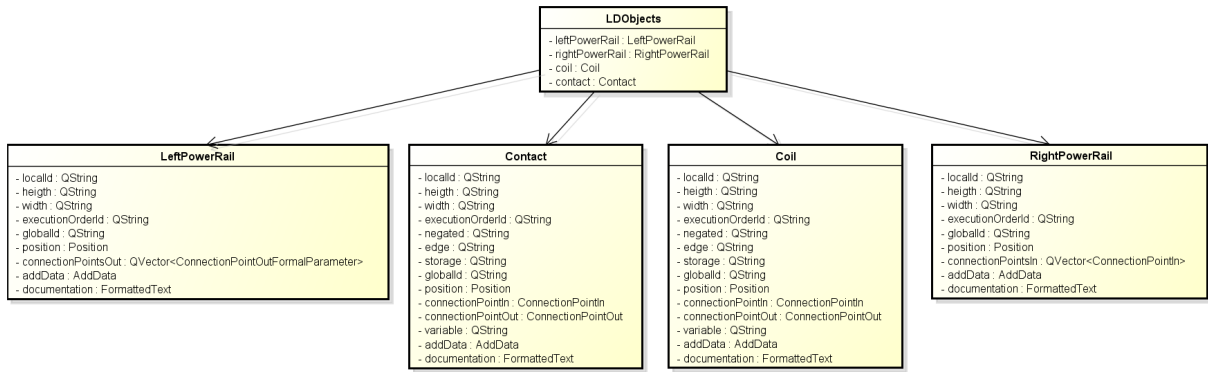

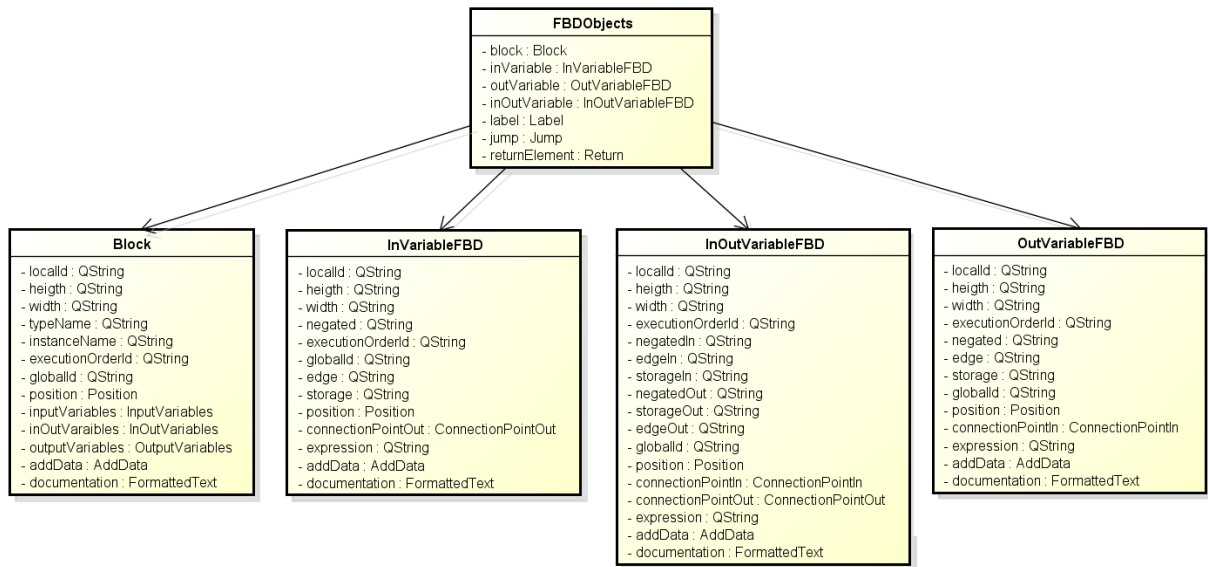Figure 42: The class LDObjects, representing the elements that are used in a LD program.

Figure 43: The class FBDObjects, representing the elements that are used in a FBD program.

APPENDIX D – CDC extra information

D.1 CDC interface

Name	Class	Type
LeadCDC	Input	BOOL
train_mode	Input	OPERATION_MODES
selected_side	Input	SIDE
OP_RS_Console	Input	BOOL
OP_LS_Console	Input	BOOL
CL_RS_Console	Input	BOOL
CL_LS_Console	Input	BOOL
OP_RS_SideButton	Input	BOOL
OP_LS_SideButton	Input	BOOL
CL_RS_SideButton	Input	BOOL
CL_LS_SideButton	Input	BOOL
turn_off_door_sign	Input	BOOL
speedZero	Input	BOOL
speedLower6	Input	BOOL
platformSafe	Input	BOOL
TL_AUTO_RS_ATC	Input	BOOL
TL_RS_OP_ATC	Input	BOOL
TL_AUTO_LS_ATC	Input	BOOL
TL_LS_OP_ATC	Input	BOOL
TL_DOORS_CL	Input	BOOL
TL_DOORS_LCK	Input	BOOL
TL_ButtomEmg_RS	Input	BOOL
TL_ButtomEmg_LS	Input	BOOL
TL_MechEmg	Input	BOOL

Name	Class	Type	Initial Value
conditions_to_open_satisfied	Local	BOOL	FALSE
conditions_to_close_satisfied	Local	BOOL	TRUE
emg_side	Local	SIDE	no_side
emergency_evaluated	Local	BOOL	FALSE
opening_operation	Local	BOOL	FALSE
closing_operation	Local	BOOL	FALSE
test_MEmg	Local	Define_MechEmergency	
test_BEmg	Local	Define_ButtonEmg	

Name	Class	Type
DOORS_CL_LCK	Output	BOOL
EMG_SIGN	Output	BOOL
DOOR_SIGN	Output	SIDE
TL_RS_OP	Output	BOOL
TL_LS_OP	Output	BOOL
TL_AUTO_RS	Output	BOOL
TL_AUTO_LS	Output	BOOL
TL_RS_CL	Output	BOOL
TL_LS_CL	Output	BOOL
TL_AUTOCL_RS	Output	BOOL
TL_AUTOCL_LS	Output	BOOL
TL_AUTO_EMG	Output	BOOL

Table 19: The interface of CDC, presenting its input, local and output variables.

Name	Class	Type
speedLower6	Input	BOOL
speedZero	Input	BOOL
platformSafe	Input	BOOL
train_mode	Input	OPERATION_MODES
return type		BOOL

Table 20: Interface of *Test_Opening*

Name	Class	Type
train_mode	Input	OPERATION_MODES
TL_RS_OP_ATC	Input	BOOL
TL_RS_OP_ATC	Input	BOOL
TL_AUTO_RS_ATC	Input	BOOL
TL_LS_OP_ATC	Input	BOOL
TL_AUTO_LS_ATC	Input	BOOL
OP_RS_Console	Input	BOOL
OP_LS_Console	Input	BOOL
return type		BOOL

Table 21: Interface of *Test_Closing*

Name	Class	Type
stopped	Input	BOOL
in_platform	Input	BOOL
doors_closed	Input	BOOL
doors_locked	Input	BOOL
authorize	Output	BOOL
signalize	Output	BOOL

Table 22: Interface of *Define_MechEmergency*

Name	Class	Type
lower6	Input	BOOL
stopped	Input	BOOL
in_platform	Input	BOOL
doors_closed	Input	BOOL
doors_locked	Input	BOOL
selectedSide	Input	SIDE
emgSide	Input	SIDE
authorize	Output	BOOL
signalize	Output	BOOL
open_right	Output	BOOL
autho_open_right	Output	BOOL
open_left	Output	BOOL
autho_open_left	Output	BOOL

Table 23: Interface of *Define_ButtonEmergency*

D.2 Doors system elements
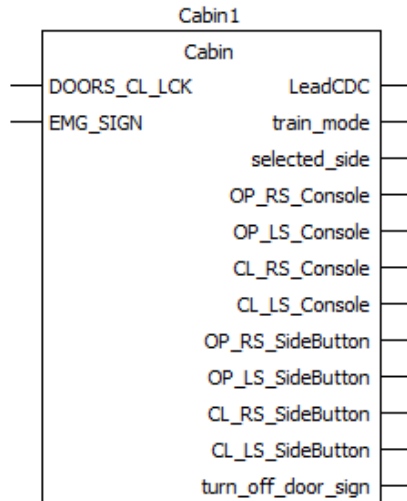
Figure 44: All the components of the train's doors system and their interactions with each other.
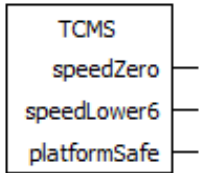
Figure 45: All the components of the train's doors system and their interactions with each other.
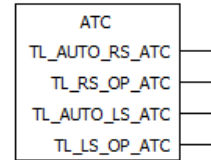

Figure 46: All the components of the train's doors system and their interactions with each other.
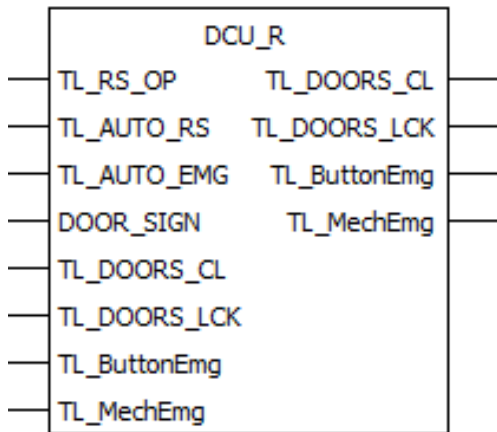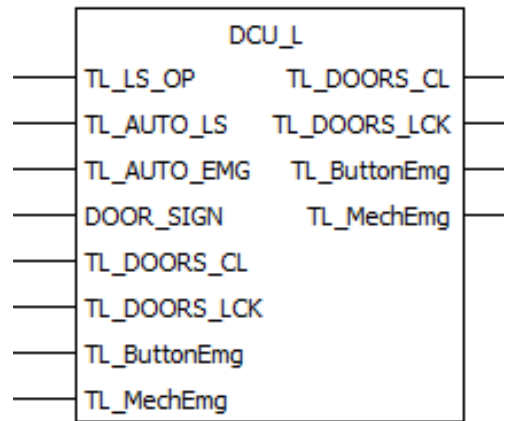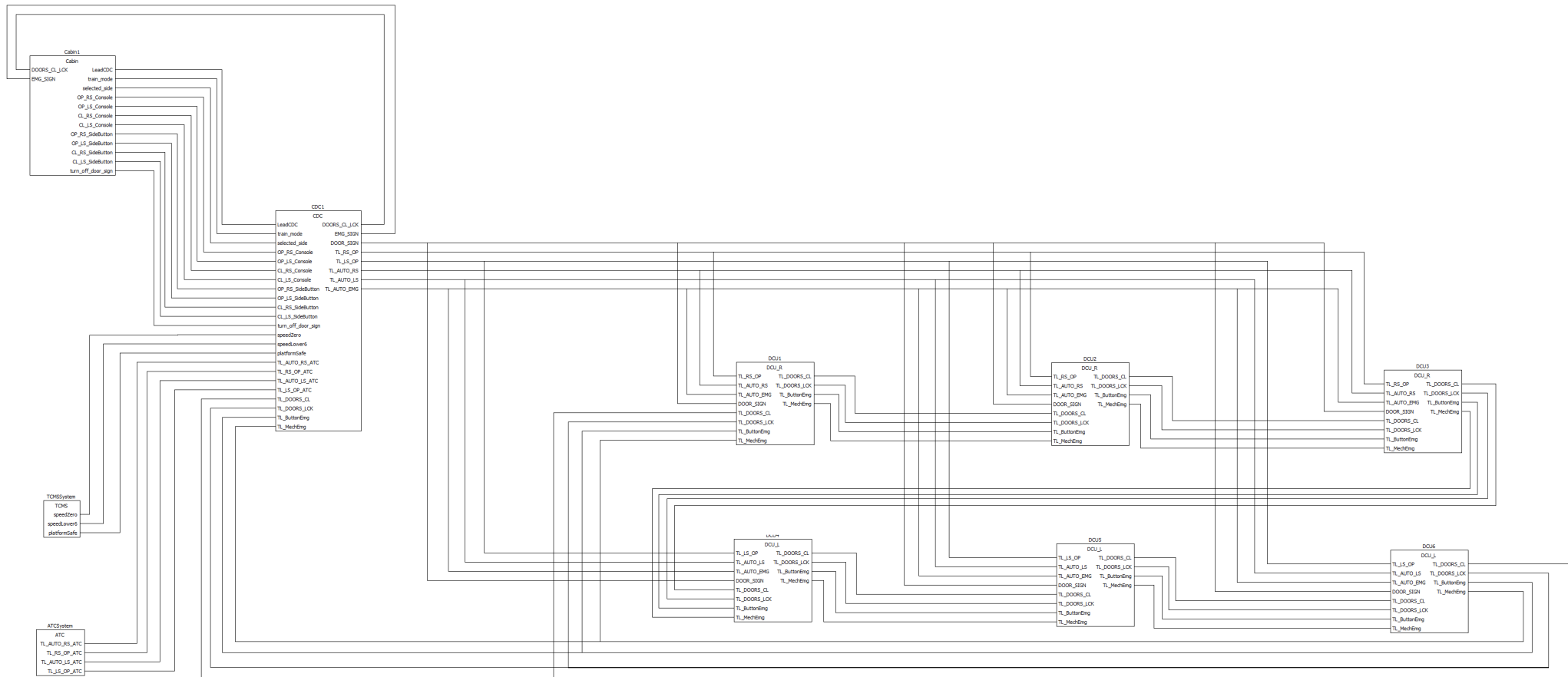

Figure 47: All the components of the train's doors system and their interactions with each other.

Figure 48: All the components of the train's doors system and their interactions with each other.

Figure 49: All the components of the train's doors system and their interactions with each other.
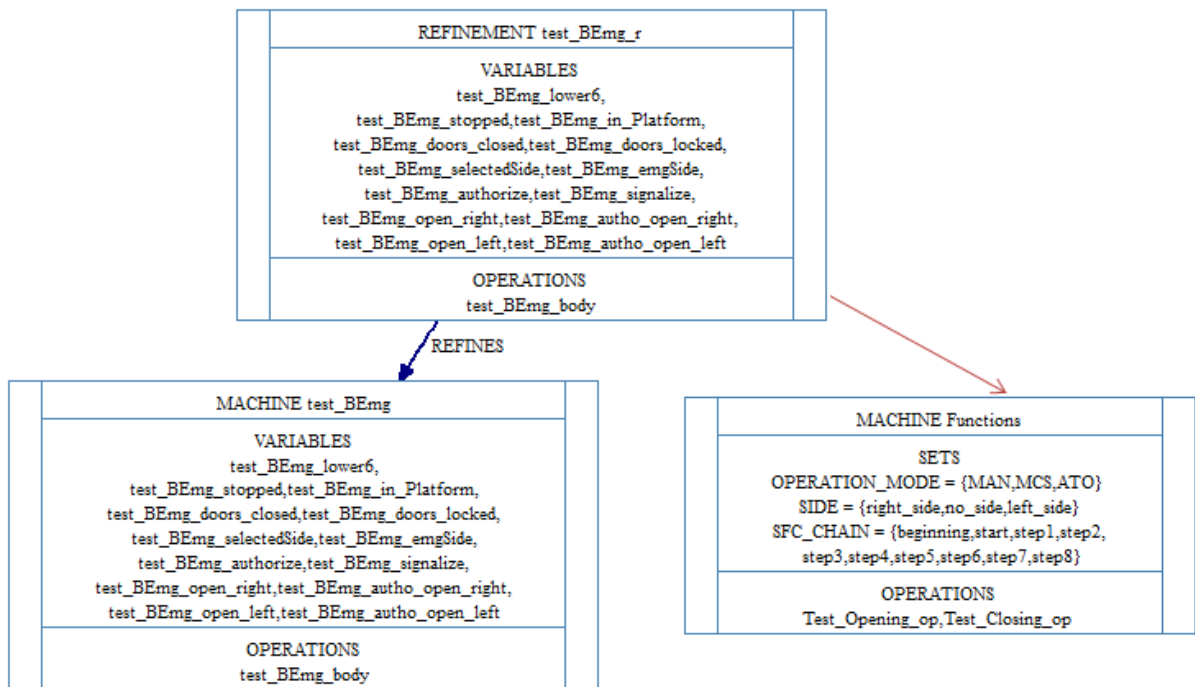
D.3 B models

Figure 50: B model for the instance of the *Define_ButtonEmergency* function block.
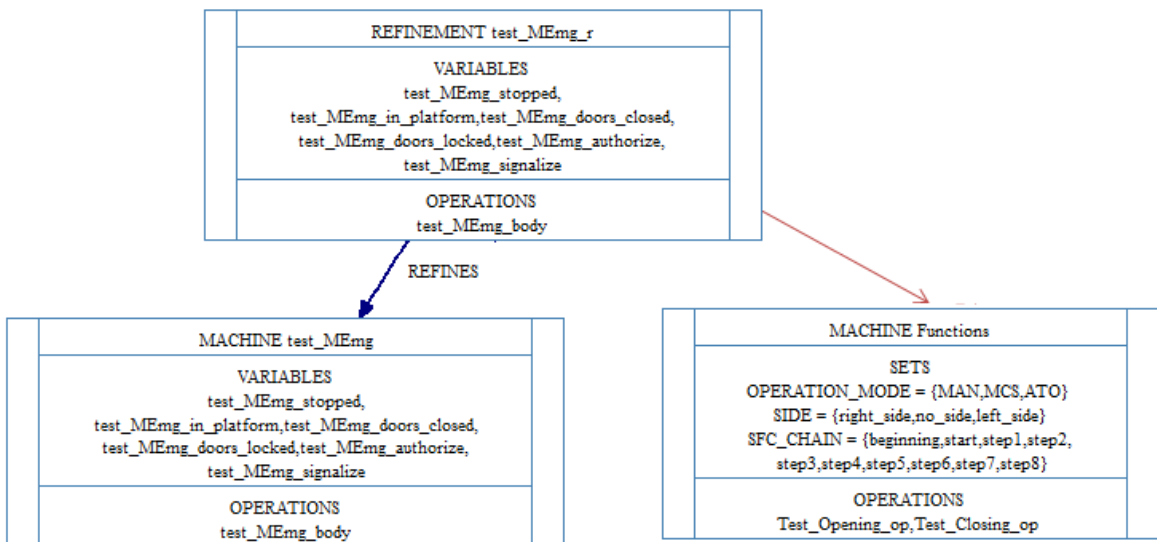

Figure 51: B model for the instance of the *Define_MechEmergency* function block.