Automatization of the Code Generation for Different Industrial Robots

Gustavo Vasconcelos Arnold

UMinho|2007

Universidade do Minho
Escola de Engenharia

Gustavo Vasconcelos Arnold

Automatization of the Code Generation for Different Industrial Robots

Novembro de 2007

Universidade do Minho
Escola de Engenharia

Gustavo Vasconcelos Arnold

Automatization of the Code Generation for Different Industrial Robots

Tese de Doutoramento em Informática
Área de Especialização:
Compiladores, Geração de Código e Robótica Industrial

Trabalho efectuado sob a orientação do
Professor Doutor Pedro Rangel Henriques e do
Professor Doutor Jaime Cruz Fonseca

Novembro de 2007

Abstract

This document presents GIRo (Grafcet - Industrial Robots), that is a generic environment for programming industrial robots off-line, that includes: a truly high-level and declarative language (Grafcet); an easy-to-use front-end (Paintgraf); an intermediate representation (InteRGIRo); the translators from Paintgraf to InteRGIRo; the generic compiler, that translates InteRGIRo to the robot target code; and the editor for the robot language inputs (to obtain the necessary information about the robot target language, to allow the generation of code to such robot).

GIRo focus on the modelling of the system, based on the Grafcet specification diagram, rather than on the robot details, improving the programming and maintenance tasks, allowing the reuse of source code, once this source code will be machine independent.

GIRo also allows the programmer to write programs in the robot language, if he is familiarized with the robot commands.

With GIRo:

- the user can program robots in a high or low level;
- the portability for the source code is granted;
- the reuse of source code for different robots is allowed;
- the programming and maintaining tasks are facilitated.

GIRo is easy-to-use. So, GIRo is "giro[1]"!

[1]Portuguese (Portugal) slang that means cool.

Resumo

Este documento apresenta um ambiente genérico de desenvolvimento de programas off-line para robôs industriais chamado GIRo (Grafcet - Industrial Robots). GIRo contém com os seguintes componentes: uma linguagem declarativa e de alto-nível (Grafcet); um front-end amigável (Paintgraf); uma representação intermédia (InteRGIRo); os tradutores para a InteRGIRo, a partir do diagrama Grafcet desenhado no Paintgraf; um compilador genérico de codígo que traduz a InteRGIRo para a linguagem de programação do robô destino; e um editor, utilizado para juntar as características e instruções da linguagem de programação do robô destino, a fim de permitir a geração de código para este robô.

GIRo foca na modelação do sistema, baseado no diagrama de especificação de automatismos Grafcet, ao invés das características físicas do robô. Desta forma, melhora-se as tarefas de desenvolvimento e manutenção de programas, uma vez que é permitido a reutilização de código fonte, já que este é independente de plataforma.

Giro também permite que o programador escreva os seus programas na linguagem do robô, caso o mesmo esteja familiarizado com os seus comandos.

Com o GIRo:

– a programação de robôs pode ser feita em alto nível (Grafcet) ou em baixo nível (linguagem do robô);
– a portabilidade do código fonte é garantida;
– a reutilização do codigo fonte para robôs diversos é permitido;
– as tarefas de programação e manutenção são facilitadas.

GIRo é fácil de utilizar. GIRo é "giro"!

Agradecimentos

Primeiramente, gostaria de agradecer a minha família (pais, irmãos e demais parentes) por todo incentivo, dedicação, suporte e conquistas. Sempre foram um porto seguro em minhas empreitadas e continuam sendo em mais uma, que está começando agora: o início de uma vida nova, com minha esposa e filhas, num país por nós desconhecido e atraente, o Canadá.

À minha esposa e filhas, por todo amor, carinho e alegria, além da paciência e compreensão durante a fase final deste doutoramento.

Aos meus orientadores, professores Pedro Rangel Henriques e Jaime Cruz Fonseca, por todo apoio, incentivo, sugestões, revisões, amizade, companheirismo, e obviamente, orientações, tanto locais como à distância. Mesmo quando foi preciso voltar para o Brasil, não deixaram de acreditar em mim e de me apoiar e incentivar.

Ao SBLP'97 (Simpósio Brasileiro de Linguagens de Programação de 1997, em Campinas - SP), ponto de partida de conversas, amizades e projetos, tendo sido este evento o berço de muitos trabalhos de mestrado e doutorado de vários brasileiros. Foi lá que conheci José Carlos Ramalho, professor da Universidade do Minho, onde trocamos idéias e iniciamos contatos para projetos futuros. E quantos projetos tiveram origem a partir deste encontro!

Aos amigos baianos e companheiros de apartamento Alfrânio (antigo colega de trabalho), Fábio e Tiago (ex-alunos), por todo apoio, incentivo e amizade, surgidos nos bons tempos que passei em Braga.

Aos amigos brasileiros e portugueses que tive o prazer de conhecer e que tornaram muito agradáveis os momentos vividos em Portugal.

Aos treinadores e amigos Nuno Reininho e Francisco Costa, por terem apostado e aprimorado as "habilidades" esportivas de um "atleta" de mais de 30 anos, e principalmente por terem ajudado a realizar um sonho adolescente: o de representar a minha instituição em eventos esportivos nacionais.

Às funcionárias da secretaria (Cristina, Goretti, Helena e Paula) por todo auxílio, presteza e rapidez na resolução de quaisquer problemas.

À equipe técnica (Aragão, Carla, Jaime, João, e Jorge Faria), por todo suporte e apoio técnico no uso de softwares, no acesso a internet e na manutenção de equipamentos, inclusive os pessoais.

Ao DI (Departamento de Informática) e à UM (Universidade do Minho), pela estrutura fornecida, pelo apoio material e pelo auxílio para participação em eventos nacionais e internacionais.

À FCT, pelo suporte financeiro imprescindível para a realização deste trabalho.

Contents

List of Figures

Chapter 1

Introduction

Today, the programming task of non-robotic systems is done in very high level languages; each of these languages try to facilitate the specification of the program, that solves the problem, allowing the programmer to concentrate on the problem, instead of the equipment where this program will be executed.

To do this, the developer should not think about the equipment that will run the program. He must be concerned about the problem, to find the correct solution for it. To make this task easy, it is used some modelling technique, appropriate for the kind of problem. The modelling technique will decompose the problem into smaller components, that will be implemented using the adequate programming language. This language should facilitate not only the programming task, but the readability, maintenance, reusability, composability and other important features for the development according to software engineering principles.

Another advantage of this approach is the existence of compilers to translate the high--level (machine independent) languages into the code of different computer platforms. With these, it is possible to run the same source code in different machines. So, the equipment can be upgraded, or changed, without the necessity of rewriting the programs.

However, the development of industrial robotic systems is still a difficult, costly, and time consuming operation. Today's industrial robots generally require a tremendous amount of programming to make them useful. The commercial robot programming environments are typically closed systems. The manipulator level is still the most widely used programming method employed in industry for manufacturing tasks. The forerunner languages, such as AML [TSM82] or AL [MGB82], have now been superseded by elaborated robot languages like ABB Rapid [ABB94]. Despite their common use, they have three important drawbacks.

1. They require detailed description of the motion of every joint in the mechanism in order to execute a desired movement.
2. They require specialized knowledge of the language.
3. The robot programs have limited portability. As a result, significant investment must be made when changing or acquiring a new robot type or simply when upgrading a

new controller from the same vendor.

One simple approach to solve some limitations is the Off-line programming environments. These environments are based in graphical simulation platforms, in which the programming and execution process is shown using models of the real objects. Consequently, the robot programmer has to learn only the simulation language and not any of the robot programming languages. Other benefits of off-line programming environments include libraries of pre-defined high-level commands for certain types of applications, such as painting or welding, and the possibility to assess the kinematics feasibility of a move, thus enabling the user to plan collision-free paths. The simulation may also be used to determine the cycle time for a sequence of movements. These environments usually provide a set of primitives commonly used by various robot manufacturers, and produce a sequence of robot manipulator language primitives such as "move" or "open gripper" that are then downloaded in the respective robot controllers. However, the current state-of-the-art off-line systems suffer from two main drawbacks. *Firstly*, they do not address the issue of sensor-guided robot actions. *Secondly*, they are limited to a robot motion simulator, which provides no advanced reasoning functionality, nor flexibility in the tasks.

1.1 Motivation

Considerable effort has been concentrated on developing new robot programming languages, and basically, there are two solutions for the creation of a general purpose robot programming language:

1. The compiler should generate code directly to the machine, where it should be necessary to know the hardware architecture of the robot, before the creation of such compiler;
2. the compiler should generate code for the industrial robot programming language, as Workspace[1] does.

This problem is also treated on the computers domain. There are some interesting solutions that are normally used and that are accepted by the society, while some others are under development and research.

Basically, there are three solutions for computer programming languages, that grants portability for the source code:

1. The compiler should generate standard C code, because standard C language is a very used and known programming language, that has translators for almost all computer platforms;
2. The compiler should generate an intermediate code, and the platforms where the program would run must have a virtual machine, responsible for interpreting this intermediate code, as JVM (Java Virtual Machine - used to interpret java programs) and CLR (Common Language Runtime - used to interpret Microsoft .Net applications) do;

[1]Workspace is a well-known robot simulation system, that allows the off-line programming of industrial robots. Workspace is a product of Flow Software Technologies.

3. The compiler should generate an intermediate code, and an specific final code generator translates it into the machine code. However, this final code generator is generated automatically by an automatic generator of code generators. To do this, it is necessary to specify the hardware architecture for this automatized tool;

Despite the existence of these five solutions, none of them could be really applied for the existing industrial robots, as it will be explained in chapter 4.

1.2 Objectives and Proposal

The aim of this PhD. work is to automatize the code generation for different industrial robots. This automatization intends to facilitate the industrial robots programming task, which means that it must be created a programming environment, with the definition of a programming language (as a front-end), that generates final code to different industrial robots (the back-ends).

The programming language (front-end) must attend the basic requisites of the industrial robots programming. It should be easy to use and to learn. It is also important that the user should actualize the environment to allow the generation of code for new industrial robots. At last, should solve the problems presented on the previous subsection.

To achieve this goals, it is necessary:

1. to specify a programming language;
2. to create a friendly programming environment;
3. to automatize the code generation process;
4. to simplify the process of obtaining the industrial robot details needed by the code generator;
5. to have a friendly interface for the user to enter the robot details that are needed by the code generator;
6. to grant efficiency and efficacy for the generated final code.

So, after some research and developments, the product of this thesis was created, that is called GIRo – Grafcet - Industrial Robots: A Generic Environment for Programming Industrial Robots Off-Line.

GIRo is inspired on a mix of two solutions that was presented on the previous section: the solution that is used by the Workspace, that generates code in the used industrial robot language (figure 4.2); and the one that generates automatically the code generator (figure 4.5). Instead of having an automatic generator of code generator, it has a generic compiler, responsible for translating code into many industrial robot programming languages (figure 1.1).

GIRo's approach can be seen on figure 1.2. It is supported by the following languages and tools that compose the architecture we want to defend:

– a truly high level and declarative language (Grafcet)
– an easy-to-use front-end (Paintgraf)

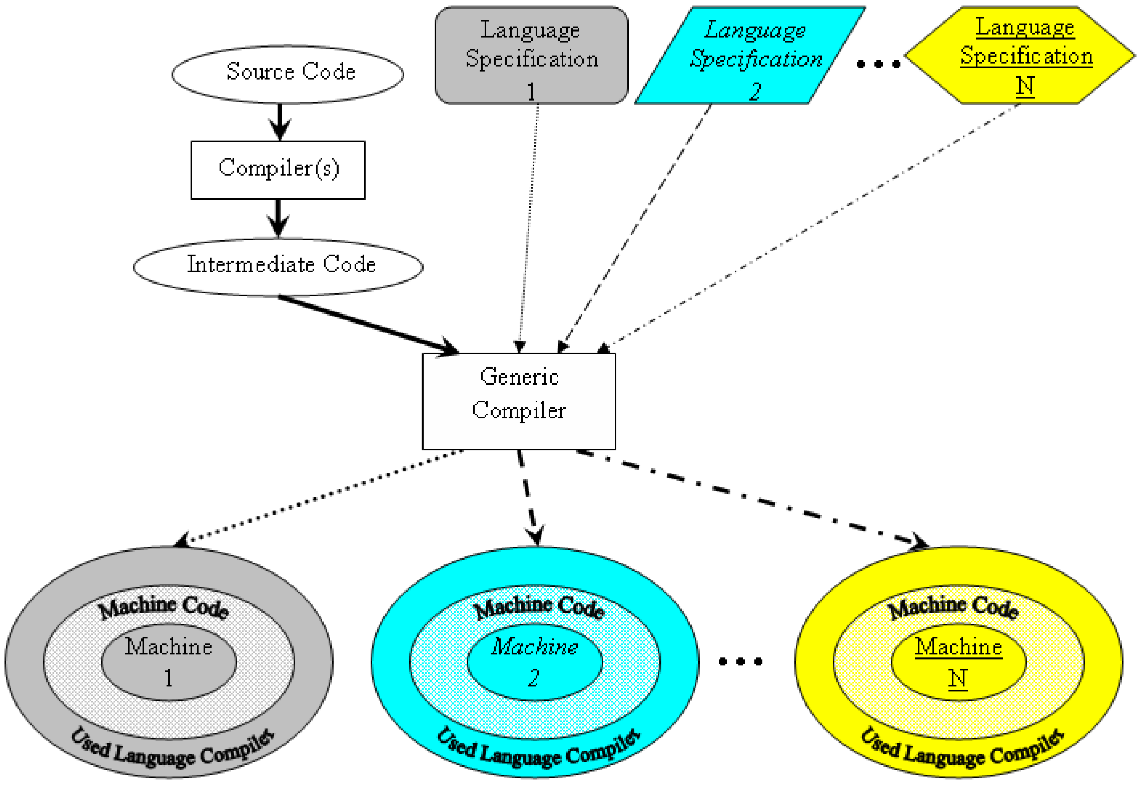


Figure 1.1: Thesis proposal.

- an intermediate representation (InteRGIRo)
- the translators to InteRGIRo
- the generic compiler
- the robot language inputs editor

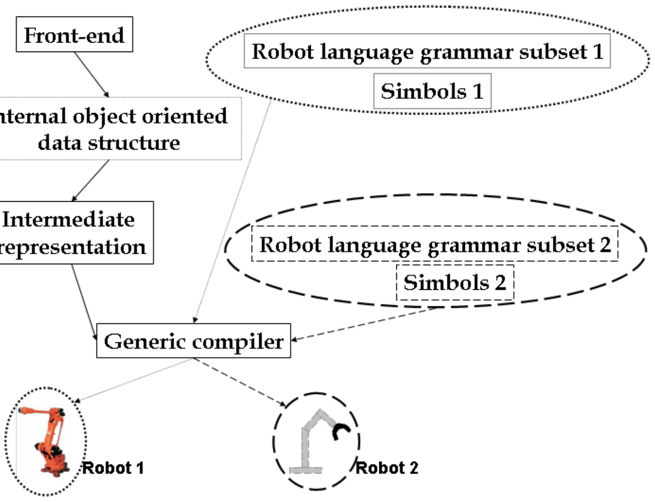

Front-end
Robot language grammar subset 1
Simbols 1
Internal object oriented data structure
Intermediate representation
Robot language grammar subset 2
Simbols 2
Generic compiler
Robot 1
Robot 2

Figure 1.2: GIRo's approach to industrial robots programming.

At the top, there is the problem to be solved. It would be used an adequate modelling technique that describes formally the overall problem. As the modelling language adopted was Grafcet, it was included in the FE a Graphical interface to aid editing the Grafcet visual description, the *Paintgraf*, an easy-to-use compiler front-end, based on Grafcet, that is a truly high level language, close to the specification instead of the robot, that interprets the specification language and generates an internal data structure, that can be translated into *InteRGIRo*, an intermediate description for the program specified. To generate final code it is also necessary to get some description of the target machine, the *robot language inputs*. With all of this inputs (InteRGIRo and robot language inputs), the final code generation is possible. To generate both the InteRGIRo and the final code, it is necessary some *translators*, like the generic compiler, that is responsible for the final code generation.

1.3 Contribution

With GIRo, the contribution of this PhD. work, the programming and maintenance tasks are facilitated, allowing the reuse of source code. The existing problems on the industrial robot programming languages, (presented in the beginning of this chapter), that motivated this thesis, are solved. Besides that, GIRo also attends other users desires like:

- it is user-friendly - because Paintgraf enables the user to describe completely the program by the use of Grafcet, that is an well known specification diagram for industrial automation aplications;
- it is easy to use and to learn - because Grafcet is a well knows specification diagram;

- it allows a programming close to the problem specification, instead to the robot details - because it is uses a programming language that is based on the Grafcet;
- it is extensible - because it is possible for the user to create high level instructions and data types;
- it is expressive - because it is possible to create programs in a truly high-level language, or even directly in the target robot language (loosing the source-code portability);
- it grants the source-code portability - because all the source programs do not depend on the robot language features. They are firstly translated into InteRGIRo, that is, then, translated into the target robot language, by the generic compiler;
- it allows the reusability of source-code - as the source-code portability is granted, it makes possible to reuse source-code.
- it facilitates the programming and maintenance tasks;
- it is platform independent - because GIRo was done by Java language;
- it allows the environment interaction - because it is possible to create programs that could interact with the environment, by the use of digital input/output signals;
- it is easy to obtain the target descriptions needed by the code generator - because it is used the target language description, instead of machine one;
- it is easier to describe programs in a high level way, instead of creating them inside simulators, that can difficult / limit the robot programming;

1.4 Thesis Organization

All of the necessary concepts of this thesis and the components of GIRo are better described in the chapters that follow this introduction. This thesis is structured in the following way:

- The basic concepts of industrial robotics with the state of the art in industrial robots programming languages, are presented in chapters:
  - 2 - Industrial Robots Programming - presents the background of industrial robotics and the ways of industrial robots programming, making a comparison of their current programming tools with the computer programming tools, showing the limitations that exist on the first ones;
  - 3 - Industrial Robots Programming Languages - classifies the industrial robot programming languages, presenting some examples of them;
  - 4 - Programming Environments and Some Existing Solutions - presents some other programming tools for the programming of industrial robots;
- The components of our initial approach, that has given the research directions, with its definitions and explanations, are presented in chapters:
  - 5 - Front-end, that presents the initial and the actual front-end used in the product of this thesis: the declarative and reactive RS language, that gave the initial idea of this research, that is to facilitate the programming task for industrial robots; and the Grafcet, one well-known description language for industrial automation applications, that is suitable to be the visual interface for the front-end;

    - 6 - Intermediate Representation - presents the importance of using an intermediate representation on the product of this thesis, with examples of some existing ones;
    - 7 - Generator of Code Generators - presents the concepts and some examples of this kind of translator, that is the principal element of this thesis: a compiler that makes possible the generation of code for different industrial robots;
- The description of the product of this thesis, that is called GIRo, with some case studies, conclusions and future works, are presented on these last chapters:
    - 8 - Our Solution: GIRo's Approach - presents the product of this thesis, with its components, some implementation details, algorithms, and examples of the translation processes;
    - 9 - Case Studies - presents some case studies that were done to validate GIRo;
    - 10 - Conclusions - presents the conclusions of this thesis, with some future works.

Chapter 2

Industrial Robots Programming

In this chapter, it will be presented the background of industrial robotics. An industrial robot will be defined, with its main features and with its ways of programming. It is also made a comparison between the industrial robots programming tools and the computers programming tools, showing the drawbacks of the first ones.

2.1 What is a robot?

A robot is an electro-mechanical device, with mechanical actuator(s), controlled by a computer. It can be programmed to perform automatically a large amount of tasks. The Robotic Industries Association [Reh97] defines an industrial robot as follows:

"An industrial robot system includes the robot(s) (hardware and software) consisting of the manipulator, power supply, and controller; the end-effector(s); any equipment, devices and sensors with which the robot is directly interfacing, any equipment, devices and sensors required for the robot to perfom its task; and any communications interface that is operating and monitoring the robot, equipment and sensors."

2.2 Some Robot Terms

To better understand robot programming, it is necessary to take care about some robot terms. These terms are described following.

End-of-Arm Tooling

Tools and grippers are the two general categories of end effectors used in robotics. In each case the robot must not only control the relative position of the tool with respect to the work as a function of time, it must also control the operation of the tool. For this purpose, the robot must be able to transmit control signals to the tool for starting, stopping, and otherwise regulating its actions.

Sensors

The most sophisticated robot available today cannot perform routine tasks, like detecting that a part has fallen to the floor or when a finished part is not ejected from a machine, without the use of sensors.

Sensors used in industrial robotics can be classified into two categories, sensors that are internal to the robot and sensors that are external to the robot [Gro87]. Although, the same types of sensors might be used in both categories.

Sensors internals to the robot are those used for controlling position and velocity of the several joints. These sensors form a feedback control loop with the robot controller. Typical sensors used to control the position of the robot arm include potentiometers and optical encoders. To control the speed of the robot arm, tachometers of several types are used.

Sensors external to the robot are used to coordinate the operation of the robot in its environment. In some cases, these external sensors are relatively simple devices such as limit switches that determine whether a part has been positioned properly in a fixture, or to indicate that a part is ready to be picked up at a conveyor. Other situations require more advanced sensor technologies, like:

– Tactile sensors - these sensors are used to determine whether contact is made between the sensor and another object. Tactile sensors can be divided into two types in robotics applications: touch sensors and force sensors. Touch sensors are those that indicate simply that contact has been made with the object. Force sensors are used to indicate the magnitude of the force with the object.
– Proximity sensors - these indicate when an object is close to the sensor. When this type of sensor is used to indicate the actual distance of the object, it is called a range sensor.
– Machine vision and optical sensors - Vision and other optical sensors can be used for various purposes. Optical sensors such as photocells and other photometric devices can be used to detect the presence or absence of objects, and are often used for proximity detection. Machine vision is used in robotics for inspection, parts identification, guidance, and other uses.
– Miscellaneous sensors - This category includes other types of sensors that might be used in robotics, including devices for measuring temperature, fluid pressure, fluid flow, electrical voltage, current, and several other physical properties.

Some authors, like [Reh97], also classifies the external sensors in another two categories: Contact and noncontact sensors. Contact sensors include the tactile ones, while the noncontact sensors include the proximity, miscellaneous and optical sensors, and machine vision.

Teach Stations

Teach stations on robots may consist of teach pendants, teach terminals, or controller front panels. Some robots permit a combination of programming devices to be used in programming the robot systems, whereas others provide only one system programming unit. In addition, some robots use IBM PC or compatible microcomputers as programming stations. Teach stations support three activities: (1) robot power-up and preparation for the programming; (2) entry and editing of programs; and (3) execution of programs in the work cell. The development of a program includes keying in or selecting menu commands in the controller language, moving the robot arm to the desired position, and recording the position in memory.

Robot Workspace

The space in which the robot gripper can move with no limitations in travel other than those imposed by the joints. Figure 2.1 shows a workspace for a spherical geometry robot in contrast to that of a human worker.

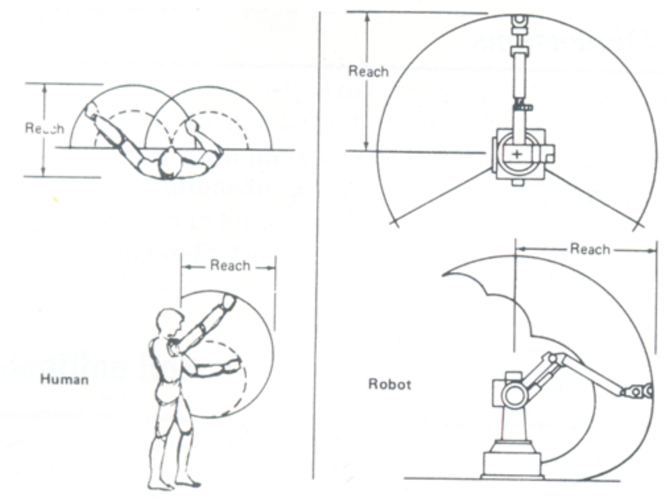


Figure 2.1: Human and industrial robot workspace.

Axis Numbering

The following axis numbering convention is used to describe the axes of motion of a robot. See figure 2.2

1. Start at the robot mounting plate (base).
2. Label the first axis of motion encountered as axis 1.
3. Progress from the base to the end-effector numbering each axis encountered successively.

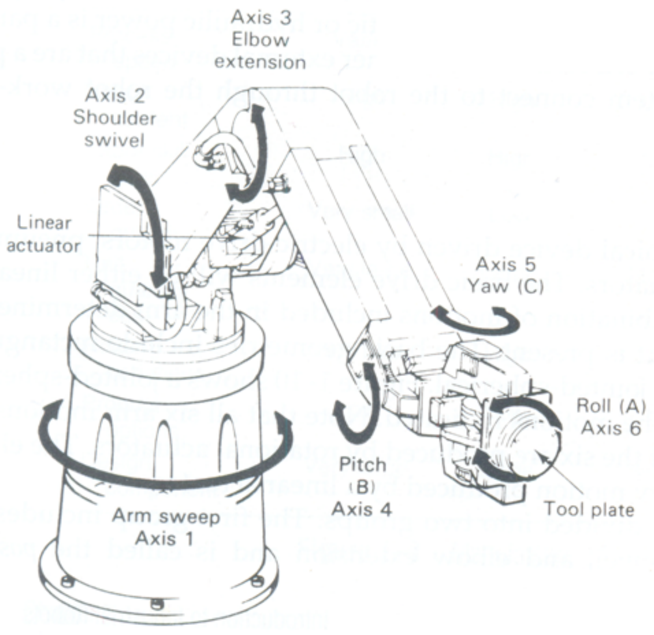


Figure 2.2: Industrial robot example.

Degree of Freedom

Every joint or movable axis on the arm is a degree of freedom. A machine with six movable joints is a robot with 6 degrees of freedom or axes.

Coordinate Systems

All points programmed in the work cell are identified by a base coordinate system that consists of three translation coordinates - X, Y and Z - that position spatially the robot; and three rotational coordinates - A, B, and C - that position the tool plate at the end of the arm. The number of work-cell coordinates required to define a programmed point is determined by the number of degrees of freedom present on the robot.

Offset (Tool Center Point - TCP)

When a tool is mounted to the robot tool plate, the points where the actions must happen can be different from the ones defined for the coordinate system of the previous subsection. The coordinate system corresponds to the location of the tool plate (figure 2.2), while the offset, or the tool center point (TCP) corresponds to the position where the tool, that is mounted at the tool plate, must act (figure 2.4). With an offset of 0, 0, 0 for the coordinates of the TCP (L, A and B respectively), the TCP is located at point TCP1 in figure 2.3. With L=10, A=5 and B=4 , the TCP is located at TCP2. When a tool is mounted to the tool plate, the distance from the TCP1 to the action point on the tool is included in the robot program. For example, the welding torch in figure 2.4 has a TCP of L = 14 inches, A = -4 inches, and B = 0 inches. The robot will control the motion at the welding electrode as the arm moves through the programmed points.

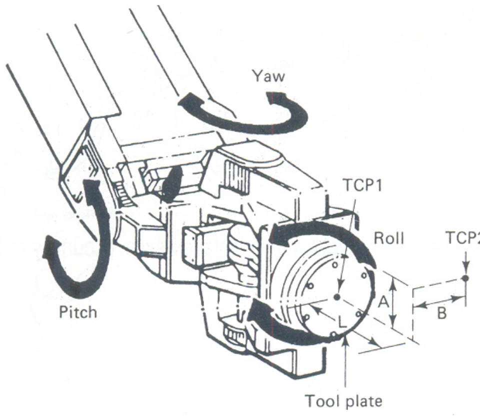


Figure 2.3: Tool center point.

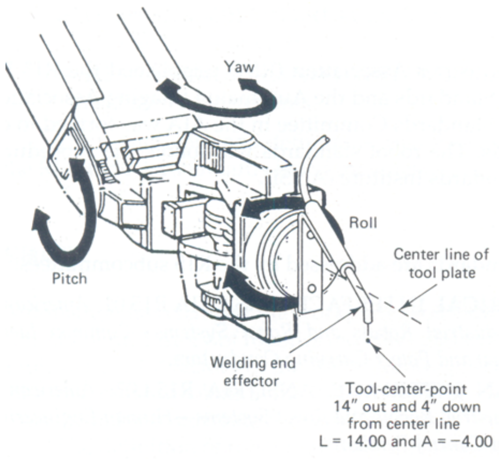

Yaw
Roll
Center line of tool plate
Pitch
Welding end effector
Tool-center-point 14" out and 4" down from center line L = 14.00 and A = −4.00

Figure 2.4: Robot with welding tool.

Control Techniques

The type of control used to position the tool separates robots into two categories: servo (closed-loop) and nonservo (open-loop) systems. An understanding of these two control techniques is fundamental to understanding the operation and programming or robots [Reh97].

1. *Open-Loop Systems:* The open-loop system provides no information to the controller concerning arm position or rate of change during the robot's motion throughout the

workspace. The arm receives only the drive signal, and the controller relies on the reliable operation of the actuators to move the arm to the programmed location. There are several advantages of an open-loop system:
   - Lower initial investment in robot hardware.
   - Well-established controller technology in the form of programmable logic controllers.
   - Less sophisticated mechanical and electronic systems with fewer maintenance and service requirements.
   - A larger pool of engineers and technicians familiar with the requirements of this type of system.
2. *Closed-Loop Systems:* A closed loop control system measures and controls the position and velocity of the robot tool center point (TCP) at every point during the robot's motion. The closed-loop uses information provided by the internal sensors attached to the robot's movable axes to calculate the current position and velocity of the TCP. These systems are much more complex and have a statistically higher chance to malfunction than open-loop systems; however, the advantages gained in the application of servo-controlled robots outweigh the problems created by the additional complexity inherent in the system. The advantages include these:
   - They provide highly repeatable positioning anywhere inside the workspace. This flexible, multiple-positioning characteristic is necessary for the implementation of robots in many production jobs.
   - The servo type of controller with computer capability can provide system control for devices and machines that are external to the robot system.
   - Powerful programming commands are available to perform complex manufacturing tasks with very simple program steps.
   - Interfacing robots to other computer-controlled systems, such as vision systems and host computers, is easier to accomplish with this controller.

Program Points

One of the main differences between computer programming languages and robotic programming languages is the necessity of storing program points, that represents the position points in the workspace, where the robot must pass in.

Normally, this points are defined by the use of the *teach box*, where the robot is moved until the desired location. At this moment, this position point can be stored. This points storage can be done at any time during the development of the program.

On-Line and Off-Line Programming

The terms On-line and off-line programming define the location where the robot program is developed.

For on-line programming, the production operation is stopped and the robot programmer puts the production machine into the programming mode, loosing production time. So, the programmer teaches the robot the required position, motion, and control sequences.

Some robot languages use variable names for the translation points and permit the control structure, moves, and program logic to be developed outside the production machine. The completed program is downloaded to the robot controller and production is stopped only to teach translation points for all the location variables named in the program. A significant reduction in lost production is achieved.

The term off-line programming means that all of the programming is performed away from the robot and the production area. Some systems use an workcell simulation software to define the program points and to test the program.

2.3 Industrial Robot Programming

A robot program can be defined as a path in space to be followed by the manipulator, combined with peripheral actions that support the work cycle, like opening and closing the gripper, for example. The program has two basic parts.

The first part is a set of points in space, to which the robot will move when the program is executed. To do this, it is used the *teach box*, where the user should move the robot throw its workspace, setting and storing points that will be necessary for the correct execution of the program. The second part consists of program statements that set the order of motion, the logic and conditions for decision, gather information on the operation of the cell, and analyze and prepare data for communication with other machines.

A robot is programmed by entering the commands into its controller memory. There are four ways to program a robot[Gro87]:

– Manual setup - the programming is done by setting limit switches and mechanical stops to control the end points of its motion. However, this does not correspond to a programming method, but to a mechanical control;
– Leadthrough programming - also called "teach by showing", programming task is done by moving the manipulator through the desired motion path during a teach procedure, entering the program into the controller memory. It is a simple way for the programming task, however, it has some disadvantages: it is necessary to interrupt the production process during the leadthrough programming; it is limited in terms of the decision-making logic that can be incorporated into the program; and since this method was developed before computer control became common for robots, these methods are not readily compatible with modern computer-based technologies, such as CAD/CAM, manufacturing data bases, and local communications networks;
– Computer like robot programming languages - Like a computer programming language, but with some specific commands to control the robot. They can solve some limitations of leadthrough programming with some functions, like: Enhanced sensor capabilities; Improved output capabilities for controlling external equipment; increase of the program logic control; Computations and data processing similar to computer programming languages; Communications with other computer systems;
– Off-line programming - It also uses its own computer like programming language, with the advantages explained above. But all the steps of de programs development,

like programming and testing, is done under a simulator environment. So, only when the development is done, and the system is corrected and tested, the program will be sent to the robot controller, with no interruption of the production process during the program development.

As it can be seen, the use of a computer like programming languages has some advantages. However, the actual programming languages, like the ones used do control the Mitsubishi, Nachi and Puma robots [Cor94], for example, do not allow a correct systems development, following the software engineering principles. It can be seen above:

– These programming languages are imperative, and some of then looks like assembly languages. It means that these languages are in a level more close to the machine than the user. So, the programmer must be care about some hardware features instead of think only about the logic of the system;
– It is used unconditional jumps (like *goto*) as execution flow control. Even in conditional jumps, the execution flow goes to any point of the program, defined by a label. It is against the principles of structured programming;
– There is no source code portability, because each programming language belongs to an specific robot;
– As a consequence, source code reutilization is only possible when using the same robots;
– Without source code portability, the programmers must be trained to use and program the machine, each time that the robot is changed;

There are some high level programming languages that are visual, friendly. Although, they are specific for a such robot. So, they do not solve the problem of source code portability.

Few standards currently exist for robot control languages. There is no interchangeability of programs among manufacturers, and in some cases, only limited interchangeability of programs between models from the same manufacturer [Reh97].

2.4 Industrial Robots Programming X Computer Programming

Today, the programming task of computer systems is done in very high level languages; each of these languages try to facilitate the specification of the program that solves the problem, allowing the programmer to concentrate on the problem, instead of the equipment where this program will be executed.

To do this, the developer should, first, do not think about the equipment that will run the program. He must be concerned about the problem, to find the correct solution for it. To make this task easy, it is used some modelling technique, appropriate for the kind of problem. The modelling technique will decompose the problem into smaller components, that will be implemented using the adequate programming language. This language should facilitate not only the programming task, but the readability, maintenance, reusability, composability and other important features for the development according to software engineering principles.

Another advantage of this approach is the existence of compilers that translate the high-level (machine independent) languages for different computer platforms. With these, it is possible to use the same source code in different machines. So, the equipment can be upgraded, or changed, without the necessity of rewriting the programs.

However, the industrial robot programming languages did not evolved in the same manner as the computer languages. The development of industrial robotic systems is still a difficult, costly, and time consuming operation. Today's industrial robots generally require a tremendous amount of programming to make them useful. Their controllers are not very sophisticated and the commercial robot programming environments are typically closed systems. The manipulator level is still the most widely used programming method employed in industry for manufacturing tasks.

One simple approach that solves some limitations is the Off-line programming environments. These environments are based in graphical simulation platforms, in which the programming and execution process are shown using models of the real objects. Consequently, the robot programmer has to learn only the simulation language and not any of the robot programming languages. Other benefits of off-line programming environments include libraries of pre-defined high-level commands for certain types of applications, such as painting or welding, and the possibility to assess the kinematics feasibility of a move, thus enabling the user to plan collision-free paths. The simulation may also be used to determine the cycle time for a sequence of movements. These environments usually provide a set of primitives commonly used by several robot vendors, and produce a sequence of robot manipulator language primitives such as "move" or "open gripper" that are then downloaded in the respective robot controllers. However, these current state-of-the-art off-line systems do not solve the problems that we want to treat.

Figure 2.5 shows a schema where can be compared the industrial robots programming tools with the computer programming tools.

2.5 Limitations of Industrial Robots Programming

It can be said that there are basically two ways for programming industrial robots:

1. by the use of the industrial robots programming languages, where the user can program directly the robot, with a language that allow him to use all the potentialities of the robot;
2. by the use of off-line programming environments, where the programmer can use some graphical development environment, where it is possible to test the program before using it in the robot. It is also possible to develop programs for different robots, but it is necessary to have a library for each robot.

However, both of them have their own drawbacks.

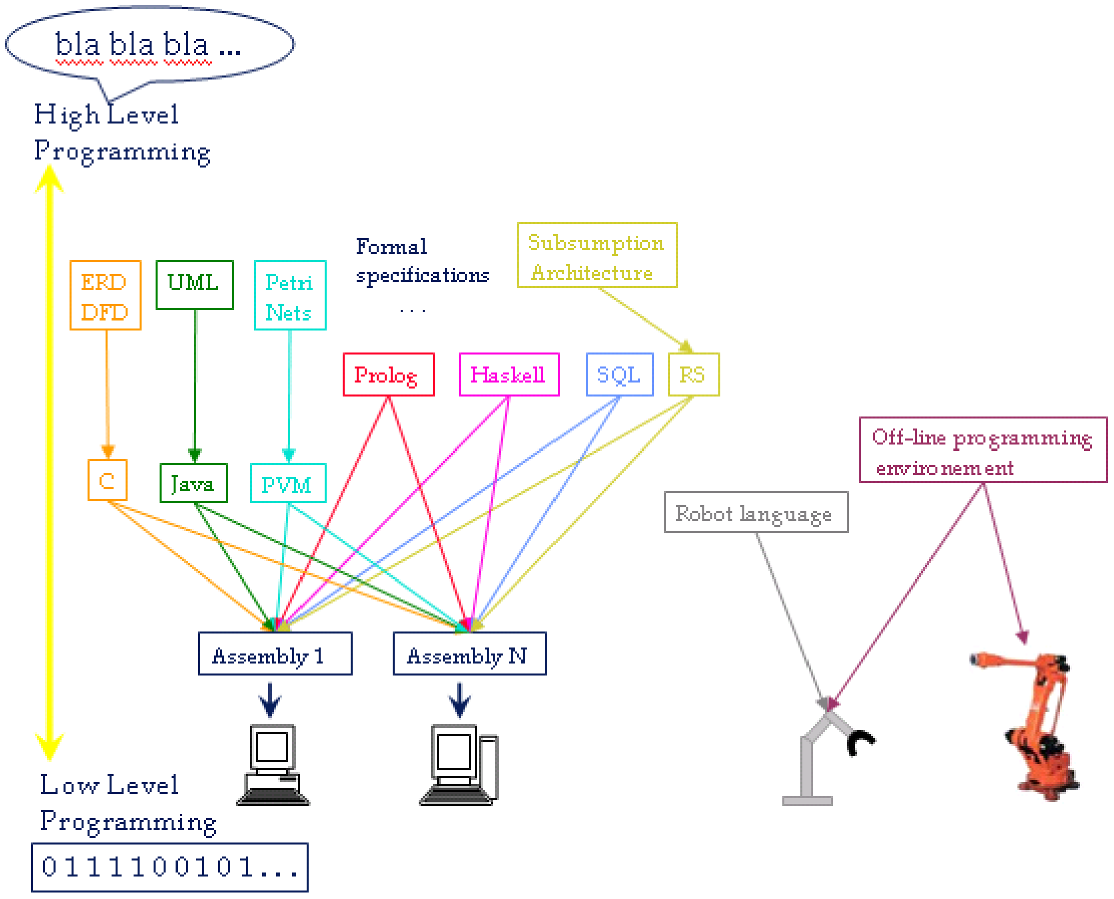


Figure 2.5: Industrial robots X Computer programming tools

2.5.1 Industrial Robots Programming Languages Drawbacks

The industrial robots programming languages did not evolved in the same manner as the computer languages. Those languages have some drawbacks:

1. The typical languages are imperative, low-level or structured;
2. Each industrial robot has its own programming language, which make difficult or even impossible, to reuse the source code;
3. although the forerunner languages, such as AML [TSM82] or AL [MGB82], have now been superseded by elaborated robot languages, like ABB Rapid [ABB94], they require detailed description of the motion of every joint in the mechanism in order to execute a desired movement;
4. the programmer in charged of the task, instead of thinking only about the problem, he has to think about the robot that will run the program, and about its programming language. Both the robot and the language limitate the specification of the problem;
5. They are more closed to the robot specification than to the problem, difficulting the problem modelling task and all other good practices that a correct software engineering should require;
6. they require specialized knowledge of the language;
7. the robot programs have limited portability, and significant investment must be made when changing or acquiring a new robot type or simply when upgrading a new controller from the same vendor.

2.5.2 Off-Line Programming Environments Drawbacks

The off-line programming environments also have some drawbacks, like:

1. They are, also, more closed to the robot specification than to the problem, difficulting the problem modelling task and all other good practices that a correct software engineering should require;
2. These tools do not solve the problem of programming the robot to interact with its environment;
3. the most recent off-line programming environments (for instance, Workspace) do not address the issue of sensor-guided robot actions;
4. they are limited to a robot motion simulator, which provides no advanced reasoning functionality, nor flexibility in the tasks;
5. Probably, it is necessary to construct a new translator always that a new robot is created, or some robot language is increased.

2.5.3 Some Principles and Criteria

According to some principles of software engineering and programming languages evaluating criteria [Seb99], it is possible to see that these development environments do not attend the user needs, as can be seen on figure 2.6.

Principles	**Off-Line Programming Environments**	**Industrial Robots Programming Languages**
User-friendly	YES	NO
Source Code Portability	YES	NO
Expressiveness	NO	YES
Environment interaction	NO	YES
Specification close to the problem	NO	NO
Reusability	NO	NO

Figure 2.6: Principles of software engineering and programming languages that must be supported

The programs created today to control industrial robots make them act as programmable logic controllers that can move; but there are much more things to explore. Maybe the problem arises from the limitations that both the industrial robots programming languages and the off-line programming environments presents, and from the low level of the programming languages available, because they do not make easy to program robots as it is with computers, and there is no reuse of source code.

2.6 Chapter's Considerations

In this chapter it was presented some robot concepts and ways of programming, with some considerations about each kind of programming. It was made a comparison between industrial robots programming tools and computer programming tools, being clear that, concerning on users facilities and their expectations, the first programming area is far way from the last one.

It is also important to say that:

- each robot model has its own programming language;
- the industries do not shows the robot architecture of their produced robots, which difficult the development of other compilers and softwares.

Chapter 3

Industrial Robots Programming Languages

In this chapter, some fundamentals about industrial robots programming will be presented, including the classification of industrial robots programming languages, showing the effort that was done for the standardization of robot programming. At the end, some examples of such languages are presented, aiming to illustrate that they must be improved.

It is important to say that there are other solutions for the programming of industrial robots, but they that are more than a language: they are programming environments, or even robot controllers. These other solutions will be presented in the next chapter.

3.1 Industrial Robot Language Classification

One way to classify the many languages used by industrial robot manufacturers is according to the level at which the user must interact with the system during the programming process. The current languages can be grouped into the four loosely formulated levels that follows [Reh97]:

3.1.1 Joint-Control Languages

Languages at this level concentrate on the physical control of robot motion in terms of joints or axes. The program commands must include the required angular change of rotational joints or the extension lengths of linear actuators. These languages usually do not support system or work-cell commands, which can be incorporated into the programs of higher-level robot languages for control of external devices.

3.1.2 Primitive Motion Languages

Point-to-point primitive motion languages are now usually confined to older robot programming languages, or they may be an optional programming mode for a more sophisticated robot language. These languages have the following characteristics:

- A program point is generated by moving the robot to a desired point and depressing a program switch;
- Program editing capability is provided;
- Teaching motion of the robot is controlled by either a teach pendant, terminal, or joystick;
- The programmed and teaching motion can occur in the cartesian, cylindrical, or hand coordinate modes;
- Interfacing to work-cell equipment is possible;
- The language permits simple subroutines and branching;
- Some languages allow parallel execution using two or more arms in the same word space.

The advantage of languages at this level is the performance on the manufacturing floor. But as disadvantages, the programming emphasis is still on robot motion rather than on the production problem, and there is no support for off-line programming. An example of this kind of languages is the VAL language.

3.1.3 Structured Programming Languages

These languages offer a major improvement over the last levels, and have become the standard for the major vendors of robots. There are several examples of this kind of languages, like VAL II, Rapid, and others. They have the following characteristics:

- A structured control format is presented;
- Extensive use of coordinate transformations and reference frames is permitted;
- Complex data structures are supported;
- Improved sensor commands and parallel processing;
- System variables (called state variables), whose value is a function of the state of position of the system, are permitted;
- The format encourages extensive use of branching and subroutines defined by the user;
- Communication capability with local area networks is improved;
- Off-line programming is supported.

3.1.4 Task-Oriented Languages

The primary function of this kind of languages is to hide from the user the commands and program structure that normally must be written by the programmer. The user need only be concerned with solving the manufacturing problem. These languages have the following characteristics:

- Programming in natural language is permitted;
- A plan generation feature allows replanning of robot motion to avoid undesirable situations;
- A world modelling system permits the robot to keep track of objects;
- The inclusion of collision avoidance permits accident-free motion;
- Teaching can be accomplished by showing the robot an example of a solution.

Currently, no languages at this level are operational, but significant efforts are underway at university and industrial research laboratories. Several experimental languages have exhibited the characteristics of this level: AUTOPASS (Automatic Programming System for Mechanical Assembly) from IBM, RAPT (Robot Automatically Programmed Tools), and LAMA (Language for Automatic Mechanical Assembly) [Reh97].

3.2 Standardization of Robot Programming Languages

As it was said before, most of the problems in the successful implementation of industrial robots are related to the difficulties in programming and the lack of integration with other factory automation equipment. Several experts have suggested the need for a standardized robot programming language, that will enable robots of different manufactures to seamlessly converse and reduce high training costs. Researchers worldwide have tried to develop a standardized robot programming language, but have been only partially successful. Most of the standardization of robot programming is done for industrial purposes (for example, IRDATA, MMS, SLIM/STROLIC), and they are directly aimed for programming the motions of the robot. [Zie01].

The standardization of robot languages has not matured yet, especially when interaction between robot and sensors is involved. Today, there are many standard robot programming languages, but they are not really used by the industrial robots. The users and researchers want this kind of language, but the industrial robot manufactures do not. All the references for those languages has, at least, 10 years, but none of them is really currently adopted by the commercial industrial robots. Maybe because of the difficulty for identifying the needed instructions, or because the industrial robot manufactures are not interested in adopting any standardization.

However, some of these standardization tries are presented above.

IRL - Industrial Robot Language

Close cooperation between robot manufacturers, robot users, and research institutes gave birth to a new high level programming language for robots. The national German standardization organization (DIN) published this language in 1993 as a national standard DIN 66312 part 1. The language is independent of any particular robot controller or robot kinematics. [Sal00]

IRL is a high-level language, designed with reference to the Pascal family of languages. It covers facilities for the modularization of programs. It enables the programming of simple tasks with easy to learn language elements.

RobotScript

Robotic Workspace Technologies (RWT) claims that the RobotScript is the first universal industrial robot programming language. It is based on the Microsoft interpreted computer language Visual Basic Scripting Edition, or VBScript. The commands are English-like. However, this language can be used only with RWTs Robot Control Solution (RCS interprets the RobotScript language to the robot controller). It is necessary to acquire an RWT Universal Robot Controller (URC), that substitutes the original robot controller. So, to control the robot, it is necessary to develop the industrial robot drivers for the URC. As URC has an open-architecture, anyone can create these drivers [Tec07b].

This solution transfers the standardization problem from the programming language to the robot drivers. RoboScript works well, there are some important users like NASA, but to use this language it is necessary to acquire the URC. And to control some industrial robot, someone has to develop the driver between this robot and URC (which is not a simple task), or an standard driver solution must be created and adopted by the industrial robots manufactures (that may not be interesting for them).

Other

Many organizations around the world made efforts to standardize programming languages for industrial robots, like the German IRDATA, French LME, Japonese SLIM (Standard Language for Industrial Manipulators) and ICR (Intermediate Code for Robotics). These languages are aimed for programming robot motions with the correct parametrization (e.g. velocity, acceleration, smoothing), program flow instructions, arithmetic and geometric calculations, and sensor data processing. They can be seen as a standardization of the robot control. An "assembler-like" intermediate code, that belongs to a lower level than other industrial robot programming languages [EFK01].

3.3 Industrial Robot Programming Languages: Examples

Some examples of industrial robot programming languages are presented following:

3.3.1 VAL

VAL [Com80] is a computer-based language designed specifically for use with Unimation Inc. industrial robots (well known as PUMA robots).

The VAL language consists of monitor commands and program instructions. The monitor commands are used to prepare the system for execution of user-written programs.

Program instructions provide everything that is necessary to create VAL programs for controlling robot actions.

The features of VAL language that influences the programming task are [Com80] [Wik07]:

- Control programs are written on the same computer that controls the robot. It means that it is not possible do develop programs in another computer. So, it does not allow off-line programming;
- To specify the control flow instructions, numbers are used as labels. So, it is necessary to assign numbers for the lines that are destines of jumps. Each line must contain only one instruction;
- The existing constants are: integer, real and point;
- To indicate a specific location for the robot, it is used point constants. Its syntax is (X, Y, Z, R, P, Ya), where X, Y and Z matches to the robot translational coordinators; and R, P and Ya matches to the robot rotational coordinators, where R corresponds to the *Roll* degree, P to the *Pitch* degree, and Ya to the *Yaw* degree;
- All the variables are global and it is not necessary to declare them previously, because despite using integer and real constants, these are stored on a numeric data type. So, there are only 2 data types:
  - numeric data types - that accepts integer and real values;
  - point data type - that accepts point values.
- There are no distinction between the numeric and point variables identifiers. As there is no variable declaration, the user must take care when using each variable;
- To modify the instructions control flow during execution, it is necessary to use jump instructions, indicating the label of the target instruction. As it was said before, the label corresponds to a number located in the beginning of the target instruction. The *GOTO* and *IF* commands use numbers that act as labels to indicate the next instruction to be executed;
- There are two conditional sentences, depending on the condition used:
  - For conditions that uses channels, like sensors for example, an *IFSIG* instruction is necessary, with the following syntax:

    ```
    IFSIG [ch1], [ch2], [ch3], [ch4] THEN line_number
    ```

    This instruction only test if a channel is active (bigger than 0) or inactive (equal or smaller than 0). A channel is represented by a positive or negative number. When it is positive, the resulting evaluation is true if this channel is active. When the channel number is negative, the resulting evaluation is true if it is inactive. It is possible to verify from one to four channels at the same instruction, but the three commas are always obligatory. The commas between the described channels works as the logic operator AND;
    There are 32 input channels, numbered 1001 to 1032, and there are 32 output channels, numbered 1 to 32;
    A *SIGNAL* command is used to modify an output channel;
  - For conditions that uses variables or numbers, a common *IF* sentence is used. This *IF* sentence only allows one relational operator (EQ, NE, LT, GT, LE or GE) per *IF*, and does not allow the use of logic operators.

– The *STOP* command stops the execution of the program, while the *END* command ends the execution of the program.

An example of a program can be seen above:

```
10 ifsig  PP1, D2,, then 80
20 ifsig  PP1, D3,, then 80
30 ifsig  PP2, D1,, then 170
40 ifsig  PP2, D3,, then 170
50 ifsig  PP3, D1,, then 270
60 ifsig  PP3, D2,, then 270
70 goto 10
80 closei
90 ifsig  GC,,, then 110
100 goto 90
110 move esquerda
120 ifsig  AP1,,, then 140
130 goto 120
140 openi
150 ifsig  GA,,, then 10
160 goto 150
170 closei
180 ifsig  D3, GC,, then 210
190 ifsig  D1, GC,, then 240
200 goto 180
210 move esquerda
220 ifsig  AP2,,, then 140
230 goto 220
240 move direita
250 ifsig  AP2,,, then 140
260 goto 250
270 closei
280 ifsig  GC,,, then 300
290 goto 280
300 move direita
310 ifsig  AP3,,, then 140
320 goto 310
```

3.3.2 Melfa Basic III

Melfa Basic III [Com] is the Mitsubishi Movemaster RV-M2 industrial robot programming language. Despite its name indicates that this language could looks like the computer Basic language, it really looks like an assembly language, which makes their programs difficult to read or write. So, to make programs easily, it could be necessary to use the COSIMIR software (a software for off-line programming of Mitsubishi robot systems that allows a

3-D modelling, programming and simulation of production cells), instead of programming directly the robot.

The features of this language that influences the programming task are:

– It is not possible to do programs off-line. There are two ways for programming robot Movemaster RV-M2: by the teach pendant; or by the COSIROP, a software that makes a connection between the robot and a computer, allowing its programming;
– All instructions are described with only two letters, which make difficult to read/write programs;
– To specify the control flow instructions, it is necessary to number the lines. Each line must contain only one instruction;
– The numeric constants are: integer, real and point;
– To indicate a specific location for the robot, it is used point constants;
– It is not used variables. On this language, the program access directly the registers and counters of the machine;
– To modify the instructions control flow during execution, it is necessary to use unconditional or conditional jumps, indicating the line number of the target instruction;
– The `GT` instruction is used as unconditional jump;
– There are conditional jumps, however, there are no `IF` like sentences. To treat a condition, the known relational and logical operators are used as commands:
  - EQ (equal) - causes a jump to occur if the internal register value is equal to the specified value;
  - NE (not equal) - causes a jump to occur if the internal register value is not equal to the specified value;
  - LG (larger than) - causes a jump to occur if the internal register value is greater than the specified value;
  - SM (smaller than) - causes a jump to occur if the internal register value is smaller than the specified value;
  - AN (and) - ANDs the internal register value and the specified value;
  - OR (or) - ORs the internal register value and the specified value;
  - XO (xor) - Exclusive ORs the internal register value and the specified value;
  - There are no >= and <= sentences or operators;
– The *ED* command ends the execution of the program.

An example of a program can be seen above:

```
10 SP 15
13 GO
15 MO 2
20 MO 1
30 GC
34 MO 2
40 MO 3
45 MO 4
50 GO
55 MO 3
60 MO 10
```

3.3.3 VAL II

VAL II [Inc86] evolved from VAL (that was also presented previously). VAL was developed in 1975. In 1978 VAL had been rewritten as VAL II, and was offered commercially with PUMA robots. VAL II is a BASIC like interpreted language. It is also composed of a mix of instructions necessary for robot control and monitor commands.

The features of VAL II language that influence the programming task are:

– Control programs, today, are probably written on the same computer that controls the robot. It means that, today, probably it is not possible do develop programs in another computer. It has one possibility for programming it off-line, but is it necessary to use an very old 8" floppy-disk to transfer the program to this Puma controller. So, today, probably is not possible to develop programs off-line;
– To specify the control flow instructions, numbers are used as labels. So, it is necessary to assign numbers for the lines that are destines of jumps. Each line must contain only one instruction;
– The existing constants are: integer, real and point;
– To indicate a specific location for the robot, it is used point constants. Its syntax is (X, Y, Z, R, P, Ya), where X, Y and Z matches to the robot translational coordinators; and R, P and Ya matches to the robot rotational coordinators, where R corresponds to the *Roll* degree, P to the *Pitch* degree, and Ya to the *Yaw* degree;
– All the variables are global and it is not necessary to declare them previously, because despite using integer and real constants, these are stored on a numeric data type. So, there are only 2 data types:
  - numeric data types - that accepts integer and real values;
  - point data type - that accepts point values.
– There are no distinction between the numeric and point variables identifiers. As there is no variable declaration, the user must take care when using each variable;
– To modify the instructions control flow during execution, it is necessary to use jump instructions, indicating the label of the target instruction. As it was said before, the label corresponds to a number located in the beginning of the target instruction. The *GOTO* and *IF* commands use numbers that act as labels to indicate the next instruction to be executed;
– There is one conditional sentence that, differently from the first version of VAL, accepts more than one relational operator (==, <>, <, >, <= or >=), separated by the logic operators. When the condition uses channels, like sensors for example, an *SIG* command is used to verify if the channel is active (bigger than 0) or inactive (equal or smaller than 0). A channel is represented by a positive or negative number. When it is positive, the resulting evaluation is true if this channel is active. When the channel number is negative, the resulting evaluation is true if it is inactive. Above, an example of a conditional sentence with the *SIG* function:

```
if sig(-1002, 1004) goto 10
```

– There are 32 input channels, numbered 1001 to 1032, and there are 32 output channels, numbered 1 to 32;

- A *SIGNAL* command used to modify an output channel, and this command can be written as *SIG*, that is the same name used to evaluate the input signals. To identify the goal of a *SIG* command, it is necessary to see the number of the channels (1001 to 1032 for input channels, while 1 a 32 for output ones);
- The *STOP* command stops the execution of the program, while the *END* command ends the execution of the program.
- VAL II can run an application command (that controls the robot moving) and calculate the I/O values in parallel. It means that it is not necessary to wait the end of a moving command to analyze the I/O channels or to do mathematical calculus. This multitasking execution can generate errors, that are not perceived when reading the source code. For example, in the following fragment of a program
  ```
  MOVE A0
  IF SIG 1003 GOTO 20
  ```
  the channel 1003 will be analyzed before the end of the moving instruction. But, if the 1003 channel is activated only when the robot arrives at the point A0? In that situation, the IF sentence will always be false, which is not visible when reading the program, because it is not usual to analyze, in an sequential program, a instruction while the previous one is not ended. To correct this situation, it is necessary to use the *BREAK* command. This command makes the controller wait the end of the last analyzed instruction to continue the execution of the program. An example of this command can be seen above:
  ```
  MOVE A0
  BREAK
  IF SIG 1003 GOTO 20
  ```

An example of a program can be seen above:

```
10 if sig( PP1) goto 30
20 goto 40
30 if sig( 1002) goto 200
40 if sig( PP1) goto 60
50 goto 100
60 if sig( 1003) goto 200
70 if sig( PP2) goto 90
80 goto 100
90 if sig( 1001) goto 290
100 if sig( PP2) goto 120
110 goto 160
120 if sig( 1003) goto 290
130 if sig( PP3) goto 150
140 goto 160
150 if sig( 1001) goto 430
160 if sig( PP3) goto 180
170 goto 190
180 if sig( 1002) goto 430
190 goto 10
```

```
200 closei
210 if sig( GC) goto 230
220 goto 210
230 move esquerda
240 if sig( AP1) goto 260
250 goto 240
260 openi
270 if sig( GA) goto 10
280 goto 270
290 closei
300 if sig( 1003) goto 320
310 goto 360
320 if sig( GC) goto 370
330 if sig( 1001) goto 350
340 goto 360
350 if sig( GC) goto 400
360 goto 300
370 move esquerda
380 if sig( AP2) goto 260
390 goto 380
400 move direita
410 if sig( AP2) goto 260
420 goto 410
430 closei
440 if sig( GC) goto 460
450 goto 440
460 move direita
470 if sig( AP3) goto 260
480 goto 470
```

3.3.4 NACHI SC15F Robot Programming Language

NACHI SC15F robot is general purpose manipulator, made by Nachi-Fujikoshi company. It can be seen on figure 3.1.

It is necessary to store the points where the robot should pass in a file before executing the program. This action is done by the use of the robot *teach box*, as described before.

Its programming language is similar to Basic language and its main features are [Cor94]:

- To specify the control flow instructions, it is necessary to number the lines (1 to 9999). Each line must contain only one instruction;
- The numeric constants are: integer, real, hexadecimal, binary, and one that express angles in degrees;
- To indicate a specific location for the robot, it is used point constants. Its syntax is (X, Y, Z, R, P, Ya), where X, Y and Z matches to the robot translational coordinators;

Figure 3.1: NACHI SC15F robot

and R, P and Ya matches to the robot rotational coordinators, where R corresponds to the *Roll* degree, P to the *Pitch* degree, and Ya to the *Yaw* degree;

– Variables have pre-defined names, that depends from the variables data types:
  - Integer variables format - Vn% or V%[n], where n must be between 1 and 200 (Ex: V1% = 10, V%[2] = 20).
  - Real variables format - Vn! or V![n] where n must be between 1 and 100 (Ex: V1! = 10,5, V![2] = 20 * V1!).
  - Character and string variables format - Vn$ or V$[n], where n must be between 1 and 20 (Ex: V1$ = "ABC", V$[2] = "Z" + V1$).
  - Point variables format - Pn or P[n], where n must be between 1 and 999 (Ex: P1 = (100, 0, 100, 0, 0, 90));

– To modify the instructions control flow during execution, it is necessary to use labels. The name of each label must start with the character (*) followed by any other character. The *GOTO* and *IF* commands use labels to indicate the next instruction to be executed;

– The *STOP* command stops the execution of the program, while the *END* command ends the execution of the program.

An example of a program can be seen above:

```
10 USE 1
20 ACC 0
30 TOOL 0
40 V1% = 0
50 *S1
51 JMPI 4, I1
60 IF V1% = 10 THEN *L2X1 ELSE *L2X2
70 *L2X1
80 V1% = 0
90 END
100 GOTO *S1
110 *L2X2
120 V1% = V1% + 1
```

```
130 MOVE P, P1, T = 3
140 MOVE P, P2, T = 3
150 SHIFTA 0, 0, 0, 100
160 MOVE P, P3, T = 3
170 SHIFTA 0, 0, 0, 0
180 GOTO *S1
190 MOVE P, P4, T = 3
200 MOVE P, P5, T = 3
210 MOVE P, P6, T = 3
220 GOTO *S1
```

3.3.5 Rapid

RAPID [Aut] is a powerful programming language used for ABB industrial robots. It is a procedural interpreting language, and syntactically it looks like Basic, C and Pascal. The RAPID interpreter admits multitasking, which means that several RAPID programs, called tasks, can be executed in parallel. Every task consists of one or more modules, which in turn consist of RAPID instructions. It also handles errors (exceptions) and user traps.

The RAPID language itself is a fairly general purpose programming language. There are predefined and installed routines, handling the robotic-specific actions (like spot welding, for example), but these are not part of the core language itself.

Its main features for writing programs are:

- It is possible to do programs off-line, which means that, it can be used another computer to develop the programs. If the RobotStudio system is used, it is also possible to simulate the robot behavior before executing it on the real robot;
- The numeric constants are integer and real. There are also logical and string ones;
- There are 41 data types: the basic ones for computer programming languages, like numeric, string and boolean; and many other specially created for ABB robot purposes, that are records composed by the basic data types. The user can also define its own records;
- To indicate a specific location for the robot, it is used point constants. As the robot is able to achieve the same position in several different ways, the axis configuration is also specified. So, its syntax is [[`trans`], [`rot`], [`robconf`], [`extax`]], where: `trans` corresponds to the position (`x`, `y` and `z`) of the tool center point expressed in millimeters; `rot` corresponds to the orientation of the tool, expressed in the form of a quaternion (`q1`, `q2`, `q3` and `q4`); `robconf` corresponds to the axis configuration of the robot (`cf1`, `cf4`, `cf6` and `cfx`) (this is defined in the form of the current quarter revolution of axis 1, axis 4 and axis 6, while the `cfx` is only used for the robot model IRB5400); and `extax`, that corresponds to the position of the external axes, that is defined for each individual axis (`eax_a`, `eax_b` ... `eax_f`);
- The variables can be local or global. It is necessary to declare them previously, in the beginning of the program, or in the beginning of each module.
- Identifiers are used to name modules, routines, data and labels, being the first

character a letter. The other characters can be letters, digits or underscores "_", with at most 16 characters;

- The execution flow during execution can be controlled in a structured way, using the commands like `for`, `while`, `if` and `test` (a `case` like sentence). It is also possible to use `goto` instructions, indicating the label of the target instruction;
- The conditional expressions accept more than one relational operator (=, <>, <, >, <= or >=), separated by the logic operators. The value of an input signal can be read directly in the program. So, there are no distinctions when manipulating variables and input signals;
- The *STOP* command stops the execution of the program, while the *EXIT* command ends the execution of the program when a program restart is not allowed.

An example of a program can be seen above:

```
10 USE 1
20 ACC 0
30 TOOL 0
40 V1\% = 0
50 *S1
51 JMPI 4, I1
60 IF V1\% = 10 THEN *L2X1 ELSE *L2X2
70 *L2X1
80 V1\% = 0
90 END
100 GOTO *S1
110 *L2X2
120 V1\% = V1\% + 1
130 MOVE P, P1, T = 3
140 MOVE P, P2, T = 3
150 SHIFTA 0, 0, 0, 100
160 MOVE P, P3, T = 3
170 SHIFTA 0, 0, 0, 0
180 GOTO *S1
190 MOVE P, P4, T = 3
200 MOVE P, P5, T = 3
210 MOVE P, P6, T = 3
220 GOTO *S1
```

3.4 Chapter's Considerations

In this chapter it was presented some fundamentals about industrial robots programming, like the classification of industrial robots programming languages. It was also presented some standardizations that exist. At the end some examples of industrial robots programming languages were presented.

For this kind of programming languages, it is important to say that:

- they are imperatives and procedurals;
- there is no standard;
- there is no reuse of source code;
- there is no portability of the source code;
- the specification is close to the robot details, instead of the problem;
- they must be improved to facilitate the programming task.

Chapter 4

Programming Environments and Some Existing Solutions

In the previous chapter, it was presented only the industrial robot programming languages, that are the ones that are used to directly program the robots.

In this chapter, some other approaches to program industrial robots, that has some advantages when compared with the industrial robot programming languages (like the higher level programming or the source code portability), are presented.

In the beginning, it will be presented the importance of modelling techniques for software engineering, and after, some declarative languages, general purpose programming languages, programming environments, and robot control systems, will be presented. All of these approaches are different from the ones presented on the previous chapter.

There are some work groups on industrial robotic domain, like OROCOS, OSACA, ORCA, and OMG-DTF. Each of them is a tentative for creating its own open source framework, allowing components sharing and integration. Other research groups are working on the creation of a neutral representation capable of describing product data throughout the life cycle of the product, and independent from any particular system, like ISO-STEP and IAI-IFC. The objectives of each one of these projects is presented above:

- **OROCOS** (Open RObot COntrol Software): aims at developing a general-purpose, free software, and modular framework for robot and machine control [Cen07];
- **OSACA** (Open System Architecture for Controls Association): aims at specifying a system architecture for open control systems which is manufacturer independent [Ass07];
- **ORCA**: is an open-source framework for developing component-based robotic systems [Orc07];
- **OMG-DTF** (Object Management Group - Robotics Domain Task Force): aims at fostering the integration of robotics systems from modular components through the adoption of OMG standards [Gro07];
- **ISO-STEP** (STandard for the Exchange of Product model data): aims at providing a mechanism that is capable of describing product data throughout the life cycle of

a product, independent from any particular system [Org07];

– **IAI-IFC** (International Alliance for Interoperability - Industry Foundation Classes): aims at assembling a project model in a neutral computer language that describes building project objects and represents information requirements common throughout all industry processes [fI07].

Any of these works belong to the same scope of this thesis because:

– The target robots are different - In this thesis, the goal is to grant portability for the source code, facilitating the programming task, for different industrial robots, which includes the existing newer and older ones. Each one of the OROCOS, OSACA, ORCA, and OMG-DTF projects, is a tentative for creating its own open source framework to control robots, allowing components sharing and integration. However, part of their work is done: or on a low-level robot control, aiming to define a free open control system, specifying, for example, the kinematics for a given robot; or specifying interfaces that must be used during the construction of the hardware components. So, using some of the presented projects would implies on changing the robot controller, or creating some interfaces for each different robot, which differs from our proposal, that is to create an environment for programming the existing robots, without changing anything on the target robots;
– The proposals are different - The other two research groups, ISO-STEP and IAI-IFC, aim to specify a neutral representation for the product data descriptions independent from any particular system. This research groups could work together, in future, with our proposal, but the goals are different: the first ones deals with the product data, while the second with the programming task.

4.1 Modelling Techniques

The use of an adequate modelling technique facilitates the development of the system, enabling the system developers and the system clients to express their ideas, allowing their communication in a known way. For the robot programming task, there are some modelling techniques. The advantage of modelling is the creation of models from the system and its behavior that can be seen in different abstraction levels, before implementing it. So, it is very important to model the system first.

If the programs are created directly, thinking on the problem and on the machine, this programs would be difficulty to write, to read, and consequently, to maintain. So, some techniques were created, like the structure analysis, where the problem is decomposed based on the data and the operations, that should be modelled separately. To model the data it is used the Entity Relationship Diagram (ERD), and to model the operations it is used the Data Flow Diagram (DFD). Today, there are some other modelling techniques, like Unified Modelling Language (UML), used to model object oriented systems. It uses: the Class Diagram, that shows the classes and their relationships in a logical view (like ERD to structure analysis); the State Transition Diagram, that shows the events that causes transition from one state to another, with its resulting actions (like DFD to structure analysis); and the Use-Cases Diagram, that shows the system's use cases and the actors that interact to them.

Some other examples of modelling techniques are:

– **Petri Nets**, that is a graphical and mathematical modelling technique used for for describing and studying systems that are characterized as being concurrent, asynchronous, distributed, parallel, nondeterministic, and/or stochastic [Pet77];
– **Grafcet** [Tel82], that is one well-known description language for modelling industrial automation applications, which allows the incorporation of logic and sequential issues in the specification [FB98b] [FB98a];
– **Finite State Machine**, that are a class of automata studied in automata theory and the theory of computation. They are significant in many different areas, including electrical engineering, linguistics, computer science, philosophy, biology, mathematics, and logic. In computer science, finite state machines are widely used in modelling of application behavior, design of hardware digital systems, software engineering, compilers, network protocols, reactive systems, and the study of computation and languages [HMH01];
– **Statecharts**, a Finit State Machine extension, that are a methodology by which complex real time systems can be specified in an intuitive graphical manner. They are relevant to the specification and design of complex discrete-event systems, such as multi-computer real-time systems, interactive software systems, communication protocols and digital control units [Har87];
– **Subsumption Architecture** [Bro86] that decomposes the mobile robot application into several simple and reactive behaviors, that work together, in parallel, in a dynamic environment. It was the first behavior based modelling technique used in the development of mobile robots. Even that it had been created to develop mobile robots, it can be used, as a high level abstraction, to model industrial applications, as a manufacturing cell [Arn00].

Among various modelling techniques, the analyst should choose the most adequate for each concrete problem, to obtain the advantages of the software engineering.

4.2 Declarative Languages

As it was said before, it is important to use a modelling technique, but is also important to have some programming languages as closed to the specification of the problem as possible.

For a language to be close to the specification, it must also have high level constructors that allow the definition of structured and complex abstract data types and mathematical operators over them.

There are, basically, two kinds of programming languages:

1. imperative languages: the underlying principle (the operational semantics) is very similar to the processor's execution cycle, being necessary to understand its architecture; the kind of available statements is also similar to the machine

instructions. The programmer should also know how to manipulate memory elements to store the necessary data;
2. declarative languages: instead of following the execution principles, those languages have as background a mathematical theory that supports data representation and operations over that data. The explicit memory manipulation is not necessary; the programmer just manipulate, in a high level, the data, without being necessary to know where this data is stored.

Declarative languages are higher level than imperative languages, more closed to the specification of the problem.

One example to show the difference between this two kind of languages is the file manipulation, that can be done in imperative languages (like C) and in declarative languages (like SQL). A simple operation, like searching a file for a specific person (in this example called "John"), in a imperative language is done in the following way:

```
Open file
Read first element
While (the name of the element is not "John",
       and the file did not reach the end)
Do    Read the next element
If John was found, print its data
```

This same operation can be written, in a declarative language, as follows:

```
Select all data from file,
       Where name is equal to "John"
```

According to the style, declarative languages are classified as functional or relational (logic). The first group is supported by the principle that a program is just a function mapping the input data into the output results; while the second family relies upon the idea that a program is a set of assertions defining the relations that hold (evaluate to true) in some world. Typical declarative languages are: Lisp, ML or Haskell (functional paradigm), and Prolog (logic paradigm).

On industrial robotics domain there are some tentatives for creating declarative languages. They are called, on such domain, as task-oriented (or task level) languages, that were explained on previous chapter.

There are some other examples of declarative languages for robotics, most of them applied to the mobile robotic domain, and almost all used for research purposes. None of them is a general purpose language. They work on specific robots to which they were created. Some examples of declarative languages are:

– **RS language** [Tos93], that is a real time language relying on the principle of productions systems, ie, on the rule-based paradigm (condition-reaction set of rules). Some experiments where made like controlling a Nachi industrial robot [Pio98], or controlling a simulated manufacturing cell [Arn00]. At present, despite it has inspired this work [AHF02], there are no more research using this language.
– **GRL** (Generic Robot Language) [Hor00], that is a language that extends traditional functional programming techniques to behavior-based systems. Most of the

characteristics of popular behavior-based robot architectures can be concisely written as reusable software abstractions in GRL. This project was ended;

- **FROB** (Functional Robotics) [PHH99], that is an Embedded Domain Specific Language for controlling robots. It is built using the principals of Functional Reactive Programming (FRP - a domain specific language embedded in Haskell that is used for programming hybrid reactive systems [WH00]). It supports rapid prototyping of new control strategies, enables software reuse through composition, and defines a system in a way that can be formally reasoned about and transformed;
- **Behavior language** [Bro90], that is a rule-based real-time parallel mobile robot programming language. It compiles into a modified and extended version of the subsumption architecture [Bro86] and thus has back-ends for a number of processors including the Motorola 68000 and 68HC11, the Hitachi 6301, and Common Lisp;
- **AUTOPASS** [LW77], that is oriented towards objects and assembly operations, rather than motions of mechanical assembly machines. It is intended to enable the user to concentrate on overall assembly sequence and to program with English-like statements using names and terminology that are familiar to him. To relate assembly operations to manipulator motions, the AUTOPASS compiler uses an internal representation of the assembly world. This representation consists of a geometric data base generated prior to compilation and updated during compilation. The AUTOPASS language is embedded in PL/I and offers many of the control and data facilities of that language. The major part of the implementation effort focused on a method for planning collision-free paths for cartesian robots among polyhedral obstacles [LPB85].

Some other research on task-oriented languages have suggested languages like LAMA [LPW77], TWAIN [LPB85] and RAPT. However, LAMA was only partially implemented; TWAIN was only a framework for the existing and future research in task planning (affirmed in 1985) where a prototype was expected around 1987; and RAPT does not deal with obstacle-avoidance, automatic grasping, or sensory operations [LPB85]. LAMA, RAPT and AUTOPASS, are the most well known examples of task-oriented languages, even that none of them can be used to completely control an industrial robot. Any other recent research on this field has achieved the results of those seventies languages.

There are also some other high-level languages that are not exactly declarative: they are XML based languages. Two examples of these are below, however they were used under research laboratories, and now, these projects reach the end.

- **XRCL** [Sal00], that is a XML based language and environment designed to allow mobile robotics researchers to share ideas by sharing code. It is an open source project, protected by the GNU Copyleft. A prototype for two simulated robots, and three hardware platforms is built: the B21R, the Pionner and the HandyBoard. Using RWI's (Real World Interface Division) CORBA interface gives the possibility of controlling any robot (or device) on the Internet;
- **RoboML** [MT00], that is an XML-based language for data representation and interchange in robotic applications. Combined with a set of communication protocols, it is aimed at providing a common interface for hardware and software robotic agents communicating via internet networks.

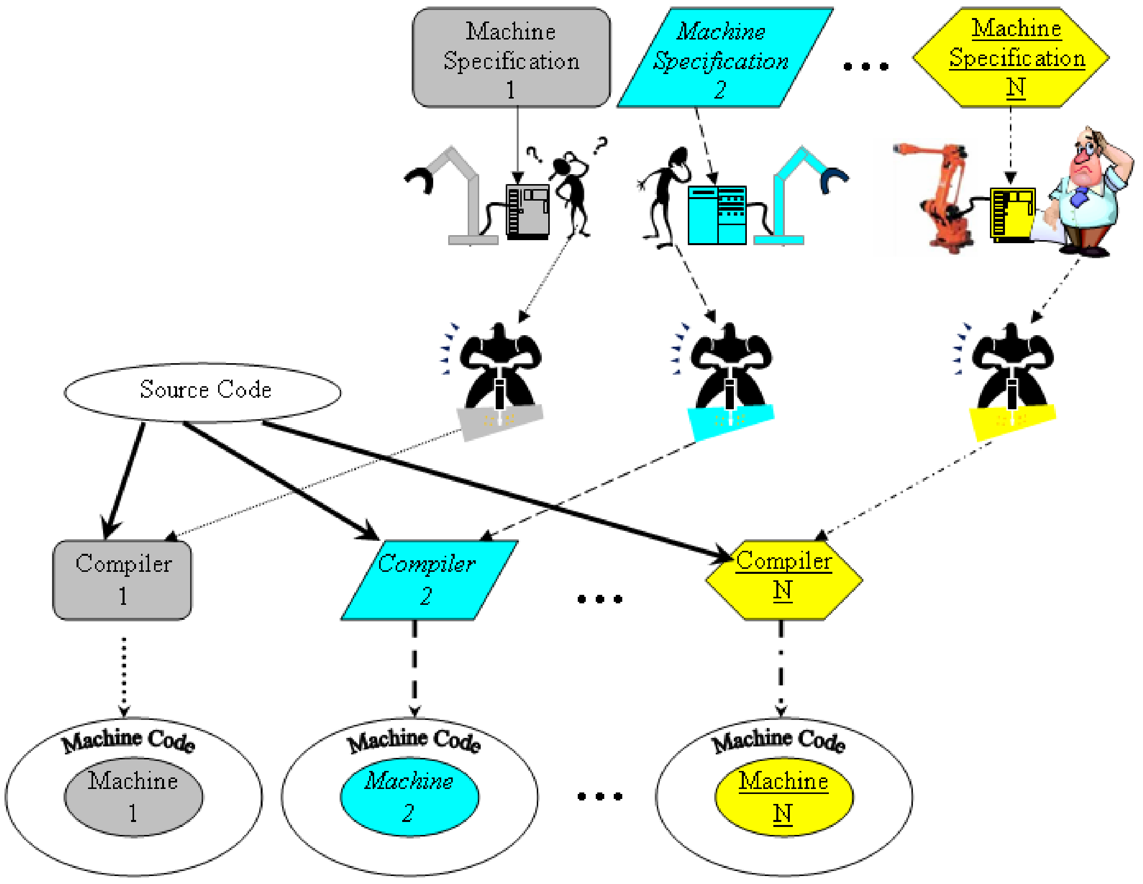


Figure 4.1: Code generation directly to the machine.

As it could be seen on this subsection, there are no current research on declarative (or task-oriented) programming languages for industrial robots purposes. The used industrial robots programming languages are the imperatives ones, like the ones presented on previous chapter.

4.3 General Purpose Programming Languages

Considerable effort has been concentrated on developing new robot programming languages, and basically, there are two approaches for the creation of a general purpose robot programming language:

1. The compiler should generate code directly to the machine, where it should be necessary to know the hardware architecture of the robot, before the creation of such compiler (figure 4.1);
2. the compiler should generate code for the industrial robot programming language, as Workspace does (figure 4.2).

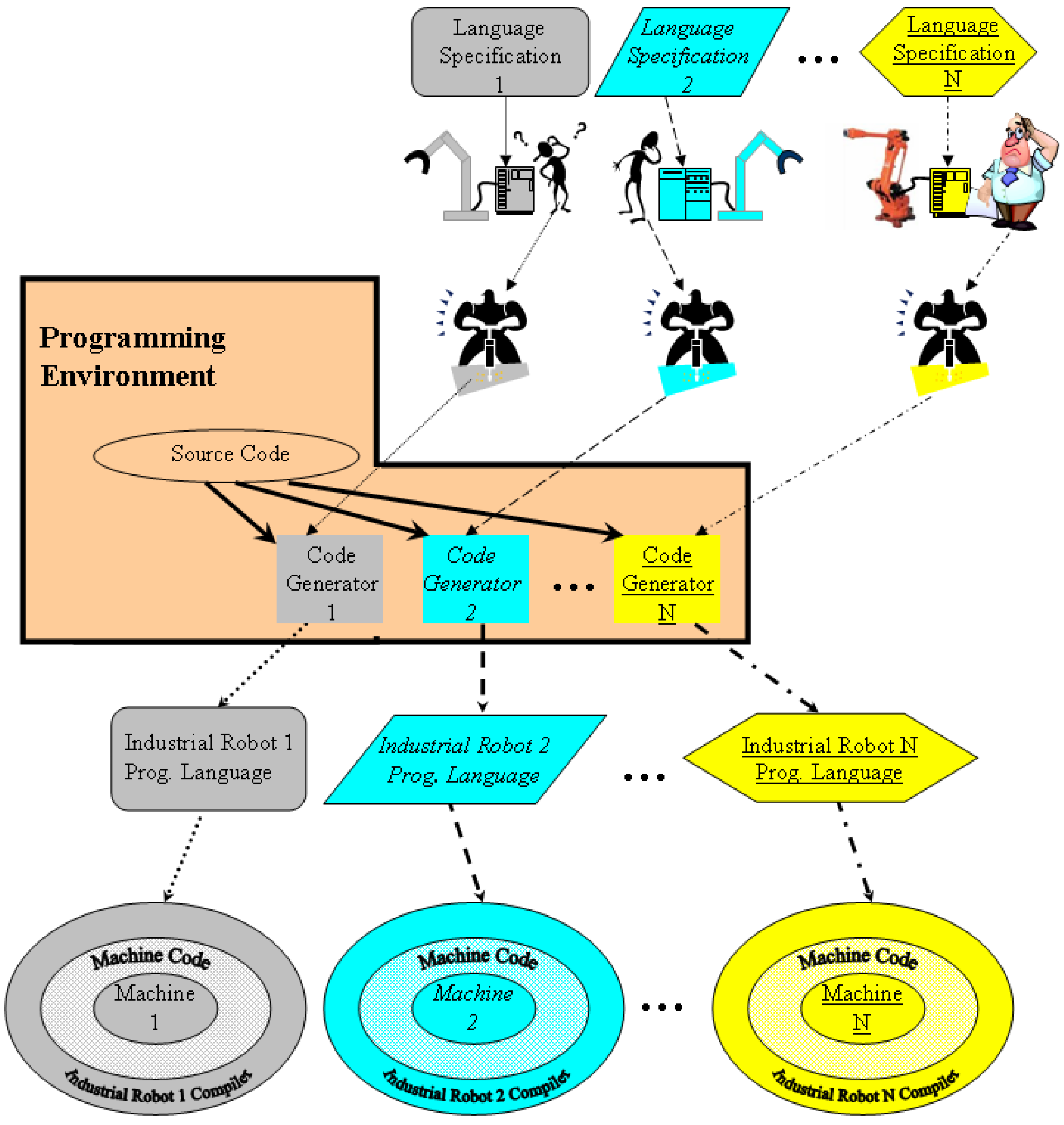


Figure 4.2: Code generation for the industrial robot programming language.

However, the first approach is not applicable for commercial industrial applications, because the industrial robots have closed architectures. This option is used on research domain, where robot controllers are substituted by the ones that has open architectures. In such case, the researchers developed various interesting tools, but they are not used in commercial industrial applications.

How all the industrial robots need to be programmed, the manufactures have to create programming languages that are able to control entirely their robots, and those languages must be given to the users. This means that, in the case of close architectures, the programming language can be seen as a window to access and control the robot hardware. This is the second approach, used for Workspace to generate code for different industrial robots.

On the other hand, Workspace does not have a general compiler, or a generator of code generators. Instead of these, the Flow Software Technologies enterprise has to create each one of the necessary translators. Which means that Workspace has various translators for each one of the industrial robot programming language that they work. And always that a new robot controller is created, or evolved, a translator has to be created, or modified, by the Flow Software Technologies enterprise.

4.3.1 Similar Computer Strategies

This problem, that is to facilitate the programming task, granting portability for the source code, is also treated on the general computing domain. There are some interesting strategies that are normally used and that are accepted by the society, while some others are under development and research.

Basically, there are three strategies for computer programming languages, that grants portability for the source code:

1. The compiler should generate standard C code, because standard C language is a very used and known programming language, that has translators for almost all computer platforms; (figure 4.3);
2. The compiler should generate an intermediate code, and the platforms where the program would run must have a virtual machine, responsible for interpreting this intermediate code, as JVM (Java Virtual Machine - used to interpret java programs) and CLR (Common Language Runtime - used to interpret Microsoft .Net applications) do (figure 4.4);
3. The compiler should generate an intermediate code, and an specific final code generator translates it into the machine code. However, this final code generator is generated automatically by an automatic generator of code generators. To do this, it is necessary to specify the hardware architecture for this automatized tool (figure 4.5);

Despite the existence of these approaches, none of them could be really applied for the existing industrial robots.

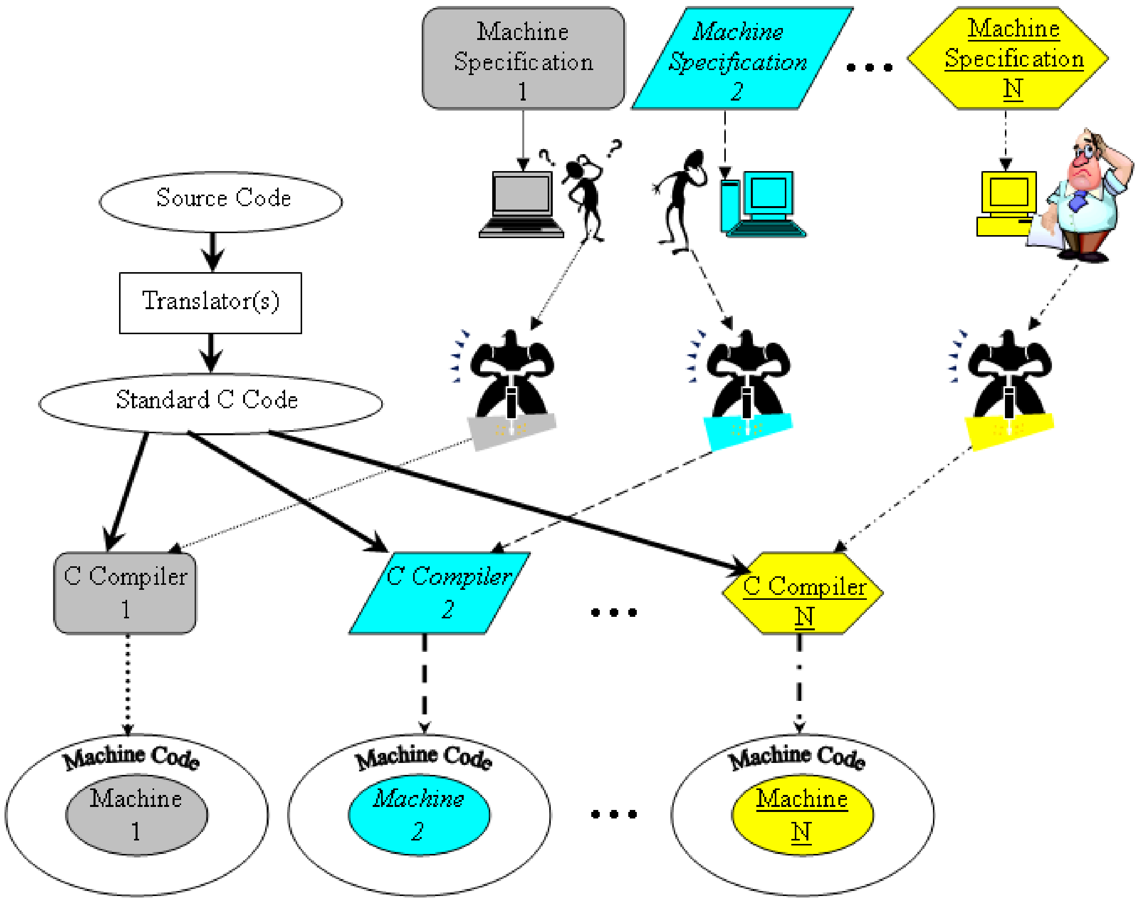


Figure 4.3: Standard C code generation strategy.

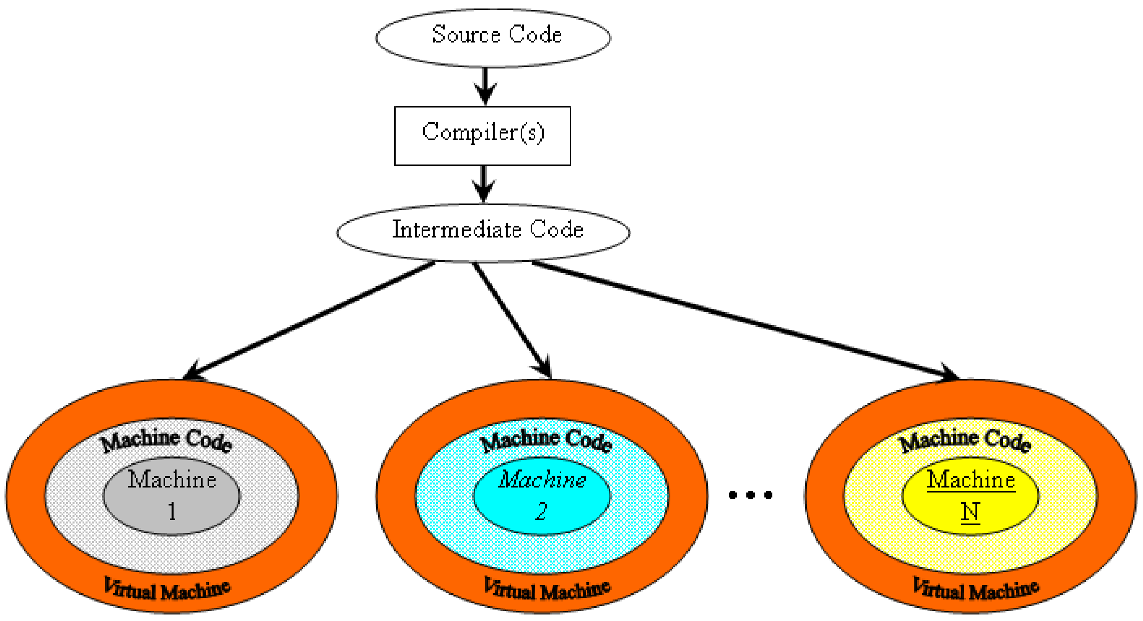


Figure 4.4: Using virtual machine.

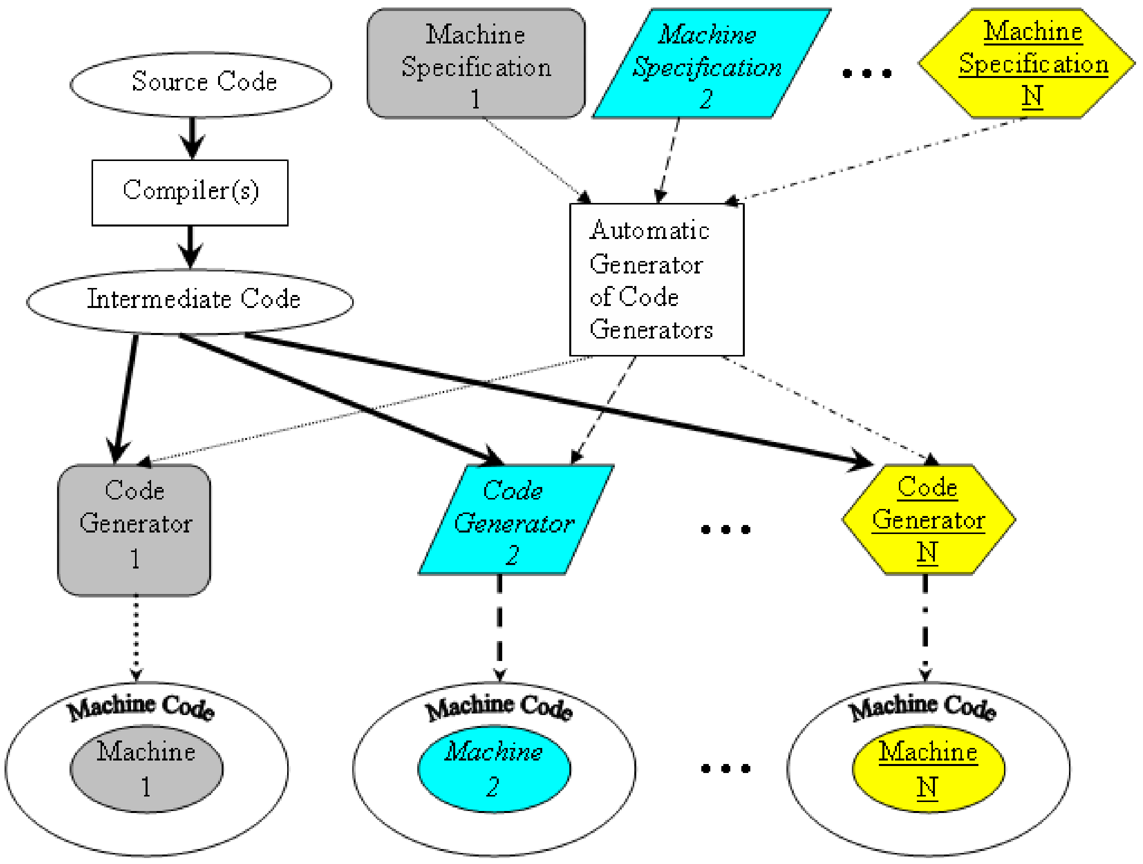


Figure 4.5: Using an automatic generator of code generators.

The first one, standard C code generation strategy, is not applicable because to control robots it is necessary specific instructions not supported by the C programming language. And even if these instructions may exist, as a C library (that exists for some robots), it would be necessary to develop C compilers for all of the existing industrial robots versions. And if a robot is modified, or a new robot version is created, the C compiler should be modified, or a new one should be developed.

The second one could be used on the next generation industrial robots, but for the current and the old ones, that are still in use, if it is possible, it must be necessary to develop many virtual machines. But for some current and old industrial robots it is impossible, because the robot controllers are limitated and/or fully used for the robot purposes.

The third one has the same problem of the one that generates code directly for the machine. The industrial robots have closed-architectures.

4.4 Off-Line Programming Environments

As the used industrial robots programming languages are imperatives, which means that the user must describe the program control flow and the robot motion, another approach was created, and it is very used, that is the off-line programming environments. First because they are simulated environments, so the user can simulate the environment where the robot will acts, and can simulate the behavior of its program on the created virtual environment, in an off-line way; and secondly because **some** existing commercial and very used environments allows the user to know only the simulation language, instead of the different ones that are used on different robots.

There are a lot of programming environments with simulation features, but none of them has an automatic tool for translators generation. So, they are limited to some robots.

Some examples of off-line programming environments are described below:

4.4.1 Workspace

Workspace is a sophisticated simulation system designed for the robotics industry. It is a 32-bit Microsoft Windows application and uses the ACIS CAD kernel to create rendered images. It allows graphical off-line programming, supporting the following list of robots: ABB, Adept, Comau, Daihen, Fanuc, Flow, Kawasaki, Kuka, Mitsubishi, Motoman, Nachi, Panasonic and Staubli. There are translators to many robot languages like [Tec07a]:

- Rapid (for ABB robots)
- TP (for Fanuc robots)
- Inform II (for Motoman robots)
- AS (for Kawasaki robots)
- KRL (for Kuka KRL robots)
- Slim (for Nachi robots)
- Pres (for Panasonic robots)

– G-Code (for Siemens robots)
– PDL2 (for Comau robots - this one is in testing phase)

With workspace it is possible to [Tec86]:

– Model work cells using the Workspace built in CAD system;
– Import CAD data from other CAD systems;
– Use robots, mechanisms, or tooling from the workspace library, or model your own;
– Automatically generate robot paths;
– Graphically edit robot programs;
– Optimize for Cycle time, reach, collision detection;
– Generate robot program off-line.

4.4.2 RobotStudio

RobotStudio [AB07] is a offline programming and simulation environment developed by ABB, to be used with its industrial robots. With RobotStudio, it is possible:

– to visualize solutions and layouts;
– to program new parts without interrupting production;
– to optimize robot programs to increase productivity;
– to detect collisions;
– to generate automatically the robot positions;
– to import data from major CAD formats including IGES, STEP, VRML, VDAFS, ACIS and CATIA;
– to download whole robot program to the real system without any translation.

RobotStudio runs on regular PCs under Microsoft Windows environment, and includes Visual Basic for Applications to customize and extend functionality.

4.4.3 COSIMIR

COSIMIR´is a PC-based software for 3-D modelling, programming and simulation of production cells with industrial robot systems, made by Festo. It runs under Windows NT/2000/XP.

With COSIMIR it is possible to [Co07]:

– Import different CAD formats, such as DXF, STL or VDAFS2.0;
– Test and work with the robot in simulation mode;
– Detect collisions and optimize cycle times;
– Teach robot movements with the mouse;
– set inputs and outputs interactively, to simulate the interaction of the robot with its environment;
– Simulate multiple robot systems;
– Simulate sensor;

- communicate with other PC applications or external control systems by DDE interface;

COSIMIR Professional supports programming in the following languages:

- IRL (DIN 66312)
- Rapid (for ABB robots)
- KRL (for Kuka KRL robots)
- V+ (for Adept and Stäubli robots)
- Movemaster Command (for Mitsubishi robots)
- Melfa Basic III (for Mitsubishi robots)
- Melfa Basic IV (for Mitsubishi robots)

4.4.4 Others

There are many other programming environments with simulation features. The ones that are presented on this section are the ones that are not really applied to the same purposes of this thesis, because they are not applicable to industrial robots; or they are only a simulation tool, without code generation.

X-Arm

X-Arm is a programming language for an OpenGL simulator of robotic arms, that has educational purposes. It does not generate code for any real robot [dBMB03];

Microsoft Robotics Studio

Microsoft Robotics Studio [Cor07] is a Windows-based environment for academic, hobbyist, and commercial developers to easily create and simulate robotics applications across a wide variety of hardware.

The features of the Microsoft Robotics Studio environment include:

- end-to-end Robotics Development Platform;
- lightweight services-oriented runtime;
- a scalable and extensible platform;
- a visual programming tool (Visual Programming Language - VPL) for creating and debugging robot applications, web-based and windows-based interfaces;
- 3D simulation;
- access to robot's sensors and actuators via a .NET-based concurrent library implementation;
- support for a number of languages including C# and Visual Basic .NET, JScript, and IronPython;
- VPL also directly supports development to target Windows CE and Windows Mobile devices;

– support for add-ons. (like Maze Simulator, a program to create worlds with walls that can be explored by a virtual robot),

As it can be seen, Microsoft is now intended for robotics software also. However, at this moment, it is not applied to industrial robots. But maybe in the future. Microsoft Robotics Studio has a lot of partners, and one of them is the KUKA Roboter GmbH. KUKA will provide an MSRS-based simulation environment for educational purposes. The goal is to teach robotic students at universities how to design and implement (virtual) robot controls. This simulation framework will contain all the services needed in order to build a complete virtual robot control which the students can use to experiment with internal parameters and to replace single services with their own implementation which can all be orchestrated using MSRS Visual Programming Language.

Player project

The Player Project (formerly the Player / Stage Project or Player / Stage / Gazebo Project) [Sof07] creates free software that enables research in robot and sensor systems. Its components include the Player network server and Stage and Gazebo robot platform simulators. It offers many similar functions of the Microsoft Robotics Studio, presented above, but has different licensing conditions.

These run on POSIX-compatible operating systems, including Linux, OS X, Solaris and the BSD variants.

Features include:

– robot platform independence across a wide variety of hardware, but not applicable for industrial robots;
– C, C++ and Python programming languages are officially supported, Java, Ada, Ruby, Octave, and others are supported by third parties;
– a minimal and flexible design;
– support for multiple devices on the same interface;
– on-the-fly server configuration.

Player is a device server that provides a powerful, flexible interface to a variety of sensors and actuators (e.g., robots). Because Player uses a TCP socket-based client/server model, robot control programs can be written in any programming language and can execute on any computer with network connectivity to the robot. In addition, Player supports multiple concurrent client connections to devices, creating new possibilities for distributed and collaborative sensing and control.

Stage is a scalable multiple robot simulator; it simulates a population of mobile robots moving in and sensing a two-dimensional bitmapped environment. Various sensor models are provided, including sonar, scanning laser rangefinder, pan-tilt-zoom camera with color blob detection and odometry. Stage devices present a standard Player interface so few or no changes are required to move between simulation and hardware. Many controllers designed in Stage have been demonstrated to work on real robots

Gazebo is a multi-robot simulator for outdoor environments. Like Stage, it is capable of simulating a population of robots, sensors and objects, but does so in a three-dimensional world. It generates both realistic sensor feedback and physically plausible interactions between objects (it includes an accurate simulation of rigid-body physics). Gazebo presents a standard Player interface in addition to its own native interface. Controllers written for the Stage simulator can generally be used with Gazebo without modification (and vise-versa).

Pyro

Pyro (Python Robotics) [Bla07] provides a programming environment for exploring advanced topics in artificial intelligence and robotics without having to worry about the low-level details of the underlying hardware. However, Pyro is used for real robotics research.

Because Pyro abstracts all of the underlying hardware details, it can be used for experimenting with several different types of mobile robots and robot simulators. It is possible to use Pyro to program many different mobile robots, allowing code to be shared across platforms as well as allowing students to experiment with different robots while learning a single language and environment.

The currently supported robots are the Pioneer family (Pioneer, Pioneer2, PeopleBot robots), the Khepera family (Khepera, Khepera 2 and Hemisson robots), the AIBO, the IntelliBrain-Bot, and the Roomba.

Pyro has the ability to define different styles of controllers, which are called the robot's brain. For example, the control system could be a neural network, behavior based, or a symbolic planner. One unique characteristic of Pyro is the ability to write controllers using robot abstractions that enable the same controller to operate robots with vastly different morphologies.

Pyro is also integrated with several existing robot simulators including Robocup Soccer, Player/Stage, Gazebo and the Khepera simulator.

4.5 Robot Control Systems

Today, the industrial robots programming tasks are done normally by the use of the own industrial robot programming language, or by the use of a proprietary off-line programming environment. However, in many research labs, there are people working on a solution to improve the industrial robot software design. And one solution leads with the creation and adoption of a new standard robot controller.

This idea leads with architecture details and the kinematics of the robot. Comparing to personal computers domain, the robot control system can be viewed as an operating system, that must have an interface for the user, and must control all the hardware. However, the industrial robot programs has basically one kind of instruction that differs them to the personal computer ones: the moving instructions. And the robot controller must deal with moving instructions, that is not a simple task. On a robotic arm there are, normally, 6 degrees of freedom. So, to move to a single position in the workspace, it is necessary to control the 6 axis, evaluating their internal sensors and controlling each individual axis, and all of these at the same time. So, it is not so easy to create a standard controller for different robots, but if one is adopted, it would be easy to create a standard language.

On this section, two robot control systems that exists, one commercially, will be presented.

4.5.1 MRROC++ Framework

MRROC++ [Zie99] is a programming framework that facilitates the implementation of single and multi-robot system controllers, for robots of different types, that execute diverse tasks. Its structure was derived by formalizing the description of the general structure of multi-robot controllers.

MRROC++ controller program consists of several concurrently running multi-threaded processes, that can be distributed over the local network.

MRROC++ is derived from C++, thus it is object oriented. Its based controller is a real-time application, currently running under control of the QNX Neutrino ver.6.3 RTOS. Therefore, MRROC++ source code can be readily ported from QNX to another real-time operating system with POSIX API.

Work is being continued on ever more complex systems, which in due time will be a basis for the creation of a fully fledged service robot. Currently the abilities of hearing and speaking are being added. The robot will be able to comprehend Polish language commands and respond to them by a synthesized voice. Moreover, two-handed manipulation using visual and force feed-back is integrated with the system [ZSW05].

4.5.2 Universal Robot Controller

The Universal Robot Controller (URC) is a PC open architecture control system for industrial robots that replaces the original industrial robot controller. It was developed to extend the life of existing production robots and redeploy them by providing an upgrade alternative to buying a new robot/controller system. All the robot kinematics information and motion control are resident in the URC [Tec07b].

To program the URC controlled robots, it is used the language RoboScript, that was presented on previous chapter.

Some features of URC are:

- Windows OS supports multi-tasking and Web services;
- Real-time Operating System (RTOS) on motion control module;
- Offline programming (OLP) and simulation software included;

As it was said before on the previous chapter, this strategy transfers the standardization problem from the programming language to the robot drivers. And despite it is an open architecture, to adopt such strategy, it is necessary to acquire the URC. The users only changes the "*proprietary*". However, it is better to have one "*proprietary*" to control various robots instead of having various "*proprietaries*" that probably do not "*talk with each other*".

4.6 Chapter's Considerations

In this chapter it was presented some other approaches and tools for the industrial robots programming, like programming environments, higher-level languages and standardization of robot controllers.

The goal of this chapter was to present some very interesting strategies for industrial robots programming. However, some of them simply changes the problem place: instead of being a compiler problem, it is a driver problem. To other, someone has to create the specified translator for a given target robot.

So, none of them facilitate the programming task: allowing a higher level programming; and working on the automatization of the code generation for different robots (granting portability for the source code). These are the main requirements that are treated on this work.

Chapter 5

Front-End

After designing the system model and writing its specification in a high level way, the program must be implemented. To do this, there are two ways: the developer writes all the program by hand (maybe applying some systematic translation rules); or he uses a compiler that transforms the specification into a runable program (machine code or still a high-level language that is then translated into the target code). To facilitate the second approach, it is recommended the use of an environment specially tailored for the application scope of that language, which contains everything necessary for the edition and compilation. This environment should be, also, easy-to-use. So, it must have a friendly interface.

The compiler is normally divided into two components: the front-end (FE) that reads the input and parses it to recognize its meaning (it implements the lexical, syntactic and semantic analysis); and the back-end (BE) that generates the target code and optimizes it.

As higher level is the front-end, the programmer can express exactly what he intends to do. Such a description should be simple, and as closed to the specification of the problem as possible.

In the context of the robot programming, an example of declarative language (proposed some years ago [Tos93]) is RS — a reactive and real time language relying on the principle of productions systems, ie, on the rule-based paradigm (condition-reaction set of rules). Some experiments where made, like controlling a Nachi industrial robot [Pio98], or controlling a simulated manufacturing cell [Arn00], in order to assess the use of RS in that context. At this moment, RS language is not a component of GIRo, but the main idea of this approach cames from some good experiments done with this language.

On the other hand, one well-known description language for industrial automation applications is the Grafcet [Tel82], that is suitable to be the visual interface for the front-end, and that is able to produce a textual description from the graphical specification.

In this chapter, a brief introduction to RS language and Grafcet will be done, explaining the importance that both of them had to the creation of GIRo.

```
P -> M+
M -> B+ | R+
B -> R+
R -> s_ext#[s_int] ===> [acção]
```

Figure 5.1: RS Grammar.

5.1 RS Language

The RS language [Tos93] is a synchronous language intended for the development of programs that must react immediately to stimulus coming from an external environment. These programs constitute the kernel that corresponds to the central and more difficult part of a reactive system. The RS language allows to program the so-called productions, i.e. the pairs <condition, action>, that describes the system's behavior.

Actually, RS language provides a suitable notation to represent reactive behaviors; a program direct specifies the internal transformations and the emission of output signals that shall occur as consequence of each external input stimulus.

As a result of the synchronism hypothesis, an RS program can be seen as if its internal operations were executed by an infinitely fast processor. This means that each reaction is accomplished in an infinitesimal time, making the output signals synchronous with the input signals. The synchronism hypothesis corresponds to consider ideal systems, that react instantaneously to each external stimulus, with a transformation of the internal state and with an emission of output signals, like a Mealy machine [HH79]. As the operations are executed in an infinitesimal time, the computer automatically satisfies the external timing constraints.

As it occurs with Esterel [BG92], Lustre [NH91], and other synchronous languages, RS is not a self-sufficient language; the I/O interface and the data handling components should be provided by a host language or by the execution environment.

5.1.1 Structure and operation

An RS program works with classical variables that are shared in a module level, and with signals that are used for external communication and for internal synchronization.

The signals are partitioned in: input signals, the only that originate reactions; output signals, used to communicate results to the environment; and internal signals, used for synchronization or internal communication. Besides these, there are special input signals, called sensors, whose values are always accessible in any reaction.

An RS program is composed by a set of modules, which are composed by a set of boxes, containing a set of reaction rules, as can be seen on figure 5.1. Modules and boxes allow to structure a program in a better way, but they are not really necessary; in fact, any program can be constituted by a single set of rules.

```
ext_sig#[int_sig] ===> [action]
```

Figure 5.2: RS reaction rule example.

A reaction rule has the form: "trigger condition" => "action", where the trigger condition is a not empty list of internal and/or input signal, having at most one input signal (figure 5.2). A given set of open (turned on) signals triggers simultaneously off all rules whose trigger conditions are contained in this set, closing (turning off) the signals specified in these conditions. In spite of having this parallel execution of rules, there is no indeterminism in the reaction; this happens because the RS language forces the correct sharing of variables and signals.

The trigger conditions allow the programmer to map the states of the external environment into internal states of the program. Each one of these internal states defines a set of rules which can be triggered by an external stimulus. The internal variables allow the system to keep information from a reaction to another. There are conditional rules which allow the system to react according to conditions that are evaluated at run time.

The two most used actions are the commands *emit* and *up*. The first one sends an output signal to the environment, while the second one turns on an internal signal.

One interesting construct of the language is the mechanism to handle exceptions, that is, conditions that cause abrupt changes in the state of the reactive system. These conditions can be raised internally or can be signalled from the external environment.

The execution of an RS program is accomplished in a sequence of steps, where each step consists of the parallel execution of all rules with firing condition true. The first action of a step is an implicit action that turns on all the signals contained in the true conditions. As the execution of a rule can turn on signals, this originates a sequence of steps that only finishes when the set of on signals is not enough to fire any reaction rule. In this situation, the program waits until some external signal arrives to start a new reaction (sequence of steps).

Some characteristics of the RS language are: it is possible to call procedures from the actions rules; it is possible to read the value of any signal within any rule; it is possible to use variables of type record (structured variables); it is possible to declare temporary internal signals which are turned off automatically at the end of each reaction; it is possible, at any state, to define a common treatment for all valid external signals which are not explicitly awaited in this state.

Another feature of the RS language is that is possible to use the subsumption architecture [Bro86] to create programs, because its control mechanisms, the inhibitor and the suppressor, were included [Arn98].

5.1.2 Simple example

To illustrate the RS syntax, an RS program is listed, in the Figure 5.3. Although the program has no practical importance, it verifies if a mouse button was pressed with a

```
module mouse: [input :[click, tick],
 output:[single, double],
 signal:[awaitClick, awaitAny],
 var   :[count],
 initially: [up(awaitClick)],
 tick#[awaitClick] ===> [up(awaitClick)],
 click#[awaitClick]===> [count:=2, up(awaitAny)],
 tick#[awaitAny]   ===> case [
   count>0 ---> [count:=count1, up(awaitAny)],
   else    ---> [emit(single), up(awaitClick)] ],
 click#[awaitAny]  ===> [emit(double), up(awaitClick)]
].
```

Figure 5.3: Example of a single RS program.

double or a single click. The program has two input signals: tick, that means a clock impulse, and click, that means the pressing of the mouse button, and two output signals: single and double.

Initially, the internal signal *awaitClick* is turned on, which means that only the first two rules can be triggered. This situation changes only when a *click* signal arrives, triggering the second rule that sets the counter the value 2 and turns on the internal signal *awaitAny*. After this, if another click signal arrives, an output signal *double* is sent to the environment. If this external signal does not arrives, after 3 *tick* signals, an output signal single is sent to the environment. After sending a signal to the environment, the program returns to the initial situation.

5.1.3 Output from the RS Compiler

The RS compiler produces a finite automaton as output, like some other reactive and synchronous languages. The automaton, that corresponds to the description of a finite state machine, is defined by the word AUTOMATON. Each line in the file has three components. The first of then, in the left side, can be: a number, that represents a single state; or the word init, that corresponds to the initial state. The second component contains the name of the external input signal that will stimulate the third component, which are the rules to be executed if the signal is active in the current state. The file that contains the rules to be executed by the automata starts with the word RULES. Each rule has the following format: a number to identify itself, which that will be used by the automata; a condition, that must be evaluated to trigger off the rule; and the action to be take, that contains one or more commands, separated by commas. The asterisks between the rules are used to delimit the ones that can be executed in parallel.

In the following figures, it can be seen the generated automata (figure 5.4) and rules (figure 5.5) for the RS program that controls a mouse (figure 5.3).

```
AUTOMATON:
  init       [1,*,go_to(1)]
   1   tick  [2,*,go_to(1)]
   1   click [3,*,go_to(2)]
   2   tick  [[4  1,*,go_to(2)], [4 - 2,*,go_to(1)] ]
   2   click [5,*,go_to(1)]
```

Figure 5.4: RS automaton generated by the RS compiler.

```
RULES:
  Module mouse:
  1. [ ] ===> [ ]
  2. [ ] ===> [ ]
  3. [ ] ===> [count:=2]
  4. Case:
  41. [ ]{count>0} ---> [count:=count1]
  42. [ ]{else} ---> [emit(single)]
  5. [ ] ===> [emit(double)]
```

Figure 5.5: RS rules generated by the RS compiler.

5.2 Programming Robots Using RS Language

To use the RS language to develop robotic programs, it is necessary:

- identify the sensors that will be necessaries and define them as input signals for the RS program;
- specify a special input signal responsible for starting the execution of the automaton. It corresponds to the robot state *ready*, which means that the robot is turned on and with a program to be executed in its memory;
- identify the operations that should be executed by the robot aiming to work correctly and defines them as output signals, that will be sent to the environment by the *emit* command. These signals may have some parameters necessaries to execute the specific function.

The points where the robot must pass are represented by identifiers. The definition of this points does not matter to the developing of the RS program, but it is needed for the execution of the program. So, this definition os points must be done on another moment, normally by the use of the teach-box, as it was presented in chapter 2.

5.3 RS Case Studies

Some case studies were worked out, to validate the use of the RS language to programm robots, like:

```
rs_prog nr_0001:
[input : [ready, sensor],
 output: [go(P), end, tool(C), use(B), shift(D,X,Y,Z)],
 module move:
 [input    : [ready, sensor],
  output   : [go(P), end, tool(C), use(B), shift(D,X,Y,Z)],
  signal   : [t1, t2, t3, t4, t5],
  var      : [count1],
  initially: [emit(use(1)), emit(tool(0)), count1:=0, up(t1)],
  ready#[t1]  ===> case
         [count1=10 ---> [count1:=0, emit(end),up(t1)],
          else      ---> [count1:=count1+1, emit(go(p1)),up(t2)],
        [t2] ===> [emit(go(p2)),emit(shift(0,0,0,100)),up(t3)],
        [t3] ===> [emit(go(p3)),emit(shift(0,0,0,0)), up(t1)],
  sensor#[t1] ===> [emit(go(p4)),up(t4)],
        [t4] ===> [emit(go(p5)),up(t5)],
        [t5] ===> [emit(go(p6)),up(t1)],
 ] ].
```

Figure 5.6: RS program to control the Nachi SC15F robot.

– An RS program to control a Mobile Robot [AT98]. This mobile robot must follow some interesting object, avoiding obstacles, and must wander randomly if there is no interesting object. It was used the Subsumption Architecture [Bro86] as a modelling technique. It was proved that the RS language is a useful development toll for mobile robots programming. This control was simulated;
– An RS program to control a Manufacturing Cell [Arn00] that contains: two robots, one conveyor belt, one camera, one lathe, and one store. The goal was to present an extended version of the RS language. This version has the control mechanisms of the Subsumption Architecture [Bro86] implemented on it. So, the system that controls the manufacturing cell was developed using the Subsumption Architecture as a modelling technique, and the RS language as a implementation tool. This control was also simulated;
– An RS program to control an Industrial Robot [ALHF02]. It was created a translator to make possible to use the RS language to control a Nachi Industrial Robot. This translator proved that the RS language is a powerful development toll for industrial robots programming;

5.3.1 RS Program Example that Controls an Industrial Robot

One example of an RS program that was used on Nachi SC15F robot can be seen on figure 5.6 [Pio98].

The aim of this program is to make the robot move through the P1, P2 and P3 points. But, if the sensor *sensor* is active, the robot should move through P4, P5 and P6 points.

```
10 USE 1
20 TOOL 0
30 V1%=0
40 *S1
41 JMPI 4, I1
50 IF V1%=10 THEN *L2X1 ELSE *L2X2
60 *L2X1
70 V1%=0
80 END
90 GOTO *S1
100 *L2X2
110 V1%=V1%+1
120 MOVE P,P1,T=3
130 MOVE P,P2,T=3
140 SHIFTA 0,0,0,100
150 MOVE P,P3,T=3
160 SHIFTA 0,0,0,0
170 GOTO *S1
180 MOVE P,P4,T=3
190 MOVE P,P5,T=3
200 MOVE P,P6,T=3
210 GOTO *S1
```

Figure 5.7: Generated program to be executed at Nachi SC15F robot.

The robot must perform these tasks 10 times, while it is on (signal *ready* is active). At the end of the cycle, the signal *end* is sent to the environment, ending the execution of this program.

As it was described in subsection 5.1.3, when this RS program is compiled, it is generated one automata. [Pio98] created a translator that can translate this generated automata into a Nachi SC15F robot program. This code can be seen of figure 5.7.

With this translator, it was proven the possibility of using the RS language to control industrial robots, allowing a faster programming task and the use of a truly high level language, where the programmer can abstract from the robot aspects.

5.4 Grafcet

To conceive automatized systems, it is used a methodology that has three principles [Tel82]:

– The system must be decomposed in two parts: a command one (corresponds to the automata of the system); and an operative one (corresponds to the physical system, that is responsible for executing the commands that comes from the command part);
– A precise description of the automatism functioning must be given by the command part. To achieve this, it is recommended to divide the description into two successive

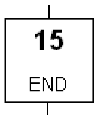


Figure 5.8: Example of Grafcet's step.

and complementary levels: The first one describe the behavior of the command part (the functional specifications that represents what the automatism must do); and the second one that adds to the functional specifications the indispensable details for the correct functioning that comes from the technological and operational specifications;

– It is necessary to adopt one language to write this description.

Grafcet is a description method of the command part of an automatized system that attends the above principles, that can be used in both levels.

Nowadays, there are several Grafcet variations. However, to achieve our goals, it will be presented the standard Grafcet, that was used in this work.

5.5 Standard Grafcet

Because Grafcet [Tel82] is a well known diagrammatic specification language for industrial automation applications, it was decided to use in GIRo environment a graphical interface that should looks like the Grafcet, as told above.

So before describing the FE interface built, we discuss the basic concepts that Grafcet uses to represent automatisms, that are [Gra00]:

– *Step*, represents a partial state of system, in which an action was performed. The step can be active or idle. The associated action is performed when the step is active, and remains asleep when the step is idle;
– *Transition*, links a precedent step (one or several) to a consequence step (one or several), and represents the actions flow. It describes a state change. Changing is under the control of two conditions:
  1. every step previous to the transition must be active (and the actions executed),
  2. a boolean condition associated with the transition, must be true.

5.5.1 Steps

The step is represented by a square, that is referenced by a number and, optionally, a symbolic name, to show the goal of this step (figure 5.8). Some steps are the initial ones. Those are represented by a square with double lines; they are active at the beginning of the execution.

The actions that must be performed when this step is active, are described literally in another square, located on the right side of the step, as can be seen on figure 5.9. The

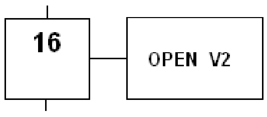


Figure 5.9: Example of Grafcet's step with action.

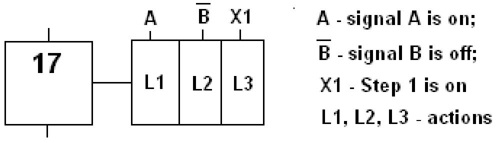


Figure 5.10: Example of Grafcet's step with action and conditions.

execution of each action can be dependent on a logic condition, on the value of input variables, auxiliary variables, and on the state of other steps (figure 5.10).

5.5.2 Transitions

The transition is represented by a thick horizontal line, that cross the oriented line between two steps. It has a condition that is responsible for the state changing of the system. Each condition is a predicate applied to exterior informations, auxiliary variables, and the state of other steps. One transition example can be seen on figure 5.11.

These conditions also can change the state of variables, where its value can change from 0 to 1, or from 1 to 0, as can be seen on figure 5.12.

It is possible to use temporal conditions, like `t/8/10`, which means that the condition holds after 10 seconds after the last activation of step 8.

5.5.3 Alternative and Simultaneous Sequences

When one transition is linked to two or more steps, like that in figure 5.13, it is called *simultaneous sequence*, which means that all the following steps will be executed simultaneously,

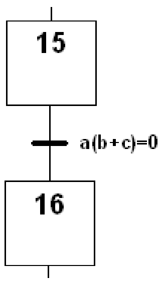


Figure 5.11: Example of Grafcet's transition.

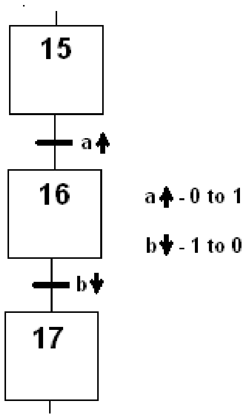


Figure 5.12: Example of Grafcet's transition that changes the state of variables.

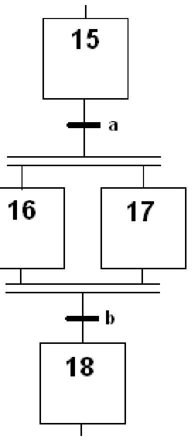


Figure 5.13: Example of Grafcet's simultaneous sequence.

until the steps are linked to only one transition. This kind of sequence is represented by a double horizontal line.

Alternative sequences are those where a step is linked to more then one transition, where these transitions should be mutual exclusive. It means that, only one of the following steps will be executed if its transition actually happens (figure 5.14).

5.5.4 Evolution Rules

To define completely Grafcet, it is also necessary to define the evolution rules. They are responsible for the evolution between the steps, deciding when each one must be active or inactive. The evolution rules are:

1. Rule 1 - Initialization: indicates which step must be active at the beginning of the automatism. This step is represented by squares with double lines, and it is activated unconditionally at the beginning of the automatism;

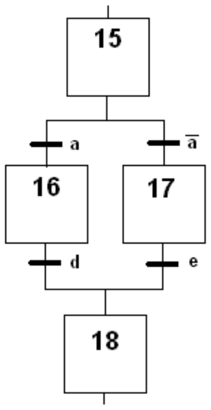


Figure 5.14: Example of Grafcet's alternative sequence.

2. Rule 2: A transition can be validated or invalidated. It is validated when all the precedent steps are actives and its conditional predicate is true;
3. Rule 3: The transposition of such transition actives all of the following steps, deactivating all of the precedent ones;
4. Rule 4: All the transitions that are simultaneously transposable are simultaneously transposed;
5. Rule 5: During the automatism, if some step must be deactivated and activated simultaneously, it remains active.

5.6 Why to Use Grafcet instead of RS Language?

Initially, the RS language [Tos93] was chosen to program robots, due to the results previously achieved [Pio98] [Arn98], and also because it has some interesting features like:

– It is oriented to the specification and implementation of reactive systems, that are drove by stimulus that comes from the external environment, that will trigger off the actions associated to them;
– It is a simple language, that uses rules like `condition => action` as its commands.
– It facilitates the programming task, allowing the programmer to worry about the logic of the system, expressing the correct reasoning about reactive systems;
– It allows for programs to be structured in a set of modules, that can be composed by a set of boxes, that contains the rules.
– It is a parallel and distributed language, as its modules can run in different processors/machines;
– It is a real time programming language, because it adopts the synchronism hypothesis, where its reactions are executed in zero time, making its output signals synchronous with the input ones;
– It has some components to handle exceptions conditions;
– It is a high level declarative language.

Based on the good results obtained from the case studies, it was started a bigger project aimed at using the RS language to control different industrial robots [AHF03] The main goal was to facilitate the programming task, and to grant portability of the source code, making possible to use the same RS program to control different robots. To achieve this, it would be developed a generator of code generators, responsible for automatically produce the code generator for different robots, based on the same intermediate representation.

To facilitate the industrial robot programming, it was proposed a graphical front-end for the RS language, based on the GRAFCET specification diagram. The goal was to use this well known diagram to create programs to control industrial robots.

The Grafcet was chosen because it is a nice diagrammatic specification approach very used on the automation areas, and because the RS language supports completely the Grafcet specifications and vice-versa.

However, the analogies between Grafcet and RS language had a consequence: the automata generated by a RS program corresponds exactly to the Grafcet diagram. It does not make sense to create a program under a Grafcet diagram, translate it into an RS program, and after compiling, to get an automata that corresponds exactly to the first Grafcet diagram. So, we concluded that it was not necessary to use both the RS language and the Grafcet.

Grafcet was chosen as the front-end of GIRo in place of RS language because of: Grafcet is a well known specification diagram, while RS language is an unknown programming language; It is easier to program in a well known graphical environment instead of writing programs by hand.

So, after some years working with the RS language, after the good results that encouraged me to do this work, it was decided to abandon this language. However, the know-how obtained helped me a lot in the progress of this Ph.D. work.

5.7 Chapter's Considerations

In this chapter it was presented the two possibilities of front-ends to be adopted: The RS language, because of the good results from previous works; and Grafcet, a well-known specification diagram.

Both possibilities were implemented, and, after the discussion supported along the chapter, Grafcet was the chosen one, according to the principles explained above.

With Grafcet, the GIRo environment will be easy to use by the industrial robot programmers.

Chapter 6

Intermediate Representation

In traditional compilers [Muc97], the interface between the front-end (FE) and the back-end (BE) is an intermediate representation (IR) that should be independent of source and target languages.

That independence makes possible the generation of programs that will be executed on different robots. The IR is composed by a set of instructions and data representations that is common to the majority of industrial robots. In traditional compiler architectures, the FE translates the source program, written by the programmer, into this intermediate representation, and then the BE will translate the IR into machine code.

This representation must be as simple as possible to make the code generation easy and efficient.

In this chapter, some intermediate representations are presented. After, InteRGIRo, that is the created intermediate representation used by GIRo environment, is presented, explaining the necessity of the creation of a new IR, instead of adopting some existing one.

6.1 Some Existing Intermediate Representations

There are several intermediate representations, which some of them are well known on the computers domain. On the following subsections, some existing intermediate representations are presented.

6.1.1 Three Address Code (TAC)

Three Address Code (TAC) [ASU86] is a popular format for intermediate languages. It is a linearized representation of a syntax tree in which explicit names correspond to the interior nodes of the graph. The reason for the name is that each statement usually contains three addresses, two for the operands and one for the result, as can be seen on the following example:

`x` := `y` *op* `z`

where `x`, `y`, and `z` are names, constants, or compiler generated temporaries; *op* stands for any operator.

x + y * z

corresponds to

t1 := y * z

t2 := x + t1

where t1 and t2 are compiler generated temporary names.

It is also possible to write other instructions in Three Address Code format, like: copy; conditional and unconditional jumps; unary operations; procedure calls; indexed variables and pointers manipulation [Mat99].

6.1.2 Register Transfer Language (RTL)

The Register Transfer Language (RTL) is a historical intermediate representation, created for the Peephole Optimizer [FD80], that is aimed at being syntactically closed to the target language (assembly), but abstract enough to be platform independent. It looks like three-address code, using simple assembly-like operations, however, these operations manipulate registers.

It was created to be independent from the front-end and also from the back-end, and to facilitate the application of optimization algorithms.

Its main features are:

– its intermediate code is represented by a list of instructions;
– its instructions are based on three-address code representation;
– its instruction set has generic operators chosen from the instructions used by the most common processors;
– it manipulates only pseudo-registers;
– it does not have physical limitations, as the amount of registers.

One of the main advantages of RTL resides on the simplicity of the operations, that allows the code optimization and the final code generation.

6.1.3 GNU Compiler Collection (GCC)

The GNU Compiler Collection [Sta00] is a set of programming language compilers that aims at being portable to any machine. It is free software and it is the standard compiler for the free software Unix-like operating systems and Apple Mac OS-X.

Originally named the GNU C Compiler (it only handled the C programming language), it was extended to compile C++, having front-ends for many other programming languages (like Fortran, Pascal, Objective-C, Java, and Ada), and back-ends for several architectures (like Alpha, SPARC, PowerPC, x86 and others) [Mat05].

For most of its history, GCC has compiled statements directly to an intermediate language GCC - RTL (GCC - Register Transfer Language, that is an intermediate representation based on the historical RTL, that was presented in the subsection 6.1.2). GCC - RTL has been a very useful intermediate language for low-level optimizations, but has significant limitations that keep it from being very useful for higher level optimizations [Mer03] [Nov04]:

– Its notion of data types is limited to machine words, without ability to deal with structures and arrays as a whole;
– It introduces the stack too soon: taking the address of an object forces it into the stack, even if later optimization removes the need for the object to be addressable;
– There is no single tree representation in GCC. Each front end defines its own trees;
– Trees are arbitrarily complex. This is a problem for optimizations, and it must take into account many different variations.

So, some other intermediate representations, rather than GCC - RTL, where developed, to simplify portability and cross-compilation.

Generic

GENERIC [Mer03] is a language-independent tree representation generated by each front end. It is used to serve as an interface between the parser and optimizer. GENERIC is a common tree representation that is able to represent all the constructs needed by the different front-ends supported by GCC, while removing all the language dependencies.

The purpose of GENERIC is simply to provide a language-independent way of representing an entire function in trees.

Gimple

GIMPLE [Mer03] is a simplified intermediate representation based on GENERIC, that is used for target and language independent optimizations (e.g., inlining, constant propagation, tail call elimination, redundancy elimination, etc). It is a very simple C-like three-address language that looks pretty straightforward to analyze and keeps all the high-level attributes of data types. The particular subset chosen (and the name) was heavily influenced by the SIMPLE intermediate language used by the McCAT compiler project at McGill University. [AL07].

GIMPLE retains much of the structure of the parse trees: lexical scopes and control constructs, such as loops, are represented as containers, rather than markers. However, expressions are broken down into a 3-address form, using temporary variables to hold intermediate values.

Gimple differs from GENERIC in that the GIMPLE grammar is more restrictive: expressions contain no more than 3 operands (except function calls), it has no control flow structures and expressions with side-effects are only allowed on the right hand side of assignments.

GCC - RTL

GCC - RTL (in this work, it is used this name to differentiate from the historical RTL, presented in the subsection 6.1.2) is a term used to describe a kind of intermediate representation that is very close to assembly language. The historical RTL has the same name and is the base of a specific intermediate language used in the GNU Compiler Collection.

Actually, it is generated from the GIMPLE representation, transformed by various passes in the GCC middle-end, and then converted to assembly language.

Most of the work of the compiler is done on GCC - RTL. In this language, the instructions to be output are described, pretty much one by one, in an algebraic form that describes what the instruction does, and some other optimizations are done, such as loop optimization, jump threading, common subexpression elimination, instruction scheduling, and others.

The GCC - RTL optimizations are of less importance with the recent addition of optimizations on GIMPLE trees: GCC - RTL optimizations have a much more limited scope, and have less high-level information.

GCC - RTL is inspired by Lisp lists. It has both an internal form (made up of structures that point at other structures), and a textual form that is used in the machine description and in printed debugging dumps. The textual form uses nested parentheses to indicate the pointers in the internal form.

An example of a GCC - RTL sentence can be seen following:

```
(set:SI (reg:SI 140) (plus:SI (reg:SI 138) (reg:SI 139)))
```

that means: "add the contents of register 138 to the contents of register 139 and store the result in register 140."

The sequence of the generated GCC - RTL code has some dependency on the characteristics of the processor for which GCC is generating final code. However, the meaning of the GCC - RTL is more-or-less independent of the target. Similarly, the meaning of the GCC - RTL doesn't usually depend on the original high-level language of the program.

6.1.4 RTL System

RTL System was developed, initially, to be the intermediate representation of the Type Smaltalk compiler. But it has evolved into an independent solution for compilers development, specially for the code optimization tasks [JML91].

RTL System uses the RTL (presented on subsection 6.1.2) language for data representation and manipulation, but it is implemented using the object-oriented paradigm, which added many functionalities.

The RTL System elements are defined as classes, that allows the addiction of many other informations, like the data dependency graph. To obtain the intermediate representation for a source program, it is necessary to instantiate these classes.

RTL System improves the original RTL with the expression trees, allowing the use of optimizations and some other code generation processes from these two representations.

6.1.5 DIR

The intermediate representation proposed in the context of the project Dolphin, the so called DIR (Dolphin Intermediate Representation), that sprang out from the previous BEDS project [Mat99], and is fully implemented at [Mat05]. This project deals with optimization and code generation tools, aiming to offer a framework to build optimized compilers, from the parsers to the code generators, for different machines based on an universal intermediate program representation.

Like RTL System, DIR is also implemented using the object-oriented paradigm, with some other functionalities, as the `view` concept, that corresponds to an abstraction level, facilitating the creation of new middle-level routines (it is possible to have many different views from the intermediate representation, like a perspective of the control flow graph, or a list of expressions trees, etc.)

6.1.6 Java Bytecode

The Java programming language is a general-purpose object-oriented concurrent language created by the Sun Microsystem's. The Java platform was initially developed to address the problems of building software for networked consumer devices. It was designed to support multiple host architectures and to allow secure delivery of software components. To meet these requirements, compiled code had to survive transport across networks, operate on any client, and assure the client that it was safe to run [LY99].

To achieve the above goals, Java does not generate machine code to any particular computer, in the sense of native hardware instructions. Rather, it generates an intermediate code, called Java Bytecode, a high-level, machine-independent code, that may be portable among the computer architectures that have Java virtual machine, that is an abstract computing machine. Like a real computing machine, it has an instruction set and manipulates various memory areas at run time. The bytecode is then interpreted, or run on the Java virtual machine.

Java bytecodes are the machine language of the Java virtual machine (JVM). They can be executed by intepretation, just-in-time compiling, or any other technique that was chosen by the designer of a particular JVM.

A bytecode stream is a sequence of instructions for the Java virtual machine. Each instruction consists of a one-byte opcode followed by zero or more operands. The opcode indicates the action to take.

Each type of opcode has a mnemonic. In the typical assembly language style, streams of Java bytecodes can be represented by their mnemonics followed by any operand values. For example, the following stream of bytecodes [Ven96]:

```
03 3b 84 00 01 1a 05 68 3b a7 ff f9
```

can be disassembled into mnemonics:

```
iconst_0      // 03
istore_0      // 3b
iinc 0, 1     // 84 00 01
iload_0       // 1a
iconst_2      // 05
imul          // 68
istore_0      // 3b
goto -7       // a7 ff f9
```

Java opcodes generally indicate the type of their operands. This allows operands to just be themselves, with no need to identify their type to the JVM. For example, instead of having one opcode that pushes a local variable onto the stack, the JVM has several. Opcodes iload, lload, fload, and dload push local variables of type int, long, float, and double, respectively, onto the stack [Ven96].

The JVM supports seven primitive data types (byte, short, int, long, float, double, char) and three kinds of reference types (class types, array types, and interface types). The values of the reference types are references to dynamically created class instances, arrays, or class instances or arrays that implement interfaces, respectively.

The bytecode instruction set was designed to be compact. The total number of opcodes is small enough so that opcodes occupy only one byte. This helps minimize the size of class files that may be traveling across networks before being loaded by a JVM. It also helps keep the size of the JVM implementation small.

All computation in the JVM centers on the stack. Because the JVM has no registers for storing arbitrary values, everything must be pushed onto the stack before it can be used in a calculation. Bytecode instructions therefore operate primarily on the stack. The JVM was designed as a stack-based machine rather than a register-based machine to facilitate efficient implementation on register-poor architectures such as the Intel 486 [Ven96].

For the sake of security, the Java virtual machine imposes strong format and structural constraints, as it could be seen above. However, any language with functionality that can be expressed in terms of bytecodes can be hosted by the Java virtual machine. Attracted by a generally available, machine-independent platform, implementors of other languages are turning to the Java virtual machine as a delivery vehicle for their languages [LY99].

6.1.7 .NET CIL

Similar to Java framework, Microsoft has created its own framework, called .NET, that has a Common Intermediate Language (CIL) as an intermediate language designed to be shared to all compilers for the .NET Framework. This framework also use its own bytecodes, has its own virtual machine, and is also entirely stack-based. The framework is intended to make it easier to develop computer applications and to reduce the vulnerability of applications and computers to security threats. The primary .NET languages were C#, Visual Basic .NET, C++/CLI, and J#.

Programming languages on the .NET Framework compile into the Common Intermediate Language. In Microsoft's implementation, this intermediate language is not interpreted, but it is compiled, in a manner known as just-in-time compilation, into native code. The combination of these concepts is called the Common Language Infrastructure (CLI).

The Common Language Infrastructure is an open specification developed by Microsoft that describes the executable code and runtime environment that form the core of the Microsoft .NET Framework. The specification defines an environment that allows multiple high-level languages to be used on different computer platforms without being rewritten for specific architectures.

To clarify, the CLI is a specification [Int06], not an implementation, while the Common Language Runtime (CLR) corresponds to the Microsoft's implementation of the CLI, which contains aspects outside the scope of the specification.

Programs written for the .NET Framework execute on CLR environment that manages the program's runtime requirements. The CLR provides the appearance of an application virtual machine, so that programmers do not need to consider the capabilities of the specific CPU that will execute the program. The CLR also provides other important services such as security mechanisms, memory management, and exception handling.

The .NET Framework introduces a Common Type System, or CTS. The CTS specification defines all possible data types and programming constructs supported by the CLR and as they may or may not interact with each other. Because of this feature, the .NET Framework supports development in multiple programming languages. The data types defined by the CTS specification are divided in value types, and reference types [Int06]. In the first ones, the values are self-contained, like boolean, char, string, float, integer, for example; while in the second ones, the value denotes the location of another value. There are for types of reference values:

- Object type, that is a reference type of a self-describing value;
- Interface type, that is always a partial description of a value;
- Pointer type, that is a compile-time description of a value whose representation is a machine address of a location;
- Built-in reference types.

Common Intermediate Language (CIL) is the lowest-level human-readable programming language in the Common Language Infrastructure and in the .NET Framework. It is a CPU (and platform) independent instruction set that can be executed in any environment supporting the .NET framework. Languages which target the .NET Framework compile to CIL, which is assembled into bytecode (like Java Bytecode, that was seen above).

6.1.8 ANSI C Language

ANSI C is used as an intermediate language because the enormous variety of C compilers for different platforms that exist. It is used by many programming languages including Eiffel, Sather, Esterel; some dialects of Lisp (Lush, Gambit), Haskell (Glasgow Haskell Compiler); Squeak's C-subset Slang, and others. Variants of C have been designed to provide C's features as a portable assembly language, including one of the two languages called C–, the C Intermediate Language and the Low Level Virtual Machine.

6.2 Why to Create Another Intermediate Representation?

With the intermediate representations, the front-end becomes independent from the back-end, and some optimizations can be done on the program, without caring about the machine and the programming environment.

There are intermediate representations that are used on compiling systems focusing on the final code generation, for example RTL, Java Bytecode and CIL (the first one is focused on a real machine, while the others are focused on virtual machines). Some others, like the DIR, and RTL System, that use object-oriented intermediate representations, are complete solutions for the development of compilers.

This work is intended to translate the source program into a program that is in another procedural program on a different programming language. So, the purpose of the intermediate representation is simply to connect the front-end with the back-end. It is not necessary to do optimizations, because the generated final code is a program in another programming language, that has a specific compiler for an specific machine. So, the optimizations are done on the target compiler. It is also not necessary to be focused on machine code generation. For the reason that industrial robot programming languages are procedural languages, it is also not necessary to work with object oriented intermediate representations.

Also, and probably the most important, there is the fact that the proposed environment will be used by industrial robot operators, that know nothing, or a little, about compilers, code generators, optimizers, hardware descriptions, intermediate representations, and so on... So, frameworks /environments for developing compilers, like the presented RTL System and DIR, and some others, like Suif [ADH$^+$00], Zephir [ADR98] (that are very interesting ones), would be difficult to use on the industrial robotic domain, because the users of these tools are compilers developers, and not other professionals.

The intermediate representation that would be used must be focused on the translation process between two procedural languages. It must be easily generated and easily translated into a target procedural code. So, none of the intermediate representations that were studied satisfies these criteria on an easily way.

Another intermediate representation was created, and it is named InteRGIRo (Intermediate Representation of GIRo). It looks like Gimple [Mer03], that is based on the Simple [AL07] intermediate language. So, InteRGIRo is also three address code based. It will be presented on the following section.

6.3 InteRGIRo

As it was said before, because the front-end must be focused on the specification of the problem, while the back-end must be focused on the robot, an intermediate representation must be used to connect the front-end with the back-end.

To connect both worlds, and to allow the back-end to generate code for different machines, the back-end needs some target specification.

Normally, the first option for this target specification would be the hardware specification of the robot. But this option was not possible because:

1. It is not a simple task the specification a robot hardware. The user would need to know some technical knowledge about the robot, and some syntactical knowledge to describe this specification for the translator, while the goal of this work is to create a user-friendly environment, accessible for the current robot programmers;
2. It is not possible to access the hardware specification of commercial industrial robots, as it was presented in section 1.1.

As the robot programming language can be seen as a window to access and control the robot hardware (section 1.1), it was chosen, as target specification, a subset of the robot language instructions. So, based on this subset instructions, InteRGIRo (Intermediate Representation between Grafcet and Industrial Robots programming language) has been created.

As InteRGIRo is integrated in a broader project, some explanations of it depends on the Generic Compiler implementation. The Generic Compiler, that is also a component of this work, will be explained in the next chapter, however, some considerations are done in this section to facilitate the understanding.

6.3.1 Basic Idea

The industrial robot programming languages are closer to the robot specification than to the specification of the problem to be solved. They look like Pascal or C computer languages, or even else a lower level than C language. These languages are imperative and procedural.

The procedural programs can be seen as formed by the following elements:

– Data types, with the respective operations that manipulates each data type. The more used data types are: numeric, character, string, boolean, record, array and pointer. For each data type, there are their operators, that normally are the same for all procedural languages (for example, the numeric data types uses arithmetic operators, the boolean ones uses boolean operators, etc). The difference between the same data types from different languages are, if exists, their syntax;

- Conditional and unconditional branches. There are many ways for implementing branches. The simplest ones, that exist in all procedural languages, is a simple `if..then` statement and the `jump` instructions;
- Some specific instructions used by the language to control some specific components that a specific machine may have. For example, all the industrial robot programming languages have a moving instruction, that is necessary to move the robot through the environment.

With this division and explanation, it is possible to say that, with a "high-level" intermediate representation (that is an intermediate representation between procedural languages, instead of one between a language and a machine), the creation of a generic translator is facilitated: the data types and their operations are normally the same; and all of the studied procedural languages has `if..then` statement and `goto` instruction.

In this way, if the intermediate representation has only these kind of instructions, a translator could simply substitute these instructions with the correspondent ones of the target language to generate the target code.

As there are few necessary commands of the target programming language, and they are described on the robot programming language manuals, it would only be necessary, for the user, to take a look at the manual to get all the necessary instructions and describes then to this generic translator, that it will be able to generate code for this new language.

What about the specific instructions? They can be seen as procedure calls, which means commands with arguments. In the robotic domains, there are some instructions, as the moving statements, that are common for all of their programming languages. These specific instructions, in this work, are called actions, and they are not part of InteRGIRo, but InteRGIRo supports them, simply ignoring them (these actions must be treated by the generic compiler, and not by the InteRGIRo).

For the generic translator to deal with these actions, it is necessary to have these kind of instructions on the source language, and someone (possibly the user, taking a look at the manual) should describe these actions, using instructions of the target language, for the generic translator. Doing this, this translator could simply continue substituting the source statements with the target ones.

6.3.2 Data Types

As the scope of this work is to deal with industrial robots programming languages, and as robotics applications normally uses few data types for programming industrial robots (basically two: the numeric and point data types), InteRGIRo was tested and validated with only atomic data types. The composed data types are not treated, at this moment. Also, the abstract data types will not be considered, because the existing commercial industrial robot programming languages do not use this kind of data type.

It is important to say that it is assumed that the point data type corresponds to a atomic one. This hypothesis is assumed to grant portability for the source code, because a point is not a atomic data type, but during the specification of the program it can be viewed as being atomic value, corresponding to a position in space. Of course that, a position in a 3D space is composed by 3 coordinates, and for industrial robot it possibly is composed by more than 3 coordinates, depending on the type of robot. Also, there could exist some other arguments associated with the position of the robot, for example, the position of the TCP. But, when the user is writing the program, he is not worried about the exactly values of each coordinate, because these values are obtained on another moment: or by the use of the teach pendant; or by the project designed in a CAD system. So, to facilitate the programming task, and to grant portability for the source code, it is assumed that a point is a atomic data type.

InteRGIRo, at this moment, supports only atomic data types. However, it is being prepared to support, in future, the structured data types, like records and arrays, and pointers. that are used to allocate variables dynamically. These data types are not supported because there is one crucial operation that was not decided yet: the access to a single value. The string data type, at this moment, is viewed as a atomic one, which means that it is not possible to access a single character in a specific position, as it was explained for the array and record data types.

After these explanations, it can be said that InteRGIRo supports the following atomic data types, with its specific operators:

- Numeric - with the arithmetic operators: +, -, *, /,
  (integer division), % (rest of the division);
- Boolean - with the logic operators: & (and), | (or), ˜(not);
- Character;
- String - with the operators: substring, + (concatenation);
- Point (represents a position in space);
- Time - with the operators: set, get;
- Input - Digital input signal;
- Output - Digital output signal;

However, as it was said above, InteRGIRo is being prepared to support the following structured and pointers data types:

- Array;
- Record;
- Memory position (pointers);

6.3.3 Identifiers

InteRGIRo has identifiers for: variables, digital input/output signals, labels and symbols.

There is no necessity for declaring any of these identifiers. However, there are some rules for naming each identifier, that will be explained on following subsections.

Variables (except point variables)

The InteRGIRo variable (within exception of point variables) identifiers must be written as follows:

1. The first character is the `@`;
2. following cames the character that corresponds to the data type, that may be one of the correspondent ones:
   - N - Numeric;
   - B - Boolean;
   - C - Character;
   - S - String;
   - A - Array
   - R - Record;
   - M - Memory position (pointers);
   - T - Time
3. following cames the `_v`;
4. at the end cames a number.

The difference between variables with the same data type is the last component, that is the number. For the first variable, this number is set to 0 and it is increased for each new encountered variable. This is done because:

- it is automatically generated by the *"variable" allocation optimizer*, during the translation process from Grafcet representation into InteRGIRo one. This optimizer identify the minimum number of variables that is necessary for such Grafcet program, and substitutes the Grafcet variables for the really needed variables in InteRGIRo, using this method of identifying variables;
- it facilitates the posterior translation work (final code generation), because there are some industrial robot programming languages with rigid restrictions for the variable identifiers.

The *"variable" allocation optimizer* will be presented in subsection 8.3.

Point Variable and Digital Input/Output Signals

The InteRGIRo point variables and I/O signals identifiers must be written as follows:

1. The first character is the `@`;
2. following cames the character that corresponds to the data type, that may be one of the correspondent ones:
   - P - Point (represents a position in space);
   - I - Digital input signal;
   - O - Digital output signal;
3. following cames the variable name.

The point variables and digital input/output signals identification are different from the other variables:

- the first ones refer to external elements, that may have specific names that may be common to all applications that use this elements. For example: a robot sensor can have a specific identification that should be used for all applications; an specific position can have a specific name to facilitate the program reading and writing; etc;

– the second ones refer to internal variables, that are important only for that program, and not for all other applications.

So, it is not possible to use the *"variable" allocation optimizer* for point variables and input/output digital signals, because these ones do not belong to the program scope, but to the industrial robot one.

Labels

It is necessary to use labels on an InteRGIRo program to modify the sequential control flow during execution. So, a label identifier must begin with the character `$`, followed by the label name, that can be written by any quantity of alphanumerical characters.

Symbols

A symbol is a name used on the front-end to facilitate the programming task and the source code portability, that may have an specific identification on each target language. For example, touch sensor can be named `touch` on the source program, but at a target program it should be changed for the digital input identifier (e.g. I1).

Symbols are accepted in InteRGIRo, but they are treated only by the Generic Compiler, that is the responsible one for mapping each symbol with the correspondent identifier on the target language. Symbol are identified, in the front-end and in the InteRGIRo, by the use of the character `%` in front of the symbol name.

6.3.4 Expressions

InteRGIRo expressions support only simple arithmetic and logical expressions (the ones with only one arithmetic or logic operator), as three address code expressions. This is done because some target languages, do not support complex expressions. As the basic operation of the Generic Compiler is to substitute instructions, all the expressions used on the front-end must be decomposed into three address code ones in InteRGIRo, using auxiliary variables or labels if necessary, as can be seen following:

1. for arithmetical expressions:
```
                                               @N_v2 := -@N_v1
@N_v2:=(-@N_v1+10)*20    corresponds to       @N_v2 := @N_v2+10
                                               @N_v2:= @N_v2*20
```
2. for logical expressions:
```
if (@N_v1=1 | @I_a & (@I_b | (@I_c & (@I_d | @I_e))))
    then goto $E2
```
   (a) it is firstly decomposed into:
```
if (@N_v1=1) then goto $E2
if (@Ia & @Ib) then goto $E2
if (@Ia & @Ic & @Id) then goto $E2
if (@Ia & @Ic & @Ie) then goto $E2
```

(b) that is also decomposed into:

```
ifv (@N_v1=1) goto $E2
ifs (@Ia) goto $Estado_E1_1if
goto $Estado_E1_else2
$Estado_E1_1if ifs (@Ib) goto $E2
$Estado_E1_else2 ifs (@Ia) goto $Estado_E1_2if
goto $Estado_E1_else3
$Estado_E1_2if ifs (@Ic) goto $Estado_E1_3if
goto $Estado_E1_else3
$Estado_E1_3if ifs (@Id) goto $E2
$Estado_E1_else3 ifs (@Ia) goto $Estado_E1_4if
goto $Estado_E1_else4
$Estado_E1_4if ifs (@Ic) goto $Estado_E1_5if
goto $Estado_E1_else4
$Estado_E1_5if ifs (@Ie) goto $E2
$Estado_E1_else4 ...
$E2 ...
```

6.3.5 Instructions

InteRGIRo accepts and treats only attribution, conditional and jump instructions. This is done because InteRGIRo is an intermediate representation used for generating an imperative program (the robot languages are imperatives). As all of the Von Newmann architectures works with attributions and conditional and unconditional jumps statements, all the studied imperative languages have these statements.

However, some other instructions, called actions in this work, that do not belong to InteRGIRo, can be used, but the responsability for treating them belongs to the generic translator.

The instructions and actions will be explained on following subsections.

Attribution

To associate a value to a variable, or to an output signal, it is necessary to use the attribution operation, that corresponds to the ":=" symbol, with the variable that will receive the value on the left side of the ":=" while the value, or an expression that results on a value, on its right side, as can be seen on the following example:

```
@N_v1:=0
@N_v2:=@N_v1+3
```

All variables must be initialized before using them.

Selection

To allow that the execution of some statements should depend on a specific condition, it is used an `if` statement. On InteRGIRo it works as a conditional jump, where the only instruction that must be executed, if the conditional results on a true value, is a jump to a specific label. So, the instructions that should be executed, after the evaluation of a logic condition expression, can be written in any place of the program, but the first one must have, on the beginning of such instruction, the target label used on the `if` statement. If the conditional expression results on a false value, the execution continues on the following instruction.

As it was explained before, about logical expressions on conditional statements, the `if` statements has only one condition (it is not permitted the use of the logical operators `AND` and `OR` to group conditions).

Depending on the condition analysis, the `if` statement will be represented by one of these two kinds:

– `ifv` if the condition uses variables;
– `ifs` if the condition uses external input signals.

An example of the two conditional instructions used on InteRGIRo can be seen following:

```
ifv (@N_v4=3) goto $E2
ifs (@Ia) goto $E1
```

All the known conditional statements can be simulated with a set of this simple InteRGIRo `if` instruction.

Loop

As it was explained before, how all of the Von Newmann architectures works with jump statements, all the studied imperative languages have this statement. So, the InterGIRo uses `goto` statements as unconditional jumps, with `if` statements when it is necessary to do a conditional jump.

With the labels (that are necessary for the use of `goto` statements), the `goto` and `if` statements, it is possible to simulate all kinds of well-known loop statements, like `while`, `repeat` and `for`.

Actions

As it was said before, actions do not belong to InteRGIRo. They can be used but they do not have specific representations. The responsability for writing them belongs to the user and to the generic translator. This means that, if the user only translates the source code into an InteRGIRo program, it is not possible to validate the actions, because they are simply copied into InteRGIRo. However, if the user translates the InteRGIRo program into a target code, the generic translator can analyze, treat and validate the actions. This is done because:

1. the target language may have some specific instructions for its own purposes that could be interesting to write these target language instructions directly on the front-end. So, InteRGIRo must allow their use. This option allows the user to control directly the target hardware using its specific instructions on the front-end, but it does not allow source code portability;
2. the user can create their own actions, with arguments if necessary, to facilitate their programming task. The generic compiler treats this actions, substituting them for the target instructions. Of course that the user must associate these actions to the specific target instructions. This actions can be viewed as macros for the generic compiler. This option allows the user to create their own "instructions" to be used on the front-end, and also grants portability, if the user associates these actions to the instructions of all the target languages that he uses. So, this option facilitates the programming task, and InteRGIRo allows it.

6.3.6 Grammar

The InteRGIRo grammar can be seen below:

```
instruction -> label statement
label -> "$" string
statement -> action | condition | jump | exp_arit
action -> string
condition -> if "(" exp ")" jump
if -> "ifs" | "ifv"
exp -> id | id op_rel id
id -> signal | point | id_var | number | symbol
op_rel -> "=" | "!=" | "<" | ">" | "<=" | ">="
jump -> "goto" label
exp_arit -> id_var ":=" right_side
right_side -> number | var_n op_arit var_n
op_arit -> "+" | "-" | "*" | "/" | "\" | "%"
var_n -> number | id_var
id_var -> "@" type_var "_v" cont
signal -> "@" type_sig string
point -> "@P" string
symbol -> "%" string
type_var -> "N" | "B" | "C" | "S" | "A" | "R" | "M" | "T"
type_sig -> "I" | "O"
cont -> number
```

6.3.7 InteRGIRo Program Examples

To present a practical example of InteRGIRo, a simple application responsible for moving ten times the robot between points p1 and p2 can be seen following:

```
$E1 @N_v1:=0
$Estado_E1 ifs (@Istart) goto $E2
goto $Estado_E1
```

```
$E2 move @Pp1
@N_v1:= @N_v1+1
move @Pp2
$Estado_E2 ifv (@N_v1>=10) goto $E3
ifv (@N_v1<10) goto $E2
goto $Estado_E2
$E3 end
goto $E_fim
$E_fim EndPrograma
```

Another example is a program that computes the factorial of a number. Despite factorial is not exactly an industrial robot application, an InteRGIRo representation was generated to better present InteRGIRo instructions, and to show that it is simple, based on this representation, to translate this program into another one of different imperative programming language.

```
$E1 @N_v1:=0
@N_v2:=1
clear_screen
write("Insert a positive number to see its factorial.")
read(@N_v1)
$Estado_E1 ifv (@N_v1<0) goto $E2
ifv (@N_v1>0) goto $E3
ifv (@N_v1=0) goto $E4
goto $Estado_E1
$E2 write("There is not fatorial for a negative number.")
goto $E_fim
$E3 @N_v2:= @N_v2*@N_v1
@N_v1:= @N_v1-1
$Estado_E3 ifv (@N_v1>0) goto $E3
ifv (@N_v1=0) goto $E4
goto $Estado_E3
$E4 write("The factorial is:")
write(@N_v2)
goto $E_fim
$E_fim EndPrograma
```

6.3.8 InteRGIRo X Three Address Code Representation

As it was said before, some target languages do not support complex expressions. So, it was decided to translate these expressions into a three address code like representation.

However, there are some differences between InteRGIRo and three address code representation, like:

1. InteRGIRo allows the use of several data types, each of them with their specific identifiers and operations;
2. InteRGIRo has two conditional jumps: one for the evaluation of variables (like the same statement in Three Address Code), and another for input/output signals;
3. InteRGIRo allows the use of instructions on its program that can be: target instructions, or actions that will be treated by the generic translator.

6.3.9 InteRGIRo X Gimple and Simple

Gimple [Mer03], that is based on Simple [AL07], and InteRGIRo were developed almost at the same time, with different purposes. While InteRGIRo is used on the translation between two procedural programming languages, Gimple is used to facilitate the translation process of the previous version of the GCC, where every front-end were responsible for translating directly into RTL, adding some target and language independent optimizations that were difficult to do directly on RTL.

Simple belongs to the McCAT project, that is intended for the efficient exploitation of parallelism for high-performance compilers, providing a unified approach to the development and performance analysis of compilation techniques and high-performance architectural features. The McCAT compiler can be used in as a source to source translator, as a complete code generating compiler for several architectures (DLX, Sparc at the moment), or produce LAST which is the low level representation used in McCAT which can be interpreted. However, industrial robot applications deals with sensors and with specific commands, like the moving one, that must be treated by the environment.

Also, InteRGIRo allows the use of sentences on its program that can be: target code sentences, or macros that will be treated by the generic translator. These sentences are not treated on Gimple and Simple intermediate representations.

6.3.10 Why "High-Level" Intermediate Representation?

According to Aho [ASU86], an *"intermediate representation should have two important properties: it should be easy to produce, and easy to translate into the target program"*. InteRGIRo attends both properties: based on a Grafcet specification, it is easy to produce, and as the target program is written on an imperative language, it is easy to translate it into the target program.

In a few words, the purpose of InteRGIRo is:

1. to represent the source code in a manner that facilitates the work of the back-end translator and the portability of the source code;
2. to be as close as possible to the target representation, that is another imperative language.

Comparing the purposes of InteRGIRo with the goals of the presented intermediate representations, the only difference between them is the second point enumerated above, that is the target representation. So, it can be said that InteRGIRo is a "high-level" intermediate representation, because InteRGIRo deals with imperative languages, and not with real or virtual machines.

6.4 Chapter's Considerations

In this chapter it was presented the importance of using an intermediate representation, describing, briefly, the most well known of them. Then, after reviewing the state-of-the-art in intermediate representations, it was explained why none of them fit well in GIRo environment, supporting the creation of a specific one, InteRGIRo.

It can be said that InteRGIRo is a "high-level" intermediate representation because it is used to represent a procedural program that will be translated into another program on a different procedural language, instead of representing the a program that will be translated into machine code.

InterGiro is the right choice for GIRo approach, as will be better explained in the next chapters.

Chapter 7

Automatization of the Code Generation

To achieve the goal of this thesis, that is to automatize the code generation for different industrial robots, based on the same source program, it is necessary, first, to choose an approach that enables the translation of the InteRGIRo program into code for different robots.

In this chapter, the possible approaches to do this process are presented, explaining why some of them could not be used. After, it is proposed the one that was used to implement the main component of the product of this thesis, that is the generic compiler.

7.1 Code Generation Approaches

To enable the code generation for different robots, one of the four following approaches should be used:

1. To develop, by hand, many code generators, one for each industrial robot;
2. To find a common programming language that could be used in many industrial robots, like C language for the computers domain, and create a compiler responsible for translating the InteRGIRo program directly into this common language;
3. To develop a single automatic generator of code generators, responsible for generating, automatically, the code generators for each industrial robot;
4. To develop a single retargetable compiler responsible for generating code for different industrial robots.

The first two items were quickly discarded: The first one because it does not belong to the purpose of this thesis, that is to automatize the code generation process; The other, that could seam to be a simple solution, it is not possible, because such common language does not exist on the industrial robotics domain.

The third one was, initially, the chosen approach to be studied and used in this project, but as it will be seen on the next section, it was also discarded after some research work.

However, it gave the ideas for the implementation of the fourth one, that is presented in section 3.

7.2 Automatic Generation of Code Generators

An automatic generator of code generators is a program that produces, as output, a set of routines that will be included in a new compiler, responsible for translating the intermediate representation into machine code. As input, the generator receives a formal specification of the target machine (architecture and instruction set).

The idea of developing code generator generators [Fra77, FW88] comes from the experience of building automatic generators for parsers and syntax directed translators. Although much more complex some important systems have been developed; for instance, BEG [ES89], BURG [FHP91, Pro95] and PAGODE [CCMD93]. In this field, also the New Jersey machine-code toolkit [RF95a, RF95b, RF96] should be referred as an important contribution. Other important work concerned with the retargeting of C compilers was discussed in [FH95] and [Sta00].

Some other compiler development systems, like the National Compiler Infrastructure Project (co-funded by DARPA and NSF), with its two components, named ZEPHYR [ADR98] and SUIF [ADH$^+$00], also has a generator of code generators among their components.

However, as it was presented in chapter 4, it is not possible to use this approach, because it is necessary to describe, on a representation defined by the tool, the target machine. And industrial robots have closed-architectures.

Also, one of the goals of this thesis is to create an environment to be used by robot programmers. If the robot architectures were opened, their description might not be a simple task for a common robot programmer. This hardware description task should be done by users with knowledge about compilers and processors.

So, it was decided to use industrial robots programming language descriptions, instead of machine ones, because all of the robots have a programming language. This descriptions can be obtained and represented on an easier way, because they are consulted on the programming language manuals. It can be said that the industrial robot programming languages can be seen as a window to access their hardware.

After this decision, it was created the Generic Compiler, that is a retargetable compiler responsible for translating an InteRGIRo program into any industrial robots programming language, that is presented on the next section.

With InteRGIRo, the Generic Compiler, and some easily obtained description of the target programming language, it is possible to generate code for different industrial robot programming languages, which means that it is possible to generate code to different industrial robots based on the same source code, that is the main goal of this research.

7.3 Generic Translator

To finally generates robot code, it is used the Generic Compiler, responsible for translating an InteRGIRo program into a specific robot program. To do this, it is necessary:

1. to specify a general and simple grammar, responsible for representing, in a neutral way, the grammar of the target languages. The Generic Translator translates the InteRGIRo program into a target one according to this unique grammar and the below target language specification;
2. to describe each target language, following the above grammar. The user must specify the target language according to the above grammar, to enable the final code generation.

The first item, the general grammar, was described after studying several industrial robot programming languages (more specifically: VAL [Com80], VAL II [Inc86], Melfa Basic III [Com], Rapid [Aut] and Nachi SC15F robot [Cor94] programming languages), and two other well-known procedural programming languages: Pascal and C ones. Its description can be seen on the following subsection.

The second item corresponds to the specification of the target language, that must be stored on some input files.

– the program structure file – that specifies the program structure of the target program (the program header, the variable declarations, the end of line character and the end of the program);
– the data types files – with one file for each data type used on the target language. In each file, it is described, for each data type: the operations, the syntax for declaring a variable of this data type, and the name of the data type to be used on the variable declaration;
– the target instructions file – that specifies the target language instructions;
– the user defined symbols file – that has the representation, for the target program, of each symbol used on the Grafcet program. A symbol is a name used on the front-end to facilitate the programming task and the source code portability, that may have an specific identification on each target language. For example, touch sensor can be named `touch` on the source program, but at a target program it should be changed for the digital input identifier (e.g. I1).

All of them are better described subsection 7.3.2.

With these files (named on this work as *robot language inputs*), the general grammar and the InteRGIRo program, the basic work of the generic compiler is to substitute the InteRGIRo program with the correspondent elements described at these inputs, according to the general grammar. If some sentence, or element, or anything in the InteRGIRo program does not exist in these input files, this sentence is treated as a robot language sentence, which means that it is simply copied. The process of final code generation is better described on subsection 7.3.5.

7.3.1 General Grammar

As it was said before, the goal of this project is to translate a source program into different industrial robot languages. To do this, it is also necessary to define a general grammar of the target language. This grammar is an imperative language general grammar, described after the research and experiments made on some industrial robot programming languages (more specifically: VAL [Com80], VAL II [Inc86], Melfa Basic III [Com], Rapid [Aut] and Nachi SC15F robot [Cor94] programming languages), and two other well-known procedural programming languages: Pascal and C ones.

It is divided into two components: the one that corresponds to the statical part of the program, like headers, variables declarations, and others, that will be seen on subsection 7.3.2, at the *Program Structure* part; and the one that corresponds to the instructions that will be executed by the robot, that can be seen following, in this subsection.

The general grammar, without the statical part, can be viewed above (the `user_defined` and the `target_spec` elements are describe following the grammar):

```
instruction -> label0 sentence
label0 -> label | &
label -> string | number
sentence -> action | condition | jump
action -> target_instruction | GIRo_instruction | expression
target_instruction -> user_defined_1
GIRo_instruction -> user_defined_2
expression -> operand attribution right_side
operand -> string
attribution -> attrib_operator | attrib_instruction
attrib_operator -> target_spec_1
attrib_instruction -> target_spec_2
right_side -> number | var_i operator var_i | op_instruction
var_i -> number | operand
operator -> target_spec_3
op_instruction -> target_spec_4
condition -> left cond_exp right jump | if_funcs
left -> target_spec_5
right -> target_spec_6
if_funcs -> target_spec_7
cond_exp -> signal | operand op_rel operand
signal -> string
op_rel -> target_spec_8
jump -> goto_sintax label
goto_sintax -> target_spec_9
```

The `user_defined` elements are created, if necessary, by the user, if he wants to facilitate its programming task, or if he wants to increase portability for its source code The `target_spec` elements are not specified in this grammar because they depend on the target language. In a few words, each of this elements means that:

- user_defined_1 - the target instruction is already written on the source program, and must be copied to the target program;
- user_defined_2 - the GIRo instruction must be substituted by a target(s) one(s), that must be described on the target instructions file. It was created only to grant portability for the source code and to facilitate de programming task;

- target_spec_1 - the attribution operator must be specified for each data type of the target language, like Pascal "`:=`" and C "`=`";
- target_spec_2 - the attribution instruction with its operands must be specified for each data type of the target language, like C "`strcpy`";
- target_spec_3 - any operator must be specified for each data type;
- target_spec_4 - the operation instruction with its operands must be specified for each data type of the target language, like C "`strcat`";
- target_spec_5 - the text that preceeds the condition must be described on the target instructions file, like C "`if (`";
- target_spec_6 - the text that succeeds the condition must be described on the target instructions file, like C "`)`";
- target_spec_7 - in case of the `if` sentence does not exist on the target language, the conditional instructions must be described, with its two operands, for each data type of the target language. Like Melfa Basic III, that uses specific relational instructions in place of the conditional sentences.
- target_spec_8 - any relational operator must be specified for each data type. In C language, they are "=", "!=", "<", ">", "<=" and ">=", while in Pascal they are ":=", "<>", "<", ">", "<=" and ">=":
- target_spec_9 - the unconditional jump instruction of the target language must be described on the target instructions file;

These inputs will be better explained on subsection 7.3.2.

As it could be seen on the above general grammar description, to generate final code, some informations about the target language must be given by the user:

- The target language uses labels or line numbering?
- There are if-like sentences? If not, how the conditional operators are expressed in the target language?
- The arithmetic, relational and attribution operators are the ones presented above? If not, they can be redefined.
- Which is the syntax of the goto instruction?

The figure 7.1 presents a translation schema between InteRGIRo and the target language, showing the needed informations that must be given by the user, and the defined elements that he can create.

These informations about the target language must be found on the files that store the robot language inputs. There are also some other informations that are needed. All of them will be seen on the following section.

7.3.2 Robot Language Inputs

Using GIRo, the first thing to do is to draw a Grafcet schema that solves the user problem. After this, the user generates an InteRGIRo representation for its Grafcet schema. To generate robot code, the user must, at this point, call the Generic Compiler. But to translate this InteRGIRo representation into robot code, the Generic Compiler needs some information about the robot language. These informations are:

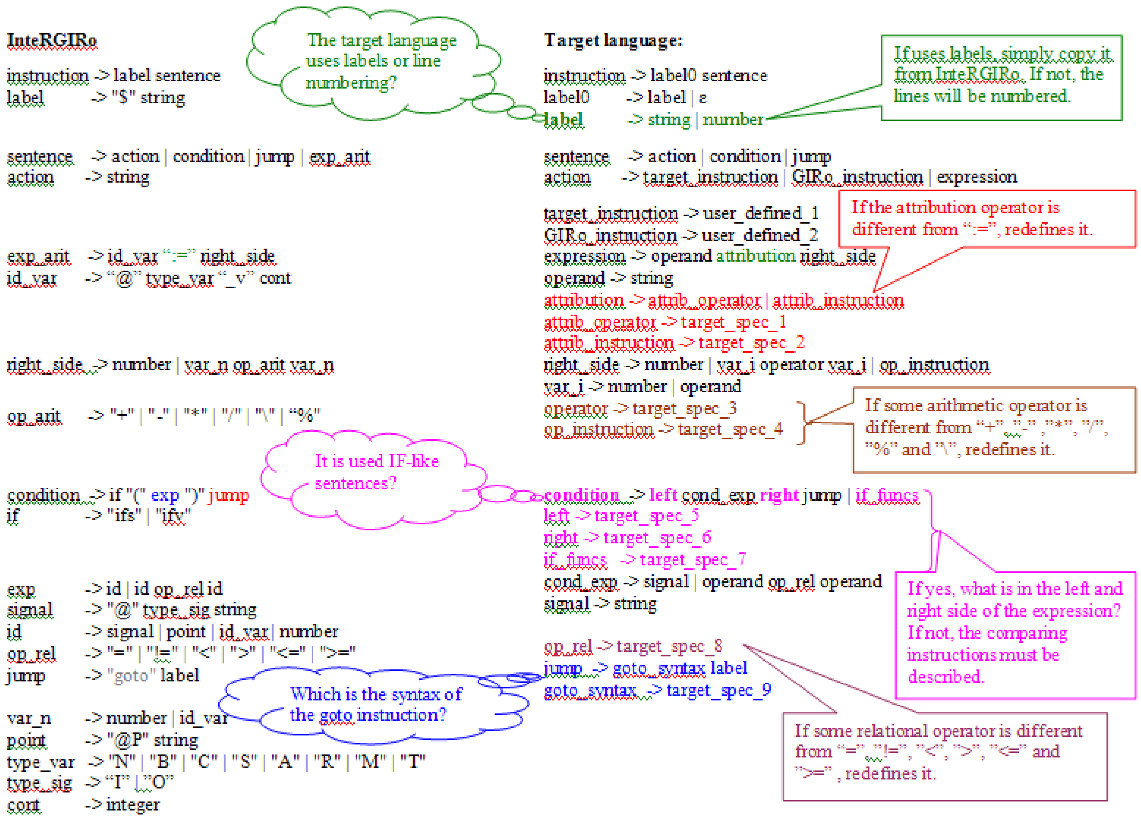

InteRGIRo
instruction -> label sentence
label -> "$" string
sentence -> action | condition | jump | exp_arit
action -> string
exp_arit -> id_var ":=" right_side
id_var -> "@" type_var "_v" cont
right_side -> number | var_n op_arit var_n
op_arit -> "+" | "-" | "*" | "/" | "\" | "%"
condition -> if "(" exp ")" jump
if -> "ifs" | "ifv"
exp -> id | id op_rel id
signal -> "@" type_sig string
id -> signal | point | id_var | number
op_rel -> "=" | "!=" | "<" | ">" | "<=" | ">="
jump -> "goto" label
var_n -> number | id_var
point -> "@P" string
type_var -> "N" | "B" | "C" | "S" | "A" | "R" | "M" | "T"
type_sig -> "I" | "O"
cont -> integer
The target language uses labels or line numbering?
It is used IF-like sentences?
Which is the syntax of the goto instruction?
Target language:
instruction -> label0 sentence
label0 -> label | ε
label -> string | number
If uses labels, simply copy it from InteRGIRo. If not, the lines will be numbered.
sentence -> action | condition | jump
action -> target_instruction | GIRo_instruction | expression
target_instruction -> user_defined_1
GIRo_instruction -> user_defined_2
expression -> operand attribution right_side
operand -> string
attribution -> attrib_operator | attrib_instruction
attrib_operator -> target_spec_1
attrib_instruction -> target_spec_2
If the attribution operator is different from ":=", redefines it.
right_side -> number | var_i operator var_i | op_instruction
var_i -> number | operand
operator -> target_spec_3
op_instruction -> target_spec_4
If some arithmetic operator is different from "+", "-", "*", "/", "%" and "\", redefines it.
condition -> left cond_exp right jump | if_fincs
left -> target_spec_5
right -> target_spec_6
if_fincs -> target_spec_7
cond_exp -> signal | operand op_rel operand
signal -> string
If yes, what is in the left and right side of the expression? If not, the comparing instructions must be described.
op_rel -> target_spec_8
jump -> goto_syntax label
goto_syntax -> target_spec_9
If some relational operator is different from "=", "!=", "<", ">", "<=" and ">=", redefines it.

Figure 7.1: Schema of the translation between InteRGIRo and the target language.

– Program structure, that represents some statical sentences that can be necessary for a program, like header, end of line character, and end of program sentences;
– Robot Language Instructions, that represents the subset of the robot language instructions that are necessary to translate completely the InteRGIRo representation into the target language;
– Data types, that defines the necessary data types, with the correspondent name on the target language, their operators, and some other things that can be specific for each data type for each target language;
– Symbols, that can represent memory positions, variable identifiers, digital input/output signals, position coordinates, or anything else that can be directly one-to-one mapped from InteRGIRo representation into target code. It works like a symbols table.

Each line in these input files maps its Giro sentence (left side) into a target sentence(s) (right side). The `@` separates both sides.

Always that the left side statement does not have a correspondent expression / operation / description on the target language, the characters `{:-(` must be written on its right side, indicating the inexistence of that left side statement on the target language. So, such sentence will be discarded. If the statement *{:-(*, or the complete expression / operation / description, is omitted (no data is presented) on the right side, it indicates that this statement is unknown for the translation process, and it is completely copied directly to the final code, because it is assumed that this is a final code instruction. This means that, using the statement *{:-(* for a given statement, always that it is used on the source program, **it will not be translated or copied** into the final code, and a error / warning message will be given to the user. Without this indication, it will be assumed that such instruction corresponds to a final one, and **it will be copied** to the final code, indicating the user with a different error / warning message.

The `%` character normally indicates the presence of a symbol or an instruction argument, but when it is necessary to represent exactly the character `%`, it is necessary to write two characters, like `%%`. This character has some other special uses, that will be presented when it is necessary.

At this moment, this informations are entered manually by the user and stored into specific files. Their file extensions are user defined. Normally it uses a part of the target language name, to facilitate its identification. Sooner, it will be created a front-end responsible for facilitating this work, that will be user-friendly. This front-end will ask some questions to the user, like the ones that was saw on previous section. All that the user must do is to take a look at the language manual and answer some questions.

Program Structure

Some programming languages needs some statically sentences in each program. These sentences must be always presented in all programs of such language. They are:

– the header, including the variable declaration;
– the end of line character;
– the end of program sentences.

To support the target languages, it is necessary to enter those informations in a file called `prog_struc.xxx`.

The first sentence represents the header of the program, that must be typed at the right side of `header @`. Some examples of headers representations are shown following (the first for a Rapid robot language, and the second and the third for computers Pascal and C languages):

```
header @ MODULE teste# %?VI%t %i;# PROC main()

header @ program teste;# uses crt;# label #    %?L%s, %f;# var # %?V%i: %t;# begin

header @ #include "stdio.h"# #include "conio.h"# void main()# {# %?V%t %i;
```

All the text included in the right side of the header are copied to the target program, excepting the following elements that can appear in any order:

– the character `#`, that indicates the carriage return, which means that a new line must be created to separate the elements of this header. The only exception occurs when the `#` element is the first character of a line. In this case, the `#` is copied to the target program;

- the `%?V`, indicating that the declaration of program variables must be done in this place. After this indication, some other elements can appear without any order, to describe how the variable declaration must be done:
  - the `I` character, after the `%?V`, indicates that the variables must be initialized. If the `I` does not appear after the `%?V`, it is not necessary to initialize the variables;
  - the `%t` represents the data type of a variable must be written;
  - the `%i` represents the variable identifier must be written;
  - all the text that appear between the `%?V` or `%?VI` and the carriage return `#`, before, after and between the `%i` and `%t` sentences, are copied on the same position for each variable declaration.
- the `%?L`, indicating that the used labels must be declared. After this indication, other two elements `%s` and `%f` are used to delimitate the element that must be the separator between the labels that will be declared. Between the `%f` and the `#`, comes the text that must be included at the end of labels declaration.

The next sentence describes the characters that must exist at the end of each instruction (for example, each Pascal or C instruction ends with the character "`;`"). These end of line characters must be described at the right side of `end_line @`.

The last sentence represents the characters that must exist at the end of the program (for example, each Pascal program ends with the characters "`end.`"). These end of program characters must be specificated at the right side of `end_prog @`.

Always that, in the program structure file, it is necessary to create a new line, the character `#` must be used. It indicates the carriage return.

Some examples of program structure files can be seen above:

- Program structure file for the Rapid target language:
```
header @ MODULE teste#%?VI%t %i;#PROC main()
end_line @ ;
end_prog @ ENDPROC#ENDMODULE
```
- Program structure file for the Pascal target language:
```
header @ program teste;# uses crt;# label #    %?L%s, %f;# var # %?V%i: %t;# begin
end_line @ ;
end_prog @ repeat until keypressed; end.
```
- Program structure file for the C target language:
```
header @ #include "stdio.h"# #include "conio.h"# void main()# {# %?V%t %i;
end_line @ ;
end_prog @ }
```

Robot Language Instructions

The robot language instructions can be seen in two different ways, that are stored into two different files:

- the elements that are responsible for the program control flow. In case of InteR-GIRo, the representation of labels, gotos, and ifs. They are stored in a file called `standard.xxx`.

– the elements that represents the actions, created by the user on the source program to facilitate its programming task and to grant portability, that must be translated into target code. They are stored in a file called `user_defined.xxx`.

Basically, this two files could be stored in only one. The only difference is the user protection rules that could be applied to these files. In the first one, if the instructions are well defined, the user should not change them, because its content has only the sentences for the program control flow: the steps and transitions. This sentences, for one kind of robot language, should be entered only once. At the second file, the user can add new robot instructions, delete and modify, any time, if he thinks that it could facilitate its programming task.

Each line in these files maps an InteRGIRo sentence (left side) into a robot code sentence (right side). The `@@` separates both sides.

1. Standard Instructions File
   The first InteRGIRo sentence to be mapped is responsible for defining the label for the jumps. It can be a line number ("n"), or a label identifier ("l"). The user should only indicates which of them is used, typing it at the right side of the sentence "$l @@", as can be seen below:
   ```
   $l @@ n
   ```
   If it is used a label identifier, the characters that can come after the ("l"), are used to specify that, always that a label is defined, at the beginning of a instruction, these characters will separate the label from the instruction, as can be seen following, with the "`:`" to be the separator:
   ```
   $l @@ l:
   ```
   Also, if it is used line numbers, the number that can come after the ("n"), is used as the first line number, and as an increment for determining each line number, as can be seen following, with the number "`5`":
   ```
   $l @@ n5
   ```
   Next, comes the `ifs` and `ifv` sentences, that represents conditional sentences, where the first one uses signals, and the second one uses variables. These two `if` sentences were created because some robot languages do this distinction, however, both are treated at the same way by the generic translator.
   The user only defines what should be at the left side of `sigs` and what should be at its right side, as can be seen following:
   ```
   ifs (sigs) @@ if sig(%sigs)
   ```
   In this example, was defined that "`if sig(`" must be in the left side of the condition, while the "`)`" must be in its right side.
   The `sigs` represents the condition that is written in each conditional sentence, and it is treated using the relational operators that are represented on the specific data type, that will be presented on next subsection.
   At the end of the standard file, there is the mapping of `goto` sentences, where the user must only indicates how this jump is described in the target language, as can be seen below:
   ```
   goto @@ GT
   ```

An example of a VAL standard instructions file can be seen following.

```
$l @@ l
ifs (sigs) @@ if sig(%sigs)
ifv (sigs) @@ if %sigs
goto @@ goto
```

2. User Defined Instructions File
   The user_defined file contains the created GIRo instructions, and the correspondent ones that may substitute them on the target language. These elements represents the GIRo actions, that are used to facilitate the programming task and to grant portability to the source code. The user can create as many actions as he wants.
   Instead of using functions on GIRo environment, it is used actions, that work as macros, substituting the actions with the correspondent program code that executes them. It is not used functions to facilitate the portability of the source code and the final code generation. Also, the program execution becomes faster, but the program size can be bigger than a correspondent one that has functions.
   Depending on the necessity of having arguments, that must be treated as parameters, these user defined instructions can be divided in two kinds of actions:
   (a) Simple actions, that are mapped directly to the corresponding robot sentence, as can be seen below, which maps the InteRGIRo `open` and `close` instructions, that intends to open and close the robot gear, into the robot instructions `openi` and `closei`.

   ```
   open @@ openi
   close @@ closei
   ```

   (b) Actions with arguments, that are mapped to the corresponding robot sentence, substituting some elements with the respective parameters. These parameters correspond to instructions arguments, that were written, between parenthesis and separated by commas, by the user on the front-end.
   The characters `%\`, located at the right side sentence, indicate the existence of arguments. The existence of only `%` characters means that such target instruction has symbols, that will be explained on User-Defined Symbols subsection.
   After the characters `%\`, may appear some elements that indicates how the arguments are passed from the InteRGIRo sentence into the target one:
      i. If comes the word `all`, it means that all the parameters from the InteRGIRo sentence are inserted into the target sentence, on the same order, separated by the characters that comes after the word `all` and before the final `%` character;
      ii. If comes a `number`, it means that the parameter located at the `number` position on the InteRGIRo sentence is inserted into the target sentence. After the number, comes another `%` character to indicate the end of this argument. When using `numbers`, it is possible to pass some or all arguments, on an order that is specified by the target language instruction. All other elements of the right side sentence that are not a `number` or `%` are copied into the target sentence, including the separators, that must be specified on it according to the target language.

   An example of these actions with arguments can be seen following:

   ```
   move @@ MV %\all,%
   shift @@ SH %\2%, %\4% - %\1%
   ```

An example of a VAL user defined instructions file can be seen following (the VAL instructions `openi` and `closei` respectively open and close the gear, while the `move` instruction is responsible for moving the robot to the specified location. In these cases, the locations %TL and %TR, that correspond to symbols, and will be treated later).

```
open @@ openi
close @@ closei
TL @@ move %TL
TR @@ move %TR
```

In all of the above user defined instructions, each left side is mapped with a target instruction, with or without arguments. However, it is also possible to map the InteRGIRo instruction with another InteRGIRo instruction, as can be seen on the following Pascal example (Pascal does not have `Open` and `Close` instructions, so they are mapped with write instructions, to simulate them, showing the user that such operation was done):

```
Close@@write("close gear")
Open@@write("open gear")
write@writeln(%\all,%)
```

There is no recursively mapping. Always that an instruction is evaluated, it is inserted into an instructions list. The instructions are treated only if they are not in this instruction list. In this example, `Close` is inserted and mapped into a `write` instruction. After, `write` is inserted and mapped into `writeln` instruction, that is a target instruction. If the write instruction was mapped with a `Close`, or with another `write` one, the evaluation would stop, because any of them are in the instructions list. The user is informed about this error, and the last mapped instruction (`Close` or `write`) will be inserted on the target code, assuming that this is the target instruction.

In all of the above user defined instructions, each left side is mapped into a single target / InteRGIRo instruction, with or without arguments. However, it is possible to map the InteRGIRo instruction into a sequence of instructions, separated by the `;`, as can be seen on the following user defined instructions for the Rapid programming language:

```
Close @@ Reset Gripper; WaitTime 0.2
Open @@ Set Gripper; WaitTime 0.2
```

As an InteRGIRo instruction can be mapped into a sequence of instructions, with arguments, it can be viewed as a macro definition, that can contain a part of a program, or even, a subprogram. To allow this, it is possible to define variables and labels in this right side sequence of instructions.

- The labels must start with the `$` character, followed by the `L` character (to be different from the ones that are in the InteRGIRo program, that start with the `$E` characters), and by a number, that differentiate the labels that are used in this sequence of instructions;
- The same occurs with the variables: first come the `@` character, followed by the correspondent type character with the `_` one. After, comes the `a` character (to be different from the ones that are in the InteRGIRo program, that uses the `v` character), and by a number, that differentiate the variables that are used in this sequence of instructions.

During these translation process, the number that is used in these defined labels is substituted by an unique one, in case that such InteRGIRo instruction is used more than one time in the InteRGIRo program. So, it is granted that each label is unique. It is not necessary to do the same with the defined variables, because they are valid and used only in this generated sequence, and can be reused.
It is also possible to add rules that test if a data is a variable or a constant. This is necessary because there are some target languages, like Melfa Basic III, that has different instructions according to the kind of data: a constant or a variable.
A rule is located between the `??` characters, and evaluate if the specified parameter satisfies the rule. If the rule is valid for such situation, the following code is translated into final code. It is only possible to assign one target instruction for each rule.
An example of an operation data type (that will be presented later but it is treated in the same way that was describe here) that has defined variables, labels and rules, that maps an InteRGIRo + operation with the correspondent sequence of instructions in Melfa Basic III, can be seen following (Melfa Basic III does not have arithmetic operators, so each of them has to be described as a subprogram).

```
+ @ ?1 is_var?   cp %1;
    ?1 is_var?   cl @N_a1;
    ?1 is_constant?   sc @N_a1 %1;
    ?2 is_var?   cp %2;
    ?2 is_var?   cl @N_a2;
    ?2 is_constant?   sc @N_a2 %2;
    ?2 is_constant?   cp @N_a2;
        sm 0 $L2;
    $L1 eq 0 $L3;
        ic @N_a1;
        dc @N_a2;
        cp @N_a2;
        gt $L1;
    $L2 eq 0 $L3;
        dc @N_a1;
        ic @N_a2;
        cp @N_a2;   gt $L2;
    $L3;
    @N_a1
```

Data Types

As it was presented before, at this moment, Giro accepts 11 data types: number; boolean; char; string; array; record; pointer (memory address); point (robot position); time; input (digital input signal); and output (digital output signal). There must be a file for each necessary data type.

How all the robot languages uses numeric and point variables and input and output signals; and how some of them uses only numeric and point variables and digital input-output signals, at this moment Giro works only with atomic data types. The decision of accepting structured data types is because, in future, it is expected that Giro will supports them. However, for this current work, it was validated only the use of numeric, point, input and output data types, which means that, despite the existence of 11 different data types, only these ones are really used today.

For each data type file, there are three obligatories inputs:

– First comes the data type identification, that maps the name of the data type used on Giro with the correspondent one in the target file. A `%` symbol should appear between two sentences of the data type identification always that the variable identifier comes between these two sentences, like the string definition of C language (`char id[]`, where `id` is a vector composed by char elements). This input is necessary to indicate that such data type is used by the target language, and for declaring variables, described at the header component of the program structure input file. If this data type does not exist on the target language, its value must be `{:-(`. In some situations, the data type exists, its operations exist, but there is no declaration for the variables of this data type. This situation normally occurs with the input, output and point data types. Always that this situation occurs, the name of the data type, on the target language, must be empty, which means that, the data type exist, but it is not defined;
– Second comes the variable identification, in case that the target language specifies how the variables identifiers must be written. On the left side comes the name of this sentence (`ID`) and on the right side comes the character(s) used on this identifier, with a `%` symbol that corresponds to where the number, generated for this identifier at the translation process into InteRGIRo, will be included.
  The @ character, used to identify variables in an InteRGIRo program, are always retired during the translation process into the target program.
  As data type identification, always that the variable identifier is composed by two sentences, one at the beginning and another at the end of the number generated for this identifier, a `%` symbol must be used to indicate where the number must appear (like the real variables used at Nachi SC15F robot programming language, identified as `Vn!` where `n` must be a number).
  In case of point data type and of input and output digital signals, this variable identification does not exist, because, as it as presented on previous chapter, these elements do not belong to the program scope, but to the physical robot. They are translated directly to the target program, or they can be substituted by another identifier defined on the User Defined Symbols File, that will be seen on this chapter;
– The third item is the initialization value for a variable of this data type, that must be used always that the target language specifies that the variables of such data type must be initialized. If in the right side of `initialization` comes a value, all variables of this data type will be initialized with this value in the beginning of the target program;

An example of these first inputs can be viewed below, where the numeric data type is defined as `int` on the target language, the variables are identified by the character `I` plus a number, and each variable of this type must be initialized with the value `0`.

```
NUMBER @ int
ID @ I
init @ 0
```

After these 3 inputs, others can be inserted. These ones are not limitated in number, and they correspond to:

1. the data type operations, mapping the InteRGIRo operators on the left side with the

target ones on the right side. There are, basically, three kinds of operators and, also, not limitated in number:

– the data type basic ones, used for the basic operations between two operands from the same data type, like the InteRGIRo arithmetic operators `+, -, *, /` for numeric data types;
– the attribution one, used to assign the value that is on the right side of the `:=` InteRGIRo operator to the operand on its left side, where the both sides have the same data type;
– the relational ones, used for comparing two operands from the same data type, like the InteRGIRo relational operators `=, !=, <, >, <=, >=` for numeric data types.

All the data type operators of the target language, must be described on the right side of the `@`, even when they have a similar syntax, changing only the operator representation, or when their descriptions correspond to the same ones used by InteRGIRo, as can be seen on the following example:

```
+ @ +
:= @ =
= @ ==
!= @ !=
```

However, when the syntax of the data type operator is different from the one at the target language, it must be described in details on the right side. How all the InteRGIRo operators are between two operands, it is used two special symbols, %
1% and %
2%, that correspond exactly to the left (first) and right (second) operands (as it happens with the robot instructions). With this two symbols, the user can describe that data type operator, using them to represent the left and right operands, as can be seen on the following C example for string data type operations. If it is necessary to use more than one sentence to represent such operation, the `;` is used to separate one sentence from another.

```
+ @ concat(%\1%, %\2%)
:= @ strcpy(%\1%, %\2%)
= @ strcmp(%\1%, %\2%)==0
!= @ strcmp(%\1%, %\2%)!=0
```

Analyzing different robot languages, it was identified a special situation: a specific robot programming language, in this case the Melfa Basic III, that do not have `if` as conditional sentence. So, in such case, to map the `ifs` and `ifv` InteRGIRo conditional sentences into Melfa Basic III, it is necessary to indicate that the robot language do not have `ifs` and `ifv` sentences, adding an `{:-(` on the right side of each `if` sentence on the standard robot input file. After this, all of the relational operators in each data type file must be redefined, because a conditional sentence depends on the result of a relational operation between two operands, the `%`
`1%` and `%`
`2%` ones, presented above. If it is necessary to get only the target label used on an InteRGIRo instruction, and insert it on the target sentence, it is used the `#` to indicate the location where such target label must be inserted on the target instruction. For example, the relational operator `=` can correspond to two sentences: one for copying the value into a register; and the other that performs the equal operation. The

<= operator has one more sentence, that corresponds to another comparison, the < operator. So, two comparisons must be done, and both of them with the same goal if one of them is true. The other elements are simply copied (like IN, EQ and SM), as can be seen above:

```
?= @ IN %\1%;EQ %\2%#
?<= @ IN %\1%;SM %\2%#;EQ %\2%#
```

2. the instructions that are lexically or syntactically dependent on the data types, like input and output of user data on C language, for example. These instructions can be described in two manners that was also presented: as the simple actions; and as the the actions with arguments. Both of them were presented previously on the subsection User Defined Instructions File. An example can be seen following (remember that the %%) means that the % character must be written):

```
write @ printf("%%i",%\1%)
read @ scanf("%%i",&%\1%)
```

3. the specific elements that are used on some data types, like the ", used to represent a string constant. In this case, it is only necessary to indicate, on the right side of the @, the representation of that element on the target language, and all of these elements will be substituted by the target element, as can be viewed on the Pascal example below.

```
" @ '
```

User-Defined Symbols

Like a symbols table, this file is responsible for mapping the InteRGIRo symbols into robot code symbols, like memory position, position coordinates, variable identifiers, digital input/output signals, or simply a higher level GIRo symbol into a low level robot code symbol. As it was presented before, a symbol can be identified, on an InteRGIRo program, with the % character at the beginning of the symbol. The point variables and the digital input/output signals can be also treated as symbols.

Each line in this file also uses the @@ to separates a GIRo symbol (left side) into a robot code symbol (right side).

Some examples can be seen below.

```
P1 @@ 142,0,-84
rest_position @@ 0, 0, 0
sensor1 @@ I1
var1 @@ I2
D1 @@ 1001
D2 @@ 1002
D3 @@ 1003
TL @@ left_point
TR @@ right_point
```

It is important to say that the symbols are the last elements that are treated by the generic translator for each GIRo instruction. This means that they can be used not only on a GIRo program, but also in all used input files: Program Structure file, Standard instructions file, User Defined instructions file, and the Data Types definitions file. The following example shows such situation on the user-defined file, where the GIRo instruction `rest_in_peace` corresponds to a move command to the initial position of the robot, represented by the symbol `rest_position`. This `rest_position` symbol must be defined at the user-defined symbols file.

```
rest_in_peace @@ move %rest_position
```

User Defined File X Data Types File: Where to Insert an Instruction

As it can be saw, an instruction can be inserted in both files. For example, the write instruction was inserted on the User_Defined file for Pascal language, while it was inserted on the Data Types file for C language. How to decide where to put the description of a specific instruction?

The answer is, it depends on the used arguments and how these instructions are used on the target language. If the instruction on the target language is written always in the same way for all used data types (without analyzing the arguments), like the `write` or `writeln` instructions on Pascal language, it can be described only once, at the User_Defined file. If it is written on different manners (without analyzing the arguments), like the `printf` in C language (one element of the `printf` instruction corresponds to how the data must be written, %d, %s, etc), or if such instruction is applicable only for some data types, and not for all, it must be described for each valid data type.

And what about an instruction be inserted in both files? Which of them will be used on the final code?

The Generic Compiler analyzes first the target language instructions, and if the specific instruction is not found, it will analyze the data types files. So, in such case, the chosen one is the instruction that is described at the target language instructions file.

7.3.3 Some Functionalities

Some useful functionalities of this environment can be viewed below:

- The possibility of simulating the robot programs on personal computers environment. For example, if a personal computer programming language, like C or Pascal, do not has a moving instruction, it can be simulated adding, on the user defined instructions file for that target language, some writing sentences that show the user the movements that must be done, which means that, it is possible to simulate the behavior of a robot program on other programming languages.
- The possibility for creating new instructions using other already existing ones. It facilitates the programming task, because, when creating the robot inputs file, it is necessary for the user to take a look, or study, a different language. If this user can create new instructions using others that were already created, he will create instructions using the ones that he already knows, instead of having to create a new instruction based on a target language that he is not familiarized to use. Of course that the user will have to create some basic operations and instructions first,

using directly the target language, but after a few new instructions, others can be specified using the "old" ones. For example, it is presented, below, the InteRGIRo `write` instruction, with the correspondent target one, in this case, the C `printf` instruction. After, the `move` command is presented with its correspondent target one, that is, in this case, an already defined InteRGIRo `write` instruction.

```
write@@printf("%%i",%\1%)
...
move@@write("move to %p %\all,%")
```

– The user can create macros. GIRo do not deal with functions, but achieves the same goals using macros. When creating instructions, it is possible to group as many instructions as he think that is necessary for creating another one. It is also possible, as it was presented before, to use parameters, to send some arguments into the macro body instructions. For example, on the InteRGIRo program, the unary input data type operation ~ corresponds to a test to identify if a sensor is inactive. The result must be true if it is inactive, or false if it is active. If some language do not have a simple operation that perform this task, but allows this test with the use two or more instructions together, a macro, or "function" can be created as can be seen on the example below, where the parameter %
1% will be substituted by the InteRGIRo sensor name:

```
~@write("Reading the sensor ");write("%\1%");read(%\1%); %\1%==0
```

7.3.4 Basic Algorithm for Treating Each Instruction

This subsection presents the basic algorithm used to translate each InteRGIRo instruction into the target one. This algorithm uses `str` as a temporary string variable. We think that the algorithm is self-explanatory and do not deserve more comments.

```
If the InteRGIRo sentence is defined as a Robot Input Instruction
   then copy the target instruction from the Robot Input Instructions File into str
        If str contains arguments
           then treat and substitute them on str
   else
      if the InteRGIRo sentence corresponds to a Data Type operation
         then copy the target instruction from the specific Data Type file into str
              If str contains arguments
                 then treat and substitute them on str
         else copy the InteRGIRo sentence into str (it is a final instruction)

if str contains symbols
   then substitute them on str with the final symbols

return str
```

7.3.5 Generic Compiler: InteRGIRo -> any Industrial Robot Language

To finally generates robot code, it is used the Generic Compiler, responsible for translating an InteRGIRo program into an specific robot program. To do this, the Generic Compiler needs some input files. The first one has the InteRGIRo program. The others must have the information about the robot language, that was explained on previous sections: the program structure file, the data types file, the standard file, the user_defined file, and the symbols file.

With these files, the basic work of the generic compiler is to substitute the InteRGIRo program with the elements that are in the robot language files. If some sentence, or element, or anything in the InteRGIRo program does not exist in the robot language files, this sentence is treated as a robot language sentence, which means that it is simply copied.

If the robot language needs line numbering, a line counter variable is used, that has the line number of the line that is being generated. A table that maps each label with its corresponding line number is also used.

To illustrate the idea, it will be presented two generated C programs from the InteRGIRo representations presented in section 6.3.8 (a simple program responsible for moving ten times the robot between points p1 and p2, and the other that calculates the factorial of a given number). As it was presented on this chapter, to generate final code, it is necessary some information about the target language, in this case, the C language. How the case studies are presented in chapter 9 in more details, at this moment it will be presented only the necessary input files that allows the final code generation for these two programs, that are:

– Program structure file:
  ```
  header@#include "stdio.h"# #include "conio.h"# void main()# {# %?V%t %i;
  end_line@;
  end_prog@}
  ```
– Standard instructions file:
  ```
  $l@@l:
  ifs@@if (%sigs)
  ifv@@if (%sigs)
  goto@@goto
  ```
– User defined instructions file
  ```
  clear_screen@@clrscr
  move@@write("move to %p %\all,%")
  end@@getchar(); getchar()
  ```
  How C language does not have moving instructions, the InteRGIRo move sentence will be translated into a writing instruction, to allow the user to see, on screen, the movements that are done. It is a very important functionality of GIRo: with the generation of robot programs into C language it is possible to simulate the behavior of the robot program on a personal computer, where the moving instructions are substituted by the writing instructions, allowing to see to where the robot must move.
  Also, this is a simple program that do not need symbols. But, only to show that it works well, a symbol `%p` was inserted on the writing instruction of the move sentence, that will be substituted by the world `point`, described on the symbols user defined file.
– Number data type file
  ```
  NUMBER@int
  ID@
  init@
  +@+
  -@-
  *@*
  /@/
  \@\
  ```

```
%@%
:=@=
=@==
!=@!=
<@<
>@>
<=@<=
>=@>=
write@@printf("%%i",%\1%)
read@@scanf("%%i",&%\1%)
```

- Input data type file

```
INPUT@int
ID@
init@
~@write("Insert a value for the sensor ");write("%\1%");read(%\1%); %\1%==0
_@write("Insert a value for the sensor ");write("%\1%");read(%\1%); %\1%!=0
:=@=
=@==
!=@!=
<@<
>@>
<=@<=
>=@>=
write@printf("%%i\n",%\1%)
read@scanf("%%i",&%\1%)
```

  Despite C language does not have digital input/output signals data types, it was created the input data type to allow the code generation from the InteRGIRo program example into C. How digital input signals always have a value, the operations responsible for analyzing a sensor value where substituted by reading commands, so, the user can enter any value from the keyboard to analyze the behavior of the robot program. It is useful to allow the test of a robot program on a personal computer. The behavior of the program can be simulated, getting the sensor values from the keyboard instead of a real sensor.

- User defined symbols file

```
p@@point
Pp2@@ponto2
```

  There are only two symbols, one that will substitute a symbol `p` from an user defined instruction, and another `Pp2` that will substitute the InteRGIRo point variable `@Pp2` with the physical point name `ponto2`.

Using these above files, the InteRGIRo program responsible for moving ten times the robot between points `p1` and `p2` was translated into the following C program:

```
#include "stdio.h"
#include "conio.h"
void main()
{
     int N_v1;
     int Ion;
E1: N_v1=0;
Estado_E1: printf("%s\n","Insert a value for the sensor ");
printf("%s\n","Ion");
scanf("%i",&Ion);
if ( Ion!=0) goto E2;
goto Estado_E1;
E2: printf("%s\n","move to point Pp1");
N_v1=N_v1+1;
printf("%s\n","move to point ponto2");
Estado_E2: if (N_v1>=10) goto E3;
if (N_v1<10) goto E2;
goto Estado_E2;
E3: getchar();
 getchar();
goto E_fim;
E_fim: ;
}
```

Also, using the same above files, the InteRGIRo program that calculates the factorial for a given number was translated into the following C program:

```
#include "stdio.h"
#include "conio.h"
void main()
{
     int N_v1;
     int N_v2;
E1: N_v1=0;
N_v2=1;
clrscr;
printf("Insert a positive number to see its factorial.");
scanf("%i",&N_v1);
Estado_E1: if (N_v1<0) goto E2;
if (N_v1>0) goto E3;
if (N_v1==0) goto E4;
goto Estado_E1;
E2: printf("There is not fatorial for a negative number.");
goto E_fim;
E3: N_v2=N_v2*N_v1;
N_v1=N_v1-1;
Estado_E3: if (N_v1>0) goto E3;
if (N_v1==0) goto E4;
goto Estado_E3;
E4: printf("The factorial is:");
printf("%i",N_v2);
goto E_fim;
E_fim: ;
}
```

Basic Algorithm

This section presents the basic algorithm of the translator described and some figures (7.2, 7.3, 7.4, 7.5 and 7.6) that illustrates each stage of this algorithm. We think that the algorithm and figures are self-explanatory and do not deserve more comments.

```
1st stage (figure 1.3):
load the content of standard and user_defined files into mem_inst
load the content of all data types files into mem_data_types
load the content of symbols file into mem_symb
load header from estr_prog file into mem_header
load end line element from prog_struc file into mem_end_line
load end program elements from prog_struc file into mem_end_prog

2nd stage (figure 1.4):
load InterGIRo program into mem_prog, loading variables into mem_vars, with their
respective data types from mem_data_types

3rd stage (figure 1.5):
for each element of mem_header
{
    if robot language uses line numbering
```

```
        add line number for each line
    if element is variable declaration
        add all variables and data types from mem_vars
    else
        add element in mem_prog
}

4th stage (figure 1.6):
for each line of mem_prog {
    if robot language uses line numbering {
        add line number
        if the first line element is a label {
            load label and line number in mem_labels
            delete label from mem_prog
        }
    }
    for each element of this line {
        find element in mem_inst
        if exist
            change element with the respective robot code
            if it has arguments
                treat and substitute them
        else find element of the respective data type on mem_data_types
            if exist
                change element with the respective robot data type operation
                if it has arguments
                    treat and substitute them
        if exist symbols in element
            substitute symbols with the respective target ones from mem_symb
    }
    if it is end of line
        add mem_end_line
    if it is the end of the program
        add mem_end_prog
}

5th stage (figure 1.7):
Substitute the InteRGIRo variable identifiers with the ones specified for each data type
Retire the @x_ characters from the variable and signals identifiers
if robot language uses line numbering
    change all labels from mem_prog for line numbers from mem_labels
```

7.3.6 Error Handling

As the generic compiler has, as one input, an InteRGIRo program, and this InteRGIRo program is generated automatically by the GIRo environment, there are no errors to be handled based on this input. However, to generate final code, it is necessary some other inputs, that correspond to the target language description, and some unexpected situations can occur and must be handled.

There are three kinds of messages that can be automatically generated:

– Errors, indicating that it is not possible to generate the final code. However, a "final" code is generated, but it contains the InteRGIRo instructions that could not be translated. The user could easily look for them and change, if necessary. The generated errors are:
    - *ERROR: XXX data type was not defined for the target language.* - This error

happens when a data type operation is used on the InteRGIRo code, but its data type file was not created;

- *ERROR: It is not possible to generate code for this target language. There is no XXX data type on it.* - This error happens when a data type operation is used on the InteRGIRo code, but its XXX data type file does not exist on the target language (the name of this data type, for this target language, was defined as `{:-()`;
- *ERROR: It is not possible to generate code for this target language. There is no YYY instruction on its XXX data type.* - This error happens when the YYY data type operation is used on the InteRGIRo code, but its XXX data type file was not defined on the target language (the data type instruction is empty, or the instruction definition is recursive);
- *ERROR: There is another LLL label defined with the same name.* - This error happens when an InteRGIRo action is described with a sequence of instructions, and it is necessary to create labels. The LLL created label already exist;
- *ERROR: The argument must be greater than zero* - This error happens when an instruction has parameters, and one of the them, that will be substitute by the InteRGIRo argument located on this position, has a value smaller or equal to zero;
- *ERROR: The target argument value is greater than the amount of the InteRGIRo instruction arguments* - This error happens when an instruction has parameters, and one of the them, that will be substitute by the InteRGIRo argument located on this position, has a value that is greater than the amount of arguments from this InteRGIRo instruction;
- *ERROR: It is necessary to initialize all variables, but there is no initial value for the XXX data type* - This error happens when it is specified, on the program structure input file, that all the variables must be initialized, but there is no initial value specified on the XXX data type input file;
- *ERROR: XXX variable has an unknown data type* - This error happens when some variable has a data type different from the accepted ones (the character that comes after the @ is different from the A, B, C, I, M, N, O, P, R, S or T);
- *ERROR: It is not possible to generate code for this target language. The YYY instruction, on the XXX data type, does not exist on this target language* - This error happens when the YYY data type instruction does not exist on the XXX data type file (the instruction was defined as `{:-()`;

– Warnings, indicating that there are some elements, on the robot input files, that could be defined, but it is not really necessary. It is possible to generate final code normally. The generated warning is:

  - *WARNING: XXX data type was not defined* - This warning happens when a XXX data type exists, its operations exist, but data type file was not defined for the target language (the name of this data type, for this target language, is empty). As it was presented before, this situation normally occurs with the input, output and point data types;

– Errors/Warnings, depending on the previous messages, these ones can be interpreted as errors or warnings. The generated errors/warnings are:

- *ERROR/WARNING: Target language do not have `IF` statements. You must verify if it has instructions that evaluate the relational operators* - This message occurs when it is indicated that the target language does not have `IF` statements (the `ifs` and `ifv` InteRGIRo instructions were defined as `{:-()`). So, there must have instructions to evaluate the relational operators. If they exist, there is no problem. But if they do not exist, it is impossible to generate target code for programs that have `IF` statements (almost all of them have). This message is important only for the user that is creating the robot inputs files, to help him to verify if these inputs are correctly specified. For other users, this message can be discarded;
- *ERROR/WARNING: There is no YYY instruction on the target language. It was discarded on the final generated code* - This message occurs always that an instruction is defined as `{:-()`, indicating that it does not exist on the target language. So, it is discarded. If this instruction is not important for the final code, (for example, a clear screen instruction), there is no problem on discarding it. But if it is important, the user must: write the final code instruction directly on the source program (loosing portability), or modify the robot inputs files to include this new instruction;
- *ERROR/WARNING: The YYY instruction was not defined! It is assumed that it is a target language instruction* - This message occurs always that the target instruction is empty, or it is not described on the robot input files, indicating that it was still not defined. So, it is assumed that it is a final instruction. If it is not a final instruction, the user must: modify the generated target program, substituting this instruction with the correct one, or modify the robot inputs files to include this new instruction.

7.4 Chapter's Considerations

In this chapter it was presented the Generic Compiler, that is, with InteRGIRo and PaintGraf, the main components of GIRo's approach.

The Generic Compiler is responsible for translating an InteRGIRo program into a industrial robot program, written on the industrial robot programming language. To do this, some information about the target language must be entered by the user. However, these information can be obtained in a simple way: reading the programming language manual.

After reading this chapter, someone can say that it is not so easy to describe the robot language inputs. They do not need only some details about the target language, but also, it is necessary some knowledge about the special syntax used to describe these inputs, (and the robot programmer may be not familiarized to it), it is necessary to take care about the parameters, and some other details. It is true, but it is also true that:

– it is easier to describe a programming language than a hardware architecture;
– it is easier to take a look at the robot programming language manual than to the description of a machine;

– it is easier to have a programming language manual than the description of the hardware;
– it is difficult to "open" the robot controller, "view" the hardware, and "create" a description of the machine.

The problem about the description of the hardware can be resumed in only one phrase: The robot controllers have closed architectures.

As it will be seen on the conclusions of these thesis, it will be implemented a graphical interface that will facilitate a lot the description of the target language. This interface will ask some simple questions that can be answered by taking a look at the manual, and based on the user answers, the robot inputs files will be automatically generated.

In the next chapters, the GIRo implementation will be presented, with some case studies that validate this work.

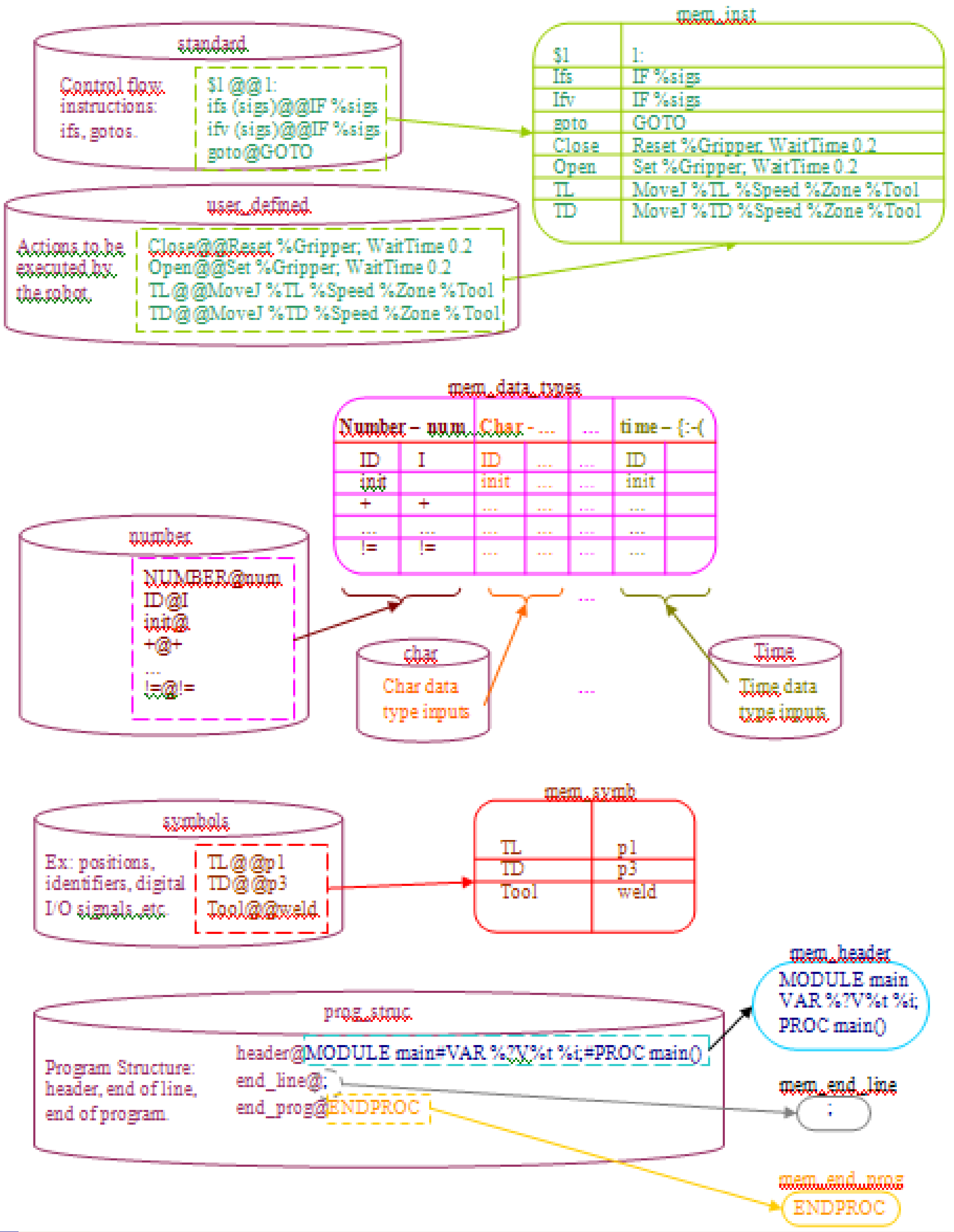

standard
Control flow instructions: ifs, gotos.
$1 @@1:
ifs (sigs)@@IF %sigs
ifv (sigs)@@IF %sigs
goto@GOTO
mem_inst
$1 1:
Ifs IF %sigs
Ifv IF %sigs
goto GOTO
Close Reset %Gripper; WaitTime 0.2
Open Set %Gripper; WaitTime 0.2
TL MoveJ %TL %Speed %Zone %Tool
TD MoveJ %TD %Speed %Zone %Tool
user_defined
Actions to be executed by the robot.
Close@@Reset %Gripper; WaitTime 0.2
Open@@Set %Gripper; WaitTime 0.2
TL@@MoveJ %TL %Speed %Zone %Tool
TD@@MoveJ %TD %Speed %Zone %Tool
mem_data_types
Number – num
Char - ...
...
time – {:-(
ID I
init
+ +
!= !=
number
NUMBER@num
ID@I
init@
+@+
...
!=@!=
char
Char data type inputs
Time
Time data type inputs
mem_symb
TL p1
TD p3
Tool weld
symbols
Ex: positions, identifiers, digital I/O signals, etc.
TL@@p1
TD@@p3
Tool@@weld
prog_struc
Program Structure: header, end of line, end of program.
header@MODULE main#VAR %?V%t %i;#PROC main()
end_line@;
end_prog@ENDPROC
mem_header
MODULE main
VAR %?V%t %i;
PROC main()
mem_end_line
;
mem_end_prog
ENDPROC

Figure 7.2: First stage of translation from InteRGIRo into industrial robot language.

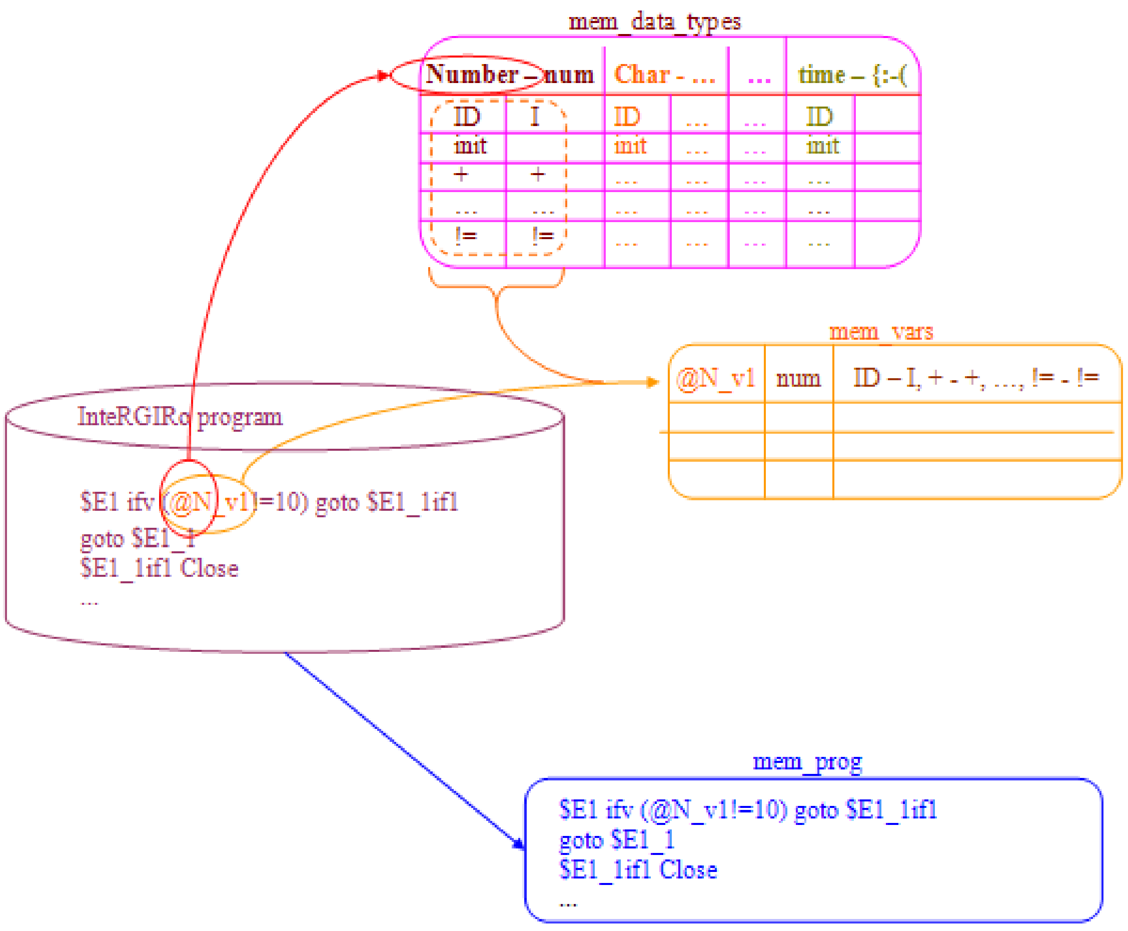


Figure 7.3: Second stage of translation from InteRGIRo into industrial robot language.

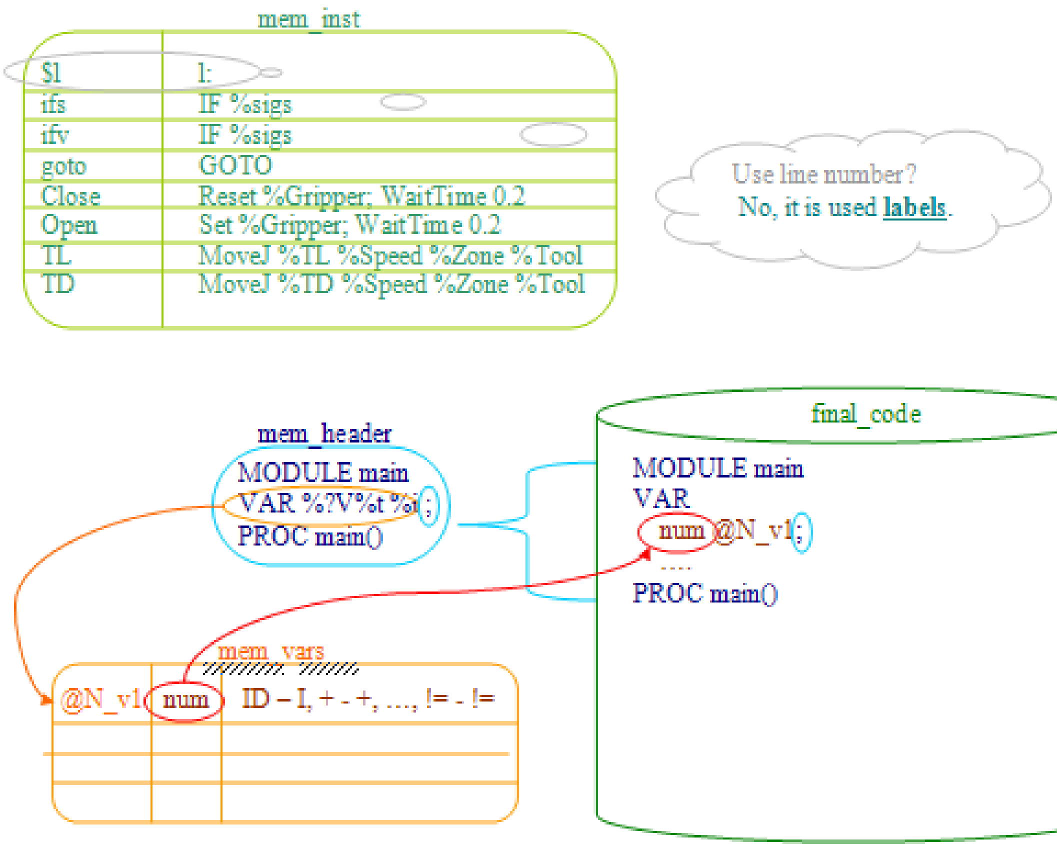


Figure 7.4: Third stage of translation from InteRGIRo into industrial robot language.

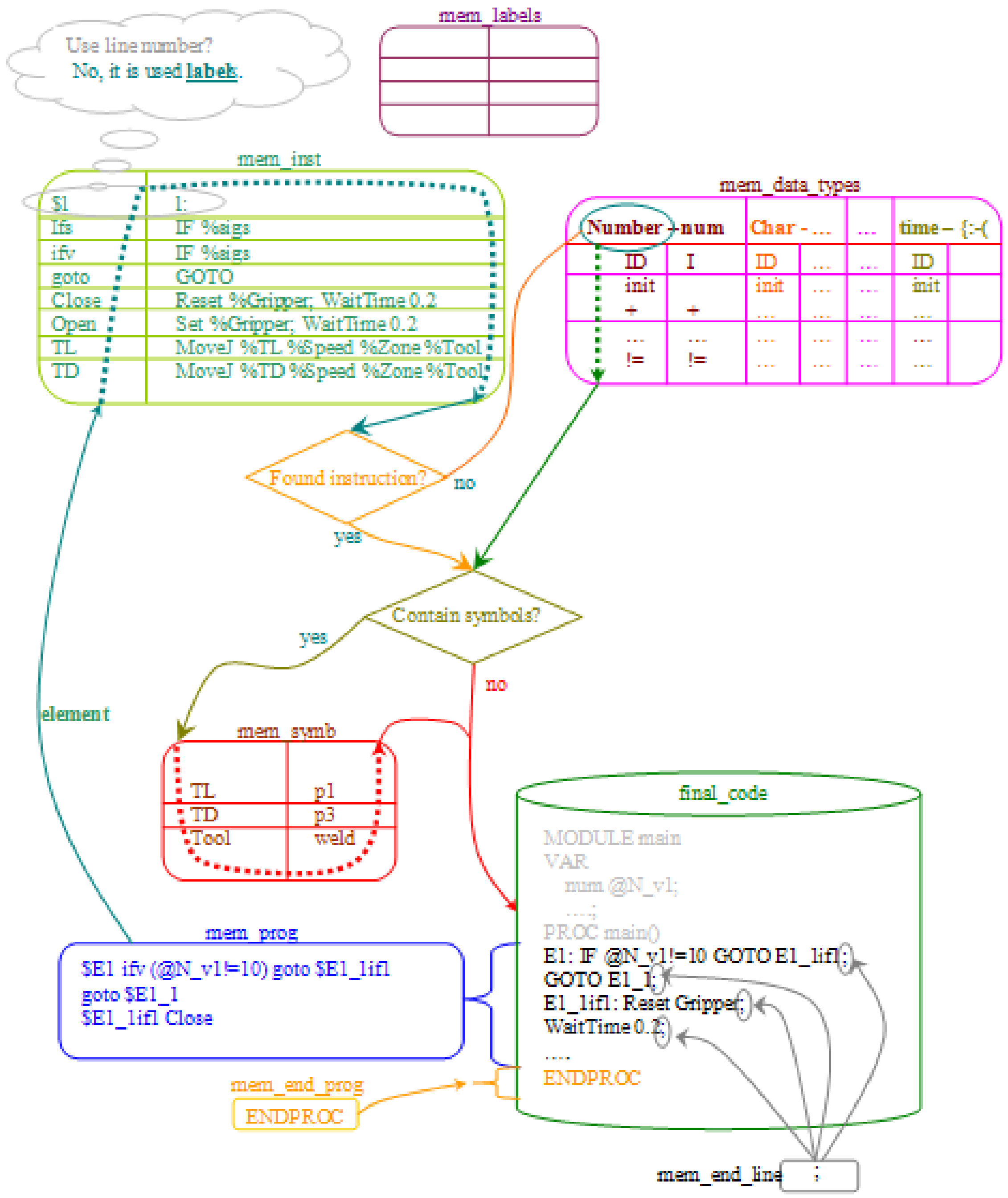
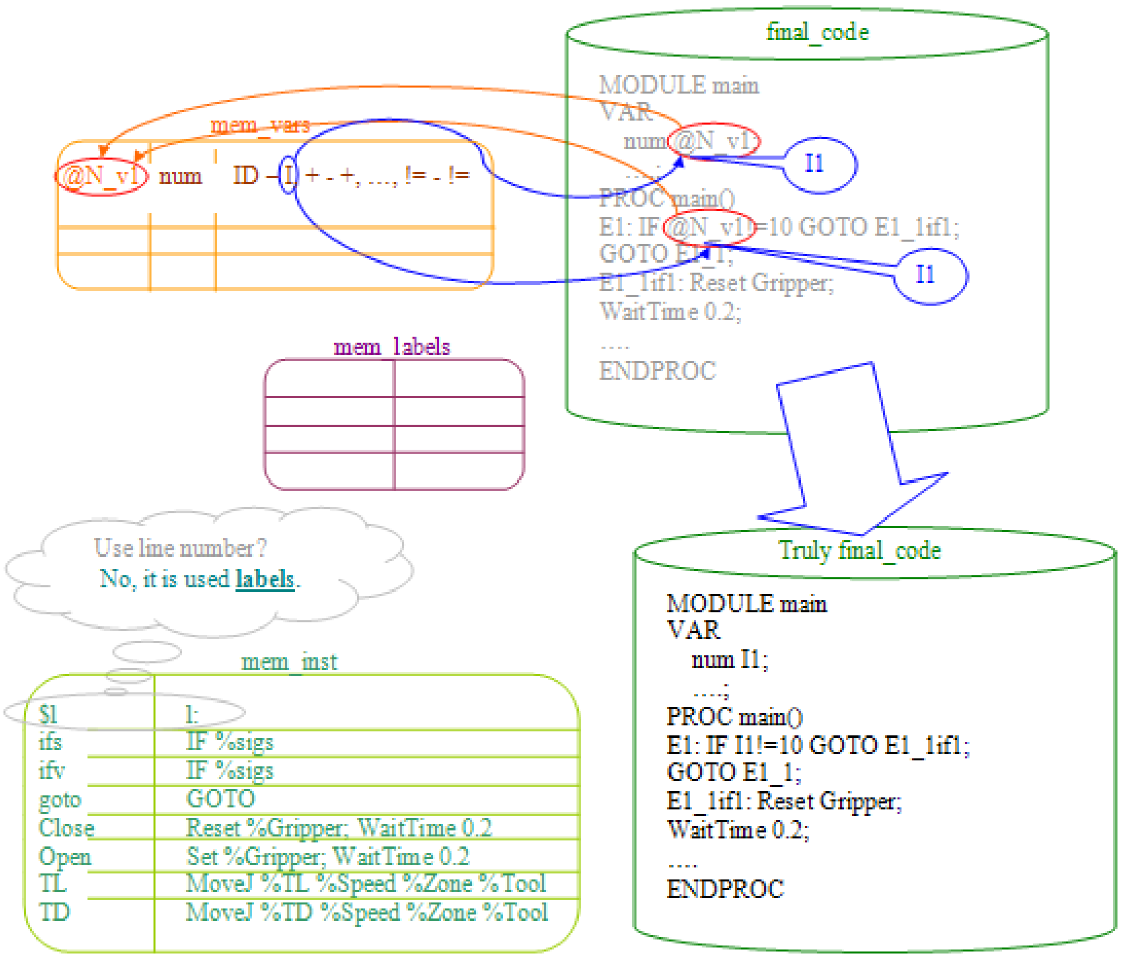


Figure 7.5: Fourth stage of translation from InteRGIRo into industrial robot language.


mem_vars
@N_v1 num ID – I1 + - +, ..., != - !=
final_code
I1
I1
mem_labels
Use line number?
No, it is used labels.
mem_inst
$1 l:
ifs IF %sigs
ifv IF %sigs
goto GOTO
Close Reset %Gripper; WaitTime 0.2
Open Set %Gripper; WaitTime 0.2
TL MoveJ %TL %Speed %Zone %Tool
TD MoveJ %TD %Speed %Zone %Tool
Truly final_code

```
MODULE main
VAR
  num @N_v1
  ....
PROC main()
E1: IF @N_v1!=10 GOTO E1_1if1;
GOTO E1_1;
E1_1if1: Reset Gripper;
WaitTime 0.2;
....
ENDPROC
```

```
MODULE main
VAR
  num I1;
  ....;
PROC main()
E1: IF I1!=10 GOTO E1_1if1;
GOTO E1_1;
E1_1if1: Reset Gripper;
WaitTime 0.2;
....
ENDPROC
```

Figure 7.6: Fith stage of translation from InteRGIRo into industrial robot language.

Chapter 8

Our Solution: GIRO's Approach

This chapter presents our solution to achieve the source code portability for industrial robots programming, the GIRo's approach.

To grant portability for the source code, the initial approach needed an automatic generator of code generators. To generate automatically code generators, it is necessary to have some information about the target machine. But, how can the programmer use the environment at the first time for a different robot? It should be necessary, for this user, to give the specification of this robot to the code generator generator. And it would not be a simple task. Maybe impossible because normally, the robot controllers have closed architectures. So, to facilitate this task, it was decided to change this input.

After some research and development, the initial approach has evolved to GIRo's approach [AHF06]. It happened because one of the main goals was to allow any user to use this environment. Instead of giving a specification of the robot, a subset of the robot language grammar must be defined. The user can get this subset taking a look at the robot language manual. And with a friendly interface, he could give this grammar in a easy way, just answering some questions.

GIRo's approach can be seen on figure 8.1. It is supported by the following languages and tools that compose the architecture we want to defend:

– a truly high level and declarative language (Grafcet)
– an easy-to-use front-end (Paintgraf)
– an intermediate representation (InteRGIRo)
– the translators to InteRGIRo
– the generic compiler
– the robot language inputs editor

At the top, there is the problem to be solved. It should be used an adequate modelling technique that describes formally the overall problem. As the modelling language adopted was Grafcet, it was included in the FE a Graphical interface to aid editing the Grafcet visual description, the *Paintgraf*, an easy-to-use compiler front-end, based on Grafcet, that is a truly high level language, close to the specification instead of the robot, that interprets

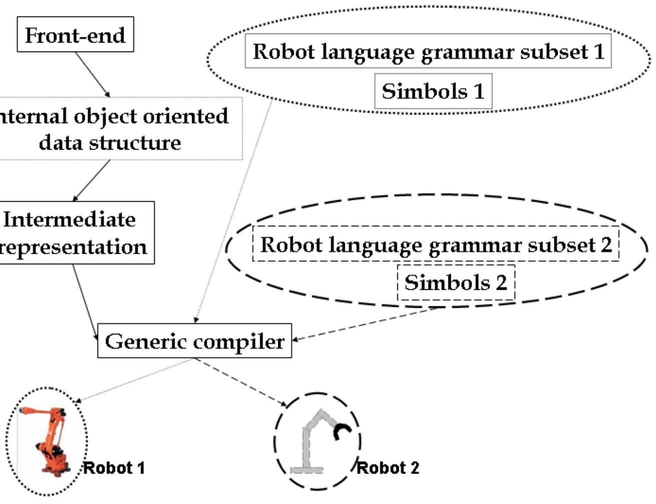

Front-end
Internal object oriented data structure
Intermediate representation
Generic compiler
Robot language grammar subset 1
Simbols 1
Robot language grammar subset 2
Simbols 2
Robot 1
Robot 2

Figure 8.1: GIRo's approach to industrial robots programming.

the specification language and generates an internal data structure, that can be translated into *InteRGIRo*, an intermediate description for the program specified. To generate final code it is also necessary to get some description of the target machine, the *robot language inputs*. With all of these inputs (InteRGIRo and robot language inputs), the final code generation is possible. To generate both the InteRGIRo and the final code, it is necessary some *translators*, like the generic compiler, that is responsible for the final code generation.

Some of these components where presented on previous chapters, like the InteRGIRo and the Generic Compiler. The other components are described on the following sections, like the PaintGraf front-end.

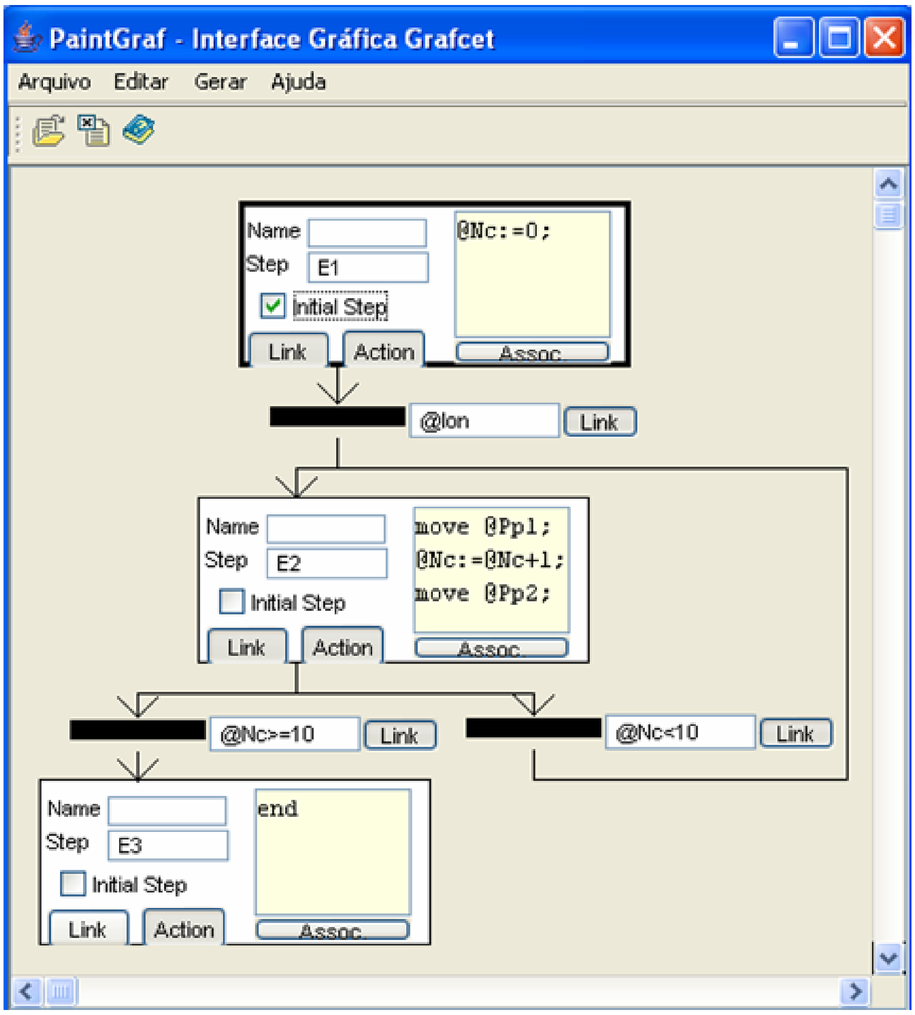
It is important to say that the standard Grafcet is completely represented in this environment. The precise description of the automatism functioning, that must be given by the command part, is achieved in the two successive and complementary levels, that were presented on section 5.4: the first one because the Paintgraf allows to describe completely the standard Grafcet; and the second one by the use of the robot language inputs files, that contains the instructions descriptions on the target language, that are used during the translation process between the InteRGIRo and the robot target language.

8.1 PaintGraf

One component of this approach is a friendly graphical interface, based on the Grafcet specification diagram [AHF05]. This interface is responsible not only for facilitating the programming task, but also to translate this diagrammatic specification into languages that belong to different levels of our approach. This interface is user-friendly, the program specification is close to the problem (instead of close to the robot), and allows environment interaction (which means that our implementation will allow the development of closed-loop systems). It is also expressive.

The interface represents all the concepts and features presented on a Grafcet diagram, and have a similar appearance. There are some syntax differences, like the necessity for using an "@" preceding the variables and digital input/output signals.

The examples used to generate the InteRGIRo and the final C program, in chapters 6 and 7, respectively, correspond to the following Grafcet representations (on PaintGraf). The program responsible for making the robot move ten times between two points (see figure 8.2), and the one that calculates the factorial of a given number (see figure 8.3).



Name
Step E1
[✓ Initial Step]

```
@Nc:=0;
```

@Ion

Name
Step E2
Initial Step

```
move @Pp1;
@Nc:=@Nc+1;
move @Pp2;
```

@Nc>=10

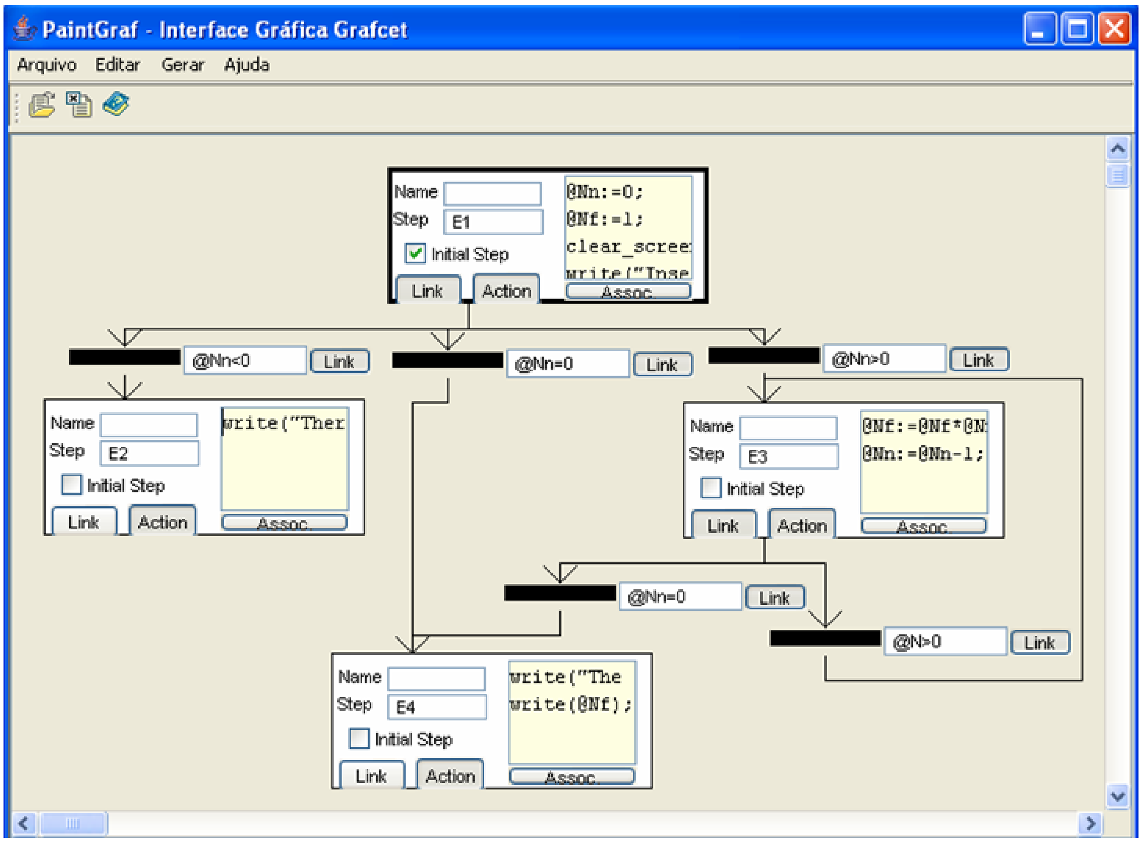
@Nc<10

Name
Step E3
Initial Step

```
end
```

Figure 8.2: Program example made on the PaintGraf graphical interface, responsible for making the robot move 10 times between two points.

In the rest of this section, it is explained how the Grafcet constructors were implemented in PaintGraf.



```
@Nn:=0;
@Nf:=1;
clear_scree
write("Inse
```

@Nn<0 @Nn=0 @Nn>0

```
write("Ther
```

```
@Nf:=@Nf*@N
@Nn:=@Nn-1;
```

@Nn=0 @N>0

```
write("The
write(@Nf);
```

Figure 8.3: Another program example made on the PaintGraf graphical interface, responsible for calculating the factorial of a given number.

8.1.1 Steps

The steps are represented as squares. Inside there are two fields: the *number* of this step, that is a mandatory field (a step number is generated automatically, but the user can modify it); and the *name* of the step, that is an optional field (the user can name the step to make it easy to understand its main goal). There is a checkbox, called *initial*, that indicates if this is an initial step. If it is, the square border becomes thicker. After inserting a new step (square) into the diagram, the editor offers two more options, that are the two buttons inside the square: the first one is the *link* button, responsible for linking this step to the next transition(s); the last one is the *action* button, responsible for associating actions to this step (figure 8.4 left). When this *action* button is pressed, the square is augmented, at the right side, and a text box appears, where the user can enter the actions code. The *associate* button inside this new right box allows to store these actions (when they are written) associated to the current step (figure 8.4 right).

Each action should be written considering the following grammar, which makes possible to represent all the conditions and actions proposed by Grafcet:

```
actions -> ( "(" condition ")" "{" (action ";")+ "}" )+
```

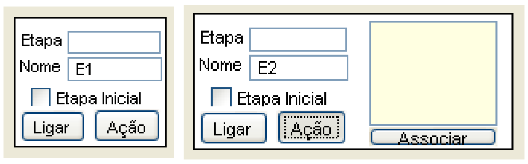


Figure 8.4: Step graphic representation (left) and step and actions graphic representation (right).

```
condition -> cond | condition op cond
cond -> iv | ~iv | n_step | var | simb
var -> "@" type_var string
type_var -> "N" | "B" | "C" | "S" | "A" | "R" | "M" | "T" | "I" | "O" | "P"
simb -> % string
op -> "," | "|"
action -> command
```

Conditions are written inside "()" and they are optional; `iv` corresponds to an input variable that must be on, while the ∼`iv` corresponds to an input variable that must be off, `n_step` corresponds to steps that must be active, `var` corresponds to a variable, and `simb` corresponds to a symbol that will be treated by the generic translator. If a condition is satisfied, the action(s) can be executed. If there are more then one action to be executed for one condition, these actions must be written inside "{}". It is necessary to use a ";" at the end of each action. Any variable must be preceded by the @ and the data type characters, while any symbol must be preceded by %. The commands, in this work called actions, will be treated only by the generic compiler, and can contain variables, signals and symbols.

8.1.2 Transitions

Each transition is represented by a thick horizontal line, followed by the *condition* field, that must be written by the user, and by a *link* button, responsible for linking this transition to the next step(s) (figure 8.5).

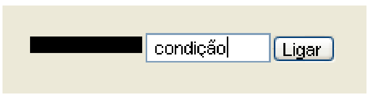


Figure 8.5: Transition graphic representation.

Each condition should be written considering the following grammar, which makes possible to represent all the functions proposed by Grafcet:

```
condition -> expr | condition op expr | "1"
expr -> v "+" | v "-" | time | "(" conditon ")" | var | var op_rel elem
time -> t "/" n_step "/" seconds
var -> "@" type_var string
type_var -> "N" | "B" | "C" | "S" | "A" | "R" | "M" | "T" | "I" | "O" | "P"
op_rel -> "=" | "!=" | "<" | ">" | "<=" | ">="
elem -> num | @var
op -> "," | "|"
```

where `v+` corresponds to set the variable `v` to 1, while the `v-` corresponds to reset the variable `v` to 0.

The number `1`, that can substitute the condition, indicates that such transition is always valid and do not need to be evaluated.

8.1.3 Alternatives and Simultaneous Sequences

The sequences are generated automatically by the system. The user must only link the steps to the transitions and vice-versa. If the user wants to link one transition to more than one step, a double line is presented, to show that this is the beginning of a simultaneous sequence. Also, if the user links more than one step to only one transition, a double line is also presented, but to indicate that this is the end of a simultaneous sequence. The other links are treated as alternative sequences. Each object can be moved to any position in the window, without loosing the links.

8.2 GIRo Translators

At this moment, there are three translators ready. The first one translates the Grafcet graphical representation into an RS program, and, as it was presented before, it is not used anymore. The other two are the main components of GIRo, responsible for the correct GIRo's functioning. The second one will be seen in the following subsection, responsible for generating the InteRGIRo program, while the third one, the Generic Compiler, responsible for generating final code, was presented on the previous chapter.

The translators that have Paintgraf as source code assume that the conditions are written using the infix notation. So, a boolean expressions analyzer was created. The third translator (InteRGIRo -> target) does not need this boolean expressions analyzer because InteRGIRo uses only simple expressions. The complex ones were turned into simple three address code ones during the translation process into InteRGIRo or RS.

Basically, the translation process of the first two translators works in the following way:

– The steps and transitions are responsible for the program control flow;
– The actions are responsible for the commands (actions) that would be executed by the robot.

The third one, basically, substitutes the InteRGIRo sentences for the correspondent ones of the robot language.

One of the main features of these translators is that they allow the user to send commands directly to the robot using robot commands, describing them at the Grafcet actions. It might be useful if the user wants to explore some specific potentialities of such a robot, but it makes the problem to be not portable. It is important to say that, any action that is not described in the robot language instructions will be interpreted as a command to the robot!

8.2.1 PaintGraf -> RS Language

As it was said in the RS language chapter, the RS language is no more used in this work. But this decision was achieved after this translator be done. So, it is possible to generates RS code, but nothing more about this work is done after this code generation. The translation process between Paintgraf and RS language is presented following.

The variables are defined previously by the user, but the input, output and internal signals are detected automatically. The RS internal signals corresponds to the steps, the RS input signals are detected during the parsing of the conditions (from actions and from transitions), and the RS output signals are detected during the parsing of the actions that must be executed.

Some auxiliary RS internal signals are also generated, if there are more then one action associated to one condition or step. They are generate to grant the sequential execution of each of these actions.

All the conditions are written using the infix notation. So, a boolean expressions analyzer was created.

It is also possible to call the RS compiler, to translate this RS program into an automaton.

Steps

Each step of Grafcet is treated as an internal signal of the RS language. So, for each step an internal signal is created, if it still does not exist, and included at the module header of the RS program. If the step is a initial one, it is added a command `up(this step)` at the `initially` sentence of the module header. This means that the respective internal signal will be activated at the beginning of the execution, starting the automaton execution at this step.

If the step contains some action, it will be parsed to detect the actions and their respective conditions, if they exist. For each action, an output signal is created and inserted at the module and program headers, and an `emit(this action)` command is added at the point where this action should be executed.

Transitions

Each Grafcet transition is parsed to detect the conditions, the kind of each element (input variables, auxiliary variables, steps), and to analyze the boolean conditions that may exist. Each input variable is added as an input signal (if it does not exist) at module and program headers; each step as internal signal at module heading (if it does not exist). The auxiliary variables where added directly by the user, when using the graphical interface, but they are included on the respective rule, to perform the correct behavior. All of these elements will be included on the left side of the respective rule, with the internal signals corresponding to the steps that precede this transition.

Alternatives and Simultaneous Sequences

The alternative sequences are detected and included on the end of each rule, by the use of the `up(next step)` command. This command is responsible for activating the internal signal associated to the next step that must be evaluated. There is no problem in closing this kind of sequence, because there is no synchronization between the steps of this sequence.

The simultaneous sequences are treated at the same way, but because one transition will trigger more than one step, it will be included more than one `up(next step)`, to activate all the internal signals that are associated to the next steps. Because it is necessary to synchronize the last steps at the end of this sequence, each of these last steps will create an internal signal and activate it. The following step will only be executed if all the internal signals from the precedent steps are active.

Basic Algorithm

This section presents the basic algorithm of the translator described.

```
for each initial step {
  add this step as an internal signal;
  add up(this step) at initially sentence;
  for each following transition
  {
    execute the Transition_evaluating_function;
  }
}

Transition_evaluating_function: if this transition was not
evaluated {
  add its elements as variables, input and
          internals signals;
  evaluate its boolean expression;
  create the rules:
  {
    each element will be on the left side;
    the actions of each next step are evaluated:
    {
      their conditions are also included on the
           left side;
      the actions are included at the right side
           by the emit(action) commands;
    }
    add this next step as an internal signal;
    add the up(next step) command on the right side;
    go to the following transitions for this next
          step and execute the
          Transition_evaluating_function;
  }
}
```

8.2.2 PaintGraf -> InteRGIRo

Basically, there are three kinds of instructions to be generated in this translation process:

- the ones that correspond to the actions. These ones may depend on the robot language. So, if there is no condition on the actions, the only work to be done will be to insert the line number and/or the label of this instruction, and copy the actions;
- the ones that correspond to the transitions. In this case, it is generated a label in the first instruction, it is generated an `if` sentence, and is is also generated another label, to jump when the condition is satisfied;
- goto sentences, that simply do a unconditional jump to the beginning of the transition instruction: when any of the instructions where validated (create a loop until some transition is satisfied); or when it is been used alternative sequences (if a transition is not validated, may have another one that must be tested).

The two Grafcet specifications presented, as examples, on figures 8.2 and 8.3, were translated into the InteRGIRo representations presented in sections 6.3.8, by the PaintGraf - InteRGIRo translator.

Steps

At each step, it is created a label to indicate that this line corresponds to the beginning of a new step, and it is inserted at the beginning of the instruction line. After this, the actions of this step are added. The actions are simply copied, because they must be executed directly by the robot. This means that, or each action is a robot instruction, or it is a identifier/instruction that will be translated into robot instructions by the generic compiler.

This translation process starts with the initial steps. After including an step (and actions), each following transition is evaluated.

Transitions

Each Grafcet transition is parsed to evaluate the boolean conditions that may exist. The condition is transformed into a sum of products condition. Because some robot languages do not allow more than one conditional expression in an `if` sentence, each element of the condition generates one `if` instruction. Each `if` instruction is generated by the following way:

1. It is created a label, by one of the following ways:
   - For the first conditional element, an `Estado + step number` label is generated. It is necessary because the program must change the step only when the transition is satisfied. So, if the transition is not satisfied, it is necessary to evaluate the transition again, without executing the actions. If the step have no actions, only the `step number` is added as a label;

  - If it is not the first conditional element, but it is the first element of each product (AND), an `step number + counter value` label is generated (each transition has its counter to create different labels for each conditional element). It is necessary because each product must be completely satisfied to validate the transition. So, it indicates the beginning of its product sentence;
  - If it is not the first element of each product, an `step number + counter 1 value + if + counter 2 value` label is generated (these 2 counters are needed to create different labels, where the counter 1 indicates that all other sentences with this step number and counter 1 belongs to the same product (AND)). This label is necessary because when a `if` sentence is validated, a jump to the next element that must be evaluated is executed;
2. Depending on the element that will be evaluated, an `ifv` or an `ifs` sentence is added. An `ifv` is used when the element has a variable, and an `ifs` is used when the element has a signal. The element will be added between ();
3. At the end of this InteRGIRo instruction, a `goto` sentence is added. The target label must correspond to:
   - the next step, if the transition is satisfied;
   - the next element of the product (AND), if the evaluated one is satisfied;
4. another line may be added after each `if` sentence to represent the `else` component of it. If there is no other element at this product (AND), but there is another product to be evaluated (sum of products), this line is not necessary, because the next program line will be executed in sequence. This line has only the `goto` instruction, and the target must correspond to:
   - the beginning of another element of the product (AND);
   - the beginning of this transition, if any of the conditions were satisfied.

Alternatives and Simultaneous Sequences

With this version of InteRGIRo, all the sequences are alternative ones. It is because of the purpose of this work, that is to generate code for different robots. How an industrial robot do not perform tasks in parallel, it was not necessary, at this moment, to work with simultaneous sequences.

Basic Algorithm

This section presents the basic algorithm of the translator described and one figure (8.6) that illustrates the algorithm. We think that the algorithm and the figure are self-explanatory and do not deserve more comments.

```
for each step (starting with the initial steps) {
  create and add a label;
  add the actions;
  for each following transition {
    evaluate its boolean expression;
    for each element of the expression {
      create and add a label,
      create a if (element) goto [next step or next element of product];
    }
    add a goto [begin of this transition or begin of another element of product]
  }
}
```

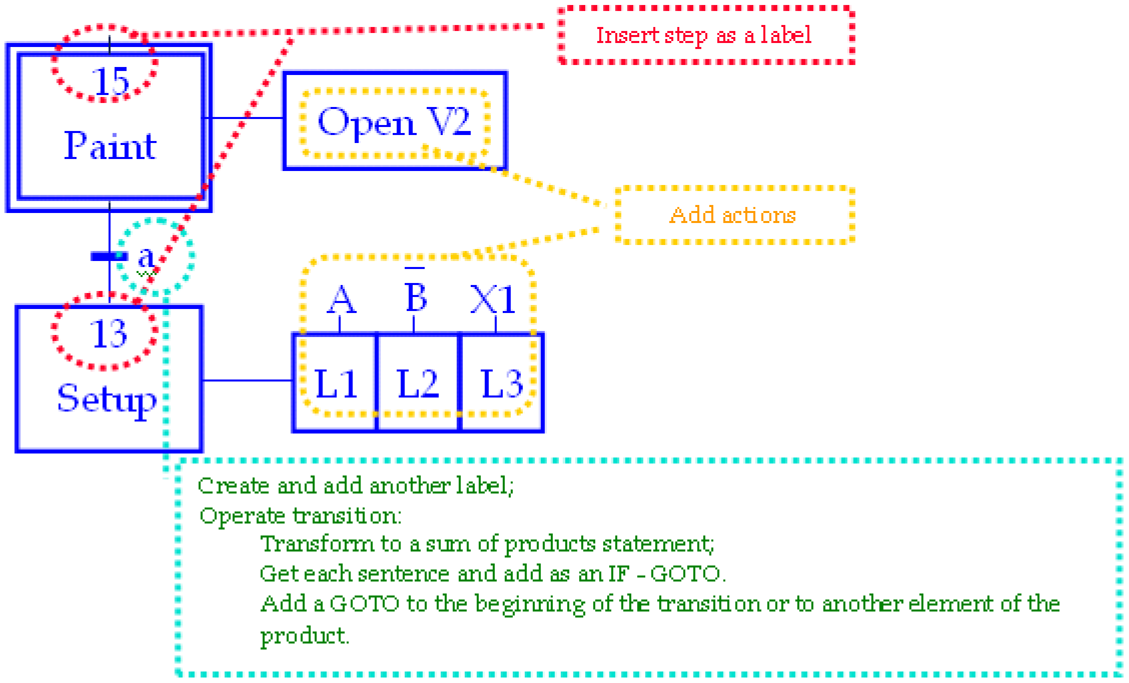

Insert step as a label
15
Paint
Open V2
Add actions
a
13
Setup
A
B̄
X1
L1
L2
L3
Create and add another label;
Operate transition:
Transform to a sum of products statement;
Get each sentence and add as an IF - GOTO.
Add a GOTO to the beginning of the transition or to another element of the product.

Figure 8.6: Schema of the translation from Paintgraf into InteRGIRo.

8.2.3 Error Handling

During the generation of an InteRGIRo program, some errors can occur and must be handled. Following comes the automatically generated errors for this translator:

– *Parenthesis error* - there is an error on the parenthesis organization: there are more or fewer parenthesis than it is necessary;
– *Unknown element* - there are some unknown or unexpected elements, that are described on the message error;
– *Type mismatch* - there are different data types on the same instruction. The instruction is evaluated and accepted, because there are target languages that accept this situation, while other do not. But it is important to inform the user about this situation;
– *XXX variable has an unknown data type* - there are some variable that has a data type different from the accepted ones (the character that comes after the @ is different from the A, B, C, I, M, N, O, P, R, S or T);
– *Variable VVV has not been initialized* - there are some variables, in such case, the VVV one, that has not been initialized. All of the variables, except the input, output, point and time ones, must have a initial value, before their first use;

– *There is no initial step.*

8.3 Optimization: "Variable" allocation

During the translation process between the PaintGraf program into the InteRGIRo one, an optimization is done, responsible for trying to detect the minimum number of variables for a given program. The goal is the same of a Register Allocation algorithm, but it is located on a higher level, dealing with variables instead of registers.

This is done because some older industrial robots have a limited hardware, that limit the number of variables to be used by their programs. So, a "variable" allocation optimizer was created, based on the existing register allocation ones.

This optimizer is executed during the translation process, from the PaintGraf source code, into the InteRGIRo program. It is done at this moment because, as InteRGIRo program is translated into several different target programs, with an intermediate program that has the minimum number of variables, the automatically generated final codes will maintain the same amount of variables.

It was based on the graph coloring algorithm, and it is implemented using as many linked lists as the existence of different data types on the source code. These linked lists will indicate the life time of each variable of that data type.

The elements of these linked lists are inserted and modified each time a variable is encountered during the source code analysis, on the following way:

– If it is the first occurrence of such variable, it is inserted two times into the linked list for its own data type;
– If it is not the first time, the second variable occurrence on its data type linked list is moved to the end of this linked list.

After analyzing all the source code, some linked lists where created, one for each used data type. The life time of a variable is indicated by the first and last occurrence of such variable: the first occurrence of a variable on some linked list indicates the first use of it on the source program, while the last occurrence corresponds to its last use. All the variables that are located between the first and last occurrence of such variable correspond to the variables that are been used during the same time, and can not share the same variable identifier. It can be viewed as a coloring graph, where the variable that is been analyzed is linked to all other that are located between its first and last occurrence on the data type linked list.

After analyzing each data type linked list, it is possible to:

– identify the minimum number of variables for each data type (minimum number based on this algorithm, that was not compared with the optimal graph coloring one to make sure that the generated number is really the minimum);
– identify the variables that can be shared;
– identify the variables lifetime.

So, it can be said that, during the Paintgraf -> InteRGIRo translation, only the minimum number of variables are generated, and not all of them that were defined in the PaintGraf source program.

8.4 Chapter's Considerations

In this chapter it was presented the GIRo approach and implementation, more specifically: the PaintGraf; the translators between PaintGraf representation and InteRGIRo, and between InteRGIRo and the target programming language; and the "variable" allocation optimizer. All the translation process and the used algorithms where described in details.

In the next chapter, some case studies are presented, to validate the proposed approach.

Chapter 9

Case Studies

This chapter presents five case studies that validate the GIRo's approach.

For each one, it is presented its: description; Grafcet representation and the corresponding program (in Paintgraf Editor); Intergiro representation; one or two target language descriptions; and two target codes that were automatically generated for the two target languages (the other five target codes are presented in the appendices), proving the main goal of this thesis.

The chosen target languages can be separated in two sets:

1. The industrial robot programming languages, that correspond to the scope of this work. It was chosen five programming languages, according to the possibility of using them at the Department of Industrial Electronics of the University of Minho, that are:
   - VAL and VAL II (Puma robots);
   - Rapid (ABB robots);
   - Melfa Basic III (Mitsubishi Movemaster RV-M2 robots);
   - Luna (Sony robots)
2. The personal computer programming languages, that does not correspond to the scope of this thesis, but they can prove that it is possible to generate final code to other different procedural languages than the tested ones. This means that GIRo ***could*** be used to generate final code to other different industrial robot programming languages, that could not be tested due to practical reasons. It was chosen the Pascal and C programming languages.

In A, B, C, D, and E, it is presented the automatically generated code to the other target languages, that were not presented on the specific case study section. Each following appendix contains the programs that were not presented on the case studies:

- A - Simple;
- B - Boxing apples;
- C - Bottles palletization;
- D - Level rail crossing;
- E - Factorial.

9.1 Robot Language Descriptions

As it was said before, to allow the automatic generation of final code, it is necessary to have some information about the target programming languages.

It was presented, in chapter 7, the information that are necessary, for the GIRo environment, to achieve such goal. These informations are distributed basically in four types of files:

– One file with the program structure data;
– Two files for the robot language instructions, one with the standard instructions, and other with the user defined ones;
– Some files for the data types specification, one for each existing data type;
– one file with the used symbols.

Normally, the first three items are defined once, and used always that is necessary. Even the user defined instructions, where the user can create new instructions if it is needed.

However, the symbols file should be specified, or verified, for each program. Different programs can use the same symbol for different elements, or positions, or digital input/output signals. For example, two programs can use the identifier P1 for a given point, but despite the same identifier, the programs can refer to different spatial positions. So, it is important to specify the symbols that are used on a single program on its symbols file. At last, verify if the symbols file maps correctly the used identifiers with its correspondent description on the target robot.

Before presenting the case studies, all of these files had to be created, one for each target programming language. They were specified only once, and used in all of the case studies. Even the symbols file, that should be specified for each program. However, each program had its own symbols file. They were merged into only one file to facilitate reading and explaining.

These robot language inputs files are better explained in the following subsections:

– VAL programming language - subsection 9.2;
– VAL II programming language - subsection 9.3;
– Rapid programming language - subsection 9.2;
– Melfa Basic III programming language - subsection 9.4;
– Luna programming language - subsection 9.5;
– Pascal programming language - subsection 9.6;
– C programming language - subsection 9.6;

9.2 A Simple Case Study

This case study implements a simple industrial robotic application, responsible for moving between three points, opening and closing the gripper. A Grafcet description of such situation, that was made using PaintGraf, can be seen on figure 9.1.

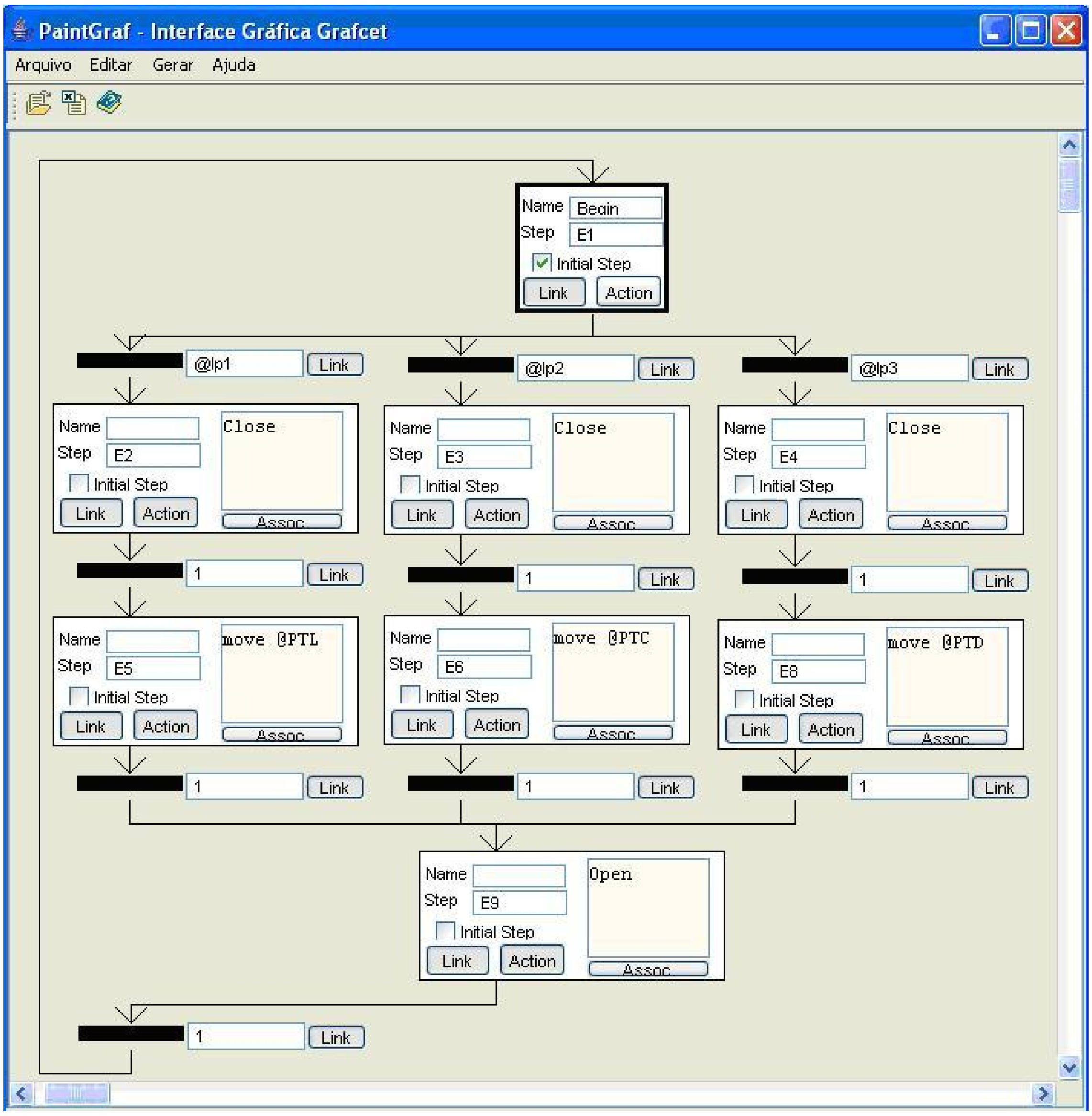


Figure 9.1: Grafcet diagram for simple case study drawn in PaintGraf editor

In this case study, a robot can move throw three points, PTL, PTC and PTR (PTC at the center, PTL to the left, and PTR to the right). There are three buttons IP1, IP2 and IP3, responsible for indicating the point to where the robot must make a transfer (IP1 for PTL, IP2 for PTC, IP3 for PRT). When one button is pressed, the robot must do the following steps:

– Close the gripper;
– Make a transfer to the a left or to the right, depending on the pressed button and the current position;
– Open the gripper when arriving at the desired position.

9.2.1 InteRGIRo Code

The InterGIRo program generated for the Grafcet specification in figure 9.1 is listed below.

```
$E1 ifs (@Ip1) goto $E2
    ifs (@Ip2) goto $E3
    ifs (@Ip3) goto $E4
    goto $E1
$E2 Close
    goto $E5
$E5 move @PTL
    goto $E9
$E9 Open
    goto $E1
$E3 Close
    goto $E6
$E6 move @PTC
    goto $E9
$E4 Close
    goto $E8
$E8 move @PTD
    goto $E9
```

9.2.2 Robot Language Inputs for VAL Programming Language

To generate VAL programs, it is necessary to specify five data types files, besides the program structure, robot language instructions, and symbols ones. The needed data types files are: Input, Numeric, Output, Point and String. Despite String is not really an existing data type, VAL has instructions responsible for writing string constants. So, it was necessary to created this fifth data type file. All of these files are presented following.

Program Structure

VAL programs do not have headers, variable declaration, special character for the end of lines, and even special statements that indicate the end of the program. So, there are no data for this file, as can be seen below.

```
header@
end_line@
end_prog@
```

Standard Instructions

VAL programs use *goto* instructions to perform unconditional jumps. To allow conditional jumps, *ifv* and *ifs*instructions are associated with the *goto* ones. The target of all jumps are specified by labels that are located on the beginning of the target instructions. As the labels are described within numbers, it is indicated, at the beginning of the standard instructions file that the VAL programs will use line numbering, with an increment of 10 for each new line. So, each line is numerated, making possible to achieve any line of the GIRo's generated VAL program.

The *ifs* sentence, with signals on its condition, has a special syntax that allows the evaluation of, at most, four digital input signals. As InteRGIRo do not use logic operators, there are no InteRGIRo *ifs* sentence that treats more than one input signal at the same time. So, three commas must be inserted after the only one used input signal in all of the VAL *ifsig* instructions.

The standard instructions file is presented below.

```
$l@@n10
ifs@@ifsig %sigs,,, then
ifv@@if %sigs then
goto@@goto
```

User Defined Instructions

For these case studies, the only needed instructions are the ones responsible for opening and closing the robot gear. A third one, responsible for cleaning the screen, was also defined because the screen is the most used device to output informations. However, VAL language do not have such instruction, and the special statement *{:-(* must be used to indicate its inexistence. With this statement, if the *clear_screen* instruction is used on the source program, **it will not be translated or copied** into the final code, and a error/warning message will be given to the user. Without this indication on this file, it would be assumed that such instruction corresponds to a final one, and **it will be copied** to the final code, indicating the user with a different error/warning message.

```
Close@@closei
Open@@openi
clear_screen@@{:-(
```

Input Data Type

There is no input data type on VAL language, but there are some operations/instructions associated to the input signals. This situation is indicated by the absence of data on the right side of the *INPUT* sentence, as can be seen below. Also, there is an absence of data for the *ID* sentence, which means that the InteRGIRo identifier, for the input data type, must be preserved.

```
INPUT@
ID@
init@{:-(
~@-%\1%
_@%\1%
=@{:-(
!=@{:-(
<@{:-(
>@{:-(
<=@{:-(
>=@{:-(
write@{:-(
read@{:-(
```

As it was explained in chapter 3, VAL allows the use of 32 input channels, numbered 1001 to 1032. This means that, the input signals used on the source program must be mapped to the respective channel number. The symbols file is responsible for this mapping, and this is why the *ID* sentence has no data on its right side: the InteRGIRo identifier is preserved during the data type evaluation, to be mapped to a input channel number at the end of the InteRGIRo program evaluation.

However, as a data type, it has some basic operations associated to it. In VAL language, there are only two basic operations for input signals, that are the ones responsible for verifying if the input signal is on or off. The InteRGIRo operator , that means absence of the input signal, has a correspondent on VAL, that is the - before the input signal, that is represented by the argument *%\1%*.

The other "operator" is not really an operator. When an input signal is written alone at InteRGIRo, it means that the input signal is present. However, to find a correspondent representation for this signal presence on the input data type file, it was necessary to create an "operator" to be found. This "operator" is represented, on the input data type file, as the character _ , and has no correspondent on VAL language. To indicate the presence of a signal, on VAL language, it has to be written only the input signal, represented by the argument *%\1%*.

The other sentences, like the initialization, the relational operators, and the read / write instructions, are not allowed on VAL language for the input signals. So, all of them do not have a correspondent one, and to indicate this situation, the characters ***{:-(*** must be associated to them.

Numeric Data Type

Different from the Input data type, there is a numeric data type on VAL language, but there is no variable declaration because VAL language deals only with numeric and point data types, and all of them are global. It is easy to identify the point variables, which means that all other variables belong to the numeric data type. This situation is also indicated by the absence of data on the right side of the *NUMBER* sentence, as can be seen below. Also, there is an absence of data for the *ID* sentence, which means that the InteRGIRo identifier, for the number data type, must be preserved.

```
NUMBER@
ID@
init@{:-(
+@+
-@-
*@*
/@/
\@{:-(
%@{:-(
:=@seti %\1%=%\2%
=@=
!=@!=
<@<
>@>
<=@<=
>=@>=
write@type/N, %\1%
read@prompt " ", %\1%
```

The basic arithmetic and relational operators has the same representation on InteRGIRo and VAL, while the assignment InteRGIRo operator is mapped to a ***seti*** instruction, preserving the left (*%\1%*) and right (*%\2%*) operands from the InteRGIRo assignment operation.

The other sentences, like the initialization, and the read / write instructions, are not allowed on VAL language for the numeric data type. Also, there is no \ (integer division) and % (remainder) operators, because the numeric data type manipulates real numbers. So, all of them do not have a correspondent one, and to indicate this situation, the characters ***{:-(*** must be associated to them.

Output Data Type

Like the input data type, there is no output data type on VAL language, but there are some operations/instructions associated to the output signals. This situation is indicated by the absence of data on the right side of the *OUTPUT* sentence, as can be seen below. Also, there is an absence of data for the *ID* sentence, which means that the InteRGIRo identifier, for the output data type, must be preserved.

```
OUTPUT@
ID@
init@{:-(
on@signal %\1%
off@signal -%\1%
:=@{:-(
=@{:-(
!=@{:-(
<@{:-(
>@{:-(
<=@{:-(
>=@{:-(
write@{:-(
read@{:-(
```

As it was explained in chapter 3, VAL allows the use of 32 output channels, numbered 1 to 32. This means that, the output signals used on the source program must be mapped to the respective channel number. The symbols file is responsible for this mapping, and this is why the *ID* sentence has no data on its right side: the InteRGIRo identifier is preserved during the data type evaluation, to be mapped to a output channel number at the end of the InteRGIRo program evaluation.

However, as a data type, it has some basic operations associated to it. In VAL language, there are only two basic operations for output signals, that are the ones responsible for setting and resetting the output signal. The InteRGIRo instruction ***on***, that sets an output signal, is described, on VAL, as *signal %\1%*; while the other InteRGIRo instruction ***off***, that resets an output signal, is describe, on VAL as *signal -%\1%*.

The other sentences, like the initialization, the relational operators, and the read / write instructions, are not allowed on VAL language for the output signals. So, all of them do not have a correspondent one, and to indicate this situation, the characters ***{:-(*** must be associated to them.

Point Data Type

Different from the Input data type, there is a numeric data type on VAL language, but there is no variable declaration because VAL language deals only with numeric and point data types, and all of them are global. It is easy to identify the point variables, which means that all other variables belong to the numeric data type.

Also like the numeric data type, there is a point data type on VAL language, but there is no variable declaration because VAL language deals only with numeric and point data types, and all of them are global. This situation is indicated by the absence of data on the right side of the *POINT* sentence, as can be seen below. It is easy to identify the point variables because they had to be specified in a specific file, manually, or by the use of the teach pendant. So, if a variable is specified on the points file, it is a point data type. If not, it is a numeric data type. Also, there is an absence of data for the *ID* sentence, which means that the InteRGIRo identifier, for the point data type, must be preserved.

```
POINT@
ID@
init@{:-(
:=@{:-(
=@{:-(
<@{:-(
>@{:-(
<=@{:-(
>=@{:-(
!=@{:-(
move@move %\1%
shift@appro %\1%, %\2%, %\3%, %\4%
move_c@move %\2%;move %\1%
```

The points used on the source program must be mapped to the respective physical points. The symbols file is responsible for this mapping, and this is why the *ID* sentence has no data on its right side: the InteRGIRo identifier is preserved during the data type evaluation, to be mapped to a position at the end of the InteRGIRo program evaluation.

However, as a data type, it has some basic operations associated to it, like the *move*, *shift* and *move_ c* (circular move), and they are described as can be seen on the point data type description, above.

The other sentences, like the initialization, the relational operators, and the read / write instructions, are not allowed on VAL language for the point data. So, all of them do not have a correspondent one, and to indicate this situation, the characters ***{:-(*** must be associated to them.

String Data Type

Also like the input data type, there is no string data type on VAL language, but there are, basically, one instruction associated to it, that is the one responsible for writing a string constant on the screen. So, this data type description was necessary. This situation is indicated by the absence of data on the right side of the *STRING* sentence, as can be seen below. Also, there is an absence of data for the *ID* sentence, which means that the InteRGIRo identifier, for the string data type, must be preserved.

```
STRING@
ID@
init@{:-(
+@{:-(
:=@{:-(
=@{:-(
<@{:-(
>@{:-(
<=@{:-(
>=@{:-(
!=@{:-(
"@"
write@type/N, %\1%
read@{:-(
```

Symbols

Below, it is described the symbols that were used in all of the case studies. Normally, it is described, on this file, only the symbols that are needed for a specific program.

```
PTL@@left_point
PTD@@right_point
PTC@@center_point
Ibp1@@1001
Ip1@@1002
Ip2@@1003
Ip3@@1004
Istop@@1005
Ise2@@1006
Ise1@@1007
Ise3@@1008
IS1@@1009
IS2@@1010
Ibox@@1011
Iapple@@1012
Ocancela@@1
Osinal@@2
Oest4@@3
Osc_box@@4
Osc_apple@@5
Osc1@@6
Osc2@@7
```

9.2.3 VAL Final Code

On this subsection, it is presented the VAL final code that was generated automatically by the GIRo environment, using the VAL robot language inputs that were escribed above. During this final translation process, some errors, or warnings, can occur, and they are stored on an errors file.

For this case study, the generation process was successful, despite the existence of warnings, that are written below.

```
WARNING: NUMBER data type was not defined
WARNING: POINT data type was not defined
WARNING: STRING data type was not defined
WARNING: INPUT data type was not defined
WARNING: OUTPUT data type was not defined
```

These kind of warnings can be generated in some situations. In the case of VAL language, they were generated because, in the robot language inputs files, it was indicated the existence of these five data types (NUMBER, POINT, STRING, INPUT and OUTPUT), but it was not identified how these data types are defined on VAL language. This was done because:

1. the NUMBER and POINT data types really exist on VAL language, but they are not really defined. VAL has this data types, but as the variables are globals and there is no variables declaration on VAL programs, there is no need for writing / defining these data types. They are identified implicitly by the VAL compiler;
2. the STRING InteRGIRo data type does not exist on VAL language. However, even that the VAL manual does not indicate the existence of string constants, they exist, and, also, there is a specific operation that writes string constants. So, this data type really do not exist on VAL language, but it exist an operation for writing string constants;
3. the INPUT and OUTPUT InteRGIRo data types do not exist on VAL language. However, there are digital input and output signals, and, also, there are specific operations for input and another for output signals. So, these data types formally do not exist on VAL language, but it is possible to identify them within a higher level abstraction.

As it can be saw, warnings were generated because there were strange situations, that did not really imply on errors. But the user have to be informed about that situation, and he would evaluate and decide if this warnings are important (must be treated) or not.

Above, it is presented the automatically generated VAL final code for the InteRGIRo representation presented on subsection 9.2.1.

```
10 ifsig 1002,,, then goto 50
20 ifsig 1003,,, then goto 110
30 ifsig 1004,,, then goto 150
40 goto 10
50 closei
60 goto 70
70 move left_point
80 goto 90
90 openi
100 goto 10
110 closei
120 goto 130
130 move center_point
140 goto 90
150 closei
160 goto 170
170 move right_point
180 goto 90
```

9.2.4 Robot Language Inputs for Rapid Programming Language

To generate RAPID programs, it is necessary to specify seven data types files, besides the program structure, robot language instructions, and symbols files. The needed data types files are: Boolean, Numeric, Input, Output, Point, String and Time.

Program Structure

Different from VAL, Rapid programs have headers, variable declaration, special character for the end of lines, and even special statements that indicate the end of the program. So, the needed data for this file can be seen below.

```
header@MODULE teste# %?V%t %i;# VAR clock clock1;# PROC main()
                       # ClrReset clock1;# ClkStart clock1;
end_line@;
end_prog@ENDPROC#ENDMODULE
```

The first statement contains the header of the rapid programs. All of them begin with the name of the *MODULE*. After, the variables are declared (*%?V*): first comes the data type (*%t*), followed by a space and the variable identifier (*%i*). A special variable declaration is done for all of the Rapid programs: *VAR clock clock1;*. This is done because, to use time instructions / functions, a specific time variable has to be created, that is the *clock1*, to store the time, in seconds. All of these instructions / functions manipulate the time using numeric variables. So, the time data type is really a numeric one. But this only one time variable has to be created because it can be seen as a input signal that stores the time, in seconds. So, this sentence is always defined, for all of the Rapid programs, because it is not possible, at this moment, to associate header instructions that depends on the existence of certain data types. At the end of the header, comes the beginning of the main program, followed by the instructions that resets and initializes the *clock1*.

The second statement contain the ";" character that must appear at the end of each instruction.

The last statement represents the sentences that must exist at the end of the program, that is the *ENDPROC* followed by the *ENDMODULE*.

Standard Instructions

Rapid programs use *goto* instructions to perform unconditional jumps. To allow conditional jumps, *ifv* and *ifs*instructions are associated with the *goto* ones. The target of all jumps are specified by labels that are located on the beginning of the target instructions.

Both *ifs* and *ifv* sentences have the same syntax that evaluate both signals and variables.

The standard instructions file is presented below.

```
$l@@l:
ifs@@IF %sigs
ifv@@IF %sigs
goto@@GOTO
```

User Defined Instructions

For these case studies, the only needed instructions are the ones responsible for opening and closing the robot gear. Both are mapped with two Rapid instructions: the first one sets or resets an output signal, that is responsible for controlling the gripper; and the second one to wait some time for the gripper to finish the previous operation. A third instruction, responsible for cleaning the screen, was also defined because the screen is the most used device to output informations. However, Rapid language do not have such instruction, and the special statement *{:-(* must be used to indicate its inexistence. The user defined instructions can be seen below.

```
Close@@Reset Gripper; WaitTime 0.2
Open@@Set Gripper; WaitTime 0.2
clear_screen@@{:-(
```

Boolean Data Type

The InteRGIRo *Boolean* data type corresponds to the *bool* in Rapid. The statement *VAR*, that precedes the *bool*, is needed always that a boolean variable is declared in Rapid. Also, there is an absence of data for the *ID* sentence, which means that the InteRGIRo identifier, for the boolean data type, must be preserved.

```
BOOLEAN@VAR bool
ID@
init@{:-(
~@NOT
:=@:=
=@=
!=@<>
write@TPWrite "" %\1%
read@{:-(
```

It is not necessary to initialize numeric variables at the beginning of the program, so the characters ***{:-(*** must be associated to it, to indicate this situation.

The logic operator correspond to the *NOT* Rapid operator, while the basic assignment and relational operators has the same representation on InteRGIRo and Rapid, except the not equal operator, that corresponds to the <> Rapid operator.

There is a write instruction, that is mapped to the *TPWrite "" %\1%* for the Rapid language. However, there is no read instruction for boolean data types, and it is mapped to the ***{:-(***.

Input Data Type

There is no input data type on Rapid language, but there are some operations/instructions associated to the input signals. This situation is indicated by the absence of data on the right side of the *INPUT* sentence, as can be seen below. Also, there is an absence of data for the *ID* sentence, which means that the InteRGIRo identifier, for the input data type, must be preserved.

```
INPUT@
ID@
init@{:-(
~@NOT TestDI(%\1%)
_@TestDI(%\1%)
=@=
!=@<>
<@<
>@>
<=@<=
>=@>=
write@{:-(
read@{:-(
```

The input signals used on the source program must be mapped to the respective channel number. The symbols file is responsible for this mapping, and this is why the *ID* sentence has no data on its right side: the InteRGIRo identifier is preserved during the data type evaluation, to be mapped to a input channel number at the end of the InteRGIRo program evaluation.

However, as a data type, it has some basic operations associated to it. In Rapid language, there are only two basic operations for input signals, that are the ones responsible for verifying if the input signal is on or off. The InteRGIRo operator , that means absence of the input signal, has a correspondent on Rapid, that is the *NOT TestDI* before the input signal, that is represented by the argument *%\1%*.

The other "operator" is not really an operator. When an input signal is written alone at InteRGIRo, it means that the input signal is present. However, to find a correspondent representation for this signal presence on the input data type file, it was necessary to create an "operator" to be found. This "operator" is represented, on the input data type file, as the character _ , and its correspondent, on Rapid language, is the *TestDI* before the input signal, that is represented by the argument *%\1%*.

It is possible to user relational operators with input signals, on Rapid, and they are the same that are used in InteRGIRo, except the not equal operator.

The other sentences, like the initialization, and the read / write instructions, are not allowed on Rapid language for the input signals. So, all of them do not have a correspondent one, and to indicate this situation, the characters ***{:-(*** must be associated to them.

Numeric Data Type

The InteRGIRo *NUMBER* data type corresponds to the *num* in Rapid. The statement *VAR*, that precedes the *num*, is needed always that a numeric variable is declared in Rapid. Also, there is an absence of data for the *ID* sentence, which means that the InteRGIRo identifier, for the number data type, must be preserved.

```
NUMBER@VAR num
ID@
init@{:-(
+@+
-@-
*@*
/@/
\@DIV
%@MOD
:=@:=
=@=
<@<
>@>
<=@<=
>=@>=
!=@<>
write@TPWrite "" %\1%
read@TPReadNum %\1%, ""
```

It is not necessary to initialize numeric variables at the beginning of the program, so the characters *{:-(* must be associated to it, to indicate this situation.

The basic assignment, arithmetic and relational operators has the same representation on InteRGIRo and Rapid, except: the \ (integer division), that corresponds to the *DIV* Rapid operator; the *%* (remainder), that corresponds to the *MOD* Rapid operator; and the not equal operator, that corresponds to the <> Rapid operator.

The last two instructions, the read / write instructions, are mapped to the respective ones for the Rapid language, where the *%\1%* corresponds to the numeric data type, or constant: *TPReadNum %\1%, ""*; and *TPWrite "" %\1%*.

Output Data Type

Like the input data type, there is no output data type on Rapid language, but there are some operations/instructions associated to the output signals. This situation is indicated by the absence of data on the right side of the *OUTPUT* sentence, as can be seen below. Also, there is an absence of data for the *ID* sentence, which means that the InteRGIRo identifier, for the output data type, must be preserved.

```
OUTPUT@
ID@
init@{:-(
on@Set %\1%
off@Reset %\1%
:=@{:-(
=@{:-(
!=@{:-(
<@{:-(
>@{:-(
<=@{:-(
>=@{:-(
write@{:-(
read@{:-(
```

The output signals used on the source program must be mapped to the respective channel number. The symbols file is responsible for this mapping, and this is why the *ID* sentence has no data on its right side: the InteRGIRo identifier is preserved during the data type evaluation, to be mapped to a input channel number at the end of the InteRGIRo program evaluation.

However, as a data type, it has some basic operations associated to it. In Rapid language, there are only two basic operations for output signals, that are the ones responsible for setting and resetting the output signal. The InteRGIRo instruction ***on***, that sets an output signal, is described, in Rapid, as *Set %\1%*; while the other InteRGIRo instruction ***off***, that resets an output signal, is describe, in Rapid as *Reset %\1%*.

The other sentences, like the initialization, the assignment operator, the relational operators, and the read / write instructions, are not allowed on Rapid language for the output signals. So, all of them do not have a correspondent one, and to indicate this situation, the characters ***{:-(*** must be associated to them.

Point Data Type

The InteRGIRo *POINT* data type corresponds to the *robtarget* in Rapid. The statement *PERS*, that precedes the *robtarget*, is needed always that a numeric variable is declared in Rapid. Also, there is an absence of data for the *ID* sentence, which means that the InteRGIRo identifier, for the point data type, must be preserved.

```
POINT@PERS robtarget
ID@
init@[[790.0,69.0,392.0],[0.0438284,0.384492,0.921829,0.0218526],
               [-1,0,0,0],[9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]]
:=@:=
=@{:-(
<@{:-(
>@{:-(
<=@{:-(
>=@{:-(
!=@{:-(
move@MoveL %\1%,v100,fine,tool0
shift@MoveL Offs(%\1%,%\2%,%\3%,%\4%),v100,fine,tool0
move_c@MoveC %\2%,%\1%,v100,fine,Tool0
```

The points used on the source program must be mapped to the respective physical points. The symbols file is responsible for this mapping, and this is why the *ID* sentence has no data on its right side: the InteRGIRo identifier is preserved during the data type evaluation, to be mapped to a position at the end of the InteRGIRo program evaluation.

All of the PERS variables have to be initialized when declaring point variables, and this initial value is defined on the right side of the *init* statement.

However, as a data type, it has some basic operations associated to it, like the *move*, *shift* and *move_ c* (circular move), and they are described as can be seen on the point data type description, above.

The other sentences, like the assignment operator, and the relational operators, are not allowed on Rapid language for the point data. So, all of them do not have a correspondent one, and to indicate this situation, the characters ***{:-(*** must be associated to them.

String Data Type

The InteRGIRo *STRING* data type corresponds to the *string* in Rapid. The statement *VAR*, that precedes the *string*, is needed always that a numeric variable is declared in Rapid. Also, there is an absence of data for the *ID* sentence, which means that the InteRGIRo identifier, for the number data type, must be preserved.

```
STRING@VAR string
ID@
init@{:-(
+@+
:=@:=
=@=
<@{:-(
>@{:-(
<=@{:-(
>=@{:-(
!=@<>
"@"
write@TPWrite "" %\1%
read@{:-(
```

The basic assignment, concatenation and equal operators has the same representation on InteRGIRo and Rapid, while the not equal operator corresponds to the <> Rapid operator.

There is a write instruction, that is mapped to the *TPWrite "" %\1%* for the Rapid language. However, there is no read instruction for string data types, and it is mapped to the ***{:-(***.

The other sentences, like the initialization, the relational operators (except the equal and not equal ones), are not allowed on Rapid language for the string data type. So, all of them do not have a correspondent one, and to indicate this situation, the characters ***{:-(*** must be associated to them.

Time Data Type

As it was said before, the InteRGIRo *TIME* data type corresponds to the *num* in Rapid. The statement *VAR*, that precedes the *num*, is needed always that a numeric variable is declared in Rapid.

```
TIME@VAR num
ID@
init@
set@{:-(
get_time@%\1%:=ClrRead(clock1)
+@+
-@-
*@*
/@/
\@DIV
%@MOD
:=@:=
=@=
!=@<>
<@<
>@>
<=@<=
>=@>=
```

It is not possible to set the time. However, as it was said before, in the Rapid program structure description, all generated Rapid programs reset and initialize the time variable *clock1*.

The InteRGIRo *get_time* instruction correspond to an assignment operation to a numeric variable. The time variable *clock1* is read, and its value is stored on the *%\1%* numeric variable.

The other sentences are the same that were previously explained, on the Rapid numeric data type.

Symbols

Below, it is described the symbols that were used in all of the case studies. Normally, it is described, on this file, only the symbols that are needed for a specific program.

```
PTL@@left_point
PTD@@right_point
PTC@@center_point
Ibp1@@di1
Ip1@@di2
Ip2@@di3
Ip3@@di4
Istop@@di5
Ise2@@di6
Ise1@@di7
Ise3@@di8
IS1@@di9
IS2@@di10
Ibox@@di11
Iapple@@di12
```

```
Ocancela@@do1
Osinal@@do2
Oest4@@do3
Osc_box@@do4
Osc_apple@@do5
Osc1@@do6
Osc2@@do7
```

9.2.5 Rapid Final Code

On this subsection, it is presented the RAPID final code that was generated automatically by the GIRo environment, using the RAPID robot language inputs that were described above. During this final translation process, some errors, or warnings, can occur, and they are stored on an errors file.

For this case study, the generation process was successful, despite the existence of warnings, that are written below.

```
WARNING: INPUT data type was not defined
WARNING: OUTPUT data type was not defined
```

As it was explained before, these kind of warnings can be generated in some situations. In the case of RAPID language, they were generated because, in the robot language inputs files, it was indicated the existence of these two data types (INPUT and OUTPUT), but it was not identified how these data types are defined on RAPID language.

This happened because the INPUT and OUTPUT InteRGIRo data types do not exist on RAPID language. However, there are digital input and output signals, and, also, there are specific operations for input and another for output signals. So, these data types formally do not exist on RAPID language, but it is possible to identify them within a higher level abstraction. So, there is no problem.

Above, it is presented the automatically generated RAPID final code for the InteRGIRo representation presented on subsection 9.2.1.

```
MODULE teste
     PERS robtarget PTL:=[[790.0,69.0,392.0],
        [0.0438284,0.384492,0.921829,0.0218526],
        [-1,0,0,0],[9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];
     PERS robtarget PTC:=[[790.0,69.0,392.0],
        [0.0438284,0.384492,0.921829,0.0218526],
        [-1,0,0,0],[9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];
     PERS robtarget PTD:=[[790.0,69.0,392.0],
        [0.0438284,0.384492,0.921829,0.0218526],
        [-1,0,0,0],[9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];
VAR clock clock1;
PROC main()
    ClrReset clock1;
    ClkStart clock1;
E1: IF TestDI(di2) GOTO E2;
```

```
    IF TestDI(di3) GOTO E3;
    IF TestDI(di4) GOTO E4;
    GOTO E1;
E2: Reset Gripper;
    WaitTime 0.2;
    GOTO E5;
E5: MoveL left_point,v100,fine,tool0;
    GOTO E9;
E9: Set Gripper;
    WaitTime 0.2;
    GOTO E1;
E3: Reset Gripper;
    WaitTime 0.2;
    GOTO E6;
E6: MoveL center_point,v100,fine,tool0;
    GOTO E9;
E4: Reset Gripper;
    WaitTime 0.2;
    GOTO E8;
E8: MoveL right_point,v100,fine,tool0;
    GOTO E9;
ENDPROC
ENDMODULE
```

The point variables must have initial values, even that these points would have different values, defined by the use of the teach pendant. Without this initialization, the program does not compile. As the normal robotic programs, these points are stored on the memory of the robot controller by the use of the teach pendant, and are used by the programs.

Also, all automatically generated Rapid programs will have, on the beginning of the program, two commands responsible for defining a clock variable, and for starting this clock variable. This must be done because, in case of using instructions that manipulates the time, the clock have to be started on the beginning of the program to have a valid value.

9.3 Boxing Apples Case Study

This second case study implements an application that uses a robot responsible for taking apples from one conveyor belt, leaving them on a box that is stopped on another conveyor belt. When the box is full of apples, i.e., the box contains ten apples, this box is substituted with another empty one.

A Grafcet description of such situation can be seen on figure 9.2, and the correspondent one, that was made using PaintGraf, can be seen on figure 9.3.

This case study starts opening the gripper and activating the boxes conveyor belt. When the sensor that detects the presence of a box is activated (Ibox is on), which means that there is an empty box on the correct position, the boxes conveyor belt is deactivated. Then, an apple counter is initialized with zero, and the apples conveyor belt is activated.

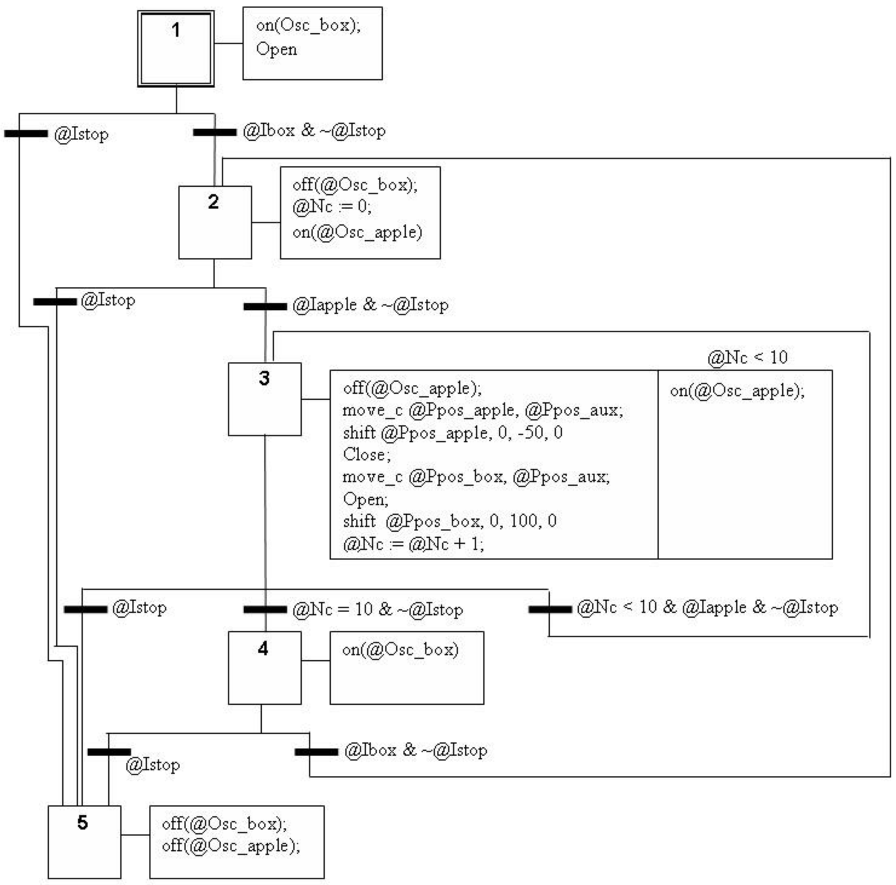
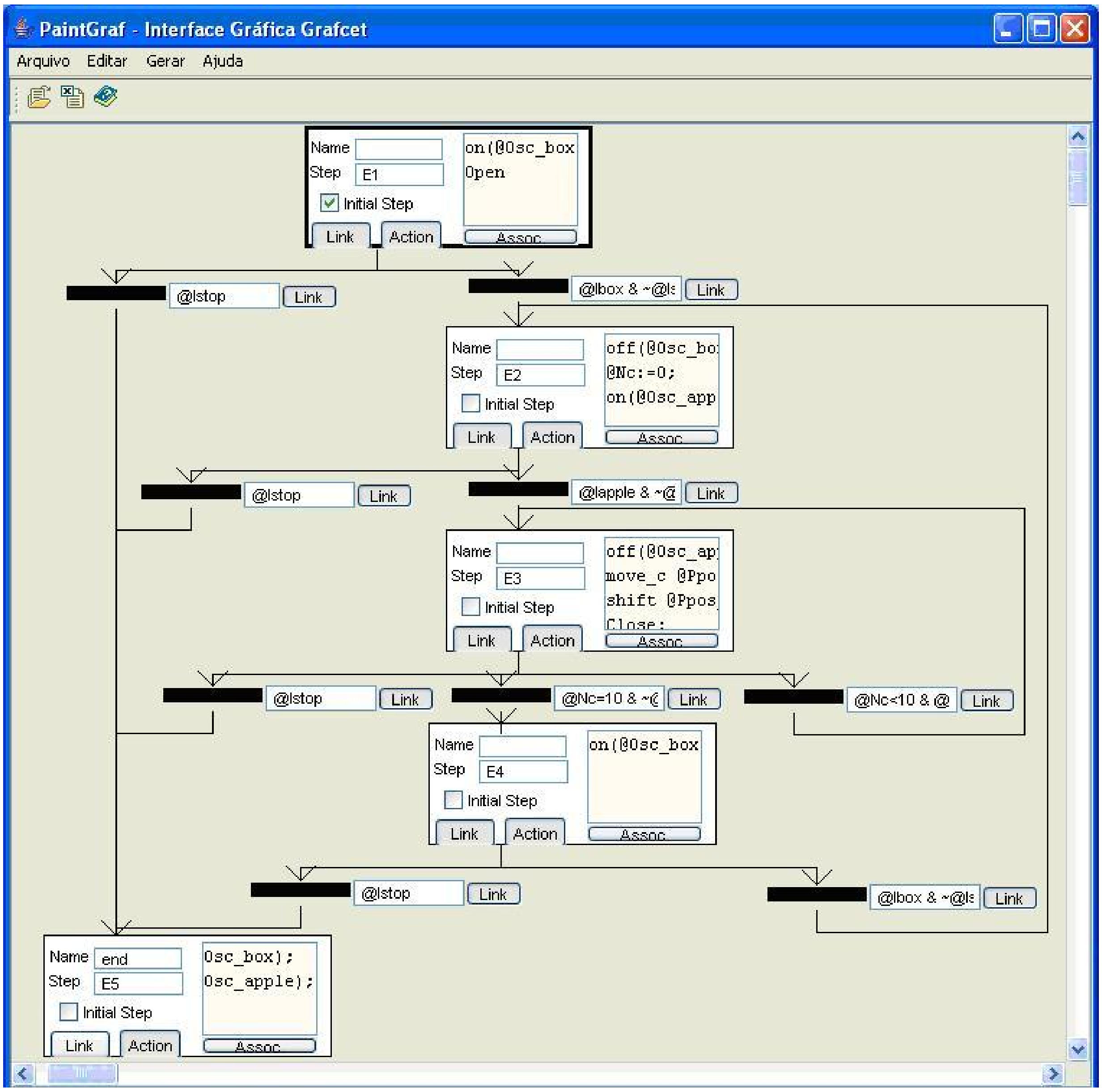


Figure 9.2: Grafcet representation of the boxing apples case study.



Name
Step E1
☑ Initial Step

```
on(@Osc_box
Open
```

@lstop

@lbox & ~@ls

Name
Step E2
☐ Initial Step

```
off(@Osc_bo:
@Nc:=0;
on(@Osc_app
```

@lstop

@lapple & ~@

Name
Step E3
☐ Initial Step

```
off(@Osc_ap;
move_c @Ppo
shift @Ppos
Close;
```

@lstop

@Nc=10 & ~@

@Nc<10 & @

Name
Step E4
☐ Initial Step

```
on(@Osc_box
```

@lstop

@lbox & ~@ls

Name end
Step E5
☐ Initial Step

```
Osc_box);
Osc_apple);
```

Figure 9.3: Grafcet diagram for boxing apples case study drawn in PaintGraf editor

Once the sensor that detects an apple is active (Iapple is on), which means that there is an apple on the correct position to be taken by the robot, the robot moves to the apple's location, in a circular trajectory, and take this apple. After, the robot still moves in a circular trajectory until the box location, where he opens the gripper, leaving the apple inside the box. The moving instruction is done in a circular way to grant that the robot will complete its movement without beating on the box, or on the conveyor belts, or in another obstacle.

After leaving the apple inside the box, the counter is incremented by one. While the counter value is smaller than ten, the apples conveyor belt is also activated, and the robot continues leaving apples on the same box. If it is equal to ten, the boxes conveyor belt is activated, substituting the full box with an empty one, continuing the same process.

This process ends only when the stop bottom is pressed (Istop is on).

9.3.1 InteRGIRo Code

The InterGIRo program generated for the Grafcet specification in figure 9.3 is listed below.

```
$E1 on(@Osc_box)
    Open
$Estado_E1 ifs (@Istop) goto $E5
    ifs (@Ibox) goto $Estado_E1_1if
    goto $Estado_E1_else2
$Estado_E1_1if ifs (~@Istop) goto $E2
$Estado_E1_else2 goto $Estado_E1
$E5 off(@Osc_box)
    off(@Osc_apple)
    goto $E_fim
$E2 off(@Osc_box)
    @N_v1:=0
    on(@Osc_apple)
$Estado_E2 ifs (@Istop) goto $E5
    ifs (@Iapple) goto $Estado_E2_1if
    goto $Estado_E2_else2
$Estado_E2_1if ifs (~@Istop) goto $E3
$Estado_E2_else2 goto $Estado_E2
$E3 off(@Osc_apple)
    move_c @Ppos_apple, @Ppos_aux
    shift @Ppos_apple,0,-50,0
    Close
    move_c @Ppos_box, @Ppos_aux
    Open
    shift @Ppos_box,0,100,0
    @N_v1:=@N_v1+1
    ifv (@N_v1<10) goto $E3_AC2
    goto $E3_AC3
$E3_AC2 on(@Osc_apple)
$E3_AC3
$Estado_E3 ifs (@Istop) goto $E5
```

```
    ifv (@N_v1=10) goto $Estado_E3_1if
    goto $Estado_E3_else2
$Estado_E3_1if ifs (~@Istop) goto $E4
$Estado_E3_else2 ifv (@N_v1<10) goto $Estado_E3_2if
    goto $Estado_E3_else3
$Estado_E3_2if ifs (@Iapple) goto $Estado_E3_3if
    goto $Estado_E3_else3
$Estado_E3_3if ifs (~@Istop) goto $E3
$Estado_E3_else3 goto $Estado_E3
$E4 on(@Osc_box)
$Estado_E4 ifs (@Istop) goto $E5
    ifs (@Ibox) goto $Estado_E4_1if
    goto $Estado_E4_else2
$Estado_E4_1if ifs (~@Istop) goto $E2
$Estado_E4_else2 goto $Estado_E4
$E_fim EndPrograma
```

9.3.2 Robot Language Inputs for VAL II Programming Language

To generate VAL II programs, it is necessary to specify five data types files, besides the program structure, robot language instructions, and symbols files. The needed data types files are: Numeric, Input, Output, Point and String. Despite String is not really an accepted data type, VAL II has instructions responsible for writing string constants. So, it was necessary to created this fifth data type file.

The explanation of such files were already done for the VAL and Rapid target languages. Only the different, important and needed explanations will be presented for these input files.

Program Structure

```
header@
end_line@
end_prog@
```

Standard Instructions

The *ifs* sentence, that has a signal on its condition, uses a VAL II function *SIG* to evaluate the used input signal.

The standard instructions file is presented below.

```
$l@@n10
ifs@@if sig(%sigs)
ifv@@if %sigs
goto@@goto
```

User Defined Instructions

```
Close@@closei
Open@@openi
clear_screen@@{:-(
```

Input Data Type

```
INPUT@
ID@
init@{:-(
~@-%\1%
_@%\1%
=@{:-(
!=@{:-(
<@{:-(
>@{:-(
<=@{:-(
>=@{:-(
write@{:-(
read@{:-(
```

Numeric Data Type

```
NUMBER@
ID@
init@{:-(
+@+
-@-
*@*
/@/
\@{:-(
%@{:-(
:=@=
=@==
!=@<>
<@<
>@>
<=@< =
>=@> =
write@type/N, %\1%
read@prompt " ", %\1%
```

Output Data Type

```
OUTPUT@
ID@
init@{:-(
on@signal %\1%
off@signal -%\1%
:=@{:-(
=@{:-(
!=@{:-(
<@{:-(
>@{:-(
<=@{:-(
>=@{:-(
write@{:-(
read@{:-(
```

Point Data Type

```
POINT@
ID@
init@{:-(
:=@{:-(
=@{:-(
<@{:-(
>@{:-(
<=@{:-(
>=@{:-(
!=@{:-(
move@move %\1%; break
shift@appro %\1%, %\2%, %\3%, %\4%
move_c@move %\2%;move %\1%
```

String Data Type

```
STRING@
ID@
init@{:-(
+@{:-(
:=@{:-(
=@{:-(
<@{:-(
>@{:-(
<=@{:-(
>=@{:-(
!=@{:-(
"@"
write@type/N, %\1%
read@{:-(
```

Symbols

```
PTL@@left_point
```

```
PTD@@right_point
PTC@@center_point
Ibp1@@1001
Ip1@@1002
Ip2@@1003
Ip3@@1004
Istop@@1005
Ise2@@1006
Ise1@@1007
Ise3@@1008
IS1@@1009
IS2@@1010
Ibox@@1011
Iapple@@1012
Ocancela@@1
Osinal@@2
Oest4@@3
Osc_box@@4
Osc_apple@@5
Osc1@@6
Osc2@@7
```

9.3.3 VAL II Final Code

On this subsection, it is presented the VAL II final code that was generated automatically by the GIRo environment, using the VAL II robot language inputs that were described above. During this final translation process, some errors, or warnings, can occur, and they are stored on an errors file.

For this case study, the generation process was also successful, despite the existence of warnings, that are written below.

```
WARNING: NUMBER data type was not defined
WARNING: POINT data type was not defined
WARNING: STRING data type was not defined
WARNING: INPUT data type was not defined
WARNING: OUTPUT data type was not defined
```

These warnings are the same that happened during the translation process into a VAL program, and were explained on subsection 9.2.3.

The generation process was successful, and the automatically generated VAL II final code for the InteRGIRo representation presented on subsection 9.3.1 can be seen following.

```
10 signal 4
20 openi
30 if sig(1005) goto 80
40 if sig(1011) goto 60
50 goto 70
60 if sig(-1005) goto 110
70 goto 30
```

```
80 signal -4
90 signal -5
100 goto 510
110 signal -4
120 N_v1=0
130 signal 5
140 if sig(1005) goto 80
150 if sig(1012) goto 170
160 goto 180
170 if sig(-1005) goto 190
180 goto 140
190 signal -5
200 move  Ppos_aux
210 move Ppos_apple
220 break
230 appro Ppos_apple, 0, -50, 0
240 closei
250 move  Ppos_aux
260 move Ppos_box
270 break
280 openi
290 appro Ppos_box, 0, 100, 0
300 N_v1=N_v1+1
310 if N_v1<10 goto 330
320 goto 340
330 signal 5
340
350 if sig(1005) goto 80
360 if N_v1==10 goto 380
370 goto 390
380 if sig(-1005) goto 450
390 if N_v1<10 goto 410
400 goto 440
410 if sig(1012) goto 430
420 goto 440
430 if sig(-1005) goto 190
440 goto 350
450 signal 4
460 if sig(1005) goto 80
470 if sig(1011) goto 490
480 goto 500
490 if sig(-1005) goto 110
500 goto 460
510
```

Like VAL language, VAL II one also do not have any instruction that does a circular trajectory moving. So, that instruction was mapped into two move sentences. It will not have a circular trajectory, but it will not beat on any conveyor belts or on the box.

9.3.4 Rapid Final Code

On this subsection, it is presented the RAPID final code that was generated automatically by the GIRo environment, using the RAPID robot language inputs, that were described in subsection 9.2. During this final translation process, some errors, or warnings, can occur, and they are stored on an errors file.

For this case study, the generation process was successful, despite the existence of warnings, that are written below.

```
WARNING: INPUT data type was not defined
WARNING: OUTPUT data type was not defined
```

These warnings are the same that happened during the translation process into a Rapid program for the simple case study, and were explained on subsection 9.2.5.

Above, it is presented the automatically generated RAPID final code for the InteRGIRo representation presented on subsection 9.3.1.

```
MODULE teste
     VAR num N_v1;
     PERS robtarget Ppos_apple:=[[790.0,69.0,392.0],
        [0.0438284,0.384492,0.921829,0.0218526],[-1,0,0,0],
        [9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];
     PERS robtarget Ppos_aux:=[[790.0,69.0,392.0],
        [0.0438284,0.384492,0.921829,0.0218526],[-1,0,0,0],
        [9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];
     PERS robtarget Ppos_box:=[[790.0,69.0,392.0],
     [0.0438284,0.384492,0.921829,0.0218526],[-1,0,0,0],
     [9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];
VAR clock clock1;
PROC main()
    ClrReset clock1;
    ClkStart clock1;
E1: Set do4;
    Set Gripper;
    WaitTime 0.2;
Estado_E1: IF TestDI(di5) GOTO E5;
    IF TestDI(di11) GOTO Estado_E1_1if;
    GOTO Estado_E1_else2;
Estado_E1_1if: IF NOT TestDI(di5) GOTO E2;
Estado_E1_else2: GOTO Estado_E1;
E5: Reset do4;
    Reset do5;
    GOTO E_fim;
E2: Reset do4;
    N_v1:=0;
    Set do5;
Estado_E2: IF TestDI(di5) GOTO E5;
    IF TestDI(di12) GOTO Estado_E2_1if;
    GOTO Estado_E2_else2;
Estado_E2_1if: IF NOT TestDI(di5) GOTO E3;
```

```
Estado_E2_else2: GOTO Estado_E2;
E3: Reset do5;
    MoveC  Ppos_aux,Ppos_apple,v100,fine,Tool0;
    MoveL Offs(Ppos_apple,0,-50,0),v100,fine,tool0;
    Reset Gripper;
    WaitTime 0.2;
    MoveC  Ppos_aux,Ppos_box,v100,fine,Tool0;
    Set Gripper;
    WaitTime 0.2;
    MoveL Offs(Ppos_box,0,100,0),v100,fine,tool0;
    N_v1:=N_v1+1;
    IF N_v1<10 GOTO E3_AC2;
    GOTO E3_AC3;
E3_AC2: Set do5;
E3_AC3: ;
Estado_E3: IF TestDI(di5) GOTO E5;
    IF N_v1=10 GOTO Estado_E3_1if;
    GOTO Estado_E3_else2;
Estado_E3_1if: IF NOT TestDI(di5) GOTO E4;
Estado_E3_else2: IF N_v1<10 GOTO Estado_E3_2if;
    GOTO Estado_E3_else3;
Estado_E3_2if: IF TestDI(di12) GOTO Estado_E3_3if;
    GOTO Estado_E3_else3;
Estado_E3_3if: IF NOT TestDI(di5) GOTO E3;
Estado_E3_else3: GOTO Estado_E3;
E4: Set do4;
Estado_E4: IF TestDI(di5) GOTO E5;
    IF TestDI(di11) GOTO Estado_E4_1if;
    GOTO Estado_E4_else2;
Estado_E4_1if: IF NOT TestDI(di5) GOTO E2;
Estado_E4_else2: GOTO Estado_E4;
E_fim: ;
ENDPROC
ENDMODULE
```

9.4 Bottles Palletization Case Study

This third case study implements an application that uses a robot responsible for taking bottles from different two conveyor belts, each one transporting one type of bottle, and putting them on the correct position of their respective boxes, that are located on one of another two conveyor belts. When a box is full, the respective conveyor belt is turned on, substituting it with another empty one.

A Grafcet description of such situation can be seen on figure 9.4, and the correspondent one, that was made using PaintGraf, can be seen on figure 9.5.

This case study use some variables, input and output signals. They are described following:

– @Is1 and @Is2 correspond to the sensors that detect the presence of a bottle on each

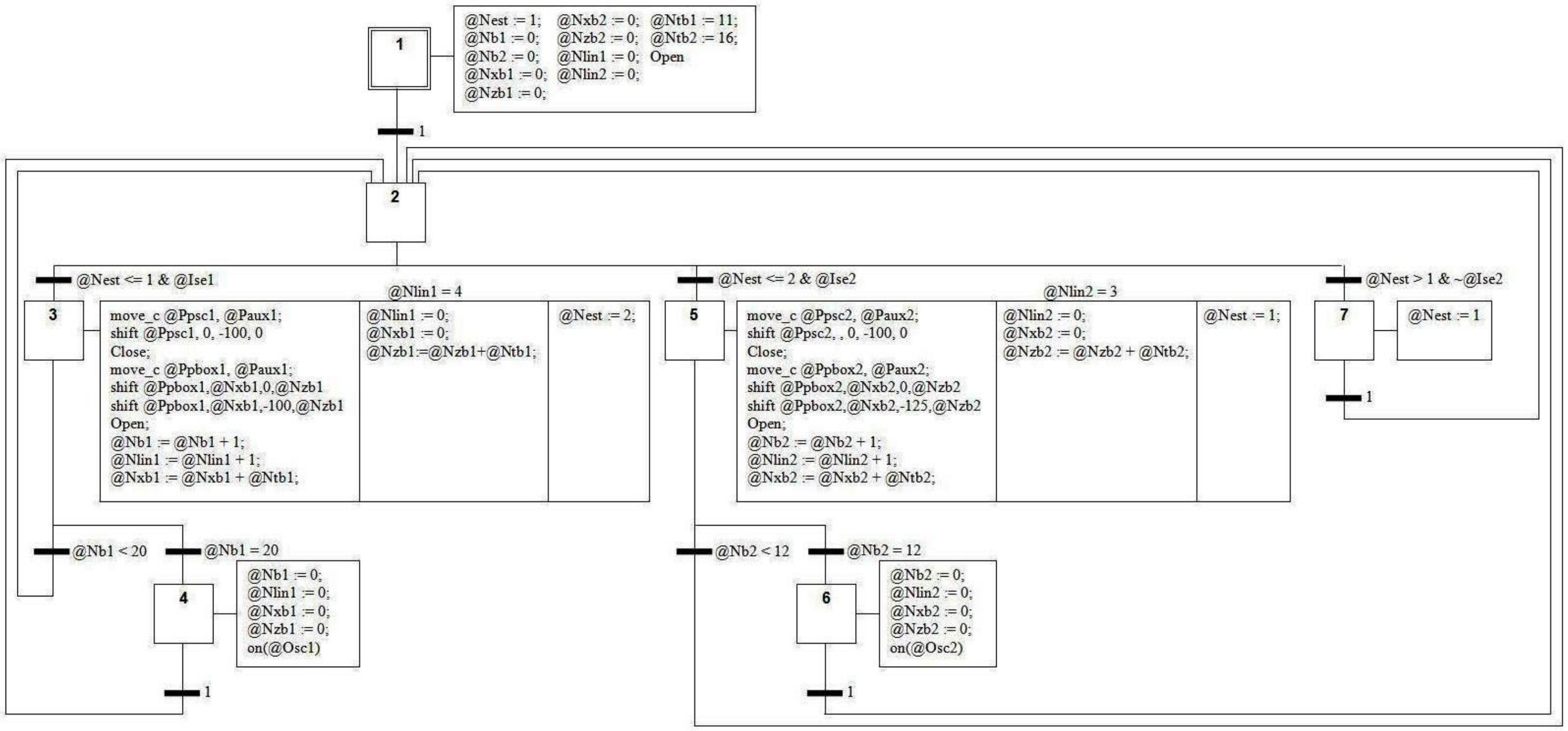
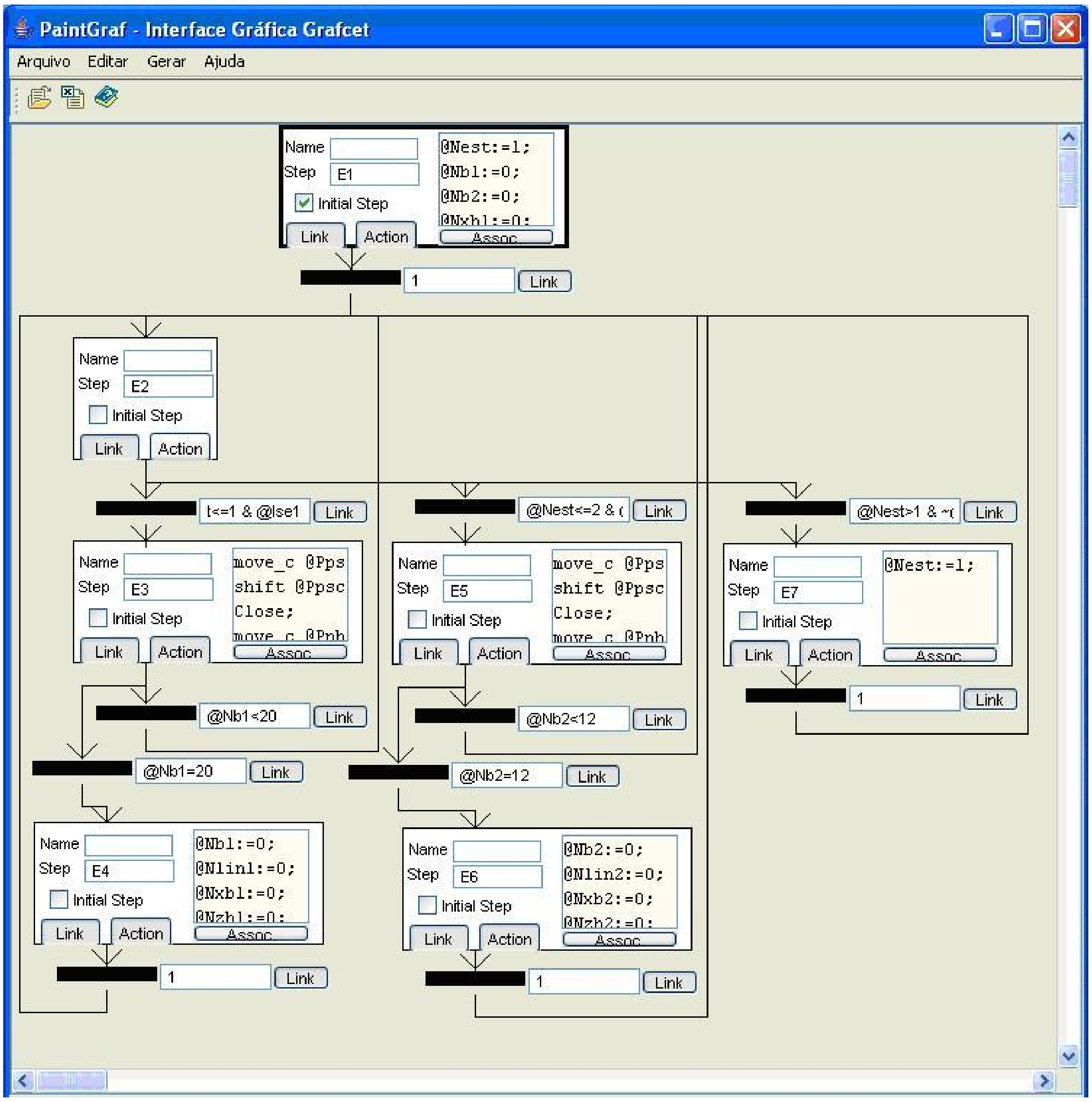


Figure 9.4: Grafcet representation of the bottles paletization case study.



Step E1 [✓ Initial Step]

```
@Nest:=1;
@Nb1:=0;
@Nb2:=0;
@Nxb1:=0;
```

Transition: 1

Step E2

Transition: t<=1 & @lse1

Step E3

```
move_c @Pps
shift @Ppsc
Close;
move_c @Ppb
```

Transition: @Nb1<20

Transition: @Nb1=20

Step E4

```
@Nb1:=0;
@Nlin1:=0;
@Nxb1:=0;
@Nzb1:=0;
```

Transition: 1

Transition: @Nest<=2 & (

Step E5

```
move_c @Pps
shift @Ppsc
Close;
move_c @Ppb
```

Transition: @Nb2<12

Transition: @Nb2=12

Step E6

```
@Nb2:=0;
@Nlin2:=0;
@Nxb2:=0;
@Nzb2:=0;
```

Transition: 1

Transition: @Nest>1 & ~(

Step E7

```
@Nest:=1;
```

Transition: 1

Figure 9.5: Grafcet diagram for bottles paletization case study drawn in PaintGraf editor

one of the two conveyor belts;
- @Osc1 and @Osc2 correspond to the output signals that control each box conveyor belt;
- @Nb1 and @Nb2 count their respective type of bottle. It was decided that the box of the type 1 can store twenty bottles, while the box of the type 2 can store twelve ones. They are initialized with zero always that the program starts, or a box is full;
- @Ntb1 and @Ntb2 contains the place size where each bottle is putted on the each box. It is a little bigger than its correspondent bottle diameter. It is assumed that @Ntc1 has a size of 30, while the @Ntc2 has a size of 50;
- @Nlin1 and @Nlin2 count the number of bottles that were putted on one of the box lines ("ou queues?"). In this case study, it is possible to put four bottles on each box line ("ou queue") of type 1, and three ones on each box line ("ou queue") of type 2. They are also initialized with zero always that the program starts, or a box line ("or queue") is full;
- @Nxb1, @Nzb1, @Nxb2 and @Nzb2 corresponds to the coordinates X and Z to where a type of bottle must be putted on the box. It was assumed that the Y coordinate corresponds to the height coordinate, and it is not needed to specify the position on the two dimensions coordinate system. Always that a bottle is putted on the box, the respective X variable is incremented by the place size of this type of bottle (@Ntb1 or @Ntb2). When a line box is full, the respective X variable is initialized and the respective Z variable is incremented by the place size of this type of bottle;
- @Nest is used to grant that each conveyor belt with bottles is treated. If the conveyor belt of type one is treated, the next one will be the type two, and vice-versa. Only in the case that a bottle is taken from one conveyor belt, and there is no bottle on the other conveyor belt, the first one can be treated again.

9.4.1 InteRGIRo Code

The InterGIRo program generated for the Grafcet specification in figure 9.5 is listed below.

```
$E1 @N_v1:=1
    @N_v2:=0
    @N_v3:=0
    @N_v4:=0
    @N_v5:=0
    @N_v6:=0
    @N_v7:=0
    @N_v8:=0
    @N_v9:=0
    @N_v10:=30
    @N_v11:=50
    Open
    goto $E2
$E2 ifv (@N_v1<=1) goto $E2_1if
    goto $E2_else1
$E2_1if ifs (@Ise1) goto $E3
$E2_else1 ifv (@N_v1<=2) goto $E2_2if
```

```
    goto $E2_else2
$E2_2if ifs (@Ise2) goto $E5
$E2_else2 ifv (@N_v1>1) goto $E2_3if
    goto $E2_else3
$E2_3if ifs (~@Ise2) goto $EE7
$E2_else3 goto $E2
$E3 move_c @Ppsc1,@Paux1
    shift @Ppsc1, 0, -100, 0
    Close
    move_c @Ppbox1,@Paux1
    shift @Ppbox1, @N_v4, 0, @N_v5
    shift @Ppbox1, @N_v4, -100, @N_v5
    Open
    @N_v2:=@N_v2+1
    @N_v8:=@N_v8+1
    @N_v4:=@N_v4+@N_v10
    ifv (@N_v8=4) goto $E3_AC2
    goto $E3_AC3
$E3_AC2 @N_v8:=0
    @N_v4:=0
    @N_v5:= @N_v5+@N_v10
$E3_AC3 @N_v1:=2
    $Estado_E3 ifv (@N_v2<20) goto $E2
    ifv (@N_v2=20) goto $E4
    goto $Estado_E3
$E4 @N_v2:=0
    @N_v8:=0
    @N_v4:=0
    @N_v5:=0
    on(@Osc1)
    goto $E2
$E5 move_c @Ppsc2,@Paux2
    shift @Ppsc2, 0, -100, 0
    Close
    move_c @Ppbox2,@Paux2
    shift @Ppbox2, @N_v6, 0, @N_v7
    shift @Ppbox2, @N_v6, -125, @N_v7
    Open
    @N_v3:=@N_v3+1
    @N_v9:=@N_v9+1
    @N_v6:=@N_v6+@N_v11
    ifv (@N_v9=3) goto $E5_AC2
    goto $E5_AC3
$E5_AC2 @N_v9:=0
    @N_v6:=0
    @N_v7:= @N_v7+@N_v11
$E5_AC3 @N_v1:=1
$Estado_E5 ifv (@N_v3=12) goto $E6
    ifv (@N_v3<12) goto $E2
    goto $Estado_E5
$E6 @N_v3:=0
```

```
    @N_v9:=0
    @N_v6:=0
    @N_v7:=0
    on(@Osc2)
    goto $E2
$EE7 @N_v1:=1
    goto $E2
```

9.4.2 Robot Language Inputs for Melfa Basic III Programming Language

To generate Melfa Basic III programs, it is necessary to specify four data types files, besides the program structure, robot language instructions, and symbols files. The needed data types files are: Numeric, Input, Output and Point.

The explanation of such files were already done for the VAL and Rapid target languages. Only the different, important and needed explanations will be presented for these input files.

Program Structure

```
header@
end_line@
end_prog@ED
```

Standard Instructions

Melfa Basic III has a particularly feature: It does not have *IF* like sentences. To deal with conditional expressions, this language uses different instructions, one for each relational operator. So, instead of having one *IF* like sentence, Melfa Basic III has many conditional sentences, as it will be seen on the each data type file. However, to indicate that Melfa Basic III does not have IF like sentences, both *ifs* and *ifv* InteRGIRo sentences are mapped to the *{:-(* statement.

The standard instructions file is presented below.

```
$l@@n10
ifs@@{:-(
ifv@@{:-(
goto@@GT
```

User Defined Instructions

```
Close@@GC
Open@@GO
clear_screen@@{:-(
```

Input Data Type

There is no input data type on Melfa Basic III language, but there are some operations/instructions associated to the input signals. This situation is indicated by the absence of data on the right side of the *INPUT* sentence, as can be seen below. Also, there is an absence of data for the *ID* sentence, which means that the InteRGIRo identifier, for the input data type, must be preserved.

```
INPUT@
ID@
init@
~@IN %\1%;EQ 0#
_@IN %\1%;NE 0#
=@IN %\1%;EQ %\2%#
!=@IN %\1%;NE %\2%#
<@IN %\1%;SM %\2%#
>@IN %\1%;LG %\2%#
<=@IN %\1%;SM %\2%#;EQ %\2%#
>=@IN %\1%;LG %\2%#;EQ %\2%#
write@{:-(
read@{:-(
```

The input signals used on the source program must be mapped to the respective digital input signal. The symbols file is responsible for this mapping, and this is why the *ID* sentence has no data on its right side: the InteRGIRo identifier is preserved during the data type evaluation, to be mapped to an input channel number at the end of the InteRGIRo program evaluation.

As it was said before, Melfa Basic III does not have *IF* like sentences. Instead of this, it has many relational instructions. Also, the relational instructions compare the second operand, that is described on the same instruction, with the one that is located at the internal register. So, before executing the relational instruction, the first operand has to be stored on such register, by the use of the *IN* instruction. When the # character is written on a data type instruction, it means that the target of a jump must be written, without the jump instruction.

The basic operations for the input data type, that is the ones that test if an input signal is active or not, are treated on the same way. The input signal is stored on the internal register, and compared to zero. If it is equal to zero, it means that the signal is inactive. If not, it is active.

The other sentences, like the initialization, and the read / write instructions, are not allowed on Melfa Basic III language for the input signals. So, all of them do not have a correspondent one, and to indicate this situation, the characters ***{:-(*** must be associated to them.

Numeric Data Type

Melfa Basic III is a very simple programming language. So simple that it does not have arithmetic operations, only the increment and decrement of one. But it is possible to create them using the GIRo environment. To do this, the user must, first, identify how to do the same with the target language. He will have to create a subprogram, using the existing instructions, that performs the same task as the desired operation, and store them in the specific data type file. As it can be seen below, some arithmetic operations are simulated using:

- the increment and decrement of 1 instructions;
- new auxiliary variables;
- new labels (starts with $);
- some rules that verifies if a data is a variable or a constant. This is necessary because Melfa Basic III that has different instructions according to the kind of data: a constant or a variable.

```
NUMBER@
ID@9
init@
+@?l is_var?  cp %\1%;?l is_var?  cl @N_a1;?l is_constant?  sc @N_a1 %\1%;
  ?r is_var?  cp %\2%;?r is_var?  cl @N_a2;?r is_constant?  sc @N_a2 %\2%;
  ?r is_constant?  cp @N_a2;sm 0 $L2;$L1 eq 0 $L3;  ic @N_a1;  dc @N_a2;
  cp @N_a2;  gt $L1;$L2 eq 0 $L3;  dc @N_a1;  ic @N_a2;  cp @N_a2;
  gt $L2;$L3;@N_a1
-@?l is_var?  cp %\1%;?l is_var?  cl @N_a1;?l is_constant?  sc @N_a1 %\1%;
  ?r is_var?  cp %\2%;?r is_var?  cl @N_a2;?r is_constant?  sc @N_a2 %\2%;?
  r is_constant?  cp @N_a2;sm 0 $L2;$L1 eq 0 $L3;  dc @N_a1;  dc @N_a2;
  cp @N_a2;  gt $L1;$L2 eq 0 $L3;  ic @N_a1;  dc @N_a2;  cp @N_a2;
  gt $L2;$L3;@N_a1
*@?l is_var?  cp %\1%;?l is_var?  cl @N_a1;?l is_constant?  sc @N_a1 %\1%;
  ?r is_constant?  cp @N_a1;  cl @N_a3;  ?r is_var?  cp %\2%;
  ?r is_var?  cl @N_a2;  ?r is_constant?  sc @N_a2 %\2%;
  ?r is_constant?  cp @N_a2;  eq 0 $L4;  $L1 eq 1 $L5;  cp @N_a3;
  cl @N_a4;  $L2 eq 0 $L3;  ic @N_a1;  dc @N_a4;  cp @N_a4;  gt $L2;
  $L3 dc @N_a2;  cp @N_a2;  gt $L1;  $L4 cp @N_a2;  cl @N_a1;  $L5; @N_a1
/@{:-(
\@?l is_var?  cp %\1%;?l is_var?  cl @N_a1;?l is_constant?  sc @N_a1 %\1%;
  ?l is_constant?  cp @N_a1;?r is_var?  cp %\2%;?r is_var?  cl @N_a2;
  ?r is_constant?  sc @N_a2 %\2%;?r is_constant?  cp @N_a2;  cl @N_a4;
  sc @N_a3 0;  cp @N_a2;  eq 0 $L3;$L1 eq 0 $L2;  dc @N_a1;  dc @N_a4;
  cp @N_a4;  gt $L1;$L2 cp @N_a1;  sm 0 $L4;  ic @N_a3;  cp @N_a1;
  eq 0 $L4;  cp @N_a2;  cl @N_a4;  gt $L1;$L3 ED;$L4;  @N_a3
%@?l is_var?  cp %\1%;?l is_var?  cl @N_a1;?l is_constant?  sc @N_a1 %\1%;
  ?l is_constant?  cp @N_a1;  cl @N_a3;  ?r is_var?  cp %\2%;?r is_var?
  cl @N_a2;  ?r is_constant?  sc @N_a2 %\2%;  ?r is_constant?  cp @N_a2;
  cl @N_a4;  eq 0 $L3;  $L1 eq 0 $L2;  dc @N_a3;  dc @N_a4;  cp @N_a4;
  gt $L1;$L2 cp @N_a3;  sm 0 $L5;  eq 0 $L4;  cl @N_a1;  cp @N_a2;
  cl @N_a4;  gt $L1;$L3 ED;$L4 sc @N_a1 0;$L5; @N_a1
:=@?r is_var?cp %\2%; ?l is_var?cl %\1%; ?r is_constant?  sc %\1% %\2%
```

```
=@?l is_var? CP %\1%; ?l is_constant? sc @N_a1 %\1%; ?l is_constant? CP @N_a1; EQ %\2%#
!=@?l is_var? CP %\1%; ?l is_constant? sc @N_a1 %\1%; ?l is_constant? CP @N_a1; NE %\2%#
<@?l is_var? CP %\1%; ?l is_constant? sc @N_a1 %\1%; ?l is_constant? CP @N_a1; SM %\2%#
>@?l is_var? CP %\1%; ?l is_constant? sc @N_a1 %\1%; ?l is_constant? CP @N_a1; LG %\2%#
<=@?l is_var? CP %\1%; ?l is_constant? sc @N_a1 %\1%; ?l is_constant? CP @N_a1;
  SM %\2%#;EQ %\2%#
>=@?l is_var? CP %\1%; ?l is_constant? sc @N_a1 %\1%; ?l is_constant? CP @N_a1;
  LG %\2%#;EQ %\2%#
write@{:-(
read@{:-(
```

The, left side, ID statement has, as correspondent, 9 in Melfa Basic III. It means that, all the numeric variables have, as identifiers, only numbers. So, the first variable is the variable 1, the second is 2, and so on.

Output Data Type

```
OUTPUT@
ID@
init@
on@OT 1 %\1%
off@OT 0 %\1%
:=@{:-(
=@{:-(
!=@{:-(
<@{:-(
>@{:-(
<=@{:-(
>=@{:-(
write@{:-(
read@{:-(
```

Point Data Type

```
POINT@
ID@
init@
:=@{:-(
=@{:-(
<@{:-(
>@{:-(
<=@{:-(
>=@{:-(
!=@{:-(
move@MO %\1%
shift@SH %\1% %\2%,%\3%,%\4%; MO %\1%; SH %\1% -%\2%,-%\3%,-%\4%
move_c@MO %\2%;MO %\1%
```

Symbols

```
PTL@@1
PTD@@3
PTC@@2
Ppos_apple@@4
Ppos_aux@@5
Ppos_box@@6
Ppsc1@@7
Paux1@@8
Ppbox1@@9
Ppsc2@@10
Paux2@@11
Ppbox2@@12
Ibp1@@4
Ip1@@1
Ip2@@2
Ip3@@3
Istop@@5
Ise2@@6
Ise1@@7
Ise3@@8
IS1@@9
IS2@@10
Ibox@@11
Iapple@@12
Ocancela@@1
Osinal@@2
Oest4@@3
Osc_box@@4
Osc_apple@@5
Osc1@@6
Osc2@@7
```

9.4.3 Melfa Basic III Final Code

On this subsection, it is presented the Melfa Basic III final code that was generated automatically by the GIRo environment, using the Melfa Basic III robot language inputs, that were described above. During this final translation process, some errors, or warnings, can occur, and they are stored on an errors file.

For this case study, the generation process was successful, despite the existence of warnings, that are written below.

```
WARNING: NUMBER data type was not defined
WARNING: POINT data type was not defined
WARNING: INPUT data type was not defined
WARNING: OUTPUT data type was not defined
WARNING: Target language do not have IF statements.
         You must verify if it has instructions that evaluate
```

```
        the relational operators.
```

These warnings are the same that happened during the translation process into a Melfa Basic III program for the simple case study A.2, and were explained on that subsection.

The generation process was successful, and the automatically generated Melfa Basic III final code for the InteRGIRo representation presented on subsection 9.4.1 can be seen following.

```
10 cp 1
20  cl 1
30 cp 0
40  cl 2
50 cp 0
60  cl 3
70 cp 0
80  cl 4
90 cp 0
100  cl 5
110 cp 0
120  cl 6
130 cp 0
140  cl 7
150 cp 0
160  cl 8
170 cp 0
180  cl 9
190 cp 30
200  cl 10
210 cp 50
220  cl 11
230 GO
240 GT 250
250 CP 1
260 SM 1 290
270 EQ 1 290
280 GT 310
290 IN 7
300 NE 0 430
310 CP 1
320 SM 2 350
330 EQ 2 350
340 GT 370
350 IN 6
360 NE 0 1540
370 CP 1
380 LG 1 400
390 GT 420
400 IN 6
410 EQ 0 2650
420 GT 250
430 MO 8
```

```
440 MO 7
450 SH 7  0, -100, 0
460 MO 7
470 SH 7 - 0,- -100,- 0
480 GC
490 MO 8
500 MO 9
510 SH 9  4, 0, 5
520 MO 9
530 SH 9 - 4,- 0,- 5
540 SH 9  4, -100, 5
550 MO 9
560 SH 9 - 4,- -100,- 5
570 GO
580 cp 2
590 cl 12
600 sc 13 1
610 cp 13
620 sm 0 680
630  eq 0 730
640 ic 12
650 dc 13
660 cp 13
670 gt 630
680  eq 0 730
690 dc 12
700 ic 13
710 cp 13
720 gt 680
730
740 cp 12
750  cl 2
760 cp 8
770 cl 12
780 sc 13 1
790 cp 13
800 sm 0 860
810  eq 0 910
820 ic 12
830 dc 13
840 cp 13
850 gt 810
860  eq 0 910
870 dc 12
880 ic 13
890 cp 13
900 gt 860
910
920 cp 12
930  cl 8
940 cp 4
```

```
950 cl 12
960 cp 10
970 cl 13
980 sm 0 1040
990  eq 0 1090
1000 ic 12
1010 dc 13
1020 cp 13
1030 gt 990
1040  eq 0 1090
1050 dc 12
1060 ic 13
1070 cp 13
1080 gt 1040
1090
1100 cp 12
1110  cl 4
1120 CP 8
1130 EQ 4 1150
1140 GT 1370
1150 cp 0
1160  cl 8
1170 cp 0
1180  cl 4
1190 cp 5
1200 cl 12
1210 cp 10
1220 cl 13
1230 sm 0 1290
1240  eq 0 1340
1250 ic 12
1260 dc 13
1270 cp 13
1280 gt 1240
1290  eq 0 1340
1300 dc 12
1310 ic 13
1320 cp 13
1330 gt 1290
1340
1350 cp 12
1360  cl 5
1370 cp 2
1380  cl 1
1390 CP 2
1400 SM 20 250
1410 CP 2
1420 EQ 20 1440
1430 GT 1390
1440 cp 0
1450  cl 2
```

```
1460 cp 0
1470  cl 8
1480 cp 0
1490  cl 4
1500 cp 0
1510  cl 5
1520 OT 1 6
1530 GT 250
1540 MO 11
1550 MO 10
1560 SH 10  0, -100, 0
1570 MO 10
1580 SH 10 - 0,- -100,- 0
1590 GC
1600 MO 11
1610 MO 12
1620 SH 12  6, 0, 7
1630 MO 12
1640 SH 12 - 6,- 0,- 7
1650 SH 12  6, -125, 7
1660 MO 12
1670 SH 12 - 6,- -125,- 7
1680 GO
1690 cp 3
1700 cl 12
1710 sc 13 1
1720 cp 13
1730 sm 0 1790
1740  eq 0 1840
1750 ic 12
1760 dc 13
1770 cp 13
1780 gt 1740
1790  eq 0 1840
1800 dc 12
1810 ic 13
1820 cp 13
1830 gt 1790
1840
1850 cp 12
1860  cl 3
1870 cp 9
1880 cl 12
1890 sc 13 1
1900 cp 13
1910 sm 0 1970
1920  eq 0 2020
1930 ic 12
1940 dc 13
1950 cp 13
1960 gt 1920
```

```
1970  eq 0 2020
1980 dc 12
1990 ic 13
2000 cp 13
2010 gt 1970
2020
2030 cp 12
2040  cl 9
2050 cp 6
2060 cl 12
2070 cp 11
2080 cl 13
2090 sm 0 2150
2100  eq 0 2200
2110 ic 12
2120 dc 13
2130 cp 13
2140 gt 2100
2150  eq 0 2200
2160 dc 12
2170 ic 13
2180 cp 13
2190 gt 2150
2200
2210 cp 12
2220  cl 6
2230 CP 9
2240 EQ 3 2260
2250 GT 2480
2260 cp 0
2270  cl 9
2280 cp 0
2290  cl 6
2300 cp 7
2310 cl 12
2320 cp 11
2330 cl 13
2340 sm 0 2400
2350  eq 0 2450
2360 ic 12
2370 dc 13
2380 cp 13
2390 gt 2350
2400  eq 0 2450
2410 dc 12
2420 ic 13
2430 cp 13
2440 gt 2400
2450
2460 cp 12
2470  cl 7
```

```
2480 cp 1
2490  cl 1
2500 CP 3
2510 EQ 12 2550
2520 CP 3
2530 SM 12 250
2540 GT 2500
2550 cp 0
2560  cl 3
2570 cp 0
2580  cl 9
2590 cp 0
2600  cl 6
2610 cp 0
2620  cl 7
2630 OT 1 7
2640 GT 250
2650 cp 1
2660  cl 1
2670 GT 250
2680 ED
```

As it was explained before, Melfa Basic III do not have arithmetic operations. Only increment and decrement of one. So, the sum operations, and there are a lot of them on this InteRGIRo code, were mapped to a set of Melfa Basic III instructions, that can perform sum operations using increment, decrement and relational instructions. This is why the final code is bigger enough than the other ones, but it achieves the expected goal.

9.4.4 Rapid Final Code

On this subsection, it is presented the RAPID final code that was generated automatically by the GIRo environment, using the RAPID robot language inputs, that were described in subsection 9.2. During this final translation process, some errors, or warnings, can occur, and they are stored on an errors file.

For this case study, the generation process was successful, despite the existence of warnings, that are written below.

```
WARNING: INPUT data type was not defined
WARNING: OUTPUT data type was not defined
```

These warnings are the same that happened during the translation process into a Rapid program for the simple case study, and were explained on subsection 9.2.5.

Above, it is presented the automatically generated RAPID final code for the InteRGIRo representation presented on subsection 9.4.1.

```
MODULE teste
     VAR num N_v1;
     VAR num N_v2;
     VAR num N_v3;
     VAR num N_v4;
     VAR num N_v5;
     VAR num N_v6;
     VAR num N_v7;
     VAR num N_v8;
     VAR num N_v9;
     VAR num N_v10;
     VAR num N_v11;
     PERS robtarget Ppsc1:=[[790.0,69.0,392.0],
        [0.0438284,0.384492,0.921829,0.0218526],[-1,0,0,0],
        [9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];
     PERS robtarget Paux1:=[[790.0,69.0,392.0],
        [0.0438284,0.384492,0.921829,0.0218526],[-1,0,0,0],
        [9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];
     PERS robtarget Ppbox1:=[[790.0,69.0,392.0],
        [0.0438284,0.384492,0.921829,0.0218526],[-1,0,0,0],
        [9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];
     PERS robtarget Ppsc2:=[[790.0,69.0,392.0],
        [0.0438284,0.384492,0.921829,0.0218526],[-1,0,0,0],
        [9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];
     PERS robtarget Paux2:=[[790.0,69.0,392.0],
        [0.0438284,0.384492,0.921829,0.0218526],[-1,0,0,0],
        [9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];
     PERS robtarget Ppbox2:=[[790.0,69.0,392.0],
        [0.0438284,0.384492,0.921829,0.0218526],[-1,0,0,0],
        [9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];
VAR clock clock1;
PROC main()
    ClrReset clock1;
    ClkStart clock1;
E1: N_v1:=1;
    N_v2:=0;
    N_v3:=0;
    N_v4:=0;
    N_v5:=0;
    N_v6:=0;
    N_v7:=0;
    N_v8:=0;
    N_v9:=0;
    N_v10:=30;
    N_v11:=50;
    Set Gripper;
    WaitTime 0.2;
    GOTO E2;
E2: IF N_v1<=1 GOTO E2_1if;
    GOTO E2_else1;
E2_1if: IF TestDI(di7) GOTO E3;
E2_else1: IF N_v1<=2 GOTO E2_2if;
    GOTO E2_else2;
E2_2if: IF TestDI(di6) GOTO E5;
```

```
E2_else2: IF N_v1>1 GOTO E2_3if;
    GOTO E2_else3;
E2_3if: IF NOT TestDI(di6) GOTO EE7;
E2_else3: GOTO E2;
E3: MoveC Paux1,Ppsc1,v100,fine,Tool0;
    MoveL Offs(Ppsc1, 0, -100, 0),v100,fine,tool0;
    Reset Gripper;
    WaitTime 0.2;
    MoveC Paux1,Ppbox1,v100,fine,Tool0;
    MoveL Offs(Ppbox1, N_v4, 0, N_v5),v100,fine,tool0;
    MoveL Offs(Ppbox1, N_v4, -100, N_v5),v100,fine,tool0;
    Set Gripper;
    WaitTime 0.2;
    N_v2:=N_v2+1;
    N_v8:=N_v8+1;
    N_v4:=N_v4+N_v10;
    IF N_v8=4 GOTO E3_AC2;
    GOTO E3_AC3;
E3_AC2: N_v8:=0;
    N_v4:=0;
    N_v5:=N_v5+N_v10;
E3_AC3: N_v1:=2;
Estado_E3: IF N_v2<20 GOTO E2;
    IF N_v2=20 GOTO E4;
    GOTO Estado_E3;
E4: N_v2:=0;
    N_v8:=0;
    N_v4:=0;
    N_v5:=0;
    Set do6;
    GOTO E2;
E5: MoveC Paux2,Ppsc2,v100,fine,Tool0;
    MoveL Offs(Ppsc2, 0, -100, 0),v100,fine,tool0;
    Reset Gripper;
    WaitTime 0.2;
    MoveC Paux2,Ppbox2,v100,fine,Tool0;
    MoveL Offs(Ppbox2, N_v6, 0, N_v7),v100,fine,tool0;
    MoveL Offs(Ppbox2, N_v6, -125, N_v7),v100,fine,tool0;
    Set Gripper;
    WaitTime 0.2;
    N_v3:=N_v3+1;
    N_v9:=N_v9+1;
    N_v6:=N_v6+N_v11;
    IF N_v9=3 GOTO E5_AC2;
    GOTO E5_AC3;
E5_AC2: N_v9:=0;
    N_v6:=0;
    N_v7:=N_v7+N_v11;
E5_AC3: N_v1:=1;
Estado_E5: IF N_v3=12 GOTO E6;
    IF N_v3<12 GOTO E2;
```

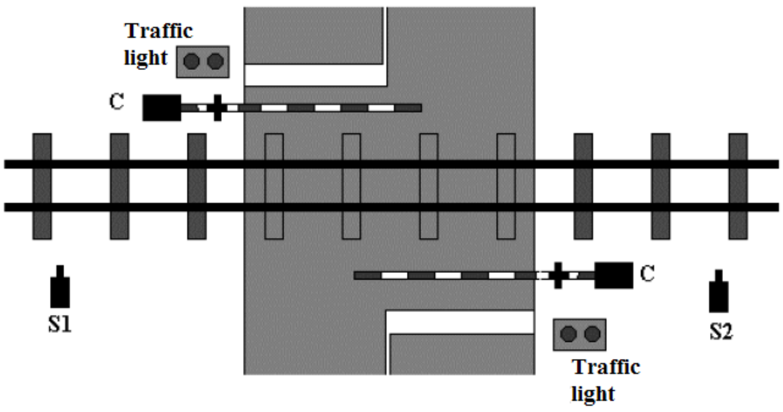


Figure 9.6: "Passagem de Nível"

```
    GOTO Estado_E5;
E6: N_v3:=0;
    N_v9:=0;
    N_v6:=0;
    N_v7:=0;
    Set do7;
    GOTO E2;
EE7: N_v1:=1;
    GOTO E2;
ENDPROC
ENDMODULE
```

9.5 A Level Rail Crossing Case Study

This fourth case study is not a industrial robot application. It is an automation application that can be implemented by the use of a PLC. The aim of this case study is to prove that the Grafcet timing functions are treated correctly, since the source code creation, by the user, until the target automatically generated code.

This application is responsible for using a industrial robot controller to control the car traffic while a train is passing on the railway. This controlling must be done to avoid collisions, because the railway pass through the road, as can be seen on figure 9.8.

When a train is passing, from right to left way, the sensor S1 detects its presence, and two actions are taken: the road is closed, because the boom barrier C is turned down; and the traffic lights indicate that the car traffic is forbidden for a while.

This situation is maintained during four seconds, because this is the expected time (for this case study) for the train to be detected by the sensor S2. After this detection, this situation is still maintained during two seconds, before evaluating again the S2 sensor. When the S2 sensor is inactive, it means that the train has passed, and the boom barrier can be turned up, and the traffic light can be turned off.

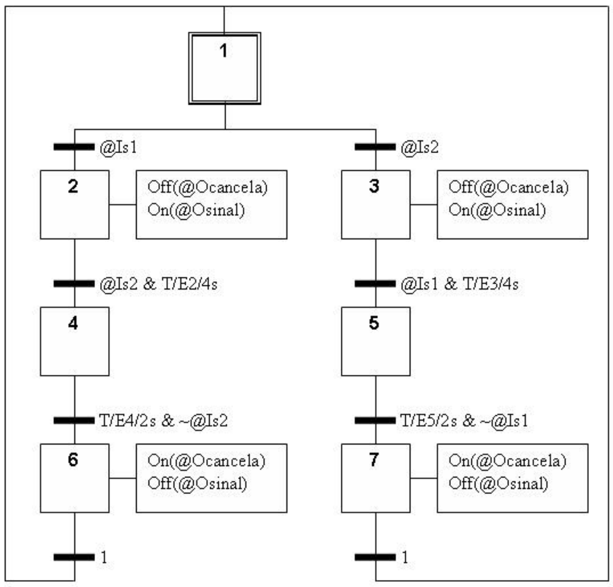


Figure 9.7: Grafcet representation of the level rail crossing case study.

If the train pass in the opposite direction, the behavior of the system is the same, only changing the sensors order.

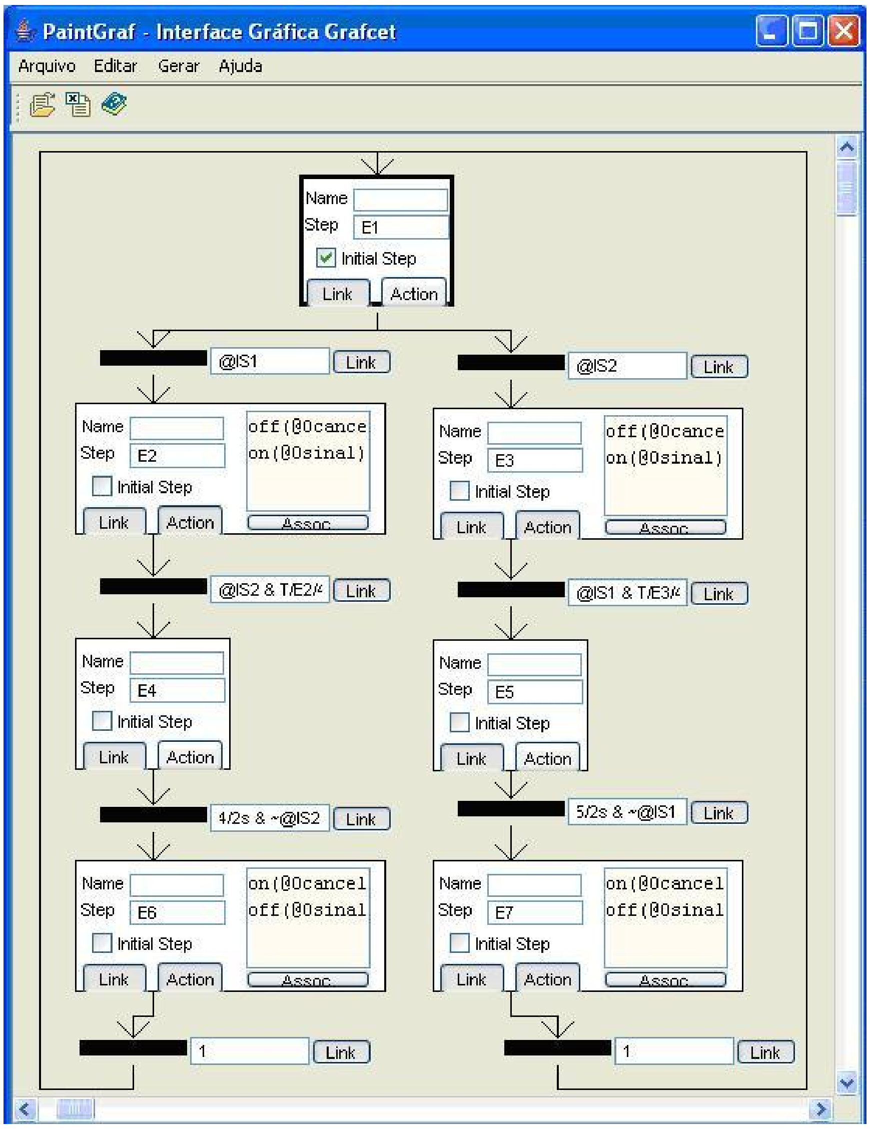
A Grafcet description of such situation can be seen on figure 9.7, and the correspondent one, that was made using PaintGraf, can be seen on figure 9.8.

As an output signal can only be turned on or off, the command responsible for turning down or up the boom barrier correspond, respectively, to turning off or on the output signal that sends this command for the boom barrier.

9.5.1 InteRGIRo Code

The InterGIRo program generated for the Grafcet specification in figure 9.8 is listed below.

```
$E1 ifs (@IS1) goto $E2
    ifs (@IS2) goto $E3
    goto $E1
$E2 get_time(@T_v1)
    off(@Ocancela)
    on(@Osinal)
$Estado_E2 ifs (@IS2) goto $Estado_E2_1if
    goto $Estado_E2_else1
$Estado_E2_1if get_time(@T_v2)
    @T_v3:=@T_v2-@T_v1
    ifv (@T_v3>=4) goto $E4
$Estado_E2_else1 goto $Estado_E2
$E4 get_time(@T_v1)
$E4T get_time(@T_v2)
```


PaintGraf - Interface Gráfica Grafcet
Arquivo Editar Gerar Ajuda
Link
Action
Assoc

Name
Step E1
[✓ Initial Step]

@IS1

Name
Step E2
Initial Step
off(@0cance
on(@0sinal)

@IS2 & T/E2/4

Name
Step E4
Initial Step

4/2s & ~@IS2

Name
Step E6
Initial Step
on(@0cancel
off(@0sinal

1

@IS2

Name
Step E3
Initial Step
off(@0cance
on(@0sinal)

@IS1 & T/E3/4

Name
Step E5
Initial Step

5/2s & ~@IS1

Name
Step E7
Initial Step
on(@0cancel
off(@0sinal

1

Figure 9.8: Grafcet diagram for level rail crossing case study drawn in PaintGraf editor

```
    @T_v3:=@T_v2-@T_v1
    ifv (@T_v3>=2) goto $E4_1if
    goto $E4_else1
$E4_1if ifs (~@IS2) goto $E6
$E4_else1 goto $E4T
$E6 on(@Ocancela)
    off(@Osinal)
    goto $E1
$E3 get_time(@T_v1)
    off(@Ocancela)
    on(@Osinal)
$Estado_E3 ifs (@IS1) goto $Estado_E3_1if
    goto $Estado_E3_else1
$Estado_E3_1if get_time(@T_v2)
    @T_v3:=@T_v2-@T_v1
    ifv (@T_v3>=4) goto $E5
$Estado_E3_else1 goto $Estado_E3
$E5 get_time(@T_v1)
$E5T get_time(@T_v2)
    @T_v3:=@T_v2-@T_v1
    ifv (@T_v3>=2) goto $E5_1if
    goto $E5_else1
$E5_1if ifs (~@IS1) goto $E7
$E5_else1 goto $E5T
$E7 on(@Ocancela)
    off(@Osinal)
    goto $E1
```

9.5.2 Robot Language Inputs for Luna Programming Language

To generate Luna programs, it is necessary to specify five data types files, besides the program structure, robot language instructions, and symbols files. The needed data types files are: Numeric, Input, Output, Point and Time.

The explanation of such files were already done for the VAL and Rapid target languages. Only the different, important and needed explanations will be presented for these input files.

Program Structure

```
header@%?V%t : %i
end_line@
end_prog@END
```

Standard Instructions

```
$l@@n
ifs@@IF %sigs
```

```
ifv@@IF %sigs
goto@@GO
```

User Defined Instructions

```
Close@@off(@Oop_gear)
Open@@on(@Oop_gear)
clear_screen@@{:-(
```

Input Data Type

```
INPUT@
ID@
init@{:-(
~@%\1%(OFF)
_@%\1%(ON)
=@{:-(
!=@{:-(
<@{:-(
>@{:-(
<=@{:-(
>=@{:-(
write@{:-(
read@{:-(
```

Numeric Data Type

```
NUMBER@INT
ID@
init@
+@+
-@-
*@*
/@/
\@{:-(
%@{:-(
:=@=
=@=
!=@<>
<@<
>@>
<=@<=
>=@>=
write@WRITE(%\all,%)
read@READ(%\all,%)
```

Output Data Type

```
OUTPUT@
ID@
init@{:-(
on@DO %\1%(OFF)
off@DO %\1%(ON)
:=@{:-(
=@{:-(
!=@{:-(
<@{:-(
>@{:-(
<=@{:-(
>=@{:-(
write@{:-(
read@{:-(
```

Point Data Type

```
POINT@POINT
ID@
init@{:-(
:=@{:-(
=@{:-(
<@{:-(
>@{:-(
<=@{:-(
>=@{:-(
!=@{:-(
move@DO LINE:%\1%
shift@DO LINE:%\1%; SHIFT %\2%,%\3%,%\4%,0
move_c@DO LINE:%\2%; DO LINE:%\1%
```

Time Data Type

```
TIME@
ID@
init@{:-(
set@TIME:=%\1%
get_time@%\1%:=TIME
+@+
-@-
*@*
/@/
\@{:-(
%@{:-(
:=@=
=@=
!=@<>
```

```
<@<
>@>
<=@<=
>=@>=
```

Symbols

```
PTL@@left_point
PTD@@right_point
PTC@@center_point
Ibp1@@I1
Ip1@@I2
Ip2@@I3
Ip3@@I4
Istop@@I5
Ise2@@I6
Ise1@@I7
Ise3@@I8
IS1@@I9
IS2@@I10
Ibox@@I11
Iapple@@I12
Ocancela@@L1
Osinal@@L2
Oest4@@L3
Osc_box@@L4
Osc_apple@@L5
 Osc1@@L6
 Osc2@@L7
 Oop_gear@@L8
```

9.5.3 Luna Final Code

On this subsection, it is presented the Luna final code that was generated automatically by the GIRo environment, using the Luna robot language inputs that were better described above. During this final translation process, some errors, or warnings, can occur, and they are stored on an errors file.

For this case study, the generation process was successful, despite the existence of warnings, that are written below.

```
WARNING: TIME data type was not defined
WARNING: INPUT data type was not defined
WARNING: OUTPUT data type was not defined
```

These warnings are the same that happened during the translation process into a Luna program for the simple case study, and were also explained on the respective subsection (A.3).

The generation process was successful, and the automatically generated Luna final code for the InteRGIRo representation presented on subsection 9.5.1 can be seen following.

```
10 IF I9(ON) GO 40
20 IF I10(ON) GO 230
30 GO 10
40 T_v1=TIME
50 DO L1(ON)
60 DO L2(OFF)
70 IF I10(ON) GO 90
80 GO 120
90 T_v2=TIME
100 T_v3=T_v2-T_v1
110 IF T_v3>=4 GO 130
120 GO 70
130 T_v1=TIME
140 T_v2=TIME
150 T_v3=T_v2-T_v1
160 IF T_v3>=2 GO 180
170 GO 190
180 IF I10(OFF) GO 200
190 GO 140
200 DO L1(OFF)
210 DO L2(ON)
220 GO 10
230 T_v1=TIME
240 DO L1(ON)
250 DO L2(OFF)
260 IF I9(ON) GO 280
270 GO 310
280 T_v2=TIME
290 T_v3=T_v2-T_v1
300 IF T_v3>=4 GO 320
310 GO 260
320 T_v1=TIME
330 T_v2=TIME
340 T_v3=T_v2-T_v1
350 IF T_v3>=2 GO 370
360 GO 380
370 IF I9(OFF) GO 390
380 GO 330
390 DO L1(OFF)
400 DO L2(ON)
410 GO 10
420 END
```

For Luna programming language, TIME is a real variable whose value progresses by 0,01 every 10 msec. It is a reserved word and need not to be described and entered. It is possible to perform assignment or arithmetic operations just as it does with real variables.

9.5.4 Rapid Final Code

On this subsection, it is presented the RAPID final code that was generated automatically by the GIRo environment, using the RAPID robot language inputs, that were described in subsection 9.2. During this final translation process, some errors, or warnings, can occur, and they are stored on an errors file.

For this case study, the generation process was successful, despite the existence of warnings, that are written below.

```
WARNING: INPUT data type was not defined
WARNING: OUTPUT data type was not defined
```

These warnings are the same that happened during the translation process into a Rapid program for the simple case study, and were explained on subsection 9.2.5.

Above, it is presented the automatically generated RAPID final code for the InteRGIRo representation presented on subsection 9.5.1.

```
MODULE teste
     VAR num T_v1;
     VAR num T_v2;
     VAR num T_v3;
VAR clock clock1;
PROC main()
    ClrReset clock1;
    ClkStart clock1;
E1: IF TestDI(di9) GOTO E2;
    IF TestDI(di10) GOTO E3;
    GOTO E1;
E2: T_v1:=ClrRead(clock1);
    Reset do1;
    Set do2;
Estado_E2: IF TestDI(di10) GOTO Estado_E2_1if;
    GOTO Estado_E2_else1;
Estado_E2_1if: T_v2:=ClrRead(clock1);
    T_v3:=T_v2-T_v1;
    IF T_v3>=4 GOTO E4;
Estado_E2_else1: GOTO Estado_E2;
E4: T_v1:=ClrRead(clock1);
E4T: T_v2:=ClrRead(clock1);
    T_v3:=T_v2-T_v1;
    IF T_v3>=2 GOTO E4_1if;
    GOTO E4_else1;
E4_1if: IF NOT TestDI(di10) GOTO E6;
E4_else1: GOTO E4T;
E6: Set do1;
    Reset do2;
    GOTO E1;
E3: T_v1:=ClrRead(clock1);
    Reset do1;
    Set do2;
```

```
Estado_E3: IF TestDI(di9) GOTO Estado_E3_1if;
    GOTO Estado_E3_else1;
Estado_E3_1if: T_v2:=ClrRead(clock1);
    T_v3:=T_v2-T_v1;
    IF T_v3>=4 GOTO E5;
Estado_E3_else1: GOTO Estado_E3;
E5: T_v1:=ClrRead(clock1);
E5T: T_v2:=ClrRead(clock1);
    T_v3:=T_v2-T_v1;
    IF T_v3>=2 GOTO E5_1if;
    GOTO E5_else1;
E5_1if: IF NOT TestDI(di9) GOTO E7;
E5_else1: GOTO E5T;
E7: Set do1;
    Reset do2;
    GOTO E1;
ENDPROC
ENDMODULE
```

9.6 A Factorial Case Study

This fifth case study is a simple program that calculates a factorial of a given number. This is not a robotic application, but it was done to show that a generic high-level programming language, like the commonly used Pascal or C, can be used as the final code to be generated by GIRo's system, and because this is a typical procedural program that can be easily tested on any personal computer. Despite that it is not a typical robotic application, some industrial robots can run the program, that was automatically generated in this case study.

A Grafcet description of such situation can be seen on figure 9.9, and the correspondent one, that was made using PaintGraf, can be seen on figure 9.10.

9.6.1 InteRGIRo Code

The InterGIRo program generated for the Grafcet specification in figure 9.10 is listed below.

```
$E1 @N_v1:=0
    @N_v2:=1
    clear_screen
    write("Insert a positive number to see its factorial.")
    read(@N_v1)
$Estado_E1 ifv (@N_v1<0) goto $E2
    ifv (@N_v1>0) goto $E3
    ifv (@N_v1=0) goto $E4
    goto $Estado_E1
$E2 write("There is no factorial for a negative number.")
    goto $E_fim
$E3 @N_v2:= @N_v2*@N_v1
    @N_v1:= @N_v1-1
```

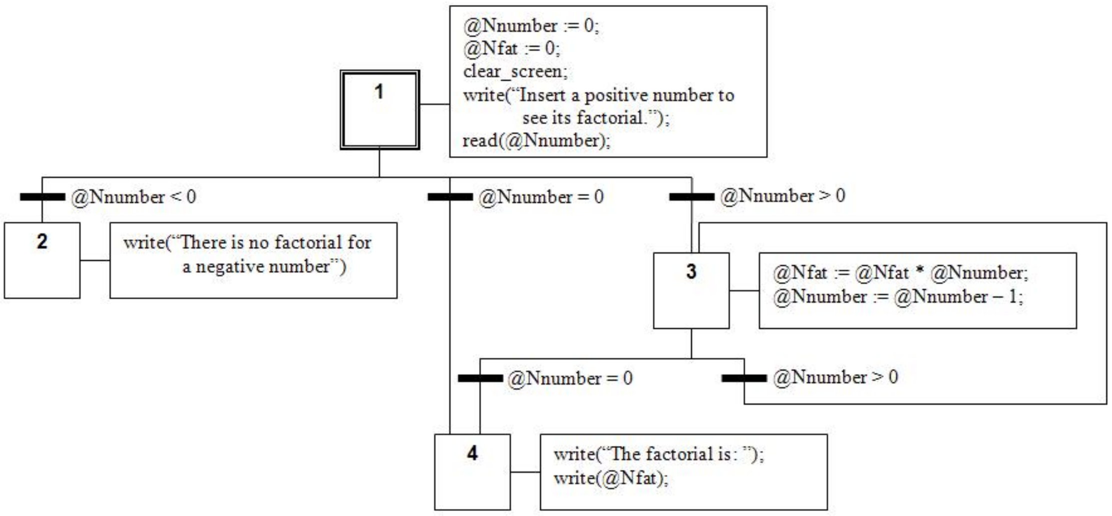

1
@Nnumber := 0;
@Nfat := 0;
clear_screen;
write("Insert a positive number to see its factorial.");
read(@Nnumber);
@Nnumber < 0
@Nnumber = 0
@Nnumber > 0
2
write("There is no factorial for a negative number")
3
@Nfat := @Nfat * @Nnumber;
@Nnumber := @Nnumber – 1;
@Nnumber = 0
@Nnumber > 0
4
write("The factorial is: ");
write(@Nfat);

Figure 9.9: Grafcet representation of the factorial case study.

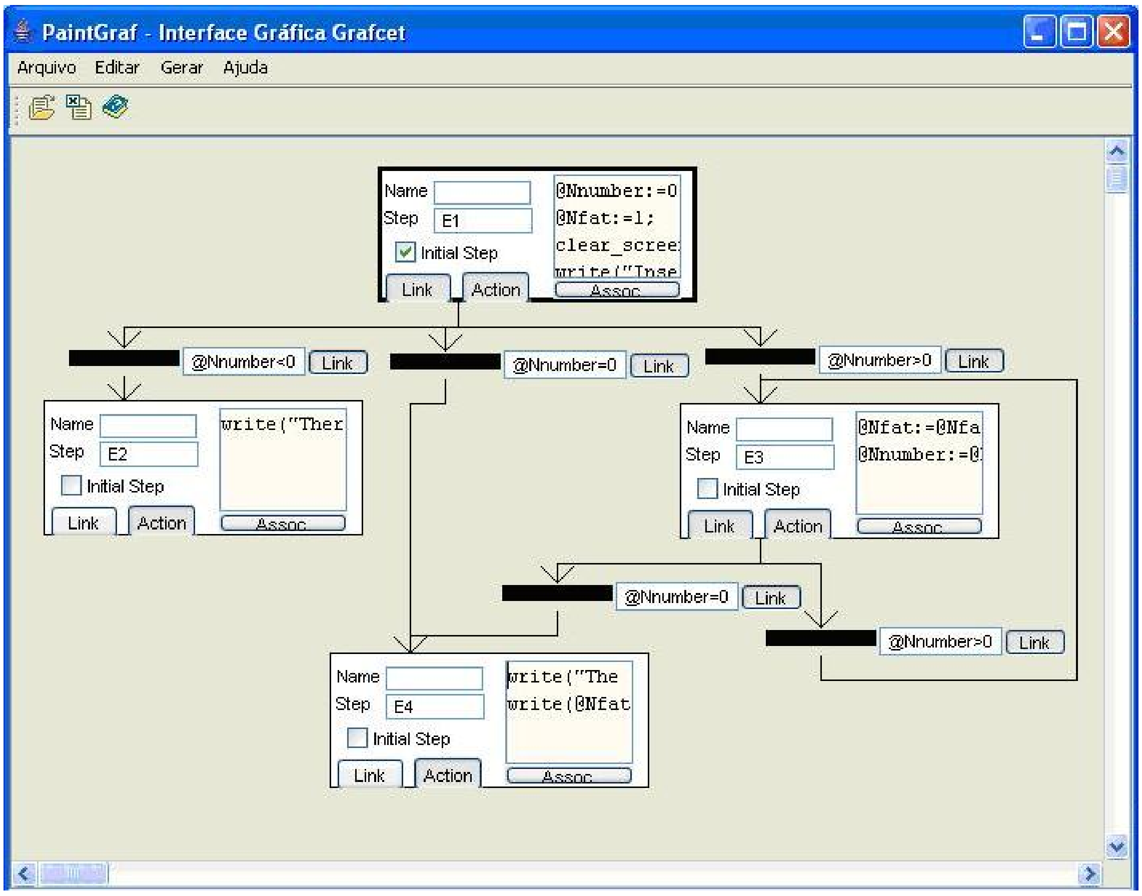


Figure 9.10: Grafcet diagram for factorial case study drawn in PaintGraf editor

```
$Estado_E3 ifv (@N_v1>0) goto $E3
    ifv (@N_v1=0) goto $E4
    goto $Estado_E3
$E4 write("The factorial is:")
    write(@N_v2)
    goto $E_fim
$E_fim EndPrograma
```

9.6.2 Robot Language Inputs for Pascal Programming Language

To generate Pascal programs, it is necessary to specify eight data types files, besides the program structure, robot language instructions, and symbols files. The needed data types files are: Boolean, Character, Numeric, Input, Output, Point, String and Time.

The explanation of such files were already done for the VAL and Rapid target languages. Only the different, important and needed explanations will be presented for these input files.

As Pascal is not an industrial robot programming language, it does not have any instruction for robotic purposes, for example, move, open and close the gripper commands; and input, output and point data types. However, it is possible to simulate, in Pascal, the functioning of a robotic program. To do this, the robotic commands have to be translated into instructions responsible for writing the commands on the screen, so the user could know all the tasks that are performed by the robot; and the input, output and point data types have to be specified as valid data types, for Pascal, without a definition and the variable identifier (as Input and Output data types for VAL language).

Program Structure

```
header@program teste;# uses crt;# label #    %?L%s, %f;# var # %?V%i: %t;
                        # begin # clrscr;
end_line@;
end_prog@repeat until keypressed; end.
```

Standard Instructions

```
$l@@l:
ifs@@if (%sigs) then
ifv@@if (%sigs) then
goto@@goto
```

User Defined Instructions

```
clear_screen@@clrscr
Close@@write("close gear")
Open@@write("open gear")
```

Boolean Data Type

```
BOOLEAN@boolean
ID@
init@{:-(
~@NOT
:=@:=
=@=
!=@<>
write@writeln(%\all,%)
read@readln(%\all,%)
```

Character Data Type

```
CHAR@char
ID@
init@{:-(
+@+
-@-
:=@:=
=@=
<@<
>@>
<=@<=
>=@>=
!=@<>
'@'
write@writeln(%\all,%)
read@readln(%\all,%)
```

Input Data Type

```
INPUT@integer
ID@
init@{:-(
~@write("Insert a value for the sensor %\1%");read(%\1%); %\1%=0
_@write("Insert a value for the sensor %\1%");read(%\1%); %\1%<>0
=@=
!=@<>
<@<
>@>
<=@<=
>=@>=
write@writeln(%\all,%)
read@readln(%\all,%)
```

Numeric Data Type

```
NUMBER@integer
ID@
init@{:-(
+@+
-@-
*@*
/@/
\@ div
%@ mod
:=@:=
=@=
!=@<>
<@<
>@>
<=@<=
>=@>=
write@writeln(%\all,%)
read@readln(%\all,%)
```

Output Data Type

```
OUTPUT@integer
ID@
init@{:-(
on@%\1%:=1;write("Output sensor %\1% was setted.");
off@%\1%:=0;write("Output sensor %\1% was resetted.");
:=@:=
=@{:-(
!=@{:-(
<@{:-(
>@{:-(
<=@{:-(
>=@{:-(
write@writeln(%\all,%)
read@{:-(
```

Point Data Type

```
POINT@
ID@
init@{:-(
:=@{:-(
=@{:-(
<@{:-(
>@{:-(
<=@{:-(
>=@{:-(
!=@{:-(
move@write("move to %p %\all,%")
shift@write("shift %\2% - x, %\3% - y and %\4% - z from point %\1%")
move_c@write("move to point %\1%, passing by the point %\2%, with a circular
              trajectory.")
```

String Data Type

```
STRING@string
ID@
init@{:-(
+@+
:=@:=
=@=
<@<
>@>
<=@<=
>=@>=
!=@<>
"@'
write@writeln(%\all,%)
read@readln(%\all,%)
```

Time Data Type

```
TIME@longint
ID@
init@
set@{:-(
get_time@write("Insert time, in seconds, for the variable %\1%.");read(%\1%)
+@+
-@-
*@*
/@/
\@DIV
%@MOD
:=@:=
=@=
!=@!=
<@<
>@>
<=@<=
>=@>=
write@writeln(%\all,%)
read@readln(%\all,%)
```

Symbols

```
PTL@@left_point
PTD@@right_point
PTC@@center_point
p@@point
```

9.6.3 Pascal Final Code

On this subsection, it is presented the Pascal final code that was generated automatically by the GIRo environment, using the Pascal robot language inputs that were described above. During this final translation process, some errors, or warnings, can occur, and they are stored on an errors file.

For this case study, the generation process was successful, despite the existence of one warning, that are written below.

```
WARNING: POINT data type was not defined
```

This warning is the same that happened during the translation process into a Pascal program for the simple case study, and was also explained on the respective subsection (A.4).

The generation process was successful, and the automatically generated Pascal final code for the InteRGIRo representation presented on subsection 9.6.1 can be seen following.

```
program teste;
uses crt;
label
   E1, Estado_E1, E2, E3, Estado_E3, E4, E_fim;
var
     N_v1: integer;
     N_v2: integer;
begin
    clrscr;
E1: N_v1:=0;
    N_v2:=1;
    clrscr;
    writeln('Insert a positive number to see its factorial.');
    readln(N_v1);
Estado_E1: if (N_v1<0) then goto E2;
    if (N_v1>0) then goto E3;
    if (N_v1=0) then goto E4;
    goto Estado_E1;
E2: writeln('There is no factorial for a negative number.');
    goto E_fim;
E3: N_v2:=N_v2*N_v1;
    N_v1:=N_v1-1;
Estado_E3: if (N_v1>0) then goto E3;
    if (N_v1=0) then goto E4;
    goto Estado_E3;
```

```
E4: writeln('The factorial is:');
    writeln(N_v2);
    goto E_fim;
E_fim: ;
    repeat until keypressed;
end.
```

9.6.4 Robot Language Inputs for C Programming Language

To generate C programs, it is necessary to specify eight data types files, besides the program structure, robot language instructions, and symbols files. The needed data types files are: Boolean, Character, Numeric, Input, Output, Point, String and Time.

The explanation of such files were already done for the VAL, Rapid and Pascal target languages. Only the different, important and needed explanations will be presented for these input files.

Program Structure

```
header@#include "stdio.h"# #include "conio.h"# #include "string.h"#
       #include "time.h"# #define TRUE 1# #define FALSE 0#
        void main()# {# %?V%t %i;# clrscr();
end_line@;
end_prog@getchar(); getchar();}
```

Standard Instructions

```
$l@@l:
ifs@@if (%sigs)
ifv@@if (%sigs)
goto@@goto
```

User Defined Instructions

```
clear_screen@@clrscr()
Close@@write("close gear")
Open@@write("open gear")
```

Boolean Data Type

```
BOOLEAN@int
ID@
init@{:-(
~@!
:=@=
=@==
!=@!=
write@printf("%%c",%\1%)
read@scanf("%%c",&%\1%)
```

Character Data Type

```
CHAR@char
ID@
init@{:-(
+@+
-@-
:=@=
=@==
<@<
>@>
<=@<=
>=@>=
!=@!=
'@'
write@printf("%%c",%\1%)
read@scanf("%%c",&%\1%)
```

Input Data Type

```
INPUT@int
ID@
init@{:-(
~@write("Insert a value for the sensor %\1%");read(%\1%); %\1%=0
_@write("Insert a value for the sensor %\1%");read(%\1%); %\1%!=0
=@==
!=@!=
<@<
>@>
<=@<=
>=@>=
write@printf("%%i\n",%\1%)
read@do { scanf("%%i",&%\1%); } while ((%\1%<0) || (%\1%>10))
```

Numeric Data Type

```
NUMBER@int
ID@
init@{:-(
+@+
-@-
*@*
/@/
\@\
%@%
:=@=
=@==
!=@!=
<@<
>@>
<=@<=
>=@>=
write@printf("%%i",%\1%)
read@scanf("%%i",&%\1%)
```

Output Data Type

```
OUTPUT@int
ID@
init@{:-(
on@%\1%:=1;write("Output sensor %\1% was setted.");
off@%\1%:=0;write("Output sensor %\1% was resetted.");
:=@=
=@{:-(
!=@{:-(
<@{:-(
>@{:-(
<=@{:-(
>=@{:-(
write@printf("%%i",%\1%)
read@{:-(
```

Point Data Type

```
POINT@
ID@
init@{:-(
:=@{:-(
=@{:-(
<@{:-(
>@{:-(
<=@{:-(
>=@{:-(
!=@{:-(
move@write("move to %p %\all,%")
shift@write("shift %\2% - x, %\3% -y and %\4% -z from point %\1%")
move_c@write("move to point %\1%, passing by the point %\2%, with a circular
              trajectory.")
```

String Data Type

```
STRING@char%[256]
```

```
ID@
init@{:-(
+@strcat(%\1%,%\2%)
:=@strcpy(%\1%,%\2%)
=@strcmp(%\1%,%\2%)==0
<@strcmp(%\1%,%\2%)<0
>@strcmp(%\1%,%\2%)>0
<=@strcmp(%\1%,%\2%)<=0
>=@strcmp(%\1%,%\2%)>=0
!=@strcmp(%\1%,%\2%)!=0
"@"
write@printf("%%s\n",%\1%)
read@scanf("%%s",&%\1%)
```

Time Data Type

```
TIME@time_t
ID@
init@
set@ @T_a1 := %\1%; stime(&@T_a1)
get_time@%\1% := time(NULL)
+@+
-@-
*@*
/@/
\@{:-(
%@{:-(
:=@=
=@==
!=@!=
<@<
>@>
<=@<=
>=@>=
write@printf("%%i\n",%\1%)
read@scanf("%%i",&%\1%)
```

Symbols

```
PTL@@left_point
PTD@@right_point
PTC@@center_point
p@@point
```

9.6.5 C Final Code

On this subsection, it is presented the C final code that was generated automatically by the GIRo environment, using the C robot language inputs that were described above. During this final translation process, some errors, or warnings, can occur, and they are stored on an errors file.

For this case study, the generation process was successful, despite the existence of one warning, that are written below.

```
WARNING: POINT data type was not defined
```

This warning is the same that happened during the translation process into a Pascal program for the simple case study, and was also explained on the respective subsection (A.4).

The generation process was successful, and the automatically generated C final code for the InteRGIRo representation presented on subsection 9.6.1 can be seen following.

```
#include "stdio.h"
#include "conio.h"
#include "string.h"
#include "time.h"
#define TRUE 1
#define FALSE 0
void main()
{
     int N_v1;
     int N_v2;
    clrscr();
E1: N_v1=0;
    N_v2=1;
    clrscr();
    printf("%s\n","Insert a positive number to see its factorial.");
    scanf("%i",&N_v1);
Estado_E1: if (N_v1<0) goto E2;
    if (N_v1>0) goto E3;
    if (N_v1==0) goto E4;
    goto Estado_E1;
E2: printf("%s\n","There is no factorial for a negative number.");
    goto E_fim;
E3: N_v2=N_v2*N_v1;
    N_v1=N_v1-1;
Estado_E3: if (N_v1>0) goto E3;
    if (N_v1==0) goto E4;
    goto Estado_E3;
E4: printf("%s\n","The factorial is:");
    printf("%i",N_v2);
    goto E_fim;
E_fim: ;
    getchar(); getchar();
}
```

9.7 Analysis of the Generated Code

After generating 35 programs for this case studies, and many other during this work for the seven presented programming languages, some considerations about the generated code can be done:

1. It is easier to generate code to higher level languages (like Rapid) than the lower level ones (like Melfa Basic III) because the second ones normally have simple instructions. If the programming language has simple instructions, there are many simple ones that perform the same operation for each specific situation. For example, for constants, the instruction is this; for registers, the instruction is that; for variables ... and so on. Normally, a higher level programming language have only one complex instruction that deals with all of this details. So, with higher level languages, it is not necessary to generate many detailed instructions (the target compiler do this), while with lower level ones, it is necessary to generate a lot of detailed instructions (the GIRo generic compiler have to do this). For example, a simple arithmetic operation for a numeric data type, that can be seen on subsection 9.4;
2. It was a right choice the decision of using only unconditional jump instructions, because: all of the used and studied programming languages have this kind of instruction; and when translating any program into an assembly or executable one, it is used only this kind of instruction;
3. It was used only the instructions that were needed for these case studies. But it can be used as many instructions as it is necessary, which means that, the instructions set is not defined and can be increased, if necessary;
4. It generates code that do exactly the same thing that a program that can be created, directly, on the target language. The programs behavior is the same;
5. It is possible to create programs that perform many different tasks. However, some newer instructions should be specified, or created, on the robot language inputs. The element that limits the application is the used industrial robot, and not the GIRo's environment.

Also, it is important to say that, using GIRo, the source code is the one that is edited using the PaintGraf. The InteRGIRo and the automatically generated program are not source codes, so the user should not read and manipulate the generated code, because it is not so easy: it is used unconditional jumps, that difficult reading; the expressions become a sequence of simple expressions, and so on. Normally, nobody is used to open and edit the assembly or the executable generated programs, so why someone would want to open and edit the GIRo's generated code? If, during the program execution, there is something wrong, the source program, and not the generated target one, should be corrected, generating a new target program.

9.8 Chapter's Considerations

In this chapter it was presented some case studies, aiming at the validation of GIRo's approach. The automatically generated target programs, for these case studies, were translated for industrial robot programming languages and for other general purpose programming languages (in this case, Pascal and C ones), which generated some interesting ideas for future works.

Chapter 10

Conclusions

This thesis presented the GIRo's approach to industrial robot programming, with its implementation. It covers all the stages of the programming task, from the modelling of the system until the robot code generation. To achieve its goals, it is used a graphical interface, based on the Grafcet, as a front-end; and it was developed an intermediate representation, called InteRGIRo, to facilitate de compiling process. At the end, some translators were done, one from Grafcet to RS language, other from Grafcet to InteRGIRo, and another one, the Generic Compiler, that is responsible for translating an InteRGIRo program into any robot program.

The translators and the Graphical interface were developed using Java language. Java was chosen because the goal of this project is to grant portability for the source code. So, it is important to use a language that is also platform independent.

As Grafcet is completely implemented, the programming task becomes easier; and because of the GIRo's compilers, it is possible to generate programs to any industrial robot with such Grafcet schema. It is also possible to generate an RS program to be used by the RS translator, that generates an efficient and simple automaton, that can be translated into a simple and efficient code, like the one presented at [Pio98].

Some case studies were done to validate GIRo's approach, more specifically generating code for the following programming languages:

– VAL 1 and VAL 2 (PUMA robots);
– Melfa Basic III (Mitsubishi Movemaster RV-M2 robot);
– Rapid (ABB robots);
– Luna (Sony robot).

As it is not possible to do case studies for all robot programming languages, it was also generated code for the Pascal and C ones, showing that it is possible to generate code for other procedural languages.

The main objective of this approach is to make the programming task easy, with the possibility of reusing the source code to program different industrial robots, allowing to explore their potentialities. The programs created to control industrial robots today make them act as programmable logic controllers that can move; but there are much more things to explore. Maybe the problem arises from the low level of the programming languages available (comparing with the general purpose programming languages), because they do not make easy the robot programming task as it is to program computers, and there is no reuse of source code. Or maybe the problem resides in the simple fact that there is no actual (economic/practical) interest in such a possibility.

The research on programming languages for industrial robots has stopped at the end of the seventies / beginning of the eighties because a program normally is done once, and it is executed for many years by the same robot, which seemed that it was not interesting to invest on this area.

For me, it was a wrong decision to stop researching on this area, because without investing on programming languages, the needed software for industrial robots was developed only by the robots manufacturers, or by some specific software houses. And all of them have economical interest, creating only proprietary solutions and maintaining a dependency on this robotics industry.

If the robots programming has improved, as the computers one has, we could have a different situation today. The robot software could have been evolved as much as the general purpose software do. Who could imagine, in the seventies, that it would be possible to have a pocket computer with a graphical interface today? Of course that the hardware has evolved a lot during these years, but software evolved even more. It can be said that the hardware has evolved to attend the software needs, because software has evolved even more. And to allow the software evolution, it is necessary to have programming languages that are closed to the problem, instead of the machine. In such situation, any programmer could design a robot software, made and test it. Also, with more programmers, more ideas are developed, and the software evolves even more.

Today, there are a lot of general purpose applications because of the computer software evolution. And what about robotics applications? They look like the same from thirty five years ago. Of course that today it is possible to have a simulation software, the robots are more robust, precise, etc, but the tasks that they do are the same from thirty five years ago. It could be possible to have also a lot of different applications, but it is not what we get today, probably because of the weak evolution of the robotics software. Why it is not possible to have an robotic arm coupled on a wheel-chair to help the physical deficients? Why we do not have robotic arms inside our garages or homes to help storing things? Probably, these situations could be possible today, if the programming task of robotics arms where easier, to facilitate the software development or to facilitate the programming task. Just for comparing, on the computers domain, we had mainframes thirty five years ago, and today we have an IPhone. What is the main difference? The software. Thirty five years ago, the software existed to facilitate the access to the hardware. Today, the hardware evolves to achieve the software needs.

10.1 Contributions

The contribution of this thesis is the GIRo environment, that allows the automatic generation of code to many different industrial robots, based on the same source code, that is developed on a friendly and well-known Grafcet graphical interface.

Below, it is listed the features that makes GIRo an innovated programming environment, and so, the contribution of this thesis:

- It covers all the stages of the programming task, from the modelling of the system until the robot code generation;
- It uses, as a front-end, a graphical interface, that is based on the well-known Grafcet;
- It allows the reuse of source code;
- It grants the source code portability;
- It is expressive;
- It allows the specification to be closed to the problem, instead of the robot;
- It enables the interaction with the environment, evaluating the digital input/output signals;
- It has an intermediate representation, InteRGIRo, that facilitates the translation process, allowing the development of another front-ends, if necessary, without the necessity of changing the back-end generator;
- It was developed some translators that allow the translation between:
    - PaintGraf internal representation - RS programming language;
    - PaintGraf internal representation - InteRGIRo representation;
    - InteRGIRo representation - any industrial robot programming language, the Generic Compiler.
- It is platform independent (all the translators were done in Java programming language, using Java Swing to develop the graphical interface)
- It is possible to generate code to various industrial robots based on the same PaintGraf source program, which facilitates and simplifies the programming task;
- It is possible to test and validate the industrial robot source programs in any computer that has the ANSI C or Pascal compilers, because it was also specified the inputs for these programming languages. This means that it is not necessary to have the industrial robot programming environment, or its software licence, in the computer, to perform the tests, because the source program is the same and the final code generation works well;
- It satisfies all of the software engineering principles presented on figure 2.6;
- It is industrial robot manufacturer independent, which means that this work was not done to satisfy any specific family of robots;
- It is easier to describe the target programming language instead of the target machine, because all of the needed information that is necessary, to allow the final code generation, can be obtained from the programming language manual. It makes this environment become safe from the economic interests;
- It is possible to use, on the source program, target commands. However, with this use, the portability of the source code is compromitted;

– It allows the use and creation of new source code instructions. Of course that, to generate final code, these new instructions have to be specified on the specific robot inputs files.

This last contribution is important because this work does not defines a set of basic instructions to be used by all of the industrial robots. Instead of this, it allows the user to create new ones, if necessary, to achieve its own goals. This means that, this work does not specify a standard language.

According to [Zie01], all of the previous researches, that tried to create standard industrial robot programming languages, were not successful because all of them tried to define a set of basic, or standard, instructions, to be used by all of the existing industrial robots.

To finish, GIRo is not completely done, which means that it can be increased, improved, and can become more powerful, as it will be seen on the *Future Work* section.

10.2 GIRo's Evaluation

At this moment, it is not possible to compare this work with any other one, because it was not found any solution with the same features and aims of the one presented on this thesis: a friendly solution that grants portability for the source code to different industrial robots.

On the generic programming domain, there are some related works, like BURG, BEG, GCC, McCat and others. However, all of them needs a hardware description, and it is not possible to get this description on industrial robots domain. Also, part of the goal of this thesis is to create a friendly and an easy-to-use environment, and it is not easy to describe an specific hardware according to a pre-defined grammar.

Other solutions, on the generic programming domain, use a virtual machine to grant portability for the source code. However, this solution is also not possible because some robots are old enough, and their controllers would not support a virtual machine.

So, this thesis presents a new, valid and useful work, on an important area of industrial robotics: its programming. The three papers that were submitted to robotics conferences (a regional one [ALHF02], an European one [AHF05], and a world one [AHF03]) where accepted and presented, demonstrating the community interest on this research.

10.3 Future Work

There are some interesting future tasks that will be done to facilitate the usability of GIRo, and also to increase the number of applications that could use GIRo as an programming environment.

These future tasks are to:

– **Improve the PaintGraf** - At this moment, PaintGraf is a prototype that must be

improved to facilitate the programming task;
- **Facilitate the Specification of the Target Language** - At this moment, the robot language inputs, that correspond to the specification of the target language, are entered manually, and the user must know their syntax to describe them correctly. It must be created friendly graphical front-ends, to facilitate these descriptions, based on questions to be answered by the user, that will take a look at the target language manuals;
- **Validate the Structured Data Types** - With this validation, it will be possible to use InteRGIRo not only with atomic data types, but with all the well-known data types;
- **Automatize the Specification of Data Types** - At this moment, it is possible to use at most eleven pre-defined data types. With this automatization, it will be possible the creation of as many data types as necessary. To do it, the user must specificate the name of the data type that he wants to use on Giro, and the correspondent name at the target language, creating the specificated data type file of the target language;
- **Validate the "Variable" Allocation Algorithm** - The used algorithm was not compared with other similar ones, like the used on registers allocation, to validate it. It was based on the graph coloring algorithm, but he was not compared with the existing ones that are implemented, to identify if it really determines the minimum number of variables as possible;
- **Allow the Specification of the Maximum Number of Variables for Each Data Type** - Some programming languages limits the amount of variables, or signals, for some specific data types. For example, Val and Val II accept only 32 input signals and 32 output ones. It might be useful to specify the maximum number of variables, or signals, for a specific data type to identify if the generated final program can really be executed by the target machine;
- **Allow Code Optimizations** - At this moment, GIRo has only one optimization, responsible for the "Variable" Allocation. Some other can also be done, allowing the generation of target programs that would consume less resources and that would be executed faster. For example, some known optimizations that could be implemented are:
  - Constant Propagation;
  - Constant Folding;
  - Algebraic Simplifications;
  - Copy Propagation;
  - Common Sub Expression Elimination;
  - Code Hoisting.

  Also, because of the translation process between PaintGraf and InteRGIRo (complex arithmetic, or relational, expressions are transformed into a sequence of simple expressions), some others could be implemented, like:
  - If a goto instruction jumps to the next line, the goto instruction could be removed;
  - If there are only two conditional sentences (*if*), and the conditions are opposed, the second conditional sentence could be removed.

- **Allow Parallel Programming** - Grafcet allows parallel execution, but the industrial robots do not, because an industrial robot can not perform more than one task, at the same time, in the environment. However, granting parallel programming will enable to generate programs to be executed in parallel, in the case of new robots that could came with this potentiality;
- **Control a Manufacturing Cell** - The main advantage of allowing the parallel programming is that it will be possible to create a system to control more than one industrial robot at the same time, in this case, a manufacturing cell;
- **Validate the Translation into other Procedural Languages** - After the successful of the case studies, generating code also for Pascal and C language, it could be useful to test and validate the use GIRo, or at least the InteRGIRo and the Generic Compiler (with another front-end), as a general programming language tool, for the programming of any computer, or robot controller. Maybe, it could be validated because all of computers / robot controllers have, at least, an assembly language, that is a procedural one;
- **Create Other Front-Ends** - Grafcet was chosen because the goal of this thesis is to facilitate and to grant source code portability for industrial robots programming. But, with the possibility for translating to other procedural languages, it could be interesting, maybe, to develop other front-ends, specifically designed for solving its kind of problems. For example, it could be interesting to create a front-end based on Petri Nets.

10.4 Other Future Ideas... Or Dreams...

In future, after being concluded the implementation of the tasks presented in previous section, some others, that today are nothing more than ideas, or dreams, could be analyzed concretely and, maybe, be brought into reality. They are:

- **Translate to Object Oriented Languages** - After the validation of the composed data types, it could be interesting to start a research that would adapt the InteRGIRo to allow the use of abstract data types, obtaining, in this case, the possibility for generating code to object oriented languages;
- **Transform GIRo into an Useful Programming Environment for Imperative Languages** - After increasing the Giro environment and allowing the generation of parallel and object oriented programs, Giro could be used to generate final code for most, or all, of the imperative languages;
- **Increase the Use of Grafcet** - After Giro environment be completely implemented, the Grafcet, or another front-end to be developed, could be studied to verify if it can be used for systems development, and not only for automatism description.

Bibliography

[AB07] ABB Flexible Automation AB. Robotstudio. http://www.abb.com/roboticssoftware", 2007.

[ABB94] ABB Flexible Automation AB. *Rapid Reference Manual 3.0*, 1994.

[ADH$^+$00] G. Aigner, A. Diwan, D. Heine, M. Lam, D. Moore, B. Murphy, and C. Sapuntzakis. An overview of the suif2 compiler infrastructure. Technical report, Computer System Laboratory, Stanford University, August 2000.

[ADR98] A. Appel, J. Davidson, and N. Ramsey. The zephyr compiler infrastructure. Technical report, Princeton University, 1998.

[AHF02] G. V. Arnold, P. R. Henriques, and J. C. Fonseca. Uma proposta de linguagem de programação para robótica. In *Anais do VI Simpósio Brasileiro de Linguagens de Programação (SBLP'02)*, pages 134–143, 2002.

[AHF03] Gustavo V. Arnold, Pedro R. Henriques, and Jaime C. Fonseca. A development approach to industrial robots programming. In *Proceedings of the 11th IEEE International Conference on Advanced Robotics (ICAR)*, volume 1, pages 167–172, Coimbra, Portugal, 2003. Institute for Systems and Robotics, University of Coimbra.

[AHF05] Gustavo V. Arnold, Pedro R. Henriques, and Jaime C. Fonseca. A graphical interface based on grafcet for programming industrial robots off-line. In *Proceedings of the 2nd International Conference on Informatics in Control, Automation and Robotics (ICINCO)*, Barcelona, Spain, 2005.

[AHF06] Gustavo V. Arnold, Pedro R. Henriques, and Jaime C. Fonseca. Giro (grafcet - industrial robots): A generic environment for programming industrial robots off-line. In *Proceedings of the XXXII Conferência Latino Americana de Informática (CLEI)*, Santiago, Chile, 2006.

[AL07] McGill University ACAPS Lab. Mccat compiler. http://www-acaps.cs.mcgill.ca/info/McCAT/McCAT.html, 2007.

[ALHF02] G. V. Arnold, G. R. Librelotto, P. R. Henriques, and J. C. Fonseca. O uso da linguagem rs em robótica. In *Electrónica e Telecomunicações*, pages 501–508, Aveiro, Portugal, 2002. Revista do Departamento de Electrónica e Telecomunicações da Universidade de Aveiro.

[Arn98] G. V. Arnold. Uma extensão da linguagem rs para o desenvolvimento de sistemas reativos baseados na arquitetura de subsunção. Dissertação de mestrado, CPGCC, UFRGS, Porto Alegre, Brasil, 1998.

[Arn00] G. V. Arnold. Controle de uma célula de manufatura através da linguagem RS estendida. In *Anais do I Congresso de Lógica Aplicada à Tecnologia (LAPTEC'2000)*, pages 145–163, São Paulo, SP, Brasil, 2000.

[Ass07] OSACA Association. Osaca: Open system architecture for controls association. http://www.osaca.org", 2007.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley Publishing Company, 1986.

[AT98] G. V. Arnold and S. S. Toscani. Using the RS language to control autonomous vehicles. In *Workshop on Intelligent Components for Vehicles*, pages 465–470, Seville, Spain, 1998. International Federation of Automatic Control.

[Aut] ABB Flexible Automation. *RAPID Reference Manual: Overview / System Data Types and Routines*. 3HAC 7783-1 / 7774-1. For BaseWare OS 4.0.

[BG92] G. Berry and G. Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19:87–152, 1992.

[Bla07] D. S. Blank. Pyro - python robotics. http://pyrorobotics.org", 2007.

[Bro86] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of robotics and Automation*, RA-2(1):14–23, 1986.

[Bro90] R. A. Brooks. The behavior language. A.I. Memo 1227, MIT AI Laboratory, 1990.

[CCMD93] P. Canalda, L. Cognard, M. Mazaud, and A. Despland. Pagode: A back-end generator for risc machines. Technical Report 152, Unité de Recherche INRIA Rocquencourt, May 1993.

[Cen07] Flanders Mechatronics Technology Centre. The orocos project. http://www.orocos.org", 2007.

[Co07] Festo Didactic GmbH & Co. Cosimir professional. http://www.festo-didactic.com/int-en/learning-systems/software-e-learning/cosimir/cosimir-professional.htm", 2007.

[Com] Mitsubishi Company. *Movemaster RV-M2 Manual*. Mitsubishi Company.

[Com80] Unimation Company. *User's Guide to VAL*. Unimation Company, 1980.

[Cor94] Nachi Fujikoshi Corporation. *Ar Controller Operating Manual*. Nachi-Fujikoshi Corporation, 1st edition, 1994.

[Cor07] Microsoft Corporation. Microsoft robotics studio. http://msdn.microsoft.com/robotics/", 2007.

[dBMB03] Raphael W. de Bettio, Alejandro Martins, and Luis H. Bogo. X-arm: uma linguagem de programação de braços robóticos. In *Anais do III Congresso Brasileiro de Computação (CBComp 2003) Robótica*, Itajaí, SC, Brasil, 2003.

[EFK01] O. Stern E. Freund, B. Lüdemann-Ravit and T. Koch. Creating the architecture of a translator framework for robot programming languages. In *Proceedings of the 2001 IEEE International Conference on Robotics & Automation*, pages 187 – 192, Seoul, Korea, 2001.

[ES89] H. Emmelmann and F. W. Schroer. Beg - a generator for efficient back ends. In *Proceedings on Programming Language Design and Implementation*, volume 24, Oregon, 1989.

[FB98a] G. Frensel and P. M. Bruijn. From modelling control systems using grafcet to analyzing systems using hybrid automata. In *Proceedings of the American Control Conference*, pages 704 – 705, Philadelphia, Pennsylvania, June 1998.

[FB98b] G. Frensel and P. M. Bruijn. The specification of a robot control system with grafcet. In *Proceedings of the UCACC International Conference on CONTROL'98*, pages 733 – 738. IEEE, September 1998.

[FD80] C. Fraser and J. Davidson. The design and application of a retargetable peephole optimizer. *ACM Transactions on Programming Languages and Systems*, 2(2):191 – 202, April 1980.

[FH95] C. Fraser and D. Hanson. *A Retargetable C Compiler: design and implementation.* Addison Wesley Publishing Company, 1995.

[FHP91] C. Fraser, R. Henry, and T. Proebsting. Burg - fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 27:68 – 76, 1991.

[fI07] International Alliance for Interoperability. Industry foundation classes. http://www.iai-tech.org/", 2007.

[Fra77] C. W. Fraser. *Automatic Generation of Code Generators.* Phd dissertation, Yale University, New Haven, 1977.

[FW88] C. W. Fraser and A. L. Wendt. Automatic generation of fast optimizing code generators. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 79 – 84, Atlanta, Georgia, 1988.

[Gra00] Grafcet. Grafcet homepage. http://www.lurpa.ens-cachan.fr/grafcet.html, 2000.

[Gro87] M. P. Groover. *Automation, Production, Systems, and Computer-Integrated Manufacturing.* Prentice Hall, Inc., 1987.

[Gro07] Object Management Group. Robotics dtf. http://robotics.omg.org/", 2007.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *The Science of Computer Programming*, (8):231 – 274, 1987.

[HH79] J. E. Hopcroft and J. D. Hullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Massachussetts, USA, 1979.

[HMH01] J. E. Hopcroft, R. Motwani, and J. D. Hullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 2nd edition, 2001.

[Hor00] I. D. Horswill. Functional programming of behavior-based systems. *Autonomous Robots*, (9):83 – 93, 2000.

[Inc86] Unimation Incorporated. *User's Guide to VAL II: Programming Manual.* A Westinghouse Company, 1986.

[Int06] ECMA International. Standard ecma-335, common language infrastructure (cli). http://www.ecma-international.org/publications/standards/Ecma-335.htm, June 2006. 4th Edition.

[JML91] R. E. Johnson, C. McConnell, and J. M. Lake. The rtl system: A framework for code optimization. In *Proceedings of the International Workshop on Code Generation*, pages 255 – 274, Dagstuhl, Germany, May 1991.

[LPB85] T. Lozano-Perez and R. A. Brooks. An approach to automatic robot programming. A.I. Memo 842, MIT AI Laboratory, April 1985.

[LPW77] T. Lozano-Perez and P. H. Winston. Lama: A language for automatic mechanical assembly. In *Fifth International Joint Conference on Artificial Intelligence*, pages 710 – 716, Massachusetts, August 1977.

[LW77] L. I. Lieberman and M. A. Wesley. Autopass: An automatic programming system for computer controlled mechanical assembly. *IBM Journal of Research Development*, 4(21):321 – 333, July 1977.

[LY99] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Sun Microsystems, Inc, second edition edition, 1999. http://java.sun.com/Series.

[Mat99] P. J. Matos. Estudo e desenvolvimento de sistemas de geração de back-ends do processo de compilação. Masters dissertation, Dpto de Informática, Universidade do Minho, Braga, Portugal, 1999.

[Mat05] P. J. Matos. *Um Modelo Arquitectónico para Desenvolvimento de Compiladores: Aplicação à Framework Dolphin.* Tese de doutoramento, Depto de Informática, Escola de Engenharia, Universidade do Minho, Guimarães, Portugal, 2005.

[Mer03] J. Merrill. Generic and gimple: A new tree representation for entire functions. pages 171–179, 2003.

[MGB82] M. S. Mujtaba, R. Goldman, and T. Binford. Stanford's al robot programming language. *Computers in Mechanical Engineering*, August 1982.

[MT00] M. Makatchev and S. K. Tso. Human-robot inteface using agents communicating in a xml-based markup language. In *Proceedings of the IEEE International Workshop on Robot and Human Interactive Communication*, pages 270 – 275, Osaka, Japan, September 2000.

[Muc97] S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, 1997.

[NH91] et al N. Halbwachs. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[Nov04] D. Novillo. From source to binary: The inner workings of gcc. *Red Hat Magazine*, December 2004.

[Orc07] Orca: Components for robotics. http://orca-robotics.sourceforge.net/index.html", 2007.

[Org07] International Standards Organization. Iso 10303:1994 - industrial automation systems and integration - product data representation and exchange. http://www.iso.ch/cate/cat.html - search on 10303 to find a list of parts of the standard", 2007.

[Pet77] J. L. Peterson. Petri nets. *ACM Computing Surveys*, 9(3):223–252, 1977.

[PHH99] J. Peterson, G. D. Hager, and P. Hudak. A language for declarative robotic programming. In *Proceedings of the International Conference on Robotics and Automation*, 1999.

[Pio98] S. J. Piola. Uso da linguagem rs no controle do robô nachi sc15f. Trabalho de Conclusão de Curso de Graduação, Departamento de Informática, UCS, Caxias do Sul, Brasil, 1998.

[Pro95] T. A. Proebsting. Burs automata generation. *ACM Transactions on Programming Languages and Systems*, 17(3):461–486, 1995.

[Reh97] J. A. Rehg. *Introduction to Robotics in CIM Systems.* Prentice Hall, Inc., 3rd edition, 1997.

[RF95a] N. Ransey and M. Fernandez. New jersey machine-code toolkit. Technical report, Departament of Computer Science, Princeton University, 1995.

[RF95b] N. Ransey and M. Fernandez. The new jersey machine-code toolkit. In *Proceedings of The USENIX Technical Conference*, pages 289 – 302, New Orleans, 1995.

[RF96] N. Ransey and M. Fernandez. New jersey machine-code toolkit architecture specifications. Technical report, Departament of Computer Science, Princeton University, 1996.

[Sal00] S. Salmi. Potential use of xml in robotics. XML Seminar Autumn, 2000.

[Seb99] Robert W. Sebesta. *Concepts of programming languages.* Addison Wesley Longman, Inc., 4nd edition, 1999.

[Sof07] Free Software. Player project. http://playerstage.sourceforge.net/index.php", 2007.

[Sta00] R. Stallman. Using and porting the gnu compiler collection (gcc). iUniverse.com, Inc, 2000.

[Tec86] Flow Software Technologies. ... Flow Software Technologies, 1986.

[Tec07a] Flow Software Technologies. Workspace. http://www.workspace5.com", 2007.

[Tec07b] Robotic Workspace Technologies. Universal robot controller. http://www.rwt.com, 2007.

[Tel82] Telemec, Portugal. *O Grafcet – Diagrama Funcional para Automatismos Sequenciais*, 1982.

[Tos93] S. S. Toscani. *RS: Uma Linguagem para Programação de Núcleos Reactivos.* Tese de doutoramento, Depto de Informática, UNL, Lisboa, Portugal, 1993.

[TSM82] R. H. Taylor, P. D. Summers, and J. M. Meyer. Aml: a manufacturing language. *The International Journal of Robotics Research*, 1(3), 1982.

[Ven96] B. Venners. Bytecode basics; a first look at the bytecodes of the java virtual machine. *JavaWorld.com*, 1996.

[WH00] Z. Wan and P. Hudak. Functional reactive programming from first principles. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 242 – 252, Vancouver, British Columbia, Canada, June 2000. ACM.

[Wik07] the free encyclopedia Wikipedia. Variable assembly language. http://en.wikipedia.org/wiki/Variable_Assembly_Language, 2007.

[Zie99] C. Zielinski. The mrroc++ system. In *1st Workshop on Robot Motion and Control, RoMoCo'99,*, pages 147 – 152, Kiekrz, Polan, June 1999.

[Zie01] C. Zielinski. By how much should a general purpose programming language be extended to become a multi-robot system programming language? *Advanced Robotics*, 15(1):71–95, 2001.

[ZSW05] C. Zielinski, W. Szynkiewicz, and T. Winiarski. Applications of mrroc++ robot programming framework. In *5th Int. Workshop on Robot Motion and Control, RoMoCo'05*, pages 251 – 257, Dymaczewo, Polan, June 2005.

Appendix A

A Simple Case Study

In this appendix, it is presented the generated final code for the VAL II, Melfa Basic III, Luna, Pascal and C programming languages, based on the Simple case study, presented on 9.2.

A.1 VAL II Final Code

On this subsection, it is presented the VAL II final code that was generated automatically by the GIRo environment, using the VAL II robot language inputs that are described in chapter 9. During this final translation process, some errors, or warnings, can occur, and they are stored on an errors file.

For this case study, the generation process was also successful, despite the existence of warnings, that are written below.

```
WARNING: NUMBER data type was not defined
WARNING: POINT data type was not defined
WARNING: STRING data type was not defined
WARNING: INPUT data type was not defined
WARNING: OUTPUT data type was not defined
```

These warnings are the same that happened during the translation process into a VAL program, and were explained on subsection 9.2.3. The generation process was successful, and the automatically generated VAL II final code for the InteRGIRo representation presented on subsection 9.2.1 can be seen following.

```
10 if sig(1002) goto 50
20 if sig(1003) goto 120
30 if sig(1004) goto 170
40 goto 10
50 closei
60 goto 70
70 move left_point
80 break
```

```
90 goto 100
100 openi
110 goto 10
120 closei
130 goto 140
140 move center_point
150 break
160 goto 100
170 closei
180 goto 190
190 move right_point
200 break
210 goto 100
```

A.2 Melfa Basic III Final Code

On this subsection, it is presented the Melfa Basic III final code that was generated automatically by the GIRo environment, using the Melfa Basic III robot language inputs that are described in chapter 9. During this final translation process, some errors, or warnings, can occur, and they are stored on an errors file.

For this case study, the generation process was successful, despite the existence of warnings, that are written below.

```
WARNING: NUMBER data type was not defined
WARNING: POINT data type was not defined
WARNING: INPUT data type was not defined
WARNING: OUTPUT data type was not defined
WARNING: Target language do not have IF statements.
         You must verify if it has instructions that evaluate
         the relational operators.
```

The first four warnings are the same that happened during the translation process into a VAL program, and were explained on subsection 9.2.3. The last one indicates that there is no *IF* instruction on the target language. It means that, to deal with conditions, the target language must have specific operations to be used as relational operators. Probably, these operations are related to each data type. If there are not this kind of instructions, an error message is presented. In this case, only a warning message was generated, to inform the user about the situation, but it is all ok.

The generation process was successful, and the automatically generated Melfa Basic III final code for the InteRGIRo representation presented on subsection 9.2.1 can be seen following.

```
10 IN 1
20 NE 0 80
30 IN 2
40 NE 0 140
50 IN 3
```

```
60 NE 0 180
70 GT 10
80 GC
90 GT 100
100 MO 1
110 GT 120
120 GO
130 GT 10
140 GC
150 GT 160
160 MO 2
170 GT 120
180 GC
190 GT 200
200 MO 3
210 GT 120
220 ED
```

Melfa Basic III operations deals with registers directly. To compare the value of an input signal with zero, for example, the value must be moved to the internal register (instruction IN) and, after, an specific instruction to compare with zero (in this case NE, that means not equal) must be executed.

A.3 Luna Final Code

On this subsection, it is presented the Luna final code that was generated automatically by the GIRo environment, using the Luna robot language inputs that are described in chapter 9. During this final translation process, some errors, or warnings, can occur, and they are stored on an errors file.

For this case study, the generation process was successful, despite the existence of warnings, that are written below.

```
WARNING: TIME data type was not defined
WARNING: INPUT data type was not defined
WARNING: OUTPUT data type was not defined
```

The first warning indicates that the TIME data type was not defined. This happened because, in Luna language, there is no TIME data type, but there are instructions that gets and sets the time. And the value that is manipulated by this instructions is numeric. So, it can be viewed that, the TIME data type does not have a specific data type in Luna, but there are operations that manipulate this kind of data. This means that it is possible to manipulate the time, without defining an specific data type.

The last two warnings are the same that happened during the translation process into a Rapid program, and were explained on subsection 9.2.5.

The generation process was successful, and the automatically generated Luna final code for the InteRGIRo representation presented on subsection 9.2.1 can be seen following.

```
10     POINT : left_point
20     POINT : center_point
30     POINT : right_point
40 IF I2(ON) GO 80
50 IF I3(ON) GO 140
60 IF I4(ON) GO 180
70 GO 40
80 DO L8(ON)
90 GO 100
100 DO LINE:left_point
110 GO 120
120 DO L8(OFF)
130 GO 40
140 DO L8(ON)
150 GO 160
160 DO LINE:center_point
170 GO 120
180 DO L8(ON)
190 GO 200
200 DO LINE:right_point
210 GO 120
220 END
```

The Sony robot that is programmed using the Luna programming language does not have a gripper, which means that Luna do not have the operations Open and Close gripper. So, they were translated into operations that sets and resets the output signal *L8*.

A.4 Pascal Final Code

On this subsection, it is presented the Pascal final code that was generated automatically by the GIRo environment, using the Pascal robot language inputs that are described in chapter 9. During this final translation process, some errors, or warnings, can occur, and they are stored on an errors file.

For this case study, the generation process was successful, despite the existence of one warning, that are written below.

```
WARNING: POINT data type was not defined
```

The only warning indicates that the POINT data type was not defined. This happened because Pascal does not have this kind of data type. However, the basic POINT data type operations can be simulated in Pascal, using writing instructions, to inform the user that the point operations that were done. This is interesting because it makes possible testing the logic of the program in any computer that has a Pascal compiler, as can be seen above, where it is presented the automatically generated Pascal final code for the InteRGIRo representation presented on subsection 9.2.1.

```
program teste;
uses crt;
```

```
label
   E1, E2, E5, E9, E3, E6, E4, E8;
var
     Ip1: integer;
     Ip2: integer;
     Ip3: integer;
begin
    clrscr;
E1: writeln('Insert a value for the sensor Ip1');
    readln(Ip1);
    if (Ip1<>0) then goto E2;
    writeln('Insert a value for the sensor Ip2');
    readln(Ip2);
    if (Ip2<>0) then goto E3;
    writeln('Insert a value for the sensor Ip3');
    readln(Ip3);
    if (Ip3<>0) then goto E4;
    goto E1;
E2: writeln('close gear');
    goto E5;
E5: writeln('move to point left_point');
    goto E9;
E9: writeln('open gear');
    goto E1;
E3: writeln('close gear');
    goto E6;
E6: writeln('move to point center_point');
    goto E9;
E4: writeln('close gear');
    goto E8;
E8: writeln('move to point right_point');
    goto E9;
    repeat until keypressed;
end.
```

Pascal also do not have INPUT and OUTPUT data types. As POINT data types, they can be simulated, redefining this data types as *integer* ones, and using the reading / writing instructions to manipulate their values. The POINT data type was not redefined as integer because that is no need for manipulating the POINT variables. There are only instructions to access that positions, and any for changing them.

A.5 C Final Code

On this subsection, it is presented the C final code that was generated automatically by the GIRo environment, using the C robot language inputs that are described in chapter 9. During this final translation process, some errors, or warnings, can occur, and they are stored on an errors file.

For this case study, the generation process was successful, despite the existence of one warning, that are written below.

```
WARNING: POINT data type was not defined
```

The only warning indicates the same problem that happened during the translation process into a Pascal program, and were explained on subsection A.4.

The generation process was successful, and the automatically generated C final code for the InteRGIRo representation presented on subsection 9.2.1 can be seen following.

```
#include "stdio.h"
#include "conio.h"
#include "string.h"
#include "time.h"
#define TRUE 1
#define FALSE 0
void main()
{
     int Ip1;
     int Ip2;
     int Ip3;
clrscr();
E1: printf("%s\n","Insert a value for the sensor Ip1");
    do { scanf("%i",&Ip1);
    } while ((Ip1<0) || (Ip1>10));
    if (Ip1!=0) goto E2;
    printf("%s\n","Insert a value for the sensor Ip2");
    do { scanf("%i",&Ip2);
    } while ((Ip2<0) || (Ip2>10));
    if (Ip2!=0) goto E3;
    printf("%s\n","Insert a value for the sensor Ip3");
    do { scanf("%i",&Ip3);
    } while ((Ip3<0) || (Ip3>10));
    if (Ip3!=0) goto E4;
    goto E1;
E2: printf("%s\n","close gear");
    goto E5;
E5: printf("%s\n","move to point left_point");
    goto E9;
E9: printf("%s\n","open gear");
    goto E1;
E3: printf("%s\n","close gear");
    goto E6;
E6: printf("%s\n","move to point center_point");
    goto E9;
E4: printf("%s\n","close gear");
    goto E8;
E8: printf("%s\n","move to point right_point");
    goto E9;
    getchar(); getchar();
}
```

C, like Pascal, also do not have INPUT and OUTPUT data types, and they are treated in the same way as Pascal does, as it was explained on subsection A.4.

Appendix B

Boxing Apples Case Study

In this appendix, it is presented the generated final code for the VAL, Melfa Basic III, Luna, Pascal and C programming languages, based on the Boxing Apples case study, presented on 9.3.

B.1 VAL Final Code

On this subsection, it is presented the VAL final code that was generated automatically by the GIRo environment, using the VAL robot language inputs that are described in chapter 9. During this final translation process, some errors, or warnings, can occur, and they are stored on an errors file.

For this case study, the generation process was successful, despite the existence of warnings, that are written below.

```
WARNING: NUMBER data type was not defined
WARNING: POINT data type was not defined
WARNING: STRING data type was not defined
WARNING: INPUT data type was not defined
WARNING: OUTPUT data type was not defined
```

These warnings are the same that happened during the translation process into a VAL program for the simple case study 9.2.3, and were explained on that subsection.

The generation process was successful, and the automatically generated VAL final code for the InteRGIRo representation presented on subsection 9.3.1 can be seen following.

```
10 signal 4
20 openi
30 ifsig 1005,,, then goto 80
40 ifsig 1011,,, then goto 60
50 goto 70
60 ifsig -1005,,, then goto 110
70 goto 30
```

```
80 signal -4
90 signal -5
100 goto 490
110 signal -4
120 seti N_v1=0
130 signal 5
140 ifsig 1005,,, then goto 80
150 ifsig 1012,,, then goto 170
160 goto 180
170 ifsig -1005,,, then goto 190
180 goto 140
190 signal -5
200 move  Ppos_aux
210 move Ppos_apple
220 appro Ppos_apple, 0, -50, 0
230 closei
240 move  Ppos_aux
250 move Ppos_box
260 openi
270 appro Ppos_box, 0, 100, 0
280 seti N_v1=N_v1+1
290 if N_v1<10 then goto 310
300 goto 320
310 signal 5
320
330 ifsig 1005,,, then goto 80
340 if N_v1=10 then goto 360
350 goto 370
360 ifsig -1005,,, then goto 430
370 if N_v1<10 then goto 390
380 goto 420
390 ifsig 1012,,, then goto 410
400 goto 420
410 ifsig -1005,,, then goto 190
420 goto 330
430 signal 4
440 ifsig 1005,,, then goto 80
450 ifsig 1011,,, then goto 470
460 goto 480
470 ifsig -1005,,, then goto 110
480 goto 440
490
```

As there is any instruction responsible for doing a circular trajectory moving on VAL language, that instruction was mapped into two move sentences. It will not have a circular trajectory, but it will not beat on any conveyor belts or on the box.

B.2 Melfa Basic III Final Code

On this subsection, it is presented the Melfa Basic III final code that was generated automatically by the GIRo environment, using the Melfa Basic III robot language inputs, that are described in chapter 9. During this final translation process, some errors, or warnings, can occur, and they are stored on an errors file.

For this case study, the generation process was successful, despite the existence of warnings, that are written below.

```
WARNING: NUMBER data type was not defined
WARNING: POINT data type was not defined
WARNING: INPUT data type was not defined
WARNING: OUTPUT data type was not defined
WARNING: Target language do not have IF statements.
         You must verify if it has instructions that evaluate
         the relational operators.
```

These warnings are the same that happened during the translation process into a Melfa Basic III program for the simple case study A.2, and were explained on that subsection.

The generation process was successful, and the automatically generated Melfa Basic III final code for the InteRGIRo representation presented on subsection 9.3.1 can be seen following.

```
10 OT 1 4
20 GO
30 IN 5
40 NE 0 110
50 IN 11
60 NE 0 80
70 GT 100
80 IN 5
90 EQ 0 140
100 GT 30
110 OT 0 4
120 OT 0 5
130 GT 870
140 OT 0 4
150 cp 0
160  cl 1
170 OT 1 5
180 IN 5
190 NE 0 110
200 IN 12
210 NE 0 230
220 GT 250
230 IN 5
240 EQ 0 260
250 GT 180
260 OT 0 5
```

```
270 MO  5
280 MO 4
290 SH 4 0,-50,0
300 MO 4
310 SH 4 -0,-50,-0
320 GC
330 MO  5
340 MO 6
350 GO
360 SH 6 0,100,0
370 MO 6
380 SH 6 -0,-100,-0
390 cp 1
400 cl 2
410 sc 3 1
420 cp 3
430 sm 0 490
440  eq 0 540
450 ic 2
460 dc 3
470 cp 3
480 gt 440
490  eq 0 540
500 dc 2
510 ic 3
520 cp 3
530 gt 490
540
550 cp 2
560  cl 1
570 CP 1
580 SM 10 600
590 GT 610
600 OT 1 5
610
620 IN 5
630 NE 0 110
640 CP 1
650 EQ 10 670
660 GT 690
670 IN 5
680 EQ 0 780
690 CP 1
700 SM 10 720
710 GT 770
720 IN 12
730 NE 0 750
740 GT 770
750 IN 5
760 EQ 0 260
770 GT 620
```

```
780 OT 1 4
790 IN 5
800 NE 0 110
810 IN 11
820 NE 0 840
830 GT 860
840 IN 5
850 EQ 0 140
860 GT 790
870
880 ED
```

Melfa Basic III do not have arithmetic operations. Only increment and decrement of one. So, the sum operation was mapped to a set of Melfa Basic III instructions, that can perform sum operations using increment, decrement and relational instructions. This is why the final code is bigger than the other ones.

B.3 Luna Final Code

On this subsection, it is presented the Luna final code that was generated automatically by the GIRo environment, using the Luna robot language inputs that are described in chapter 9. During this final translation process, some errors, or warnings, can occur, and they are stored on an errors file.

For this case study, the generation process was successful, despite the existence of warnings, that are written below.

```
WARNING: TIME data type was not defined
WARNING: INPUT data type was not defined
WARNING: OUTPUT data type was not defined
```

These warnings are the same that happened during the translation process into a Luna program for the simple case study, and were also explained on the respective subsection (A.3).

The generation process was successful, and the automatically generated Luna final code for the InteRGIRo representation presented on subsection 9.3.1 can be seen following.

```
10     INT : N_v1
20     POINT : Ppos_apple
30     POINT : Ppos_aux
40     POINT : Ppos_box
50 DO L4(OFF)
60 DO L8(OFF)
70 IF I5(ON) GO 120
80 IF I11(ON) GO 100
90 GO 110
100 IF I5(OFF) GO 150
110 GO 70
120 DO L4(ON)
```

```
130 DO L5(ON)
140 GO 550
150 DO L4(ON)
160 N_v1=0
170 DO L5(OFF)
180 IF I5(ON) GO 120
190 IF I12(ON) GO 210
200 GO 220
210 IF I5(OFF) GO 230
220 GO 180
230 DO L5(ON)
240 DO LINE: Ppos_aux
250 DO LINE:Ppos_apple
260 DO LINE:Ppos_apple
270 SHIFT 0,-50,0,0
280 DO L8(ON)
290 DO LINE: Ppos_aux
300 DO LINE:Ppos_box
310 DO L8(OFF)
320 DO LINE:Ppos_box
330 SHIFT 0,100,0,0
340 N_v1=N_v1+1
350 IF N_v1<10 GO 370
360 GO 380
370 DO L5(OFF)
380
390 IF I5(ON) GO 120
400 IF N_v1=10 GO 420
410 GO 430
420 IF I5(OFF) GO 490
430 IF N_v1<10 GO 450
440 GO 480
450 IF I12(ON) GO 470
460 GO 480
470 IF I5(OFF) GO 230
480 GO 390
490 DO L4(OFF)
500 IF I5(ON) GO 120
510 IF I11(ON) GO 530
520 GO 540
530 IF I5(OFF) GO 150
540 GO 500
550
560 END
```

B.4 Pascal Final Code

On this subsection, it is presented the Pascal final code that was generated automatically by the GIRo environment, using the Pascal robot language inputs that are described in chapter 9. During this final translation process, some errors, or warnings, can occur, and they are stored on an errors file.

For this case study, the generation process was successful, despite the existence of one warning, that are written below.

```
WARNING: POINT data type was not defined
```

This warning is the same that happened during the translation process into a Pascal program for the simple case study, and was also explained on the respective subsection (A.4).

The generation process was successful, and the automatically generated Pascal final code for the InteRGIRo representation presented on subsection 9.3.1 can be seen following.

```
program teste;
uses crt;
label
   E1, Estado_E1, Estado_E1_1if, Estado_E1_else2, E5, E2, Estado_E2,
   Estado_E2_1if, Estado_E2_else2, E3, E3_AC2, E3_AC3, Estado_E3,
   Estado_E3_1if, Estado_E3_else2, Estado_E3_2if, Estado_E3_3if,
   Estado_E3_else3, E4, Estado_E4, Estado_E4_1if, Estado_E4_else2, E_fim;
var
     Osc_box: integer;
     Istop: integer;
     Ibox: integer;
     Osc_apple: integer;
     N_v1: integer;
     Iapple: integer;
begin
    clrscr;
E1: Osc_box:=1;
    writeln('Output sensor Osc_box was setted.');
    ;
    writeln('open gear');
Estado_E1: writeln('Insert a value for the sensor Istop');
    readln(Istop);
    if (Istop<>0) then goto E5;
    writeln('Insert a value for the sensor Ibox');
    readln(Ibox);
    if (Ibox<>0) then goto Estado_E1_1if;
    goto Estado_E1_else2;
Estado_E1_1if: writeln('Insert a value for the sensor Istop');
    readln(Istop);
    if (Istop=0) then goto E2;
Estado_E1_else2: goto Estado_E1;
E5: Osc_box:=0;
    writeln('Output sensor Osc_box was resetted.');
    ;
    Osc_apple:=0;
    writeln('Output sensor Osc_apple was resetted.');
    ;
    goto E_fim;
E2: Osc_box:=0;
    writeln('Output sensor Osc_box was resetted.');
    ;
    N_v1:=0;
    Osc_apple:=1;
    writeln('Output sensor Osc_apple was setted.');
    ;
Estado_E2: writeln('Insert a value for the sensor Istop');
    readln(Istop);
    if (Istop<>0) then goto E5;
    writeln('Insert a value for the sensor Iapple');
    readln(Iapple);
    if (Iapple<>0) then goto Estado_E2_1if;
    goto Estado_E2_else2;
Estado_E2_1if: writeln('Insert a value for the sensor Istop');
    readln(Istop);
    if (Istop=0) then goto E3;
Estado_E2_else2: goto Estado_E2;
E3: Osc_apple:=0;
    writeln('Output sensor Osc_apple was resetted.');
    ;
    writeln('move to point Ppos_apple, passing by the point  Ppos_aux,
             with a circular trajectory.');
    writeln('shift 0 - x, -50 - y and 0 - z from point Ppos_apple');
    writeln('close gear');
    writeln('move to point Ppos_box, passing by the point  Ppos_aux,
             with a circular trajectory.');
    writeln('open gear');
    writeln('shift 0 - x, 100 - y and 0 - z from point Ppos_box');
    N_v1:=N_v1+1;
    if (N_v1<10) then goto E3_AC2;
    goto E3_AC3;
E3_AC2: Osc_apple:=1;
    writeln('Output sensor Osc_apple was setted.');
    ;
E3_AC3: ;
Estado_E3: writeln('Insert a value for the sensor Istop');
    readln(Istop);
    if (Istop<>0) then goto E5;
    if (N_v1=10) then goto Estado_E3_1if;
    goto Estado_E3_else2;
Estado_E3_1if: writeln('Insert a value for the sensor Istop');
    readln(Istop);
    if (Istop=0) then goto E4;
Estado_E3_else2: if (N_v1<10) then goto Estado_E3_2if;
    goto Estado_E3_else3;
Estado_E3_2if: writeln('Insert a value for the sensor Iapple');
    readln(Iapple);
    if (Iapple<>0) then goto Estado_E3_3if;
    goto Estado_E3_else3;
```

```
Estado_E3_3if: writeln('Insert a value for the sensor Istop');
    readln(Istop);
    if (Istop=0) then goto E3;
Estado_E3_else3: goto Estado_E3;
E4: Osc_box:=1;
    writeln('Output sensor Osc_box was setted.');
    ;
Estado_E4: writeln('Insert a value for the sensor Istop');
    readln(Istop);
    if (Istop<>0) then goto E5;
    writeln('Insert a value for the sensor Ibox');
    readln(Ibox);
    if (Ibox<>0) then goto Estado_E4_1if;
    goto Estado_E4_else2;
Estado_E4_1if: writeln('Insert a value for the sensor Istop');
    readln(Istop);
    if (Istop=0) then goto E2;
Estado_E4_else2: goto Estado_E4;
E_fim: ;
    repeat until keypressed;
end.
```

B.5 C Final Code

On this subsection, it is presented the C final code that was generated automatically by the GIRo environment, using the C robot language inputs that are described in chapter 9. During this final translation process, some errors, or warnings, can occur, and they are stored on an errors file.

For this case study, the generation process was successful, despite the existence of one warning, that are written below.

```
WARNING: POINT data type was not defined
```

This warning is the same that happened during the translation process into a Pascal program for the simple case study, and was also explained on the respective subsection (A.4).

The generation process was successful, and the automatically generated C final code for the InteRGIRo representation presented on subsection 9.3.1 can be seen following.

```
#include "stdio.h"
#include "conio.h"
#include "string.h"
#include "time.h"
#define TRUE 1
#define FALSE 0
void main()
{
     int Osc_box;
     int Istop;
     int Ibox;
     int Osc_apple;
     int N_v1;
     int Iapple;
clrscr();
E1: Osc_box=1;
    printf("%s\n","Output sensor Osc_box was setted.");
    ;
    printf("%s\n","open gear");
Estado_E1: printf("%s\n","Insert a value for the sensor Istop");
    do { scanf("%i",&Istop);
    } while ((Istop<0) || (Istop>10));
    if (Istop!=0) goto E5;
    printf("%s\n","Insert a value for the sensor Ibox");
    do { scanf("%i",&Ibox);
    } while ((Ibox<0) || (Ibox>10));
    if (Ibox!=0) goto Estado_E1_1if;
    goto Estado_E1_else2;
Estado_E1_1if: printf("%s\n","Insert a value for the sensor Istop");
    do { scanf("%i",&Istop);
    } while ((Istop<0) || (Istop>10));
    if (Istop==0) goto E2;
Estado_E1_else2: goto Estado_E1;
E5: Osc_box=0;
    printf("%s\n","Output sensor Osc_box was resetted.");
    ;
    Osc_apple=0;
    printf("%s\n","Output sensor Osc_apple was resetted.");
    ;
    goto E_fim;
E2: Osc_box=0;
    printf("%s\n","Output sensor Osc_box was resetted.");
    ;
    N_v1=0;
    Osc_apple=1;
    printf("%s\n","Output sensor Osc_apple was setted.");
    ;
Estado_E2: printf("%s\n","Insert a value for the sensor Istop");
    do { scanf("%i",&Istop);
    } while ((Istop<0) || (Istop>10));
    if (Istop!=0) goto E5;
    printf("%s\n","Insert a value for the sensor Iapple");
    do { scanf("%i",&Iapple);
    } while ((Iapple<0) || (Iapple>10));
    if (Iapple!=0) goto Estado_E2_1if;
    goto Estado_E2_else2;
Estado_E2_1if: printf("%s\n","Insert a value for the sensor Istop");
    do { scanf("%i",&Istop);
    } while ((Istop<0) || (Istop>10));
    if (Istop==0) goto E3;
```

```
Estado_E2_else2: goto Estado_E2;
E3: Osc_apple=0;
    printf("%s\n","Output sensor Osc_apple was resetted.");
    ;
    printf("%s\n","move to point Ppos_apple, passing by the point  Ppos_aux,
            with a circular trajectory.");
    printf("%s\n","shift 0 - x, -50 -y and 0 -z from point Ppos_apple");
    printf("%s\n","close gear");
    printf("%s\n","move to point Ppos_box, passing by the point  Ppos_aux,
            with a circular trajectory.");
    printf("%s\n","open gear");
    printf("%s\n","shift 0 - x, 100 -y and 0 -z from point Ppos_box");
    N_v1=N_v1+1;
    if (N_v1<10) goto E3_AC2;
    goto E3_AC3;
E3_AC2: Osc_apple=1;
    printf("%s\n","Output sensor Osc_apple was setted.");
    ;
E3_AC3: ;
Estado_E3: printf("%s\n","Insert a value for the sensor Istop");
    do { scanf("%i",&Istop);
    } while ((Istop<0) || (Istop>10));
    if (Istop!=0) goto E5;
    if (N_v1==10) goto Estado_E3_1if;
    goto Estado_E3_else2;
Estado_E3_1if: printf("%s\n","Insert a value for the sensor Istop");
    do { scanf("%i",&Istop);
    } while ((Istop<0) || (Istop>10));
    if (Istop==0) goto E4;
Estado_E3_else2: if (N_v1<10) goto Estado_E3_2if;
    goto Estado_E3_else3;
Estado_E3_2if: printf("%s\n","Insert a value for the sensor Iapple");
    do { scanf("%i",&Iapple);
    } while ((Iapple<0) || (Iapple>10));
    if (Iapple!=0) goto Estado_E3_3if;
    goto Estado_E3_else3;
Estado_E3_3if: printf("%s\n","Insert a value for the sensor Istop");
    do { scanf("%i",&Istop);
    } while ((Istop<0) || (Istop>10));
    if (Istop==0) goto E3;
Estado_E3_else3: goto Estado_E3;
E4: Osc_box=1;
    printf("%s\n","Output sensor Osc_box was setted.");
    ;
Estado_E4: printf("%s\n","Insert a value for the sensor Istop");
    do { scanf("%i",&Istop);
    } while ((Istop<0) || (Istop>10));
    if (Istop!=0) goto E5;
    printf("%s\n","Insert a value for the sensor Ibox");
    do { scanf("%i",&Ibox);
    } while ((Ibox<0) || (Ibox>10));
    if (Ibox!=0) goto Estado_E4_1if;
    goto Estado_E4_else2;
Estado_E4_1if: printf("%s\n","Insert a value for the sensor Istop");
    do { scanf("%i",&Istop);
    } while ((Istop<0) || (Istop>10));
    if (Istop==0) goto E2;
Estado_E4_else2: goto Estado_E4;
E_fim: ;
    getchar(); getchar();
}
```

Appendix C

Bottles Palletization Case Study

In this appendix, it is presented the generated final code for the VAL, VAL II, Luna, Pascal and C programming languages, based on the Bottles Palletization case study, presented on 9.4.

C.1 VAL Final Code

On this subsection, it is presented the VAL final code that was generated automatically by the GIRo environment, using the VAL robot language inputs that are described in chapter 9. During this final translation process, some errors, or warnings, can occur, and they are stored on an errors file.

For this case study, the generation process was successful, despite the existence of warnings, that are written below.

```
WARNING: NUMBER data type was not defined
WARNING: POINT data type was not defined
WARNING: STRING data type was not defined
WARNING: INPUT data type was not defined
WARNING: OUTPUT data type was not defined
```

These warnings are the same that happened during the translation process into a VAL program for the simple case study 9.2.3, and were explained on that subsection.

The generation process was successful, and the automatically generated VAL final code for the InteRGIRo representation presented on subsection 9.4.1 can be seen following.

```
10 seti N_v1=1
20 seti N_v2=0
30 seti N_v3=0
40 seti N_v4=0
50 seti N_v5=0
60 seti N_v6=0
70 seti N_v7=0
```

```
80 seti N_v8=0
90 seti N_v9=0
100 seti N_v10=30
110 seti N_v11=50
120 openi
130 goto 140
140 if N_v1<=1 then goto 160
150 goto 170
160 ifsig 1007,,, then goto 240
170 if N_v1<=2 then goto 190
180 goto 200
190 ifsig 1006,,, then goto 510
200 if N_v1>1 then goto 220
210 goto 230
220 ifsig -1006,,, then goto 780
230 goto 140
240 move Paux1
250 move Ppsc1
260 appro Ppsc1,  0,  -100,  0
270 closei
280 move Paux1
290 move Ppbox1
300 appro Ppbox1,  N_v4,  0,  N_v5
310 appro Ppbox1,  N_v4,  -100,  N_v5
320 openi
330 seti N_v2=N_v2+1
340 seti N_v8=N_v8+1
350 seti N_v4=N_v4+N_v10
360 if N_v8=4 then goto 380
370 goto 410
380 seti N_v8=0
390 seti N_v4=0
400 seti N_v5=N_v5+N_v10
410 seti N_v1=2
420 if N_v2<20 then goto 140
430 if N_v2=20 then goto 450
440 goto 420
450 seti N_v2=0
460 seti N_v8=0
470 seti N_v4=0
480 seti N_v5=0
490 signal 6
500 goto 140
510 move Paux2
520 move Ppsc2
530 appro Ppsc2,  0,  -100,  0
540 closei
550 move Paux2
560 move Ppbox2
570 appro Ppbox2,  N_v6,  0,  N_v7
580 appro Ppbox2,  N_v6,  -125,  N_v7
```

```
590 openi
600 seti N_v3=N_v3+1
610 seti N_v9=N_v9+1
620 seti N_v6=N_v6+N_v11
630 if N_v9=3 then goto 650
640 goto 680
650 seti N_v9=0
660 seti N_v6=0
670 seti N_v7=N_v7+N_v11
680 seti N_v1=1
690 if N_v3=12 then goto 720
700 if N_v3<12 then goto 140
710 goto 690
720 seti N_v3=0
730 seti N_v9=0
740 seti N_v6=0
750 seti N_v7=0
760 signal 7
770 goto 140
780 seti N_v1=1
790 goto 140
```

C.2 VAL II Final Code

On this subsection, it is presented the VAL II final code that was generated automatically by the GIRo environment, using the VAL II robot language inputs that are described in chapter 9. During this final translation process, some errors, or warnings, can occur, and they are stored on an errors file.

For this case study, the generation process was also successful, despite the existence of warnings, that are written below.

```
WARNING: NUMBER data type was not defined
WARNING: POINT data type was not defined
WARNING: STRING data type was not defined
WARNING: INPUT data type was not defined
WARNING: OUTPUT data type was not defined
```

These warnings are the same that happened during the translation process into a VAL program, and were explained on subsection 9.2.3.

The generation process was successful, and the automatically generated VAL II final code for the InteRGIRo representation presented on subsection 9.4.1 can be seen following.

```
10 N_v1=1
20 N_v2=0
30 N_v3=0
40 N_v4=0
50 N_v5=0
60 N_v6=0
```

```
70 N_v7=0
80 N_v8=0
90 N_v9=0
100 N_v10=30
110 N_v11=50
120 openi
130 goto 140
140 if N_v1< =1 goto 160
150 goto 170
160 if sig(1007) goto 240
170 if N_v1< =2 goto 190
180 goto 200
190 if sig(1006) goto 550
200 if N_v1>1 goto 220
210 goto 230
220 if sig(-1006) goto 860
230 goto 140
240 move Paux1
250 break
260 move Ppsc1
270 break
280 appro Ppsc1,  0,  -100,  0
290 closei
300 move Paux1
310 break
320 move Ppbox1
330 break
340 appro Ppbox1,  N_v4,  0,  N_v5
350 appro Ppbox1,  N_v4,  -100,  N_v5
360 openi
370 N_v2=N_v2+1
380 N_v8=N_v8+1
390 N_v4=N_v4+N_v10
400 if N_v8==4 goto 420
410 goto 450
420 N_v8=0
430 N_v4=0
440 N_v5=N_v5+N_v10
450 N_v1=2
460 if N_v2<20 goto 140
470 if N_v2==20 goto 490
480 goto 460
490 N_v2=0
500 N_v8=0
510 N_v4=0
520 N_v5=0
530 signal 6
540 goto 140
550 move Paux2
560 break
570 move Ppsc2
```

```
580 break
590 appro Ppsc2,  0,  -100,  0
600 closei
610 move Paux2
620 break
630 move Ppbox2
640 break
650 appro Ppbox2,  N_v6,  0,  N_v7
660 appro Ppbox2,  N_v6,  -125,  N_v7
670 openi
680 N_v3=N_v3+1
690 N_v9=N_v9+1
700 N_v6=N_v6+N_v11
710 if N_v9==3 goto 730
720 goto 760
730 N_v9=0
740 N_v6=0
750 N_v7=N_v7+N_v11
760 N_v1=1
770 if N_v3==12 goto 800
780 if N_v3<12 goto 140
790 goto 770
800 N_v3=0
810 N_v9=0
820 N_v6=0
830 N_v7=0
840 signal 7
850 goto 140
860 N_v1=1
870 goto 140
```

C.3 Luna Final Code

On this subsection, it is presented the Luna final code that was generated automatically by the GIRo environment, using the Luna robot language inputs that are described in chapter 9. During this final translation process, some errors, or warnings, can occur, and they are stored on an errors file.

For this case study, the generation process was successful, despite the existence of warnings, that are written below.

```
WARNING: TIME data type was not defined
WARNING: INPUT data type was not defined
WARNING: OUTPUT data type was not defined
```

These warnings are the same that happened during the translation process into a Luna program for the simple case study, and were also explained on the respective subsection (A.3).

The generation process was successful, and the automatically generated Luna final code for the InteRGIRo representation presented on subsection 9.4.1 can be seen following.

```
10     INT : N_v1
20     INT : N_v2
30     INT : N_v3
40     INT : N_v4
50     INT : N_v5
60     INT : N_v6
70     INT : N_v7
80     INT : N_v8
90     INT : N_v9
100     INT : N_v10
110     INT : N_v11
120     POINT : Ppsc1
130     POINT : Paux1
140     POINT : Ppbox1
150     POINT : Ppsc2
160     POINT : Paux2
170     POINT : Ppbox2
180 N_v1=1
190 N_v2=0
200 N_v3=0
210 N_v4=0
220 N_v5=0
230 N_v6=0
240 N_v7=0
250 N_v8=0
260 N_v9=0
270 N_v10=30
280 N_v11=50
290 DO L8(OFF)
300 GO 310
310 IF N_v1<=1 GO 330
320 GO 340
330 IF I7(ON) GO 410
340 IF N_v1<=2 GO 360
350 GO 370
360 IF I6(ON) GO 710
370 IF N_v1>1 GO 390
380 GO 400
390 IF I6(OFF) GO 1010
400 GO 310
410 DO LINE:Paux1
420 DO LINE:Ppsc1
430 DO LINE:Ppsc1
440 SHIFT  0, -100, 0,0
450 DO L8(ON)
460 DO LINE:Paux1
470 DO LINE:Ppbox1
480 DO LINE:Ppbox1
```

```
490 SHIFT  N_v4, 0, N_v5,0
500 DO LINE:Ppbox1
510 SHIFT  N_v4, -100, N_v5,0
520 DO L8(OFF)
530 N_v2=N_v2+1
540 N_v8=N_v8+1
550 N_v4=N_v4+N_v10
560 IF N_v8=4 GO 580
570 GO 610
580 N_v8=0
590 N_v4=0
600 N_v5=N_v5+N_v10
610 N_v1=2
620 IF N_v2<20 GO 310
630 IF N_v2=20 GO 650
640 GO 620
650 N_v2=0
660 N_v8=0
670 N_v4=0
680 N_v5=0
690 DO L6(OFF)
700 GO 310
710 DO LINE:Paux2
720 DO LINE:Ppsc2
730 DO LINE:Ppsc2
740 SHIFT  0, -100, 0,0
750 DO L8(ON)
760 DO LINE:Paux2
770 DO LINE:Ppbox2
780 DO LINE:Ppbox2
790 SHIFT  N_v6, 0, N_v7,0
800 DO LINE:Ppbox2
810 SHIFT  N_v6, -125, N_v7,0
820 DO L8(OFF)
830 N_v3=N_v3+1
840 N_v9=N_v9+1
850 N_v6=N_v6+N_v11
860 IF N_v9=3 GO 880
870 GO 910
880 N_v9=0
890 N_v6=0
900 N_v7=N_v7+N_v11
910 N_v1=1
920 IF N_v3=12 GO 950
930 IF N_v3<12 GO 310
940 GO 920
950 N_v3=0
960 N_v9=0
970 N_v6=0
980 N_v7=0
990 DO L7(OFF)
```

```
1000 GO 310
1010 N_v1=1
1020 GO 310
1030 END
```

C.4 Pascal Final Code

On this subsection, it is presented the Pascal final code that was generated automatically by the GIRo environment, using the Pascal robot language inputs that are described in chapter 9. During this final translation process, some errors, or warnings, can occur, and they are stored on an errors file.

For this case study, the generation process was successful, despite the existence of one warning, that are written below.

```
WARNING: POINT data type was not defined
```

This warning is the same that happened during the translation process into a Pascal program for the simple case study, and was also explained on the respective subsection (A.4).

The generation process was successful, and the automatically generated Pascal final code for the InteRGIRo representation presented on subsection 9.4.1 can be seen following.

```
program teste;
uses crt;
label
   E1, E2, E2_1if, E2_else1, E2_2if, E2_else2, E2_3if, E2_else3, E3, E3_AC2,
   E3_AC3, Estado_E3, E4, E5, E5_AC2, E5_AC3, Estado_E5, E6, EE7;
var
     N_v1: integer;
     N_v2: integer;
     N_v3: integer;
     N_v4: integer;
     N_v5: integer;
     N_v6: integer;
     N_v7: integer;
     N_v8: integer;
     N_v9: integer;
     N_v10: integer;
     N_v11: integer;
     Ise1: integer;
     Ise2: integer;
     Osc1: integer;
     Osc2: integer;
begin
    clrscr;
E1: N_v1:=1;
    N_v2:=0;
    N_v3:=0;
    N_v4:=0;
    N_v5:=0;
    N_v6:=0;
    N_v7:=0;
    N_v8:=0;
    N_v9:=0;
    N_v10:=30;
    N_v11:=50;
    writeln('open gear');
    goto E2;
E2: if (N_v1<=1) then goto E2_1if;
    goto E2_else1;
E2_1if: writeln('Insert a value for the sensor Ise1');
    readln(Ise1);
    if (Ise1<>0) then goto E3;
E2_else1: if (N_v1<=2) then goto E2_2if;
    goto E2_else2;
E2_2if: writeln('Insert a value for the sensor Ise2');
    readln(Ise2);
    if (Ise2<>0) then goto E5;
E2_else2: if (N_v1>1) then goto E2_3if;
    goto E2_else3;
E2_3if: writeln('Insert a value for the sensor Ise2');
    readln(Ise2);
    if (Ise2=0) then goto EE7;
E2_else3: goto E2;
E3: writeln('move to point Ppsc1, passing by the point Paux1,
             with a circular trajectory.');
    writeln('shift  0 - x,  -100 - y and  0 - z from point Ppsc1');
    writeln('close gear');
    writeln('move to point Ppbox1, passing by the point Paux1,
             with a circular trajectory.');
    writeln('shift  N_v4 - x,  0 - y and  N_v5 - z from point Ppbox1');
    writeln('shift  N_v4 - x,  -100 - y and  N_v5 - z from point Ppbox1');
    writeln('open gear');
    N_v2:=N_v2+1;
    N_v8:=N_v8+1;
    N_v4:=N_v4+N_v10;
    if (N_v8=4) then goto E3_AC2;
    goto E3_AC3;
E3_AC2: N_v8:=0;
    N_v4:=0;
    N_v5:=N_v5+N_v10;
E3_AC3: N_v1:=2;
Estado_E3: if (N_v2<20) then goto E2;
    if (N_v2=20) then goto E4;
    goto Estado_E3;
E4: N_v2:=0;
    N_v8:=0;
    N_v4:=0;
    N_v5:=0;
    Osc1:=1;
    writeln('Output sensor Osc1 was setted.');
    ;
    goto E2;
E5: writeln('move to point Ppsc2, passing by the point Paux2,
             with a circular trajectory.');
    writeln('shift  0 - x,  -100 - y and  0 - z from point Ppsc2');
    writeln('close gear');
    writeln('move to point Ppbox2, passing by the point Paux2,
             with a circular trajectory.');
    writeln('shift  N_v6 - x,  0 - y and  N_v7 - z from point Ppbox2');
    writeln('shift  N_v6 - x,  -125 - y and  N_v7 - z from point Ppbox2');
    writeln('open gear');
    N_v3:=N_v3+1;
    N_v9:=N_v9+1;
    N_v6:=N_v6+N_v11;
    if (N_v9=3) then goto E5_AC2;
    goto E5_AC3;
E5_AC2: N_v9:=0;
    N_v6:=0;
    N_v7:=N_v7+N_v11;
E5_AC3: N_v1:=1;
Estado_E5: if (N_v3=12) then goto E6;
    if (N_v3<12) then goto E2;
    goto Estado_E5;
E6: N_v3:=0;
    N_v9:=0;
    N_v6:=0;
    N_v7:=0;
    Osc2:=1;
    writeln('Output sensor Osc2 was setted.');
    ;
    goto E2;
EE7: N_v1:=1;
    goto E2;
    repeat until keypressed;
end.
```

C.5 C Final Code

On this subsection, it is presented the C final code that was generated automatically by the GIRo environment, using the C robot language inputs that are described in chapter 9. During this final translation process, some errors, or warnings, can occur, and they are stored on an errors file.

For this case study, the generation process was successful, despite the existence of one warning, that are written below.

```
WARNING: POINT data type was not defined
```

This warning is the same that happened during the translation process into a Pascal program for the simple case study, and was also explained on the respective subsection (A.4).

The generation process was successful, and the automatically generated C final code for the InteRGIRo representation presented on subsection 9.3.1 can be seen following.

```
#include "stdio.h"
#include "conio.h"
#include "string.h"
#include "time.h"
#define TRUE 1
#define FALSE 0
void main()
{
     int N_v1;
     int N_v2;
     int N_v3;
     int N_v4;
     int N_v5;
     int N_v6;
     int N_v7;
     int N_v8;
     int N_v9;
     int N_v10;
     int N_v11;
     int Ise1;
     int Ise2;
     int Osc1;
     int Osc2;
clrscr();
E1: N_v1=1;
    N_v2=0;
    N_v3=0;
    N_v4=0;
    N_v5=0;
    N_v6=0;
    N_v7=0;
    N_v8=0;
    N_v9=0;
    N_v10=30;
    N_v11=50;
    printf("%s\n","open gear");
    goto E2;
E2: if (N_v1<=1) goto E2_1if;
    goto E2_else1;
E2_1if: printf("%s\n","Insert a value for the sensor Ise1");
    do { scanf("%i",&Ise1);
    } while ((Ise1<0) || (Ise1>10));
    if (Ise1!=0) goto E3;
E2_else1: if (N_v1<=2) goto E2_2if;
```

```
    goto E2_else2;
E2_2if: printf("%s\n","Insert a value for the sensor Ise2");
    do { scanf("%i",&Ise2);
    } while ((Ise2<0) || (Ise2>10));
    if (Ise2!=0) goto E5;
E2_else2: if (N_v1>1) goto E2_3if;
    goto E2_else3;
E2_3if: printf("%s\n","Insert a value for the sensor Ise2");
    do { scanf("%i",&Ise2);
    } while ((Ise2<0) || (Ise2>10));
    if (Ise2==0) goto EE7;
E2_else3: goto E2;
E3: printf("%s\n","move to point Ppsc1, passing by the point Paux1,
            with a circular trajectory.");
    printf("%s\n","shift  0 - x,  -100 -y and  0 -z from point Ppsc1");
    printf("%s\n","close gear");
    printf("%s\n","move to point Ppbox1, passing by the point Paux1,
            with a circular trajectory.");
    printf("%s\n","shift  N_v4 - x,  0 -y and  N_v5 -z from point Ppbox1");
    printf("%s\n","shift  N_v4 - x,  -100 -y and  N_v5 -z from point Ppbox1");
    printf("%s\n","open gear");
    N_v2=N_v2+1;
    N_v8=N_v8+1;
    N_v4=N_v4+N_v10;
    if (N_v8==4) goto E3_AC2;
    goto E3_AC3;
E3_AC2: N_v8=0;
    N_v4=0;
    N_v5=N_v5+N_v10;
E3_AC3: N_v1=2;
Estado_E3: if (N_v2<20) goto E2;
    if (N_v2==20) goto E4;
    goto Estado_E3;
E4: N_v2=0;
    N_v8=0;
    N_v4=0;
    N_v5=0;
    Osc1=1;
    printf("%s\n","Output sensor Osc1 was setted.");
    ;
    goto E2;
E5: printf("%s\n","move to point Ppsc2, passing by the point Paux2,
            with a circular trajectory.");
    printf("%s\n","shift  0 - x,  -100 -y and  0 -z from point Ppsc2");
    printf("%s\n","close gear");
    printf("%s\n","move to point Ppbox2, passing by the point Paux2,
            with a circular trajectory.");
    printf("%s\n","shift  N_v6 - x,  0 -y and  N_v7 -z from point Ppbox2");
    printf("%s\n","shift  N_v6 - x,  -125 -y and  N_v7 -z from point Ppbox2");
    printf("%s\n","open gear");
    N_v3=N_v3+1;
    N_v9=N_v9+1;
    N_v6=N_v6+N_v11;
    if (N_v9==3) goto E5_AC2;
    goto E5_AC3;
E5_AC2: N_v9=0;
    N_v6=0;
    N_v7=N_v7+N_v11;
E5_AC3: N_v1=1;
Estado_E5: if (N_v3==12) goto E6;
    if (N_v3<12) goto E2;
    goto Estado_E5;
E6: N_v3=0;
    N_v9=0;
    N_v6=0;
    N_v7=0;
    Osc2=1;
    printf("%s\n","Output sensor Osc2 was setted.");
    ;
    goto E2;
EE7: N_v1=1;
    goto E2;
    getchar(); getchar();
}
```

Appendix D

A Level Rail Crossing Case Study

In this appendix, it is presented the generated final code for the VAL, VAL II, Melfa Basic III, Pascal and C programming languages, based on the Level Rail Crossing case study, presented on 9.5.

D.1 VAL Final Code

On this subsection, it is presented the VAL final code that was generated automatically by the GIRo environment, using the VAL robot language inputs that are described in chapter 9. During this final translation process, some errors, or warnings, can occur, and they are stored on an errors file.

For this case study, the generation process was not successful, because there are errors, that are written below.

```
WARNING: NUMBER data type was not defined
WARNING: POINT data type was not defined
WARNING: STRING data type was not defined
WARNING: INPUT data type was not defined
WARNING: OUTPUT data type was not defined
ERROR: It is not possible to generate code for this target language.
       There is no TIME data type on it.
ERROR/WARNING: The get_time instruction was not defined! It is assumed
               that it is a target language instruction.
```

The first five warnings are the same that happened during the translation process into a VAL program, and were explained on subsection 9.2.3.

The next one, and also the first error message, says that it is impossible to generate final code, because the InteRGIRo code manipulates a specific data type (in this case, the TIME one), but it does not exist on the target language. So, it is not possible to use any operations from this data type.

As can be seen on the last error / warning, the InteRGIRo program uses an operation (in this case, a *get_ time* one) that was not defined. This message could be viewed as an error, or a warning. As a warning because this InteRGIRo instruction could be exactly a final code instruction, that would be copied directly to the final code. And as an error because the target language could not support that instruction. So, the user should evaluate such situation to identify if the generated code is valid or not, and, in this case, such situation corresponds to an error, because of the previous error message, where it was indicated that an instruction of a non defined data type is being used.

So, this generation process was **not** successful, but the automatically generated VAL final code, for the InteRGIRo representation presented on subsection 9.5.1, can be seen following, with the *get_ time* instructions that caused the errors.

```
10 ifsig 1009,,, then goto 40
20 ifsig 1010,,, then goto 230
30 goto 10
40 get_time(T_v1)
50 signal -1
60 signal 2
70 ifsig 1010,,, then goto 90
80 goto 120
90 get_time(T_v2)
100 T_v3T_v2-T_v1
110 if T_v3>=4 then goto 130
120 goto 70
130 get_time(T_v1)
140 get_time(T_v2)
150 T_v3T_v2-T_v1
160 if T_v3>=2 then goto 180
170 goto 190
180 ifsig -1010,,, then goto 200
190 goto 140
200 signal 1
210 signal -2
220 goto 10
230 get_time(T_v1)
240 signal -1
250 signal 2
260 ifsig 1009,,, then goto 280
270 goto 310
280 get_time(T_v2)
290 T_v3T_v2-T_v1
300 if T_v3>=4 then goto 320
310 goto 260
320 get_time(T_v1)
330 get_time(T_v2)
340 T_v3T_v2-T_v1
350 if T_v3>=2 then goto 370
360 goto 380
370 ifsig -1009,,, then goto 390
380 goto 330
390 signal 1
```

```
400 signal -2
410 goto 10
```

D.2 VAL II Final Code

On this subsection, it is presented the VAL II final code that was generated automatically by the GIRo environment, using the VAL II robot language inputs that are described in chapter 9. During this final translation process, some errors, or warnings, can occur, and they are stored on an errors file.

For this case study, the generation process was not successful, because there are errors, that are written below.

```
WARNING: NUMBER data type was not defined
WARNING: POINT data type was not defined
WARNING: STRING data type was not defined
WARNING: INPUT data type was not defined
WARNING: OUTPUT data type was not defined
ERROR: It is not possible to generate code for this target language.
       There is no TIME data type on it.
ERROR/WARNING: The get_time instruction was not defined! It is assumed
               that it is a target language instruction.
```

These warnings are the same that happened during the translation process into a VAL program, in this same case study, and were explained on subsection D.1.

The generation process was **not** successful, but the automatically generated VAL II final code, for the InteRGIRo representation presented on subsection 9.5.1, can be seen following, with the instructions that caused the errors.

```
10 if sig(1009) goto 40
20 if sig(1010) goto 230
30 goto 10
40 get_time(T_v1)
50 signal -1
60 signal 2
70 if sig(1010) goto 90
80 goto 120
90 get_time(T_v2)
100 T_v3T_v2-T_v1
110 if T_v3>=4 goto 130
120 goto 70
130 get_time(T_v1)
140 get_time(T_v2)
150 T_v3T_v2-T_v1
160 if T_v3>=2 goto 180
170 goto 190
180 if sig(-1010) goto 200
190 goto 140
200 signal 1
```

```
210 signal -2
220 goto 10
230 get_time(T_v1)
240 signal -1
250 signal 2
260 if sig(1009) goto 280
270 goto 310
280 get_time(T_v2)
290 T_v3T_v2-T_v1
300 if T_v3>=4 goto 320
310 goto 260
320 get_time(T_v1)
330 get_time(T_v2)
340 T_v3T_v2-T_v1
350 if T_v3>=2 goto 370
360 goto 380
370 if sig(-1009) goto 390
380 goto 330
390 signal 1
400 signal -2
410 goto 10
```

D.3 Melfa Basic III Final Code

On this subsection, it is presented the Melfa Basic III final code that was generated automatically by the GIRo environment, using the Melfa Basic III robot language inputs, that are described in chapter 9. During this final translation process, some errors, or warnings, can occur, and they are stored on an errors file.

For this case study, the generation process was not successful, because there are errors, that are written below.

```
WARNING: NUMBER data type was not defined
WARNING: POINT data type was not defined
WARNING: INPUT data type was not defined
WARNING: OUTPUT data type was not defined
WARNING: Target language do not have IF statements.
         You must verify if it has instructions that evaluate
         the relational operators.
ERROR: It is not possible to generate code for this target language.
       There is no TIME data type on it.
ERROR/WARNING: The get_time instruction was not defined! It is assumed
               that it is a target language instruction.
```

The first five warnings are the same that happened during the translation process into a Melfa Basic III program for the simple case study A.2, and were explained on that subsection.

Also, the last two errors / warnings are the same that happened during the translation process into a VAL program for this same case study D.1, and were also explained on that subsection.

So, the generation process was **not** successful, but the automatically generated Melfa Basic III final code for the InteRGIRo representation presented on subsection 9.5.1 can be seen following, with the instructions that caused the errors.

```
10 IN 9
20 NE 0 60
30 IN 10
40 NE 0 270
50 GT 10
60 get_time(T_v1)
70 OT 0 1
80 OT 1 2
90 IN 10
100 NE 0 120
110 GT 150
120 get_time(T_v2)
130 T_v3T_v2-T_v1
140 T_v3>=4 GT 160
150 GT 90
160 get_time(T_v1)
170 get_time(T_v2)
180 T_v3T_v2-T_v1
190 T_v3>=2 GT 210
200 GT 230
210 IN 10
220 EQ 0 240
230 GT 170
240 OT 1 1
250 OT 0 2
260 GT 10
270 get_time(T_v1)
280 OT 0 1
290 OT 1 2
300 IN 9
310 NE 0 330
320 GT 360
330 get_time(T_v2)
340 T_v3T_v2-T_v1
350 T_v3>=4 GT 370
360 GT 300
370 get_time(T_v1)
380 get_time(T_v2)
390 T_v3T_v2-T_v1
400 T_v3>=2 GT 420
410 GT 440
420 IN 9
430 EQ 0 450
440 GT 380
450 OT 1 1
460 OT 0 2
```

```
470 GT 10
480 ED
```

D.4 Pascal Final Code

On this subsection, it is presented the Pascal final code that was generated automatically by the GIRo environment, using the Pascal robot language inputs that are described in chapter 9. During this final translation process, some errors, or warnings, can occur, and they are stored on an errors file.

For this case study, the generation process was successful, despite the existence of one warning, that are written below.

```
WARNING: POINT data type was not defined
```

This warning is the same that happened during the translation process into a Pascal program for the simple case study, and was also explained on the respective subsection (A.4).

The generation process was successful, and the automatically generated Pascal final code for the InteRGIRo representation presented on subsection 9.5.1 can be seen following.

```
program teste;
uses crt;
label
   E1, E2, Estado_E2, Estado_E2_1if, Estado_E2_else1, E4, E4T, E4_1if, E4_else1,
   E6, E3, Estado_E3, Estado_E3_1if, Estado_E3_else1, E5, E5T, E5_1if, E5_else1, E7;
var
     IS1: integer;
     IS2: integer;
     T_v1: longint;
     Ocancela: integer;
     Osinal: integer;
     T_v2: longint;
     T_v3: longint;
begin
    clrscr;
E1: writeln('Insert a value for the sensor IS1');
    readln(IS1);
    if (IS1<>0) then goto E2;
    writeln('Insert a value for the sensor IS2');
    readln(IS2);
    if (IS2<>0) then goto E3;
    goto E1;
E2: writeln('Insert time, in seconds, for the variable T_v1.');
    readln(T_v1);
    Ocancela:=0;
    writeln('Output sensor Ocancela was resetted.');
    ;
    Osinal:=1;
    writeln('Output sensor Osinal was setted.');
    ;
Estado_E2: writeln('Insert a value for the sensor IS2');
    readln(IS2);
    if (IS2<>0) then goto Estado_E2_1if;
    goto Estado_E2_else1;
Estado_E2_1if: writeln('Insert time, in seconds, for the variable T_v2.');
    readln(T_v2);
    T_v3:=T_v2-T_v1;
    if (T_v3>=4) then goto E4;
Estado_E2_else1: goto Estado_E2;
E4: writeln('Insert time, in seconds, for the variable T_v1.');
    readln(T_v1);
E4T: writeln('Insert time, in seconds, for the variable T_v2.');
    readln(T_v2);
    T_v3:=T_v2-T_v1;
    if (T_v3>=2) then goto E4_1if;
    goto E4_else1;
E4_1if: writeln('Insert a value for the sensor IS2');
    readln(IS2);
    if (IS2=0) then goto E6;
E4_else1: goto E4T;
E6: Ocancela:=1;
    writeln('Output sensor Ocancela was setted.');
    ;
    Osinal:=0;
    writeln('Output sensor Osinal was resetted.');
    ;
    goto E1;
E3: writeln('Insert time, in seconds, for the variable T_v1.');
    readln(T_v1);
    Ocancela:=0;
    writeln('Output sensor Ocancela was resetted.');
    ;
    Osinal:=1;
    writeln('Output sensor Osinal was setted.');
    ;
Estado_E3: writeln('Insert a value for the sensor IS1');
    readln(IS1);
    if (IS1<>0) then goto Estado_E3_1if;
    goto Estado_E3_else1;
Estado_E3_1if: writeln('Insert time, in seconds, for the variable T_v2.');
    readln(T_v2);
    T_v3:=T_v2-T_v1;
    if (T_v3>=4) then goto E5;
Estado_E3_else1: goto Estado_E3;
E5: writeln('Insert time, in seconds, for the variable T_v1.');
    readln(T_v1);
E5T: writeln('Insert time, in seconds, for the variable T_v2.');
    readln(T_v2);
    T_v3:=T_v2-T_v1;
    if (T_v3>=2) then goto E5_1if;
    goto E5_else1;
E5_1if: writeln('Insert a value for the sensor IS1');
    readln(IS1);
    if (IS1=0) then goto E7;
E5_else1: goto E5T;
E7: Ocancela:=1;
    writeln('Output sensor Ocancela was setted.');
    ;
    Osinal:=0;
    writeln('Output sensor Osinal was resetted.');
    ;
    goto E1;
    repeat until keypressed;
end.
```

D.5 C Final Code

On this subsection, it is presented the C final code that was generated automatically by the GIRo environment, using the C robot language inputs that are described in chapter 9. During this final translation process, some errors, or warnings, can occur, and they are stored on an errors file.

For this case study, the generation process was successful, despite the existence of one warning, that are written below.

```
WARNING: POINT data type was not defined
```

This warning is the same that happened during the translation process into a Pascal program for the simple case study, and was also explained on the respective subsection (A.4).

The generation process was successful, and the automatically generated C final code for the InteRGIRo representation presented on subsection 9.5.1 can be seen following.

```
#include "stdio.h"
#include "conio.h"
#include "string.h"
#include "time.h"
#define TRUE 1
#define FALSE 0
void main()
{
     int IS1;
     int IS2;
     int T_v1;
     int Ocancela;
     int Osinal;
     int T_v2;
     int T_v3;
```

```
    clrscr();
E1: printf("%s\n","Insert a value for the sensor IS1");
    do { scanf("%i",&IS1);
    } while ((IS1<0) || (IS1>10));
    if (IS1!=0) goto E2;
    printf("%s\n","Insert a value for the sensor IS2");
    do { scanf("%i",&IS2);
    } while ((IS2<0) || (IS2>10));
    if (IS2!=0) goto E3;
    goto E1;
E2: T_v1 =time(NULL);
    Ocancela=0;
    printf("%s\n","Output sensor Ocancela was resetted.");
    ;
    Osinal=1;
    printf("%s\n","Output sensor Osinal was setted.");
    ;
Estado_E2: printf("%s\n","Insert a value for the sensor IS2");
    do { scanf("%i",&IS2);
    } while ((IS2<0) || (IS2>10));
    if (IS2!=0) goto Estado_E2_1if;
    goto Estado_E2_else1;
Estado_E2_1if: T_v2 =time(NULL);
    T_v3=T_v2-T_v1;
    if (T_v3>=4) goto E4;
Estado_E2_else1: goto Estado_E2;
E4: T_v1 =time(NULL);
E4T: T_v2 =time(NULL);
    T_v3=T_v2-T_v1;
    if (T_v3>=2) goto E4_1if;
    goto E4_else1;
E4_1if: printf("%s\n","Insert a value for the sensor IS2");
    do { scanf("%i",&IS2);
    } while ((IS2<0) || (IS2>10));
    if (IS2==0) goto E6;
E4_else1: goto E4T;
E6: Ocancela=1;
    printf("%s\n","Output sensor Ocancela was setted.");
    ;
    Osinal=0;
    printf("%s\n","Output sensor Osinal was resetted.");
    ;
    goto E1;
E3: T_v1 =time(NULL);
    Ocancela=0;
    printf("%s\n","Output sensor Ocancela was resetted.");
    ;
    Osinal=1;
    printf("%s\n","Output sensor Osinal was setted.");
    ;
Estado_E3: printf("%s\n","Insert a value for the sensor IS1");
```

```
    do { scanf("%i",&IS1);
    } while ((IS1<0) || (IS1>10));
    if (IS1!=0) goto Estado_E3_1if;
    goto Estado_E3_else1;
Estado_E3_1if: T_v2 =time(NULL);
    T_v3=T_v2-T_v1;
    if (T_v3>=4) goto E5;
Estado_E3_else1: goto Estado_E3;
E5: T_v1 =time(NULL);
E5T: T_v2 =time(NULL);
    T_v3=T_v2-T_v1;
    if (T_v3>=2) goto E5_1if;
    goto E5_else1;
E5_1if: printf("%s\n","Insert a value for the sensor IS1");
    do { scanf("%i",&IS1);
    } while ((IS1<0) || (IS1>10));
    if (IS1==0) goto E7;
E5_else1: goto E5T;
E7: Ocancela=1;
    printf("%s\n","Output sensor Ocancela was setted.");
    ;
    Osinal=0;
    printf("%s\n","Output sensor Osinal was resetted.");
    ;
    goto E1;
    getchar(); getchar();
}
```

Appendix E

A Factorial Case Study

In this appendix, it is presented the generated final code for the VAL, VAL II, Rapid, Melfa Basic III, and Luna programming languages, based on the Factorial case study, presented on 9.6.

E.1 VAL Final Code

On this subsection, it is presented the VAL final code that was generated automatically by the GIRo environment, using the VAL robot language inputs that are described in chapter 9. During this final translation process, some errors, or warnings, can occur, and they are stored on an errors file.

For this case study, it can be said that the generation process was successful, despite the existence of errors and warnings, that are written below.

```
WARNING: NUMBER data type was not defined
WARNING: POINT data type was not defined
WARNING: STRING data type was not defined
WARNING: INPUT data type was not defined
WARNING: OUTPUT data type was not defined
ERROR/WARNING: There is no clear_screen instruction on the target
               language. It was discarded on the final generated code.
```

The first five warnings are the same that happened during the translation process into a VAL program for the simple case study 9.2.3, and were explained on that subsection.

The last error / warning indicates that the InteRGIRo program uses an operation (in this case, a *clear_ screen* one) that does not exist on the target language, and it is discarded. This message could be viewed as an error, or a warning. It depends on the instruction, if it is really necessary to be used on the target language, or not, and the user might decide about its importance. If this instruction is really necessary, obviously that it can be viewed as an error. If not, it is just a warning that indicates that such instruction was discarded. In this case, such situation can be viewed as a warning, because *clear_ screen* is not really a needed instruction to be translated to the target code, and can be discarded without influencing the goal of such program.

```
10 seti N_v1=0
20 seti N_v2=1
30
40 type/N, "Insert a positive number to see its factorial."
50 prompt " ", N_v1
60 if N_v1<0 then goto 100
70 if N_v1>0 then goto 120
80 if N_v1=0 then goto 170
90 goto 60
100 type/N, "There is no factorial for a negative number."
110 goto 200
120 seti N_v2=N_v2*N_v1
130 seti N_v1=N_v1-1
140 if N_v1>0 then goto 120
150 if N_v1=0 then goto 170
160 goto 140
170 type/N, "The factorial is:"
180 type/N, N_v2
190 goto 200
200
```

E.2 VAL II Final Code

On this subsection, it is presented the VAL II final code that was generated automatically by the GIRo environment, using the VAL II robot language inputs that are described in chapter 9. During this final translation process, some errors, or warnings, can occur, and they are stored on an errors file.

For this case study, it can be said that the generation process was successful, despite the existence of errors and warnings, that are written below.

```
WARNING: NUMBER data type was not defined
WARNING: POINT data type was not defined
WARNING: STRING data type was not defined
WARNING: INPUT data type was not defined
WARNING: OUTPUT data type was not defined
ERROR/WARNING: There is no clear_screen instruction on the target
               language. It was discarded on the final generated code.
```

These warnings are the same that happened during the translation process into a VAL program, and were explained on subsection E.1.

The generation process was successful, and the automatically generated VAL II final code for the InteRGIRo representation presented on subsection 9.6.1 can be seen following.

```
10 N_v1=0
20 N_v2=1
```

```
30
40 type/N, "Insert a positive number to see its factorial."
50 prompt " ", N_v1
60 if N_v1<0 goto 100
70 if N_v1>0 goto 120
80 if N_v1==0 goto 170
90 goto 60
100 type/N, "There is no factorial for a negative number."
110 goto 200
120 N_v2=N_v2*N_v1
130 N_v1=N_v1-1
140 if N_v1>0 goto 120
150 if N_v1==0 goto 170
160 goto 140
170 type/N, "The factorial is:"
180 type/N, N_v2
190 goto 200
200
```

E.3 Rapid Final Code

On this subsection, it is presented the RAPID final code that was generated automatically by the GIRo environment, using the RAPID robot language inputs, that are described in chapter 9. During this final translation process, some errors, or warnings, can occur, and they are stored on an errors file.

For this case study, it can be said that the generation process was successful, despite the existence of errors and warnings, that are written below.

```
WARNING: INPUT data type was not defined
WARNING: OUTPUT data type was not defined
ERROR/WARNING: There is no clear_screen instruction on the target
               language. It was discarded on the final generated code.
```

The first two warnings are the same that happened during the translation process into a Rapid program for the simple case study, and were explained on subsection 9.2.5, while the last one is the same that happened during the translation process into a VAL program, for this case study, and was explained on subsection E.1.

Above, it is presented the automatically generated RAPID final code for the InteRGIRo representation presented on subsection 9.6.1.

```
MODULE teste
     VAR num N_v1;
     VAR num N_v2;
VAR clock clock1;
PROC main()
    ClrReset clock1;
    ClkStart clock1;
E1: N_v1:=0;
```

```
    N_v2:=1;
    ;
    TPWrite "" "Insert a positive number to see its factorial.";
    TPReadNum N_v1, "";
Estado_E1: IF N_v1<0 GOTO E2;
    IF N_v1>0 GOTO E3;
    IF N_v1=0 GOTO E4;
    GOTO Estado_E1;
E2: TPWrite "" "There is no factorial for a negative number.";
    GOTO E_fim;
E3: N_v2:=N_v2*N_v1;
    N_v1:=N_v1-1;
Estado_E3: IF N_v1>0 GOTO E3;
    IF N_v1=0 GOTO E4;
    GOTO Estado_E3;
E4: TPWrite "" "The factorial is:";
    TPWrite "" N_v2;
    GOTO E_fim;
E_fim: ;
ENDPROC
ENDMODULE
```

E.4 Melfa Basic III Final Code

On this subsection, it is presented the Melfa Basic III final code that was generated automatically by the GIRo environment, using the Melfa Basic III robot language inputs, that are described in chapter 9. During this final translation process, some errors, or warnings, can occur, and they are stored on an errors file.

For this case study, the generation process was not successful, because there are errors, that are written below.

```
WARNING: NUMBER data type was not defined
WARNING: POINT data type was not defined
WARNING: INPUT data type was not defined
WARNING: OUTPUT data type was not defined
WARNING: Target language do not have IF statements.
         You must verify if it has instructions that evaluate
         the relational operators.
ERROR/WARNING: There is no clear_screen instruction on the target
               language. It was discarded on the final generated code.
ERROR: It is not possible to generate code for this target language.
       There is no STRING data type on it.
ERROR/WARNING: The write instruction was not defined! It is assumed
               that it is a target language instruction.
ERROR: It is not possible to generate code for this target language.
       There is no read instruction on its NUMBER data type.
ERROR/WARNING: The read instruction was not defined! It is assumed
               that it is a target language instruction.
```

```
ERROR: It is not possible to generate code for this target language.
       There is no write instruction on its NUMBER data type.
```

The first five warnings are the same that happened during the translation process into a Melfa Basic III program for the simple case study A.2, and were explained on that subsection.

The next one is the same that happened during the translation process into a VAL program, for this case study, and was explained on subsection E.1. The *clear_screen* instruction is not really needed and it can be discarded.

The next two errors / warnings was also explained on subsection D.3, and says that it is not possible to generate final code because there is no STRING data type, and its operations, on the target language. After, some other errors are presented, showing that it is not possible to generate final code because some instructions (described on each error message) simply do not exist on the target language.

So, the generation process was **not** successful, but the automatically generated Melfa Basic III final code for the InteRGIRo representation presented on subsection 9.6.1 can be seen following, with the instructions that caused the errors.

```
10 cp 0
20  cl 1
30 cp 1
40  cl 2
50
60 write("Insert a positive number to see its factorial.")
70 read(1)
80 CP 1
90 SM 0 150
100 CP 1
110 LG 0 170
120 CP 1
130 EQ 0 620
140 GT 80
150 write("There is no factorial for a negative number.")
160 GT 650
170 cp 2
180 cl 3
190 cl 5
200 cp 1
210 cl 4
220 eq 0 340
230  eq 1 360
240 cp 5
250 cl 6
260  eq 0 310
270 ic 3
280 dc 6
290 cp 6
300 gt 260
310  dc 4
```

```
320 cp 4
330 gt 230
340  cp 4
350 cl 3
360
370 cp  3
380  cl 2
390 cp 1
400 cl 3
410 sc 4 1
420 cp 4
430 sm 0 490
440  eq 0 540
450 dc 3
460 dc 4
470 cp 4
480 gt 440
490  eq 0 540
500 ic 3
510 dc 4
520 cp 4
530 gt 490
540
550 cp 3
560  cl 1
570 CP 1
580 LG 0 170
590 CP 1
600 EQ 0 620
610 GT 570
620 write("The factorial is:")
630 write(2)
640 GT 650
650
660 ED
```

E.5 Luna Final Code

On this subsection, it is presented the Luna final code that was generated automatically by the GIRo environment, using the Luna robot language inputs that are described in chapter 9. During this final translation process, some errors, or warnings, can occur, and they are stored on an errors file.

For this case study, the generation process was not successful, because there are errors, that are written below.

```
WARNING: TIME data type was not defined
WARNING: INPUT data type was not defined
WARNING: OUTPUT data type was not defined
```

```
ERROR/WARNING: There is no clear_screen instruction on the target
               language. It was discarded on the final generated code.
ERROR: It is not possible to generate code for this target language.
       There is no STRING data type on it.
ERROR/WARNING: The write instruction was not defined! It is assumed
               that it is a target language instruction.
```

The first three warnings are the same that happened during the translation process into a Luna program for the simple case study A.3, and were explained on that subsection.

The next one is the same that happened during the translation process into a VAL program, for this case study, and was explained on subsection E.1. The *clear_screen* instruction is not really needed and it can be discarded.

The last two errors / warnings was also explained on subsection D.1, and says that it is not possible to generate final code because there is no STRING data type, and its operations, on the target language.

So, the generation process was **not** successful, but the automatically generated Luna final code for the InteRGIRo representation presented on subsection 9.6.1 can be seen following, with the instructions that caused the errors.

```
10     INT : N_v1
20     INT : N_v2
30 N_v1=0
40 N_v2=1
50
60 write("Insert a positive number to see its factorial.")
70 READ(N_v1)
80 IF N_v1<0 GO 120
90 IF N_v1>0 GO 140
100 IF N_v1=0 GO 190
110 GO 80
120 write("There is no factorial for a negative number.")
130 GO 220
140 N_v2=N_v2*N_v1
150 N_v1=N_v1-1
160 IF N_v1>0 GO 140
170 IF N_v1=0 GO 190
180 GO 160
190 write("The factorial is:")
200 WRITE(N_v2)
210 GO 220
220
230 END
```