A Grafcet Interface for the Programming of Different Industrial Robots

Gustavo V. Arnold
Department of Informatics
University of Minho
Braga, Portugal
gva@di.uminho.pt

Pedro R. Henriques
Department of Informatics
University of Minho
Braga, Portugal
prh@di.uminho.pt

Jaime C. Fonseca
Department of Industrial Electronic
University of Minho
Braga, Portugal
Jaime.Fonseca@dei.uminho.pt

ABSTRACT

This paper presents the current development stage of an approach to industrial robot programming, that includes: a truly high level and declarative language; an easy-to-use front-end; an intermediate representation; an automatic generator of the robot code generators. This approach focus on the modelling of the system, rather than on the robot. So, it will improve the programming and maintenance tasks, allowing the reuse of source code.

Keywords

Compilers, Graphical Interface, Industrial Robots Programming Language, Code Generator

1. INTRODUCTION

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. The forerunner languages, such as AML [26] or AL [19], have now been superseded by elaborated robot languages like ABB Rapid [9]. 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 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 various 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, 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.

So, we proposed an integrated, formal and high-level approach to industrial robot programming [5], that would solve the above problems. One component of this approach is a friendly graphical interface, based on the Grafcet specification diagram. This interface will be responsible not only for facilitating de programming task, but also to translate this diagram specification into languages that belongs to different levels of our approach. The aim of this paper is to discuss that graphical interface based on the Grafcet, and the associated translator to RS language [28] (a reactive systems programming language).

In the following section, some theoretical aspects (like our approach to industrial robot programming, in section 2, and the Grafcet in section 3), and some implemented components (the graphical interface in section 4, and some developed translators in section 5) are presented. At the end (section 6), appear the conclusions and future works.

2. OUR APPROACH TO INDUSTRIAL ROBOT PROGRAMMING

```
                  Problem
                     |
                     V
              High level Model
                     |
                     V
                 Front-end
                     |
                     V
         Declarative representation
                     |
                     V
     Declarative intermediate translator
                     |
                     V
         Intermediate representation
        /            |            \
       /             |             \
      /              |              \
    Code           Code      ...     Code
 generator 1   generator 2       generator N
     |               |                |
     V               V                V
  Robot 1         Robot 2    ...   Robot N
```

Figure 1: Proposed approach to industrial robots programming

Today, the industrial robot programming task is done basically in two ways:

1. The programmer in charged of the task can use some modelling technique but, instead of thinking only about the problem, it is necessary to think about the robot that will run the program, and about its programming language. Both the robot and the language will limitate the specification of the problem; moreover it is not possible to reuse the same program in a different robot.

2. The programmer uses some graphical development environment, where 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. Even with this functionalities, these tools do not solve the problem of programming the robot to interact with its environment.

The industrial robot programming languages did not evolved in the same manner as the computer languages. Those languages, and environments, have some drawbacks:

- The typical languages are imperative, low-level or structured. Both of them 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.

- Each industrial robot has its own programming language, which make difficult or even impossible, to reuse the source code.

To solve these problems, we proposed an approach, described diagrammatically in figure 1. That process should

```
 Specification of robot N
            |
            V
 Code generator generator
            |
            V
 Code generator for robot N
```

Figure 2: Automatic generator of code generators.

be supported by the following languages and tools that compose the architecture we want to defend.

- a truly high level and declarative language (Grafcet, RS)

- an easy-to-use front-end

- an intermediate representation (DIR)

- the translators to RS and to DIR

- the code generator

- an automatic generator of the robot code generators

- a robot specification language

At the top, there is the problem to be solved. It will be used an adequate modelling technique, responsible for decompose the problem into simple problems, that would be easily programmed; formal models are employed to describe data structures and operations necessary to solve the subproblems. To describe formally the overall problem and the subproblems, a truly high level language, closed to the specification instead of the robot, should be used. Then we advocate the use of an easy-to-use compiler front-end, like Grafcet, that can interpret the specification language and generate a declarative description, that can be translated into an intermediate description for the program specified. An intermediate representation is used because the front-end must be focused on the specification of the problem, and not on the robot. So, there will be another component responsible for translating this intermediate code into the code of a robot. Because this compiler back-end is specific for a single robot, it is necessary to have a lot of back-ends, each of them adapted to a specific target (robot's architecture and machine code). To create these code generators, it will be used an automatic generator, as can be seen in figure 2. This tool, based on the known intermediate representation, and on the robot specification, that must be included someway, will produce an optimal code generator for the specified robot.

These components will be better presented in the following subsection:

2.1 Modeling Techniques

The use of an adequate modelling technique will facilitate 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. One of them, used in the development of mobile robots, the Subsumption Architecture [10], was used to model a manufacturing cell, composed by two robots and some others components [3]. The subsumption architecture was the first behavior based modelling technique and, even it had been created to develop mobile robots, they can be used, as a high level abstraction, to model industrial applications.

One well-known description language for industrial automation applications is the Grafcet [27]. In our approach, the Grafcet was chosen because it is a nice diagrammatic specification very used on the automation areas, and because the RS language supports completely the Grafcet specifications.

2.2 Compiler 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 the modelling language adopted was Grafcet, we included in the FE a Graphical interface to aid editing the Grafcet visual description. The FE then generates the corresponding textual description using the declarative RS language.

2.3 Declarative Language

As it was said before, it is important to use a modelling technique, but is also important to have some language that would allow the programmer to express exactly what he intends to do.

Such a language 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 languages (proposed some years ago [28]) 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 [20], or controlling a simulated manufacturing cell [3], in order to assess the use of RS in that context.

2.4 Intermediate Representation

In traditional compilers[18], the interface between FE and BE is an intermediate representation (IR) that should be independent of source and target languages; see for instance the well known RTL language [16].

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 our case, the FE has one more stage because the input is a visual specification. First it translates the Grafcet specification into RS, and then translates that source text into the IR.

The IR that will be used is the one proposed in the context of the project Dolphin, the so called DIR (Dolphin Intermediate Representation) that sprang out from the previous BEDS project [17]. This project deals with optimization and code generation tools, aiming to offer a framework to build optimized code generators for different machines based on an universal intermediate program representation; industrial robotics is one of its applications. The universal intermediate program representation proposed is based on previous work on back-ends generator [17].

2.5 Automatic Generator 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[14, 15] 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 [11] and BURG [13, 21]. In this field, also the New Jersey machine-code toolkit[22, 23, 24] should be referred as an important contribution. Other important work concerned with the retargeting of C compilers was discussed in [12] and [25].

3. GRAFCET

Because Grafcet [27] is a well known specification diagram for industrial automation applications, it was decided to use 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 [1]:

- *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.

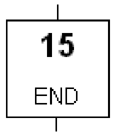

Figure 3: Example of Grafcet's step.

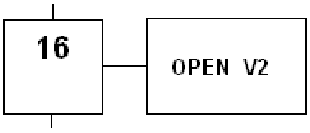

Figure 4: Example of Grafcet's step with action.

3.1 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 3). 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 4. The 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).

3.2 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 changing the state of the system. Each condition is a predicate applied to exterior informations, auxiliary variables, ant the state of others steps. Some examples of transitions can be seen on figure 6.

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 7.

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

3.3 Alternatives and Simultaneous Sequences

When one transition is linked to two or more steps, like that in figure 8, it is called *simultaneous sequence*, which means that all the following steps will be executed simultaneously, until the steps are linked to only one transition. This kind of sequence is represented by a double horizontal line.

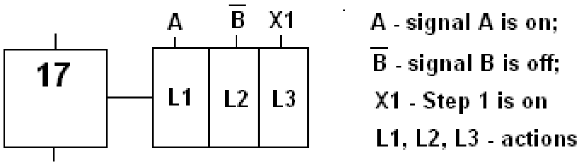

Figure 5: Example of Grafcet's step with action and conditions.

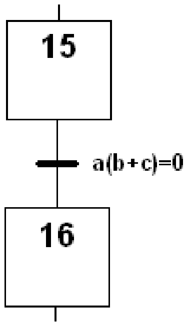

Figure 6: Example of Grafcet's transition.

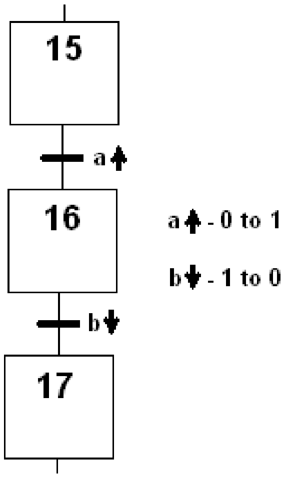

Figure 7: Example of Grafcet's transition that changes the state of variables.

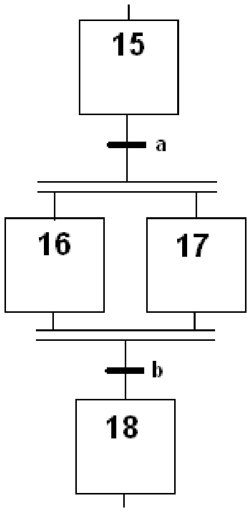

Figure 8: Example of Grafcet's simultaneous sequence.

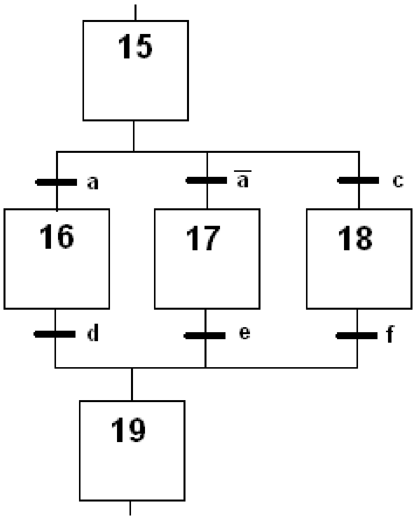

Figure 9: Example of Grafcet's alternative sequence.

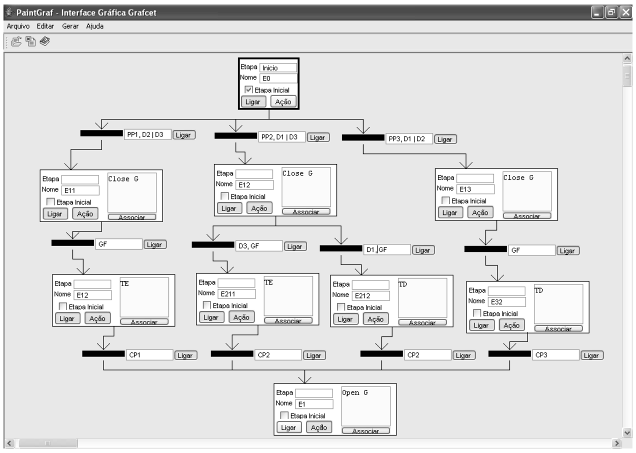

Figure 10: Graphical interface based on Grafcet.

Alternatives sequences are those where a step is linked to more then one transition. It means that, the following steps will only be executed if the transition actually happens (figure 9).

4. GRAPHICAL INTERFACE

The interface that was implemented represents all the concepts, behaviors and features of the Grafcet diagram, and have a similar appearance. An example of this interface can be seen on figure 10

In the rest of this section, we explain how those Grafcet constructors were implemented in our editor.

4.1 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 line around the square becomes thicken. There are also two buttons: 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 11). When this *action* button is pressed, the square is augmented, in the right side, and appear a text box, where the user can enter the actions, and another buttons, *associate*, responsible for storing this actions when it is written, linked to the current step (figure 12).

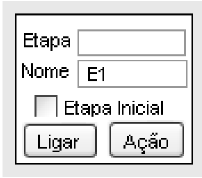

Figure 11: Step graphic representation.

Figure 12: Step and actions graphic representation.

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 ";")+ "}" )+
condition -> cond | condition op cond
cond -> iv | "$\sim$" iv | n\_step
op -> "," | "|"
action -> command
```

where the conditions are between the `()` 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, and `n_step` corresponds to steps that must be active. 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 between {}. It is necessary to use a `;` at the end of each action.

4.2 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 13).

Each condition should be written considering the following grammar, which makes possible to represent all the func-

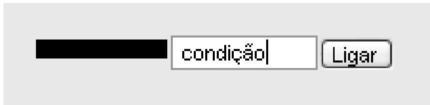

Figure 13: Transition graphic representation.

tions proposed by Grafcet:

```
condition -> expr | condition op expr
expr -> v "+" | v "-" | time | "(" conditon ")"
time -> t "/" n\_step "/" seconds
op -> "," | "|"
```

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

4.3 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. The user can move each object to any position of the window.

5. TRANSLATOR

At this moment, there is only one translator ready, that is the one that translates the Grafcet graphical representation into an RS program. The Grafcet elements implemented on the graphical interface were discussed on previous sections, while the RS language and automaton can be seen on [28, 6, 4, 7, 3, 2, 8]

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.

To illustrate the idea, figure 14 shows the generated RS program for the sample Grafcet specification presented in figure 10.

5.1 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. The rules are executed if the signals (internal or input) on the left side of the rule are on. The internal signals can be signalized internally, while the input ones are set by the external environment.

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. The output signals are responsible for sending commands to the external environment (as a consequence of the `emit` command).

5.2 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.

5.3 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.

5.4 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;
```

```
rs_prog testando_programaRS:
  input :PP1, D2, D3, GF, CP1, PP2, D1, CP2,
         PP3, CP3;
  output:Close G, TE, Open G, TD;
  module testando_modulo:
    input   :PP1, D2, D3, GF, CP1, PP2, D1, CP2,
             PP3, CP3;
    output  :Close G, TE, Open G, TD;
    t_signal:EE0, EE11, EE12, EE1, EE211, EE212,
             EE13, EE32;
    initially: up(EE0);
    PP1, D2, EE0 ==> emit(Close G), up(EE11);
    PP1, D3, EE0 ==> emit(Close G), up(EE11);
    GF, EE11 ==> emit(TE), up(EE12);
    CP1, EE12 ==> emit(Open G), up(EE1);
    PP2, D1, EE0 ==> emit(Close G), up(EE12);
    PP2, D3, EE0 ==> emit(Close G), up(EE12);
    D3, GF, EE12 ==> emit(TE), up(EE211);
    CP2, EE211 ==> emit(Open G), up(EE1);
    D1, GF, EE12 ==> emit(TD), up(EE212);
    CP2, EE212 ==> emit(Open G), up(EE1);
    PP3, D1, EE0 ==> emit(Close G), up(EE13);
    PP3, D2, EE0 ==> emit(Close G), up(EE13);
    GF, EE13 ==> emit(TD), up(EE32);
    CP3, EE32 ==> emit(Open G), up(EE1);
  end module;
end rs_prog.
```

Figure 14: Example of a generated RS program.

```
    go to the following transitions for this next
          step and execute the
          Transition_evaluating_function;
  }
}
```

6. CONCLUSIONS

This paper presented a graphical interface based on the Grafcet, and the respective translator into RS language. This work is a part of an approach to industrial robot programming, that will cover all the stages of the programming task, from the modelling of the system until the robot code generation.

The system is still under construction, but the front-end including a graphical editor and the translator to RS language is done. When all of the components of this project end, the approach will have the following features:

- It will be possible to generate code for different robots, based on the same source program;
- It will be available a high-level declarative language to specify the programs;
- It will provide a friendly front-end, to make the programming task easy;
- It will offer a automatic generator for robot code generators.

The following steps are the development of the automaton RS to DIR translator, and the development of the code generator generator.

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, because they do not make easy to program robots as it is to program computers, and there is no reuse of source code.

7. REFERENCES

[1] Grafcet homepage, 2000.
[2] G. V. Arnold. Uma extensão da linguagem rs para o desenvolvimento de sistemas reativos baseados na arquitetura de subsunção. Dissertao de mestrado, CPGCC, UFRGS, Porto Alegre, Brasil, 1998.
[3] G. V. Arnold. Controle de uma clula de manufatura atravs da linguagem RS estendida. In *Anais do I Congresso de Lgica Aplicada Tecnologia (LAPTEC'2000)*, pages 145–163, So Paulo, SP, Brasil, 2000.
[4] G. V. Arnold, P. R. Henriques, and J. C. Fonseca. Uma proposta de linguagem de programao para robtica. In *Anais do VI Simpsio Brasileiro de Linguagens de Programao (SBLP'02)*, pages 134–143, 2002.
[5] G. V. Arnold, P. R. Henriques, and J. C. Fonseca. A development approach to industrial robots programming. In *Proceedings of the 11th International Conference on Advanced Robotics*, volume 1, pages 167–172, Coimbra, Portugal, 2003. Institute for Systems and Robotics, University of Coimbra.
[6] G. V. Arnold, G. R. Librelotto, and P. R. Henriques. Rs compiler. In *Proceedings of the Fourth Congress on Logic Applied to Technology (LAPTEC'2003)*, Marlia, SP, Brasil, 2003.
[7] G. V. Arnold, G. R. Librelotto, P. R. Henriques, and J. C. Fonseca. O uso da linguagem rs em robtica. In *Electrnica e Telecomunicaes*, pages 501–508, So Paulo, SP, Brasil, 2002. Revista do Departamento de Electrnica e Telecomunicaes da Universidade de Aveiro.
[8] 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.
[9] A. F. Automation. *Rapid Reference Manual 3.0*, 1994.
[10] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of robotics and Automation*, RA-2(1):14–23, 1986.
[11] 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.
[12] C. Fraser and D. Hanson. *A Retargetable C Compiler: design and implementation*. Addison Wesley Publishing Company, 1995.
[13] C. Fraser, R. Henry, and T. Proebsting. Burg - fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 27:68 – 76, 1991.

[14] C. W. Fraser. *Automatic Generation of Code Generators*. Phd dissertation, Yale University, New Haven, 1977.

[15] 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.

[16] 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.

[17] 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.

[18] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[19] M. S. Mujtaba, R. Goldman, and T. Binford. Stanford's al robot programming language. *Computers in Mechanical Engineering*, August 1982.

[20] S. J. Piola. Uso da linguagem rs no controle do rob nachi sc15f. Trabalho de Concluso de Curso de Graduao, Departamento de Informtica, UCS, Caxias do Sul, Brasil, 1998.

[21] T. A. Proebsting. Burs automata generation. *ACM Transactions on Programming Languages and Systems*, 17(3):461–486, 1995.

[22] N. Ransey and M. Fernandez. New jersey machine-code toolkit. Technical report, Departament of Computer Science, Princeton University, 1995.

[23] N. Ransey and M. Fernandez. The new jersey machine-code toolkit. In *Proceedings of The USENIX Technical Conference*, pages 289 – 302, New Orleans, 1995.

[24] N. Ransey and M. Fernandez. New jersey machine-code toolkit architecture specifications. Technical report, Departament of Computer Science, Princeton University, 1996.

[25] R. Stallman. Using and porting the gnu compiler collection (gcc). iUniverse.com, Inc, 2000.

[26] R. H. Taylor, P. D. Summers, and J. M. Meyer. Aml: a manufacturing language. *The International Journal of Robotics Research*, 1(3), 1982.

[27] Telemec. *O Grafcet – Diagrama Funcional para Automatismos Sequenciais*. Portugal, 1982.

[28] S. S. Toscani. *RS: Uma Linguagem para Programação de Núcleos Reactivos*. Tese de doutoramento, Depto de Informática, UNL, Lisboa, Portugal, 1993.