# AdaptPack studio translator: translating offline programming to real palletizing robots

*João Pedro Carvalho de Souza*
Faculty of Engineering of University of Porto (FEUP) and INESC TEC, Porto, Portugal
*André Luiz Castro and Luís F. Rocha*
INESC TEC, Porto, Portugal, and
*Manuel F. Silva*
INESC TEC and Departamento de Engenharia Electrotécnica, ISEP – Instituto Superior de Engenharia do Porto, Porto, Portugal

**Abstract**

**Purpose** – This paper aims to propose a translation library capable of generating robots proprietary code after their offline programming has been performed in a software application, named AdaptPack Studio, running over a robot simulation and offline programming software package.

**Design/methodology/approach** – The translation library, named AdaptPack Studio Translator, is capable to generate proprietary code for the Asea Brown Boveri, FANUC, Keller und Knappich Augsburg and Yaskawa Motoman robot brands, after their offline programming has been performed in the AdaptPack Studio application.

**Findings** – Simulation and real tests were performed showing an improvement in the creation, operation, modularity and flexibility of new robotic palletizing systems. In particular, it was verified that the time needed to perform these tasks significantly decreased.

**Practical implications** – The design and setup of robotics palletizing systems are facilitated by an intuitive offline programming system and by a simple export command to the real robot, independent of its brand. In this way, industrial solutions can be developed faster, in this way, making companies more competitive.

**Originality/value** – The effort to build a robotic palletizing system is reduced by an intuitive offline programming system (AdaptPack Studio) and the capability to export command to the real robot using the AdaptPack Studio Translator. As a result, companies have an increase in competitiveness with a fast design framework. Furthermore, and to the best of the author's knowledge, there is also no scientific publication formalizing and describing how to build the translators for industrial robot simulation and offline programming software packages, being this a pioneer publication in this area.

**Keywords** Palletizing, Offline programming, Simulation, Industry 4.0, Industrial application, Robot language translator

**Paper type** Research paper

## 1. Introduction

The current competitive global trade scenario demands fast, efficient, and flexible solutions from the modern industry [1]. This production environment leads to the increasing adoption of robotic solutions in the factory shop floor, aligned with the best practices and concepts of the Industry 4.0 methodology. In particular, the fast-moving consumer goods industrial sector is constantly improved by the usage of robots at the production scale. The adoption of robotic palletizing systems gives to this industrial segment the solution to handle the palletization with high performance and efficiency. However, these solutions are not modular, lack flexibility and are usually focused on one type of robot and product, making them unattractive for companies' investment when considering highly dynamic production environments. Given this, the studies and proposals of techniques that support the fast and efficient development and deployment of modular robotic cells and, that are easily reconfigured, adapted and re-parameterized, are a current trend in robotics. It was to respond to these requests, particularly for the robotized palletizing industry, that the AdaptPack Translator, described here, was developed. The AdaptPack Project, which started at the beginning of 2016, has as its main objective the development of tools, organized in a modular architecture, to expedite the development, programming and the shop floor deployment of palletizing cells with robots from distinct brands.

Focusing on the robot programming procedure, today there exist two main methodologies: online and offline programming. Online programming, the most commonly used, consists of manually teaching movement points to a robot, supervised by a human operator. This approach can be implemented using a Teach Pendant (by guiding the robot using a controller device and then storing the points and movement instructions) or through Lead-through Programming (physically guiding the robot into the workspace and storing the path(s)). These methods depend on the operator's skill, and the development

The current issue and full text archive of this journal is available on Emerald Insight at: **https://www.emerald.com/insight/0143-991X.htm**

Industrial Robot: the international journal of robotics research and application
 [ISSN 0143-991X]
[DOI 10.1108/IR-12-2019-0253]

Received 12 December 2019
Revised 5 March 2020
2 July 2020
Accepted 2 July 2020

and test are time-consuming, expensive, tedious, and repetitive. Furthermore, collisions may occur in case of programming errors, which may damage the robot and its end-effector and the robotic cell equipment. Thus, several works have been presented in the literature, which are an example of studies aiming to facilitate this task (Choi and Lee, 2001; Sugita *et al.*, 2004; Schraft and Meyer, 2006; Ferreira *et al.*, 2016). The offline programming method is commonly based on developing the robot programming in a simulated environment. It allows more complex, fast and flexible solutions, besides the realization of tests in different situations (Pan *et al.*, 2010). This approach consists of a chain of steps, including the 3D modeling of the robotic workcell, trajectory and process planning, simulation, calibration and post-processing (translation of the robot(s) programs to native code of its controller; Pan *et al.*, 2010). One drawback of the offline programming methodology is the post-processing phase, and it concerns the programs translation to the real robots.

There are several simulator and robot brands available on the market and the effort to standardize both their interfaces and the robot programming languages has been quite modest. The Industrial Robot Language (IRL) (Industrial Robot Language DIN Standard 66312, 1996) is an attempted example of standardization that is not really used by any robot manufacturer, according to the authors' knowledge. Therefore, this bottleneck goes against the competitive exigencies of the global market, as it is necessary to program and test different routines, according to the robot used in the real cell. Aiming to help in this issue, some authors proposed a framework for offline robot programming, that generates native code to several robot brands (Bottazzi and Fonseca, 2005; Bruccoleri *et al.*, 2007); however, only some basic commands are translated. Another framework is presented in Freund *et al.* (2001). This tool interprets different native robot languages, improving the simulation step, i.e., it is a native language translator to a simulation language. However, it still requires the knowledge of the proprietary languages.

In what respects tools to expedite the development of robotic cells, the usual approach is the adoption of simulation software applications. Presently, there are two "different classes" of software packages for robot simulation and offline programming of industrial robots: (*i*) there are software applications developed by companies, such as Asea Brown Boveri (ABB) (RobotStudio), and FANUC (FANUC RoboGuide), etc., that develop robots and simulation software specifically for their robots – these software applications only allow to simulate and offline program robots from the company brand; and, (*ii*) there are software applications developed by companies independent of the robot manufacturers, that develop their applications for working with robots from distinct brands and that also allow their offline programming.

Concerning commercial products on the market of this "second" class, Delmia (Dassault Systemes, 2018), RobotExpert (Siemens, 2018), RoboDK (RoboDK, 2018) and Visual components (VC) (Visual Components, 2018) are a few examples of tools that allow the development of offline programming tasks, including the ability to translate its neutral language into proprietary codes of different industrial robots. Delmia and RobotExpert are proprietary simulators, i.e. present "closed environments" that do not offer an API or support to the development of new features and simulations. The RoboDK is more limited regarding the simulation of fully complex scenarios (as the case of palletizing cells) and has a reduced robot model library in comparison with the VC software. These were the main reasons that led to choose to conduct the AdaptPack project using VC.

A literature review and a market consultation was performed when this project started (beginning of 2016), and no tool was found, at the date, able to transform a generic offline robot program in a custom proprietary code for distinct robot brands. Besides VC, that at the date did not had the required translators, RoboDK was also considered then, but its capabilities were still very limited, also in the number of robot models available in its library. Therefore, to comply with the project requirements, the authors had to develop, by its own, a translator able to translate code from the adopted software for the project (VC). In the AdaptPack complete solution, complex codes are generated automatically (Castro *et al.*, 2020), being the program composed by different basic commands, and the focus in the presented paper is the capability to transform generic codes in proprietary codes.

Bearing the above presented ideas on mind, this paper proposes the AdaptPack Studio Translator. This library is focused on the automatic generation of native code to the ABB, FANUC, Keller und Knappich Augsburg (KUKA) and Yaskawa robot brands, using the simulation and offline programming software VC (Visual Components, 2018). This tool is an invisible layer capable of translating all statements generated by a human operator, or by automatic programming methods [such as AdaptPack Studio (Castro *et al.*, 2020)], in an offline programming stage. Therefore, one requirement is not to interfere in the programming layer, increasing the reliability in the translation procedure. The focus of this tool is also to easily convert the programming code, reuse it in other robots and create this code without the need to stop the production cell, i.e. avoid production downtime and productivity losses.

A comparative evaluation of the proprietary robot languages used in this translation library was performed and presented in a previous work (Souza *et al.*, 2019). This library is an extension of the AdaptPack Studio project, presented in Silva *et al.* (2017), Castro *et al.* (2019), whose main objective was the development of a modular framework, aimed at accelerating the design, development and programming of palletizing robot cells.

The remainder of the paper is structured as given. AdaptPack Studio Translator library system description and its explanation is presented in Section 2. The experiments performed in simulation and in the real environment, including the achieved results, are given in Section 3. The conclusion discussing the results and applicability of the proposed library is presented in Section 4.

## 2. AdaptPack studio translator library

As stated above, within the AdaptPack project was developed a modular tool, based on a brand independent robot simulation and offline programming software package (VC), for the automatic development of programs for palletizing robots (named AdaptPack Studio) (Castro *et al.*, 2020) and a module for the translation of the developed programs for distinct robot brands (the AdaptPack Translator).

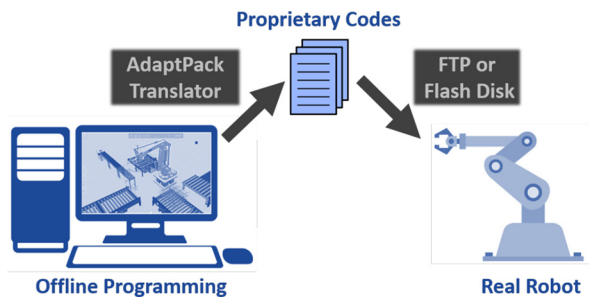
Figure 1 depicts the translation procedure proposed in an offline post-programming phase, using the AdaptPack Studio Translator running on top of the VC software. The user only needs to choose to which language should the simulated program be translated, by clicking on a button in the translator's section (area marked in red on Figure 2), embedded into the VC Program tab menu. After the code generation, the program can be downloaded into the robot controller using a flash disk or the File Transfer Protocol (FTP).

This library is developed in C# as a dynamic-link library for the VC software. Three processing entities compose the translation library architecture: the core, the parser and the translators. Each one of these entities will be explained in the sequel.

### 2.1 Core

The core code is the management entity that starts when any translation button is pressed (Figure 2). First, it checks if a robot has been selected (and which one), identifies the selected language, activates the parser, runs the correspondent translation procedure and exports the generated code to a user predefined folder. Any warnings, information messages and advise windows are also displayed by this core. The flowchart depicted on Figure 3 illustrates this explanation.

**Figure 1** Translation procedure sequence




### 2.2 Parser

The parser is responsible to identify the data and instructions used in the VC program. More specifically, it is a software structure that can get all information from the 3 D simulation environment (robots, grippers, frames, signals, etc.) and programming code (variables and statements) and organize all this data to be, later, used by the translator. For this task, the parser code has a dependency on the .NET API that implements the Robot Sequence Language (RSL) (Visual Components, 2004). With this API it is possible to detect and access all routines and subroutines inside a VC program, besides the instructions, variables and any associated parameters. After the access to all useful information, the parser creates in memory a data structure to be used later by other process entities of the AdaptPack Studio Translator library.

The data structure is basically an ordered stack of data in the memory, with a transcription of all programming instructions and its parameters, based on the 3D simulation model of the palletizing cell. This data is based on three classes: the program, variable and instruction classes. All program data, such as the main structure and subroutines, path and created file name are defined by the program class. The variable class defines the data associated with all variables used in the program and stores it in the data structure. The parser supported variables are of the types: integer, double, string and Boolean. The properties

**Figure 2** AdaptPack translator section in VC's program tab (from left to right: RAPID, KAREL, Inform and KRL selection buttons)

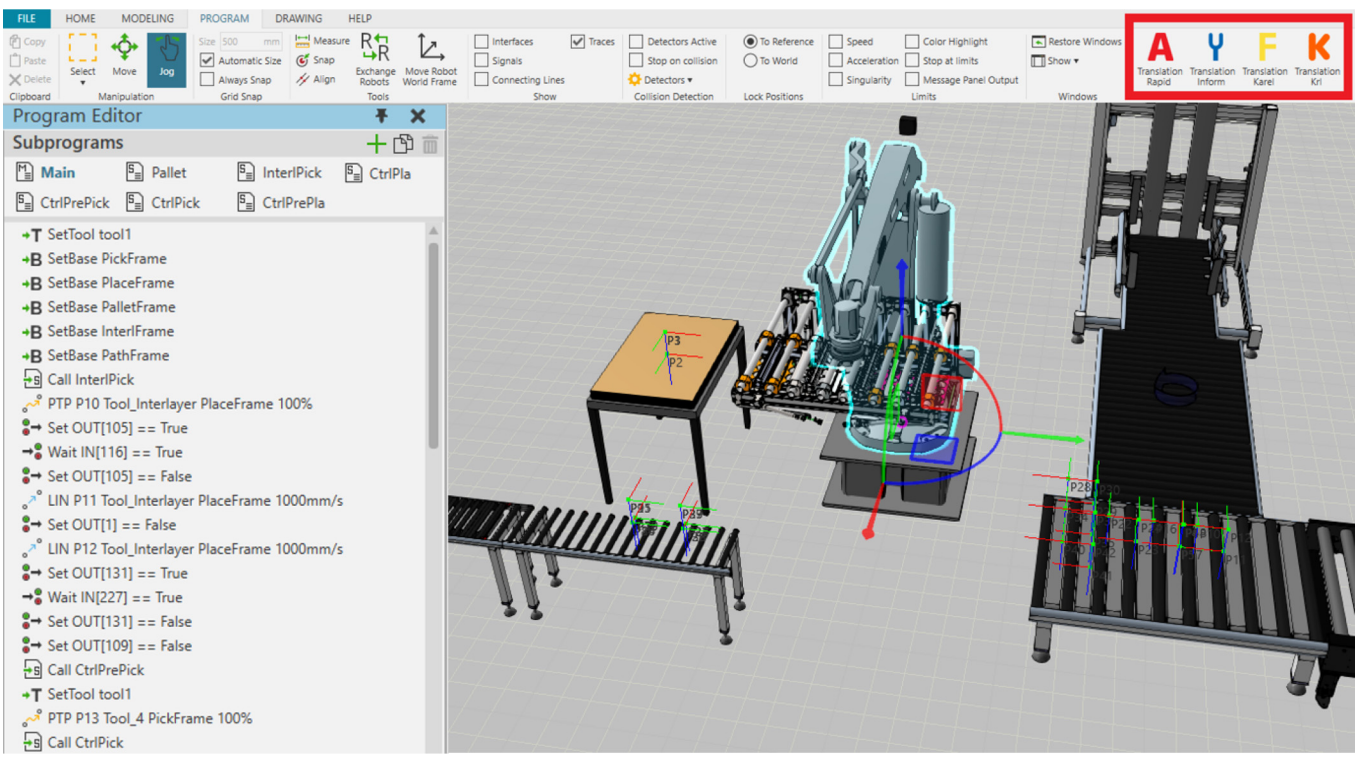
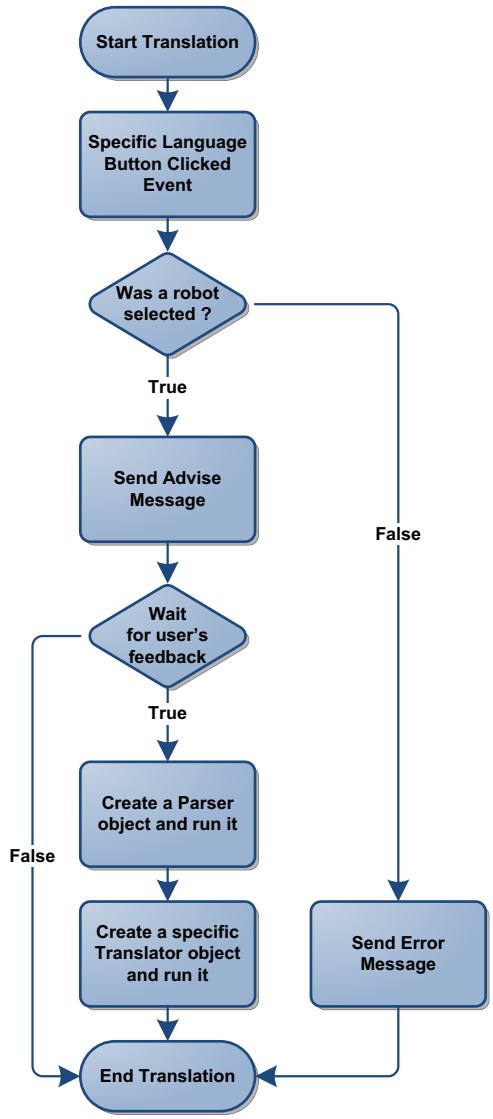


**Figure 3** Core flowchart




of each variable stored in the parser structure are as follows: name and value. Furthermore, the parser also assigns a unique identification to each variable, since some native languages use the register declaration method. The instruction class registers the necessary parameters that define a programmed operation instruction.

Some of the supported instructions are presented in the example code depicted on the left panel in Figure 2. These instructions are commentaries, waiting for a digital signal, delay command, motion order, flow control statements, subroutines call, assignment of values and signals (including arithmetical operations among them), configuration of base and tool frames, break and continue statements. Following the requirements of the AdaptPack project (Silva *et al.*, 2017; Castro *et al.*, 2019), all required instructions and its properties are presented in Table 1. Furthermore, Table 2 presents the data types extracted from the instructions that use the target, base and tool frames. These frames also have an ID assigned by

**Table 1** Parser supported instructions

| Instructions | Properties |
|---|---|
| **Comment** | Text |
| **Wait BIN** | Port and digital binary value |
| **Delay** | Time |
| **Linear Motion** | Target*, speed, base and tool frames* |
| **Point to Point Motion** | Target*, speed, base and tool frames* |
| **Set BIN** | Port and digital value |
| **While** | Condition and scope |
| **If/Else** | Condition and scope |
| **Call Routine** | Routine's name |
| **Assign** | Operators |
| **SetBase** | Base's name, ID and pose |
| **SetTool** | Tool's name, ID and pose |
| **Print** | Text |
| **Break** | — |
| **Continue** | — |

**Notes:** *See Table 2

**Table 2** Target, base and tool frames supported by the parser

| Data | Properties |
|---|---|
| **Target** | Target's name, pose, joints and configuration |
| **Base Frame** | Base frame's name, ID and pose |
| **Tool Frame** | Tool frame's name, ID and pose |

the parser. This ID is based on the first incidence of the frame in the VC program to languages that depend on the register declaration method. It is important to note that, some frames and its relationships are calculated in the step.

The flowchart presented on Figure 4 shows how the parsing procedure is performed. First, it checks if the VC robot's program is not empty and creates the program's data stack, already discussed in this section. Later, for each routine, the parser stores each variable declared succeeding by the instructions, following the incidence order in the program's code.

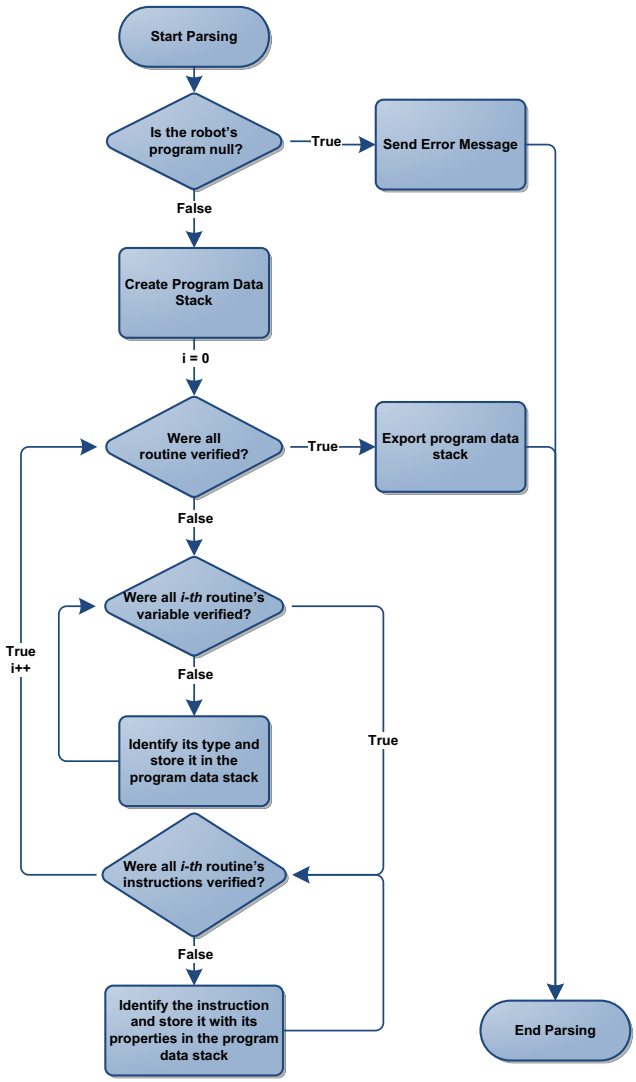
### 2.3 Translator

The translator is the third entity class in the AdaptPack Studio Translator. Generally, a translator is responsible for examining a sequence of instruction statements and data inside all routines and subroutines, added in the data structure stored in the memory by the parser. Afterwards, it translates them to the intended language. This concept can be seen as an "abstract" entity and in fact there are four translation processing entities, i.e., one for each language adopted in the project: RAPID from ABB, Inform from Yaskawa, KAREL from FANUC and KUKA Robot Language (KRL) from KUKA. All these translators will be presented in the subsequent sections and, the flowcharts of Figures 5, 6 and 7 elucidate each translation procedures. RAPID and KAREL are grouped in a single flowchart since its core is equal although minor differences are noted in practical.

#### *2.3.1 RAPID translator*

As with any translator, the RAPID translator will get all sequence of instructions and variables from the database, and it

**Figure 4** Parser flowchart

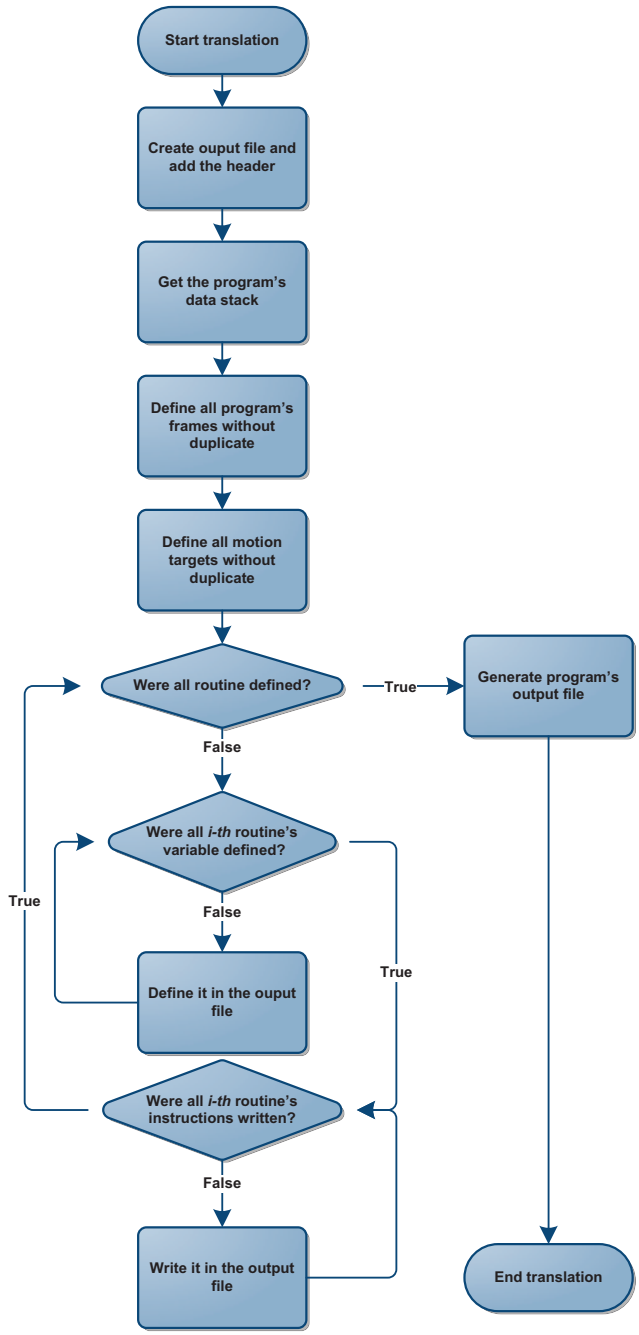



writes the program by the RAPID syntax. The extension of the created file is ".mod". In detail, this translator writes the program headers, followed by the base frames, tool frames and target positions (identified by the motion commands) declarations and definitions (as seen in Figure 5). As the RAPID language changes the records of these frames definitively if any instruction in the code modifies them, this translator creates backups of the base frames and tool frames as the next steps. The applicability of these backups concerns the restore of the original values, at the end of the program execution; this way, the code can be rerun. Subsequently, the main routine is written with its scope, followed by all subroutines (all these routines are placed in a unique file). Since the variables are local, the RAPID translator declares and defines them inside the scope on which they are used.

*2.3.2 Inform translator*

Since Inform is a low-level language [more detailed description in Souza *et al.* (2019)] and some statements of the VC software are not in conformity with it, this translator works according to

**Figure 5** RAPID and KAREL translator flowchart




a different process. It generates at least three files: one file with the extension ".JBI" for the main code, another file with the extension ".JBI" for each subroutine (if it exists), and two other files for base and tool frames declaration and definition, both with the ".CND" extension.

The first generated codes in this translator are responsible to declare and define the base frames and the tool frames into ".CND" files. The frame registers, in these both files, are set with the ID of each one, based on the first incidence in the VC program. Afterwards, the main code and all subroutines are created into ".JBI" files. For each routine, the translator writes

**Figure 6** KRL translator flowchart

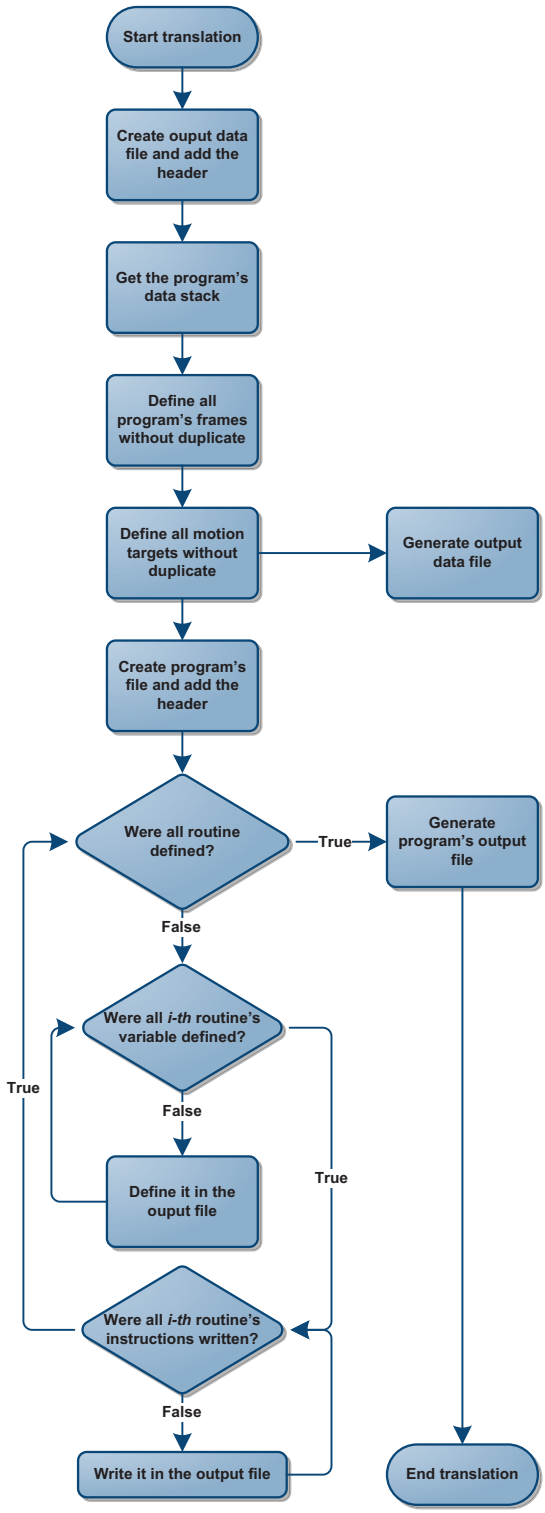
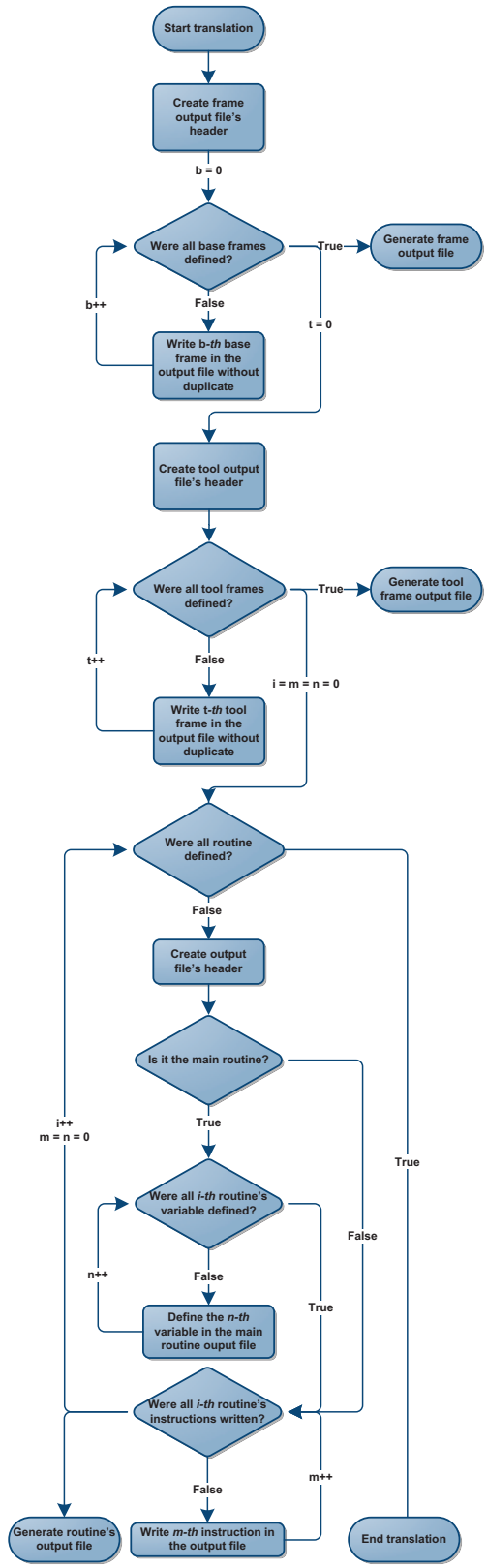



**Figure 7** Inform translator flowchart




the header that defines all the targets registers used in the scope, followed by the instructions, as described in the flowchart of Figure 7. The variables are defined in the main routine. The if-else and the control loops statements are mapped using pointers instructions.

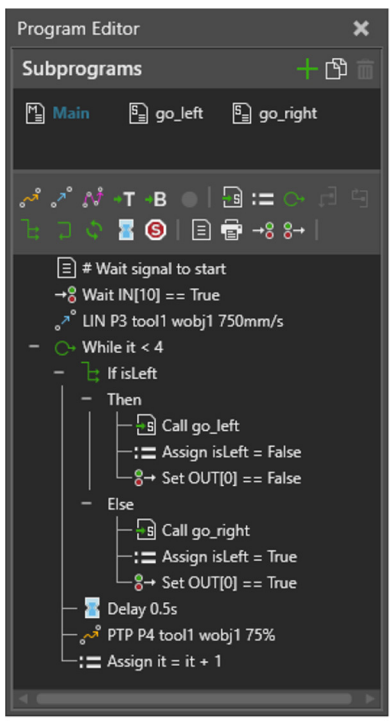
### 2.3.3 KAREL translator

After reading the database, the KAREL translator generates a file with the extension ".kl", in the folder specified by the user, based on the KAREL language syntax. It starts by writing the headers and the declaration of the frames (base and tool), target positions (identified by the motion commands) and variables. In the case of this translator, the subroutines are defined and transcribed before the main one. Inside the main routine are defined all frames, targets and global variables, besides the instructions that constitute the program. The flowchart of Figure 5 shows the processing flow core idea of this translator. The KAREL translator performs the declaration and definition of target positions using registers. Therefore, the user can perform modifications and adjustments, using the teach pendant to access the register, without the need to translate the code again. The registers have an index based on the first incidence of the data in the VC program.

### 2.3.4 KRL translator

The KRL translator generates the code into two different files, with extensions ".dat" (the file where are declared and defined all the program data) and ".src" (the file where the program and routines are defined), as represented in the flowchart of Figure 6. First, the KRL translator creates the ".dat" file with its header, followed by the declaration and definition of the base frames, tool frames and target positions, identified by the VC motion commands. It also includes the declaration and definition of the properties that constitute these motions. The KRL translator defines the base and tool frames using registers; thus, the translations are done according to the first incidence of the data in the VC program. The generation of the ".src" file starts with its header, according to the KRL syntax. The main scope of the code is defined and written, as for all other subroutines. It is worth noting that all routines are transcribed in the same file. The Inline Form Documentation method

**Figure 8** VC program test




```
24  ' Wait signal to start
25  WAIT IN#(101)=ON
26  MOVL P0000 V=750 PL=0  //P3
27  *0
28  ' BEGIN WHILE
29  JUMP *1 IF B0000
30  'GO TO IF
31  JUMP *2
32  ' GO TO ELSE
33  *1
34  ' BEGIN IFTHEN
35  CALL JOB:go_left
36  SET B0000 0
37  DOUT OT#(01)=OFF
38  JUMP *3
39  ' END IFTHEN
40  *2
41  ' BEGIN ELSETHEN
42  CALL JOB:go_right
43  SET B0000 1
44  DOUT OT#(01)=ON
45  JUMP *3
46  ' END ELSETHEN
47  *3
48  ' END IFELSE STRUCT
49  'Timer in seconds
50  TIMER T=0.5
51  MOVJ P0001 VJ=75 PL=0  //P4
52  SET I0000 I0000 + 1
53  JUMP *0 IF I0000 < 4
54  ' END WHILE
```

**Code Script 1.** Excerpt of the Inform sample code

```
25      ! Wait signal to start
26      WaitDI di10,high;
27      MoveL P3,v600,fine,tool1\WObj:=wobj1;
28      WHILE it < 4 DO
29        IF isLeft THEN
30          go_left;
31          isLeft:=False;
32          SetDO do0,low;
33        ELSE
34          go_right;
35          isLeft:=True;
36          SetDO do0,high;
37        ENDIF
38        WaitTime 0.5; ! seconds
39        MoveJ P4, vmax \V := vmax.v_tcp*0.75,fine,
              tool1\WObj:=wobj1;
40        it:=it + 1;
41      ENDWHILE
```

**Code Script 2.** Excerpt of the Rapid sample code

```
70      --Wait signal to start
71      WAIT FOR DIN[11] = ON
72      WITH $TERMTYPE = FINE, $MOTYPE = LINEAR,  $SPEED
            = 750, $UFRAME = Uframe1, $UTOOL = Utool1
          MOVE TO GenPos[1]
73      WHILE (it < 4) DO
74        IF (isLeft) THEN
75          go_left
76          isLeft = False
77          DOUT[1] = OFF
78        ELSE
79          go_right
80          isLeft = True
81          DOUT[1] = ON
82        ENDIF
83        DELAY 500 -- ms
84         WITH $TERMTYPE = FINE, $MOTYPE = JOINT,
              $SPEED = 0.75*$PARAM_GROUP[1].
              $SPEEDLIMJNT, $UFRAME = Uframe1, $UTOOL =
               Utool1 MOVE TO GenPos[2]
85        it = it + 1
86      ENDWHILE
```

**Code Script 3.** Excerpt of the Karel sample code

**Notes:** Code Script 1. Excerpt of the Inform sample code, Code Script 2. Excerpt of the Rapid sample code, Code Script 3. Excerpt of the Karel sample code

(Souza *et al.*, 2019) is also created. Hence, the KRL code instructions are exposed in the robot's teach pendant.

## 3. Tests and evaluation results

A simple test program was created using VC, adopting the offline programming procedure. Afterwards, the translation step was performed by the AdaptPack Translator. The programs in the robot's native code, resulting from the translation phase, were submitted to both simulated and real controlled scenarios. For evaluation purposes, during these tests, the proprietary simulators adopted were ABB RobotStudio, FANUC Roboguide and Yaskawa MotosimEG. The robots used in the tests were the ABB IRB 2600–20/1.6, FANUC LR Mate 200iD 7 L and Motoman YR-HP6-B10/NX100. The simulation step in proprietary simulators were used to check the functionality and syntax of the generated programs before they are uploaded in the real robots.

Figure 8 depicts the VC test program. The full code generated by the AdaptPack Translator, for each robot programming language, is extensive to be included in the paper; therefore, only some excerpts of it are presented. The resulting codes excerpts for RAPID, Inform, and KAREL languages are presented in Figure 8. The following link shows how the translation procedure is performed after the offline programming: http://bit.ly/2PJfzgh

No difference between the VC and the proprietary simulators was detected using the translated codes, i.e. the robot has the same behavior and realizes the same trajectory in both cases. Besides that, the real robots also behaved as predicted, validating the proposed translator library. Furthermore, it was verified that the programs generated by the AdaptPack Studio Translator allowed modification through the teach pendant. This feature is essential since, in the post-process translation phase, it is necessary to perform the calibration process. This process tries to correct the differences between the simulation and the real environments.

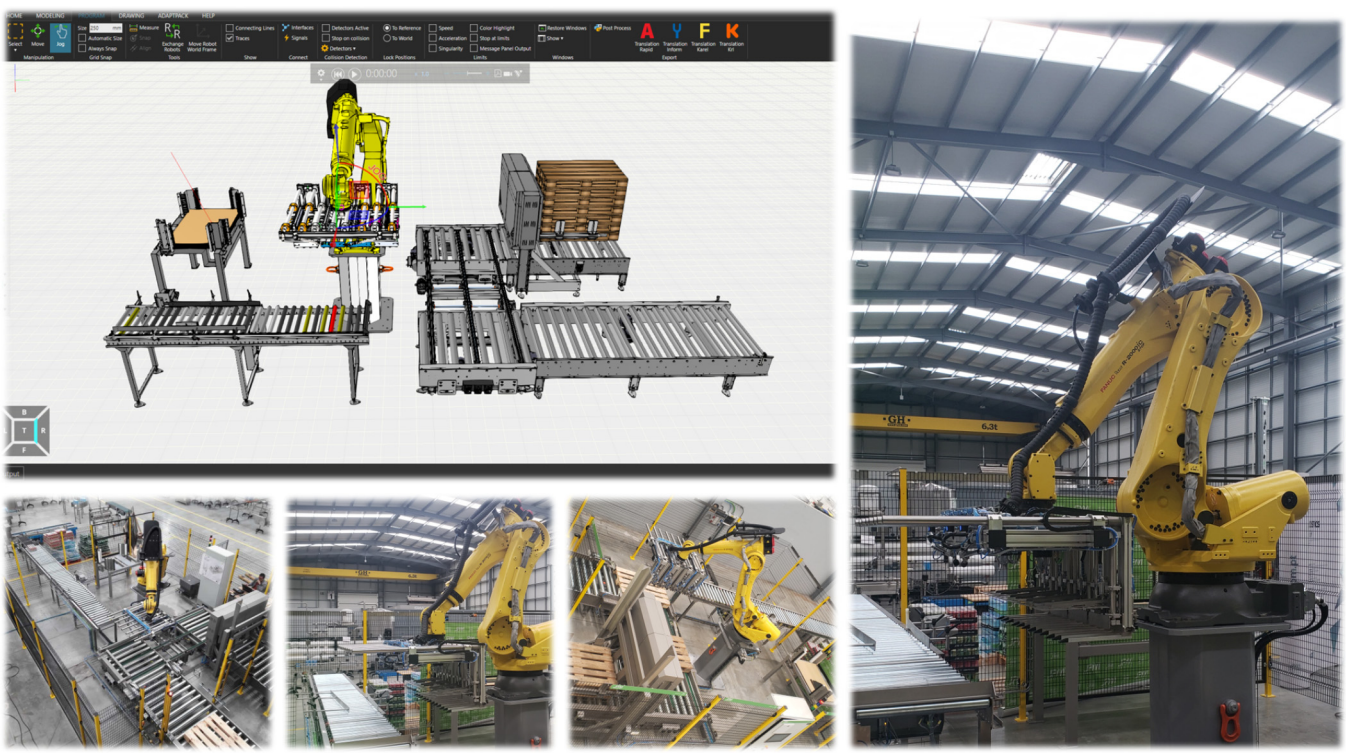
### 3.1 Industry environment tests

A more complex real scenario validation was performed in the JPM Industry company industrial facilities. During these validation tests, it was used a real palletizing cell, with the FANUC Robot R-2000iC/270F, as can be seen in Figure 9. The AdaptPack Studio allowed to automatically generate the program, following the procedure explained in Silva *et al.* (2017) and Castro *et al.* (2019) and the tests were realized in the VC software. After some calibration adjustments between the simulated and the real scenarios, the KAREL code was generated by the AdaptPack Studio Translator, and the robot task was performed as predict during the offline simulation phase.

## 4. Conclusion

This paper presented the AdaptPack Studio Translator. The focus of this library is the automatic generation of the robot controller native codes, after offline programming has been performed in the VC software. The native robot languages supported are RAPID (ABB), KAREL (FANUC), KRL

**Figure 9** AdaptPack Studio Translator real experimental tests and validation at JPM industry industrial facilities, on a palletizing cell with a FANUC robot



**Notes:** Simulated cell in VC (top left). Robot used (right). Different views of the real cell (bottom)

(KUKA) and Inform (Yaskawa). The performed tests in native simulators, in a controlled scenario, and in the real industrial environment allowed us to validate this library.

This proposal improves the post-process procedure during offline programming and supports engineers in the project development. Furthermore, this methodology can handle the present demands of companies for fast design of flexible solutions, as verified during the tests at the JPM Industry facilities, with the proprietary palletizing cell.

Currently, and based on the feedback from the industrial users, we are considering adding functionalities to allow the motions instructions to accept variables as parameters for the motion targets coordinates, instead of accepting only real values. Furthermore, we are also considering adding instructions to allow performing relative motions.

## Note

1. This is an extended and improved version of a paper that was originally presented at the ICARSC 2019 conference, that took place last April in Porto, and which received, ex-aequo, the best paper award granted by the Industrial Robot journal to the papers presented at this event. The original version of the paper was entitled "Converting Robot Offline Programs to Native Code Using the AdaptPack Studio Translators" and is available for download from the IEEE Xplore, at the following URL: https://ieeexplore.ieee.org/document/8733631/

## References

Bottazzi, V.S. and Fonseca, J.F.C. (2005), "Off-line robot programming framework", *Joint International Conference on Autonomic and Autonomous Systems and International Conference on Networking and Services – (icas-isns'05)*, p. 71, doi: 10.1109/ICAS-ICNS.2005.70.

Bruccoleri, M., D'onofrio, C. and U. L, C. (2007), "Off-line programming and simulation for automatic robot control software generation", *5th IEEE Int. Conf. on Industrial Informatics*, Vol. 1, pp. 491-496.

Castro, A., Souza, J.P., Rocha, L. and Silva, M.F. (2019), "AdaptPack studio: automatic offline robot programming framework for factory environments", *19th IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC 2019)*, *IEEE*, Vol. 1

Castro, A., Souza, J.P., Rocha, L. and Silva, M.F. (2020), "AdaptPack studio: an automated intelligent framework for offline factory programming", *Industrial Robot*, Vol. ahead-of-print No. ahead-of-print, doi: 10.1108/IR-12-2019-0252.

Choi, M.H. and Lee, W.W. (2001), "A force/moment sensor for intuitive robot teaching application", *IEEE International Conference on Robotics and Automation (ICRA)*, *IEEE*, Vol. 4, pp. 4011-4016.

Dassault Systemes (2018), "DELMIA robotic simulation and offline programming solutions", available at: www.3ds.com/events/single-eseminar/delmia-robotic-simulation-and-offline-programming-solutions/

Ferreira, M., Costa, P., Rocha, L. and Moreira, A.P. (2016), "Stereo-based real-time 6-DoF work tool tracking for robot programing by demonstration", *The International Journal of Advanced Manufacturing Technology*, Vol. 85 Nos 1/4, pp. 57-69.

Freund, E., Ludemann-Ravit, B., Stern, O. and Koch, T. (2001), "Creating the architecture of a translator framework for robot programming languages", *IEEE Int. Conf. on Rob. and Aut*, *IEEE*, Vol. 1, pp. 187-192.

Industrial Robot Language (IRL). DIN Standard 66312 (1996), "Deutsche norm".

Pan, Z., Polden, J., Larkin, N., Van Duin, S. and Norrish, J. (2010), "Recent progress on programming methods for industrial robots", *41st International Symposium on Robotics (ISR) and 6th German Conference on Robotics (ROBOTIK)*, *VDE*, pp. 1-8.

RoboDK (2018), "RoboDK, simulation and OLP for robots", available at: https://robodk.com/

Schraft, R.D. and Meyer, C. (2006), "The need for an intuitive teaching method for small and medium enterprises", *Vdi Berichte 1956*, p. 95.

Siemens (2018), "RobotExpert", available at: www.dex.siemens.com/plm/robotexpert

Silva, R., Rocha, L.F., Relvas, P., Costa, P. and Silva, M.F. (2017), "Offline programming of collision free trajectories for palletizing robots", *Iberian Robotics conference*, *Springer*, pp. 680-691.

Souza, J.P., Castro, A., Rocha, L., Relvas, P. and Silva, M.F. (2019), "Converting robot offline programs to native code using the AdaptPack studio translators", *2019 IEEE International Conference on Autonomous Robot Systems and Competitions*.

Sugita, S., Itaya, T. and Takeuchi, Y. (2004), "Development of robot teaching support devices to automate deburring and finishing works in casting", *The International Journal of Advanced Manufacturing Technology*, Vol. 23 Nos 3/4, pp. 183-189.

Visual Components (2004), "3D simulation software – quick start guide 3.1. 3.1", Visual Components.

Visual Components (2018), "Visual components", available at: www.visualcomponents.com/

## Corresponding author

**Manuel F. Silva** can be contacted at: mss@isep.ipp.pt