



POLITECNICO DI TORINO
Repository ISTITUZIONALE

Task Oriented Programming and Service Algorithms for Smart Robotic Cells

Original

Task Oriented Programming and Service Algorithms for Smart Robotic Cells / Trapani, Stefano. - (2019 Jul 18), pp. 1-160.

Availability:

This version is available at: 11583/2743230 since: 2019-07-23T15:12:47Z

Publisher:

Politecnico di Torino

Published

DOI:

Terms of use:

openAccess

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)



ScuDo
Scuola di Dottorato ~ Doctoral School
WHAT YOU ARE, TAKES YOU FAR



Doctoral Dissertation
Doctoral Program in Computer and Control Engineering (31.th cycle)

Task Oriented Programming and Service Algorithms for Smart Robotic Cells

Converting old production lines into smart factories

Stefano Trapani

* * * * *

Supervisors

Prof. Marina Indri, Academic Supervisor
Ing. Rosario Cassano, Company Supervisor

Doctoral Examination Committee:

Prof. Antoni Grau, Referee, UPC Barcelona
Prof. Antonio Visioli, Referee, Università di Brescia
Prof. Lucia Pallottino, Università di Pisa
Prof. Bartolomeo Montrucchio, Politecnico di Torino
Prof. Carlo Novara, Politecnico di Torino

Politecnico di Torino
2019

This thesis is licensed under a Creative Commons License, Attribution - Noncommercial-NoDerivative Works 4.0 International: see www.creativecommons.org. The text may be reproduced for non-commercial purposes, provided that credit is given to the original author.

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

.....

Stefano Trapani
Turin, 2019

Summary

The research activity presented in this thesis has been carried out within a PhD Project in Apprenticeship in collaboration with COMAU, co-funded by Regione Piemonte. The topics addressed in such a context are mainly related to Smart Factories and Industry 4.0, and are included in a wider research activity aimed at the conversion of standard production lines (e.g., in automotive context) into smart factories, without changing the initial layout of the line [47]. This means that the introduction of new kinds of machinery and devices is not expected; any changes should be compatible with the current production line, avoiding too invasive, setup phases. In this scenario the research activity was divided into two main fields of research: 1) the development of a new programming paradigm, and 2) the development of advanced service algorithms.

The first topic aims at providing an offline programming methodology, which allows to (re-) program the robotic cell in a flexible way. The new approach should be able to provide the fast reprogramming of the production line, with possible relocation of machinery and resources, on the basis of the current conditions. Even if such approach is less flexible than the Flexible Manufacturing Systems, it is easily applicable to standard production lines.

The goal of the second topic is to develop advanced functionalities to be implemented in standard industrial manipulators. Also in this case the idea is to avoid deep changes in the robotic cell. In this way industrial robots that were initially developed to perform, more or less, the same activity for all their life, now they could be used in different contexts, and for new kinds of applications.

In order to keep the discussion about such topics clear and well divided, the thesis has been organized into three parts: Part [I](#)) Automatic Task Oriented Programming, Part [II](#)) Advanced Robotic Service Algorithms, and Part [III](#)) general conclusions.

Contents

List of Tables	VII
List of Figures	VIII
I Automatic Task Oriented Programming	1
1 Introduction to robot programming	3
2 Task Analysis	11
3 Proposed approach	15
3.1 Task Modeling	18
3.1.1 Model Entities	19
3.1.2 Spatial Constraints Graph	23
3.1.3 Functional Link Graph	25
3.1.3.1 Complex blocks management	29
3.1.4 High Level Model	30
3.2 Work Flow Modeling	31
3.2.1 Automatic conversion HLM \rightarrow WFM	33
3.3 Case Study	40
4 The optimization phase	45
4.1 Integration with a production efficiency tool	46
4.1.1 Basics of scalable production efficiency tool	46
4.1.2 The proposed architecture	48
4.1.3 Computation of the efficiency parameters	49
4.1.3.1 Sigmoid function	50
4.1.3.2 Computation of P_{eff}	52
4.1.3.3 Computation of Q_{eff}	52
4.1.3.4 Computation of A_{eff}	52
4.1.4 Case Study	53
4.1.5 Automatic conversion of the WFM into a subsystems' tree	60

4.1.5.1	Propaedeutic definitions and notations	60
4.1.5.2	Conversion algorithm	62
II	Advanced Robotic Service Algorithms	75
5	Service algorithms for industrial robots	77
6	Collision Detection	81
6.1	The proposed approach for collision detection	82
6.1.1	Model Error Estimation	84
6.1.2	Monitoring of the currents behavior through a FSM	87
6.1.3	Monitoring of the currents time derivatives	88
6.2	Automatic learning and adaptation of the sensor sensitivity to collisions	91
6.3	Whole structure of the virtual collision sensor	96
6.4	Experimental results	101
7	Payload Check	109
7.1	The proposed Payload Check algorithm	111
7.2	Experimental results	114
8	Post-collision reaction and Manual Guidance	117
8.1	Sensor-less approach to Manual Guidance	117
8.1.1	Monitoring state	118
8.1.1.1	Condition <i>mg_enter</i>	119
8.1.1.2	Condition <i>cr_enter</i>	120
8.1.2	Manual Guidance state	121
8.1.2.1	Condition <i>mg_exit</i>	122
8.1.3	Collision Reaction state	123
8.1.3.1	Condition <i>cr_exit</i>	124
8.1.4	Waiting state	124
8.2	Experimental results	124
9	Friction modeling	131
III	Conclusions	135
	Bibliography	141

List of Tables

3.1	Effects of the three types of actions on the HLM	18
3.2	Translation rules	28
3.3	Mapping of the the real objects into the corresponding model entities	37
3.4	Mapping between some technical specifications of a COMAU Racer 7 - 1.4 and of a COMAU NJ4 110 - 2.2	40
4.1	Mapping between real object and HLM entities	55
4.2	Mapping between WFM blocks and sub-systems	55
4.3	Mapping between OVR , OEE , and the A_{eff} , P_{eff} , and Q_{eff} factors composing the OEE for the specific example interpretation of the picking task of the NJ110 - 2.2 unit, corresponding to the $e2$ subsystem.	58
4.4	Selection of implemented actions, for the higher Q_{eff} preference case.	59
4.5	n^{th} degree fundamental structures	60
4.6	Description of the notations used in the translation algorithm	67
4.7	Description of the functions adopted in the translation algorithm	68
4.8	Comparison between the collision Detection Times (DT) of the basic version and the adaptive one	102
6.1	Comparison of the number of axes which are able to detect the collision using the basic collision detection procedure and the proposed virtual sensor, including the learning and adaptive functionalities	103
6.2	Timetables for collision detection during different experiments	107
6.3	Parameters of the varying threshold function $Th_{cd}(t)$ in the experimental implementation.	125
8.1	Values of a_s and b_s for the three reaction strategies experimentally applied.	126
8.2	List of publications	139

List of Figures

1.1	Example of an automotive welding application	3
3.1	Architecture of the Task Oriented Programming	16
3.2	The three steps of the programming methodology	16
3.3	CAD Design of both the robotic cell and the process	17
3.4	The four phases of the Automatic Programming methodology	17
3.5	Structure of the object entity	22
3.6	Example of definition of a Spatial-Constraints-Graph	24
3.7	Possible example of a Functional Link Graph	25
3.8	Using V_SYNC entities in a FLG in order to synchronize different parts of the process	26
3.9	Example of a possible deadlock caused by an incorrect usage of the V_SYNC entities between two FLG models	27
3.10	Definition of a possible <i>joining</i> action between two sub-tasks belonging to different FLG models	29
3.11	Modeling of a <i>complex</i> block	30
3.12	Overall HLM model	32
3.13	Example of a WFM composed by both basic and complex blocks	33
3.14	Example of HLM with some steps of the scrolling algorithm	35
3.15	Sketch of a WFM obtained from a HLM with three Positioners called P0, P1 and P2	38
3.16	Graph $G(V, E)$ corresponding to the SPG of the proposed example	39
3.17	Sketch of the robotic cell of the case study	41
3.18	TASK_st_exec block corresponding to the ST1	42
3.19	TASK_st_exec block corresponding to the ST2	43
3.20	TASK_st_exec block defining the last iteration	43
4.1	The four fundamental structures	47
4.2	An example of tree-based relationship between parent and children structures	48
4.3	The adopted sigmoid function	51
4.4	Representation of the proposed robotic cell obtained using Blender	54
4.5	HLM corresponding to the proposed case study	56
4.6	Conversion of the WFM into a corresponding OTE systems' tree	57

4.7	OTE evolution using actions that favor higher P_{eff} and A_{eff} (in blue), or higher Q_{eff} (in red)	59
4.8	Basic structure included in the work flows	61
4.9	HMT for the <i>basic structure</i>	62
4.10	General structure of the WFM	63
4.11	General structure of the work flow	64
6.1	Virtual collision sensor scheme.	82
6.2	Behavior of the current residue of the second joint of a NJ4 110 in the <i>steady</i> (white background) and <i>unsteady</i> (blue background) states	84
6.3	Comparison between $\hat{m}_{err,1}(t)$ computed for <i>steady</i> state and for <i>unsteady</i> ones.	86
6.4	FSM scheme with sketch of the currents behavior in the various states (I_i in red and $I_{DM,i}$ in blue).	88
6.5	Examples of the behavior of $I_i(t)$ and $I_{DM,i}(t)$ during the moving state.	89
6.6	Examples of the behavior of $I_i(t)$ and $I_{DM,i}(t)$ during the impulse state.	89
6.7	Moving state: the time derivative error is within the noise limits.	90
6.8	Moving state: both the time derivatives are outside the limits but having the same sign.	90
6.9	Impulse state. The highest bounds (dashed red lines) are overcome by the time derivative error.	91
6.10	Reversing state. In the highlighted region (dashed magenta line) the time derivatives have different signs	91
6.11	Activity diagram of the <i>Bias Estimation</i> block for the i -th joint	93
6.12	FSM for the sensor sensitivity adaptation.	95
6.13	Example of a cycle involving init, learning, set and adapting phases.	96
6.14	General activity diagram of the collision detection procedure	97
6.15	Activity diagram of the <i>virtual collision sensor</i> block	98
6.16	Activity diagram of the <i>Computes $m_{err,i}(t)$</i> block	99
6.17	Activity diagram of the <i>Collision Check</i> block	100
6.18	Behavior of threshold functions and current residue during the adaptation phase for the second joint of a NS12 manipulator	101
6.19	Cartesian path of the robot tool center point in the four considered cases.	102
6.20	Cartesian velocity of the robot tool center point in the four considered cases.	103
6.21	Velocity of all the axes of a Comau NS-12 in the four considered cases: 1) left \rightarrow right EOS (blue line), 2) left \rightarrow right NR (red line), 3) top \rightarrow bottom EOS (green line), and 4) top \rightarrow bottom NR (black line)	104
6.22	<i>SMART5 Arc 4</i>	105
6.23	<i>SMART5 NJ4 110</i>	106

6.24	Behaviour of current I compared with I_{DM} related to the first joint of a SMART Arc4 during a collision.	106
6.25	Collision detection warning.	107
6.26	Behaviour of current I compared with I_{DM} related to the fifth joint of a NJ4 110 during a collision.	108
7.1	Estimation of the payload error using the lowpass filter (blue, solid line) and as average (red, dashed line)	112
7.2	Difference between the values returned by the filter and as average.	113
7.3	Behavior of the filtering action and the average during the transient.	114
7.4	Behavior of the filtering action and the average after the transient.	115
8.1	Basic state diagram for the state machine.	118
8.2	First plot from the top: filtered forces (solid blue) and $Th1$ on the x-axis (dotted blue); middle plot: filtered forces (solid orange) and $Th1$ on the y-axis (dotted orange); lowest plot: filtered forces (solid yellow) and $Th1$ on the z-axis (dotted yellow). All the three plots include the transition Flag (magenta).	127
8.3	Forces slope on the x-axis (blue), forces slope on the y-axis (orange), forces slope on the z-axis (yellow), transition Flag (magenta), slope threshold (dotted violet).	127
8.4	First plot from the top: filtered forces (solid blue) and $Th2$ on the x-axis (dotted blue); middle plot: filtered forces (solid orange) and $Th2$ on the y-axis (dotted orange); lowest plot: filtered forces (solid yellow) and $Th2$ on the z-axis (dotted yellow). All the three plots include the transition Flag (magenta).	128
8.5	First plot from the top: filtered forces (solid blue) and $Th2$ on the x-axis (dotted blue); middle plot: filtered forces (solid orange) and $Th2$ on the y-axis (dotted orange); lowest plot: filtered forces (solid yellow) and $Th2$ on the z-axis (dotted yellow). All the three plots include the transition Flag (magenta).	129
8.6	Forces slope on the x-axis (blue), forces slope on the y-axis (orange), forces slope on the z-axis (yellow), transition Flag (magenta), slope threshold (dotted violet).	129
9.1	Overall scheme of the friction identification procedure.	133

Part I

**Automatic Task Oriented
Programming**

Chapter 1

Introduction to robot programming

Programming complex manufacturing systems (see Figure 1.1), like robotic cells, can be a very hard work. Nowadays in the industrial scenario there are two main robot programming methodologies [76]: Online programming and Offline programming (OLP).

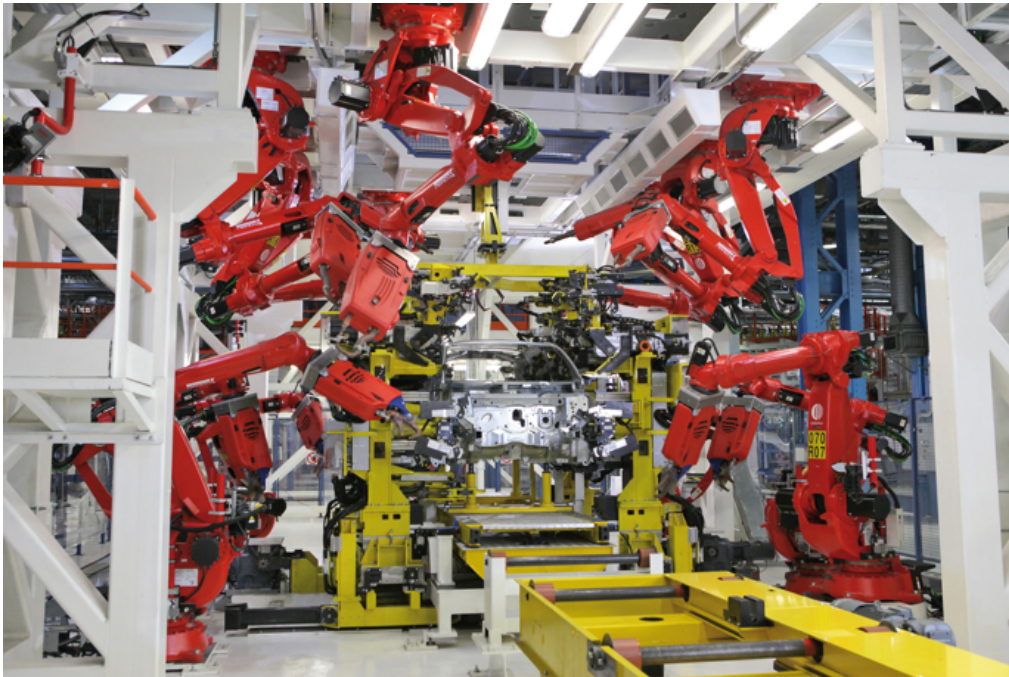


Figure 1.1: Example of an automotive welding application

Online programming is the classic programming approach, in which the programmer, using the Teach Pendant, moves the robot to each significant point,

choosing both position and orientation of the end-effector, and storing the corresponding information (i.e., the teaching phase); such operation is repeated for each robot involved in the process. It is easy to imagine that this approach is time-consuming, because of the possible very high number of points to be managed, as well as the necessity of both avoiding collisions with other robots (whose work-points could be not known, yet), and with the environment, and achieving good performance of the process, usually in terms of cycle time. In practice such kind of approach leads to a Trial & Error methodology, in which each point already saved could be subsequently modified in order to avoid a collision, or to improve the performance of the process; points and trajectories of each robot are then iteratively modified several times before reaching a good trade-off between process accuracy and cycle time. Such approach is suitable for uncomplicated process onto a simple work-piece, but sometimes is the only available methodology especially for small enterprises. Online programming is usually performed by hard skilled programmers, that must be prepared on several aspects:

- decomposition of the whole process into a sequence of simple sub-tasks;
- assignment of each sub-task to the available machinery, like robots, taking into account both the physical constraints (e.g., degrees of freedom, operational area) and the compatibility between the specific sub-task and the adopted machinery;
- definition of the trajectories of all the involved robots, taking into account possible limitations of the robots, as well as avoiding collisions;
- optimization of the process in some way. The cycle time is usually minimized in several industrial contexts, but other criteria could be taken into account (e.g., energy consumption);
- writing of the program to be run in the specific control unit.

Programs structure is defined by the programmer in order to properly sequence/synchronize the sub-tasks composing the whole required process; it can be very complicated on the basis of the complexity of the required application.

Offline robot programming is based on the usage of 3-D CAD data, in order to generate and simulate robot programs. Such approach is quite different from the classic one, where all the programming phases were made by hand; here there is a higher level of automation of the programming process, given by the possibility to: 1) define/import the 3-D model of the robot cell, 2) define the trajectories of the robots, and 3) simulate the plant. The main advantage of such approach is that programming does not require actual robots, so that this phase can be done earlier, and can be also repeated while the line is producing. A second benefit is that programs are more flexible, since they can be easily modified. The further advantage is about

the plant simulation, which is also very important in order to achieve faster (re-) programming of complex robotic lines, as well as the possibility to check the results of a priori programming, thus avoiding possible programming errors and improving the quality of the process itself. Also in this case programmers must be very skilled, since they have to perform almost the same activities; nevertheless the writing of the source code could be no longer required, in fact OLP softwares often support code generation. Such additional feature slightly change the role of the robotic line programmer, whose activity is no more concerning the writing of the source code, but is only focused about the technological processes and the related robotic issues. Very interesting works can be found about such kind of programming approach; in [70], [72], [71] a common 3-D CAD package is used to import information about the robotic cell into a more advanced robot programming interface, helping the programmer to define the task and to create the corresponding program. CAD models can be used as source of data for robot programming, as stated in [32], thus process information can be included in the CAD models (e.g., in [79] an automatic robot welding programming is proposed), and used within a OLP methodology to define the programming of the whole robotic cell. The main drawback of such kind of methodology is that when the CAD model does not reproduce exactly the real scenario, it is not possible to obtain satisfactory performance; such a problem can be solved by eliminating or minimizing the errors between the real scenario and the virtual one, e.g., using sensory feedback as proposed in [73]. Offline programming is suitable to compute the robot paths for typical industrial tasks, using the 3-D information of the work-piece, e.g., for welding [57], thermal spraying [27], and finishing [49]. Such approach can be adopted to define cooperative tasks, as well; e.g., [36] proposes the motion planning of two robots for a cooperative operation, in which both position and posture must be coordinated. OLP approach can be interfaced with other systems in order to improve its performance; e.g., in [10] an OLP platform has been integrated with a three-dimensional vision system able to provide part recognition and pose estimation of the objects in the robotic cell.

Nowadays robot programming methodologies are moving toward smart solutions, able to understand the human body language in order to make the programming process of complex tasks more comfortable, in particular for soft skilled programmers. Within such a scenario the programmer becomes the teacher of the manufacturing system, that learns the task observing or reproducing the operations performed by the human operator. Such approach is usually called Programming by Demonstration (PbD) or Learning from Demonstration (LfD) [6], but it can be found with different names in literature. Two main methodologies can be adopted to teach the task to the robotic system. The first one is based on the *demonstration* of the task; in this case there are two common approaches: i) *teleoperation*, where the robot can be guided by the teacher (e.g., using joystick or speech dialog), and ii) *shadowing*, where the robot mimics the teacher's demonstrated motions

in order to perform the task; regardless of the type of the adopted approach, sensors information of the robot is stored during the teaching process. A second LfD approach is based on the *imitation* of the task; in such a case external sensors are used to observe the tasks (e.g., wearable sensors applied on the programmer or vision systems recording the procedure) and to store the related information. Such approach is still in development, and mixed approaches can also be found, e.g., in [67] where different complementary approaches, like PbD, Feedback and Transfer, are all exploited together, and in [5] where a different PbD framework is proposed, in which demonstration is provided once, while additional task information is provided explicitly. In [33] teaching is instead achieved by using the spatial language to describe a set of predefined kinds of movements, in this way the user is not required to interact with the robot. Such approach seems to be border line in the field of PbD, but it provides information that can be very interesting for a task based programming approach. Also very interesting is the management of the teaching information, which can be encoded into sequence of predefined symbols, representing elementary actions that are exploited to obtain a generalized task structure, e.g., a hierarchy structure [95], or a topological task graph [4]. However such approaches require the specification of all the parts of the whole process (although they are performed by demonstration), so that there is no separation between the actual required tasks (e.g., a welding task) and any other intermediate parts of the process (e.g., free movements).

A more recent approach exploits Augmented Reality (AR) to program robots [59],[75],[89]. Such a methodology is quite new and despite its potential is not fully explored, it could be an important innovation in the programming of production systems [29]. AR can be seen as a middle way between the online and the offline programming [76]; it can be combined with offline programming, exploiting CAD models of the robotic cell to create the virtual environment. Using programming devices (e.g., tablet) the programmer can carry out the teaching process in the virtual robotic cell. Important advantage are introduced with respect to the classic approach, like no needs of the physical robots (since virtual robots are used), the possibility of easily programming large robots (such as airplane washing robots) and collision-free paths [21].

Current trend of manufacturing companies is to move towards wider markets, involving the production of various products in order to meet quickly the customer demand. Such a phenomenon is leading towards two main technical solutions, to make standard production lines more flexible. Different schools of thought can be found, based on: 1) Flexible Manufacturing Systems (FMS), where the production line is able to adapt its production process to possible market variations, and 2) OLP, where, using offline software tools, production line can be reprogrammed on the basis of new market conditions. The two approaches are very different; the first one is off-course more adaptive, but it requires the usage of intelligent systems inside the production line, able to change their behavior on the basis of

external information. This means that, in general, is not possible to convert a classic production line into a flexible one, without making heavy changes to the line itself. In the second case OLP is a not reactive approach (in fact the human operator decides by himself when the line must be reprogrammed), nevertheless it allows to obtain a new production process compliant with current market condition, using the same production line. In practice, using OLP the intelligent part of the system is centralized in the offline tool, instead of distributed into smart machinery (e.g., robots, AGV).

Automatizing the programming process through a OLP approach (AOLP) is quite complicated, since different issues must be taken into account at the same time, like: i) to fulfill physical constraints of the robotic cell (e.g., to avoid collisions), ii) to obtain the required performance (e.g., cycle time) and iii) to carry out the required task. An approach in which the programmer defines the sequence of sub-tasks using high level tools (e.g., CAD software), so ignoring the issues related to physical constraints, task sequencing or performance optimization, makes the programming process accessible to soft skilled programmers, mainly experienced about the process issues only. The overall objective of the research activity (carried out in collaboration with COMAU S.p.A.) is to develop an automatic offline task-oriented programming approach suitable for generic robotic cells, which can be used by soft skilled programmers during the programming of the required application. The approach must be intended as a new kind of programming approach, supporting the programmers only during the initial programming phase; flexible manufacturing systems are out of the scope of such a research, even if possible physical changes of the robotic cell (e.g., the failure of a machinery) could be possibly managed using such a task oriented approach to quickly reprogram the robotic cell. Starting from some basic information about the robotic cell and the required tasks, the goal of the task-oriented programming is to provide the planning of all the movements and actions to be performed by each machinery inside the robotic cell. The proposed goal has been reached by a three step approach given by: i) the definition of a model including both the process and the environment, ii) the definition of all the feasible work-flows, and iii) the choice of the best work-flow according to some criteria.

The greatest efforts in such a work have been devoted to define a task model able to represent a generic industrial task and to automatize the programming process; certainly several further issues are related to such a research activity, but while some of them have been already studied and a lot of material can be found in the state of the art, about task modeling, to the best of the authors' knowledge, there is no previous approach in literature with exactly the same objectives of the proposed one, even if some related paradigms and guidelines about task modeling can be found. In [8] four main elements (Resource, Skill, Process and Product) are used to obtain both a task model and a world model, in which the Skill element is defined by means of five main actions (i.e., move, connect, compare, store and

change). The proposed five main actions seem to be too specific to allow a high abstraction level, even if they could be exploited to define a set of virtual actions. In [77] three models are built: i) Process model, ii) Workcell model and iii) Object model; CAD information is also used to set the geometrical constraints and the assembly steps, as well as to store information about dimensions and polygon mesh of the involved objects, which are used to implement further checks (e.g., collision avoidance). In this work however no procedure is proposed to automatically map the tasks into corresponding robot actions or to automatically choose the robots involved in the tasks.

In [51] product CAD models and technological knowledge are exploited to obtain assembly features and build the assembly tasks. A description of all the geometrical parts by means of the triangle meshes (without using any CAD information) are also exploited to carry out a collision avoidance check. On the whole, such a work allows an automatic verification of the plan feasibility from the technological and geometrical point of view, and of the stability, but it does not address the case of cooperating multiple robots.

In all the discussed works [8], [77] and [51], CAD models are used to import a set of information that is employed to build the internal models and to define the geometrical constraints.

In [24] the task model is obtained as a set of assembly features, which are used to define an elementary skill. Through a specific graphical user interface the user can easily “write” a program by creating new skills and composing them properly. Such a work proposes an interesting solution for a possible user-friendly interface, but it does not include any automation in the task definition, which is fully defined by the user.

In [94] the authors proposed an interesting subdivision of the task planning into two phases: i) the target task is divided in simpler tasks (called primitive tasks), and ii) each primitive task is divided in simple paths using a heuristic algorithm. Such an approach could be used as starting point to define a hierarchical model of the task.

A second important activity was devoted to the definition of all the possible work-flows carrying out the required process. Typically a work-flow is modelled as a net of nodes having different meanings. Usually work-flow models include a task node defining a specific action, as well as nodes defining different relations between the tasks, e.g., choice, merge, fork, synchronization and sequence, as highlighted in [37], [83], [56]. According to such criteria, the proposed Work Flow Model (WFM) includes specific blocks allowing to split or join the work-flows, so to define the parallel or the execution in mutual exclusion of generic task blocks. In addition, the WFM is characterized by a recursive structure that allows to extend the model itself by introducing more complex blocks. The formal verification of a Workflow-net (WF-net) is also an important issue, in order to avoid possible structural conflicts

in the process model, e.g., deadlock and lack of synchronization; the choice of a work-flow model suitable for a process verification is then essential. In case of WF-nets, a set of graph reduction rules can be exploited to identify structural conflicts in the process, as proposed in [84]; other approaches (e.g., [3]) are based on the usage of Petri Nets (PN) theory and tools to verify work-flow graphs. As highlighted in [2] and [86], PN are a very suitable tool to map work-flows (e.g., in [93] an application for production systems is reported), thanks to their formal semantic, the property to be state-based, and the abundance of the related analysis techniques; also their formalism can be used to model the available resources, as illustrated in [25]. Possible conversion between a work flow model, like the WFM, and an equivalent PN can also be achieved, if necessary; e.g., in [19] an AND/OR graph is converted into a PN. The proposed WFM is suitable for conversion into PN, but it can be also adopted for the inclusion of the modeled robotic task in an overall process, by the imposition of some high level goals during the programming phase (e.g., the reduction of possible bottlenecks). Some bottleneck detection approaches, like the one proposed in [14], can be very suitable if applied to a work-flow model characterized by a recursive structure, as the proposed one. In fact, such a factory performance diagnostics, based on the computation of a proper metric called Overall Throughput Effectiveness (OTE), can be carried out recursively to all the levels of the work-flow model as presented in [46], where such an approach has been tested for a real industrial application.

Chapter 2

Task Analysis

Defining the features of a specific task involved in an industrial application is the starting point to find a generalized description of a generic task, and hence its implementation by means of a unified tool. Such a goal can be achieved by studying different types of industrial applications, in order to find a set of minimal actions defining a generic task; however the differences between some types of tasks could make difficult to obtain a unified modeling. The analysis of the tasks is then focused to: i) find the set of common features among all the tasks, ii) define procedures to translate non common features in a standard form. The most important industrial applications as listed in the websites of the main robot constructors (e.g., ABB, COMAU and KUKA) are: *Arc and spot welding, Assembly, Cosmetic sealing, Foundry, Handling and Packaging, Laser welding/cutting, Machine tending, Plasma cutting and Water jet, Polishing and deburring, Press brake bending, Interpress, Processing machining, Painting.*

The tasks show at least four different characteristics: i) tasks requiring a path, ii) tasks working on a single point of the work-space, iii) tasks involving only the motion of the work-piece, and iv) tasks needing further elaborations to obtain a work-path. On the basis of such criteria the full list of tasks can then be organized in four classes:

1. Tasks requiring a path:
 - Arc welding,
 - Cosmetic sealing,
 - Polishing deburring,
 - Laser welding cutting,
 - Plasma cutting/Water jet.
2. Task working on a single point of the work-space:

- Spot welding.
3. Tasks involving only the motion of the work-piece:
 - Machine tending,
 - Handling,
 - Processing machining,
 - Press brake bending,
 - Interpress,
 - Foundry.
 4. Tasks that need further elaborations to obtain a work-path:
 - Assembly,
 - Packaging,
 - Painting.

Analyzing the proposed classification, it can be seen that most of the tasks belong to the first and third categories; if the spot welding task is managed as a task requiring a path, whose starting and final points are coincident, it could be included in the first category as well. The fourth category defines a set of tasks that need some pre-elaborations in order to obtain the paths defining the whole process, e.g., in the painting process the paths are often defined by a specific elaboration of the shape of the work-piece, taking into account the features of the specific tool used for the application.

The analysis highlights that two main types of tasks can be defined: i) the *Standard* tasks (i.e. those belonging to the first and third category), which can involve both processes carried out on a specific path or those simply devoted to carry the work-piece, and ii) the *Special* tasks, which need some pre-elaboration. Such a result imposes some requirements on the tool that is going to be developed to generate the whole task, like the inclusion of a standardization layer handling the pre-elaboration of the *Special* tasks, so that the same structure of input values can be maintained for all the types of tasks. Such an approach allows to develop a general task model unrelated to the specific task. A second important requirement is related to the necessary simultaneous ability to manage tasks belonging to the first and the third classes. The task model must handle cases in which a real machine (e.g., a manipulator) performs a process along a predefined path using a specific tool (i.e., the *worker* of the process), as well as processes in which the machine is equipped with a proper gripper, using which it is able to carry the work-pieces from a point to another one of the work-space (i.e., the *positioner* of the process) in order to either allow some *worker* to perform its task or to simply pick or place the work-piece at

predefined points.

Further requirements rise in the applications that involve structural modification of the work-piece, as in the following tasks, in which the physical characteristics of the work-piece (could) change: i) arc welding ii) spot welding, iii) laser welding cutting, iv) plasma cutting/water jet v) processing machining, vi) press brake bending, and vii) interpress.

Chapter 3

Proposed approach

The proposed approach has been developed taking into account different factors, e.g., issues dealt with by other works in the state of the art, results of the tasks analysis, and constraints related to the context in which the methodology will be applied. Works analyzed in the literature highlight the need to model both the environment (i.e., the robotic cell in our case), and the process, and to save their internal states while computing the automatic programming; in this way it is possible to know the configuration of the robotic cell at each step of the required process. The task analysis presented in Chapter 2 also showed that a unique management of the input data is possible if a proper pre-processing phase is applied. In general it is also possible that the process can be defined in different ways in the given robotic cell, since in practice different robots could be able to perform the same sub-task of the process; a proper optimization procedure it is then required to choose the best way to perform the process. The task analysis also showed the necessity to model different kinds of machinery, like workers (e.g., robots that perform a process on the work-piece using their tools) and positioners (e.g., robots that move the work-piece using a suitable gripper). Finally a proper collision free path planning module is also necessary to guarantee the movements of each machinery involved in the process in absence of collisions. In terms of functional blocks, the proposed Task-Oriented Programming methodology can be represented as in Figure 3.1, where it can be noticed the presence of a pre-elaboration block, a process model block, and an optimization block. The path planning has been included in the optimization block, since its result is required to compute a set of performance indices, used in the optimization process.

The complete functionality should also include a high level layer providing the input data, and a low level layer that generates the user programs using the outputs of the CORE functionality (Figure 3.1). A possible work flow is represented in Figure 3.2, in which the proposed Task-Oriented Programming Approach has been included in the overall three-steps programming process.

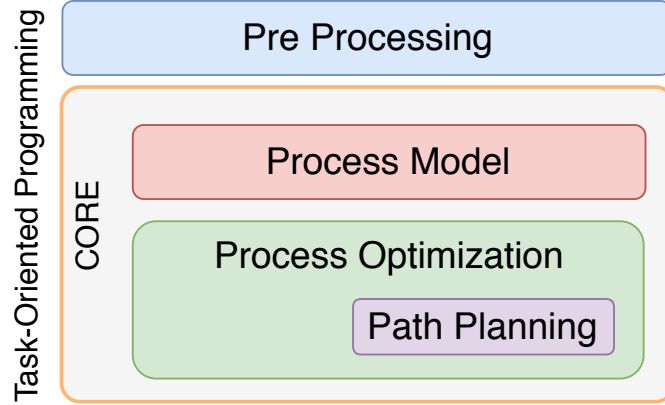


Figure 3.1: Architecture of the Task Oriented Programming



Figure 3.2: The three steps of the programming methodology

CAD Design In this step the programmer designs the robotic cell using a CAD software tool as represented in Figure 3.3; in this way the physical characteristics of the robotic cell (e.g., cell dimensions, robot characteristics, robot location, etc.) are available through a proper description file (e.g., a xml file). In such a phase the programmer also includes information about the process, which can be related to both physical characteristics, like the path of an arc welding task, and functional features, e.g., the correct sequencing tasks in the overall process. Even if the Task-Oriented Programming approach aims at automatizing the programming process, some information cannot be deducted; for this reason it is important that the programmer includes functional characteristics of the process in the description file. Functional characteristics, like the correct sequencing of the tasks, are in fact related to the specific application, and depend on unpredictable factors, e.g., in a spot welding application of a car door, it could be needed to weld some points before others. Such kind of decisions are left to the programmer in the CAD Design phase. Since the methodology has been developed to be task-oriented, only process movement are actually designed by the programmer in this phase.

Task-Oriented Programming In this step the description file previously obtained is used as input of the proposed Task-Oriented Programming approach,

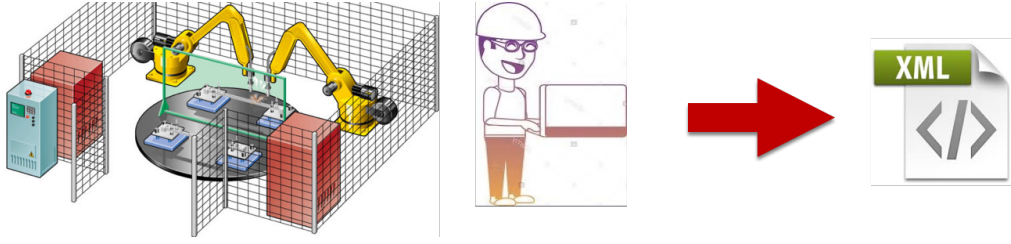


Figure 3.3: CAD Design of both the robotic cell and the process

which exploits such information to define the movements of all the machinery involved in the process. The approach builds the model of the process, taking into account both the physical characteristics of the robotic cell and the functional characteristics of the process, in order to provide one feasible solution. The model is exploited to both define the movements (including non-process ones) of the machinery, and to optimize the process itself. The Automatic Programming is achieved by applying the four phases, shown in Figure 3.4.

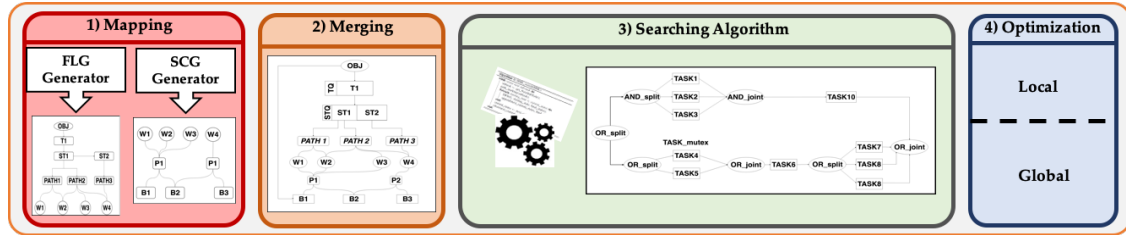


Figure 3.4: The four phases of the Automatic Programming methodology

The first two phases (i.e., Mapping and Merging) allow to build a process model, called High Level Model (HLM), taking into account both physical characteristics of the robotic cell and functional characteristics of the process. The third phase (i.e., Searching Algorithm) elaborates the HLM providing all the feasible work-flows carrying out the required process in the given robotic cell; such work-flows are defined by means of a specific model, denoted as Work Flow Model (WFM). The last phase (i.e., Optimization) applies an optimization process to the WFM in order to select the best work-flow according to some criteria (e.g., cycle time or energy consumption).

Deploy into the robotic line Aim of this step is the creation of the user program, written for a specific industrial controller (e.g., PDL2 programming language for the COMAU controller), implementing the movements generated using the Task Oriented Programming methodology. The output of this step is then the source code of the user program.

The proposed work was mainly focused on the development of the CORE functionality of the Task-Oriented Programming approach, and in particular the greatest efforts were made to automatize the building of the process model. The discussion does not address some issues, like the CAD design phase and the generation of the programs. The pre-processing and path planning are not managed, as well, since several works in the state of the art already deal with such kind of issues.

3.1 Task Modeling

The main goal of the task model is to provide a unified structure of a generic task, which allows to obtain the whole sequence of actions defining it. The High Level Model (HLM) is based on the assumption that each task can be divided into a sequence of four basic steps (or by a subset of them): i) picking of the work-piece, ii) positioning of the work-piece within a sub-set of the working-area compatible with the execution of the task, iii) working, iv) placing of the work-piece. In order to perform such steps, the model must include entities defining the points for picking and placing the work-piece, as well as entities defining a class of objects able to grip the work-piece and to carry it within the work-space, and further entities corresponding to objects equipped with a proper tool for the execution of the process. On the basis of such hypothesis and the requirements discussed in Section 2, five main classes of entities are introduced: *Buffers*, *Virtual*, *Positioners*, *Workers* and *Objects*, whose roles are detailed in Table 3.1.

Table 3.1

Entities	Role
Buffers	Real objects used to store the work-piece
Virtual	Virtual elements used to define synchronization and physical connections
Positioners	Real objects able to grip the work-piece
Workers	Real objects able to perform a specific process (e.g., a manipulator equipped with a proper tool)
Objects	Real objects that need to be processed (i.e., the work-piece)

An addition element, called PATH, defines the geometrical trajectory along which the processing has to be performed, with the addition of some features relative to the processing itself (i.e., the number of the involved work-pieces, the type of tool to be used, etc.). The HLM of the task uses the listed entities and the PATH elements (described in detail in Section 3.1.1) and takes into account the main aspects of its definition, i.e., the sequence of sub-tasks defining the user process and the physical constraints (e.g., the distance between robots inside the robotic cell),

merging two different models: i) the Spatial Constraints Graph (SCG) discussed in Section 3.1.2, and ii) the Functional Link Graph (FLG) detailed in Section 3.1.3.

3.1.1 Model Entities

The formal description of the five classes of entities is given hereafter by using italic fonts with the first letter capitalized for the classes, uppercase bold fonts for the entities, and lowercase italic fonts for the features, after the description of the main features of the PATH element.

PATH Such an element corresponds to the minimum task that can be executed by a machinery, so that it could be just a portion of the whole task; complex tasks can be obtained by a proper sequence of simple PATHs

Main features:

- ◇ *starting frame*: frame placed at the beginning of the path defining the starting position and attitude;
- ◇ *end frame*: frame placed at the end of the path defining the final point and attitude;
- ◇ *cartesian path*: curve to be followed during processing;
- ◇ *id_work*: identifier of the type of work involved (it is related to the type of tool to be used);
- ◇ *asbly_ref*: represents the number of work-pieces that are involved in the same path (processing); it is used when the processing affects different work-pieces, e.g., during the welding of two pieces;
- ◇ *action_type*: *joining*, *splitting*, *none*; it indicates whether the processing modifies the physical structure of the work-pieces, e.g., during welding two pieces could be joined but during a cutting process the work-piece can be split;
- ◇ *sync_ref*: defines a possible connection between the PATH and the **V_SYNC** node, which will be introduced later.

Buffers They are defined through five class features and three logical entities.

Class features:

- ◇ *buffer frame (BF)*: represents the reference frame of the buffer with respect to a common frame;

- ◊ *n connect frame (CF)*: defines n frames referring to the BF, each of which represents a possible location of the work-piece;
- ◊ *cf_status*: keeps the buffer states;
- ◊ *size*: corresponds to the number of objects the buffer can keep;
- ◊ *delay*: represents a time delay, which can be used to model a waiting time between the placing phase of the work-piece and the subsequent picking one (if present).

Logical entities:

- **IN (I)**: represents an input buffer for the process, i.e., a place where the work-piece can be picked by an object equipped with a proper gripper in order to start the process.
- **OUT (I)**: represents an output buffer for the process, i.e., a place where the work-piece can be placed at the end of the process.
- **In_Out (I_O)**: represents a buffer that can be used to store a work-piece during the execution of a task; it does not represent the starting or final point of the process but an intermediate one, so allowing the possible splitting of the task into sub-tasks that can be executed by different machineries. Such an entity can also be useful when the process implies the placing of the work-piece, e.g., when it is necessary to perform some measurements on the work-piece after a portion of the process. Sometimes such a procedure involves the usage of a specific instrument, which implies to place the piece in a predefined point; such a point can be modeled using the I_O entity.

Virtual This class includes two logical entities used in the definition of complex tasks.

Logical entities:

- **Virtual_SYNC (V_SYNC)**: it defines a synchronization between different tasks; it is used when there are portions of different processes (i.e., tasks carried out on different work-pieces) that must be synchronized; such an entity is then used to connect parts of the model needing a synchronization in order to obtain its mapping.

- **Virtual_Assembly (V_ASBLY)**: such an entity is used to manage a physical connection between different work-pieces; its usage will be discussed more extensively in future works.

Positioners During a process the positioner must take the work-piece from the input buffer and carry it into the common working area of all the machines involved in the process. During the working the positioner can slightly move the work-piece in order to facilitate the processing; finally, when the task is finished, it places the work-piece in the proper output buffer. The Positioners are defined through five class features and only one logical entity given by the **Positioner (P)** itself.

Class features:

- ◊ *positioner frame (PF)*: represents the reference frame of the positioner with respect to a common frame;
- ◊ *n connect frame (CF)*: defines n frames, one for each gripper, attached to the positioner (concept of virtual application point);
- ◊ *n gripper type (GT)*: identifies the type of gripper for each CF;
- ◊ *dof*: defines the number of degrees of freedom of the positioner in the Cartesian space;
- ◊ *wa*: defines the working-area of the positioner.

Workers They include manipulators equipped with the required tool, and are defined through four class features and only one logical entity, given by the **Worker (W)** itself.

Class features:

- ◊ *worker frame (WF)*: represents the reference frame of the worker with respect to a common frame;
- ◊ *tool type (TT)*: identifies the type of tool attached to the worker;
- ◊ *dof*: defines the number of degrees of freedom of the worker in the Cartesian space;
- ◊ *wa*: defines the working-area of the worker.

Objects They represent the work-pieces that must be processed, as defined by a set of PATH structures as shown in Figure 3.5. The object is a passive entity (i.e., it cannot perform any action alone), so that it is always connected to an entity able to keep it or carry it (e.g., *Positioner* or *Buffer*).

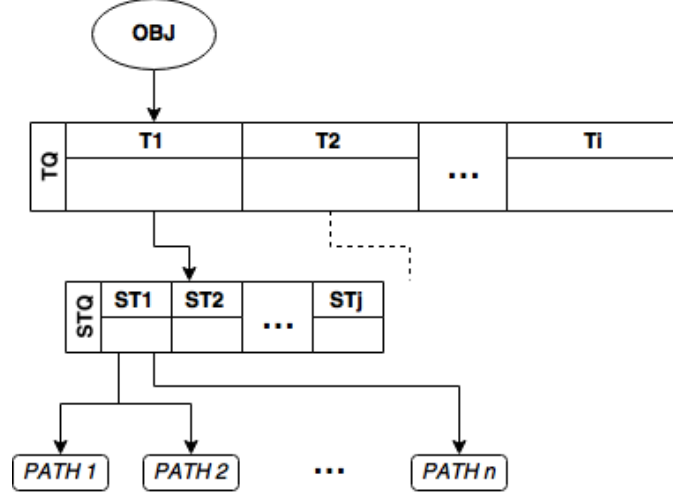


Figure 3.5: Structure of the object entity

The Objects are defined through five class features and only one logical entity, given by the **Object (OBJ)** itself.

Class features:

- ◇ *object frame (OF)*: represents the reference frame of the object with respect to the frame of the entity on which it is connected (i.e., the WF or the BF). Such a frame is used as a common base for all the other frames of the object;
- ◇ *n connect frame (CF)*: defines n frames with respect to OF, one for each possible coupling point of the object (i.e., the position that must be reached by a positioner in order to grip the object);
- ◇ *and_or_flag*: defines whether the coupling points must be all gripped at the same time (AND) or if only one coupling point is available (OR);
- ◇ *gripper_id*: identifies the type of gripper required for each CF;
- ◇ *task queue (TQ)*: represents a queue containing the list of tasks to be performed, so defining the order of execution. The *TQ* is characterized by:
 - *input buffer (I)*: represents the input buffer of the task;
 - *output buffer (O)*: represents the output buffer of the task;

- *sync_ref*: defines a possible connection with a **V_SYNC** node;
- *sub-task queue (STQ)*: represents a queue containing the list of Sub-Tasks (ST), each of which is composed by a set of PATH structures (minimum action executable by a worker) which can be performed in parallel execution. The STQ establishes the precedence of execution of the paths; its structure can be defined as:
 - * *path frame (PF)*: represents a reference frame defined with respect to OF, which is used as basis for the frames present in the PATH structure;
 - * *PATH*: as already defined.

The defined entities can be connected according to two criteria: i) the type of task they can perform, ii) their level of interaction, that can be related to the actual distance between the real objects; such a distance defines in some way whether a pair of objects (and hence of entities) can interact each other in order to perform a task. The possible relationship between a pair of entities can be defined by using a specific graph, in which each edge determines the presence of such a relation; two different data structures are then used to specify the two types of relationships: i) the Functional Link Graph defining the functional relations, and ii) the Spatial Constraints Graph defining a possible interaction between entities, as detailed in the next subsections.

Remark 1 In the proposed approach a general task is composed by a queue of tasks that properly sequences the operations (e.g., in automotive context a possible general welding task could be composed by three welding processes executed in a specific order by three different work-stations). Each task can be further divided into an ordered sequence of sub-tasks that allows a further level of abstraction in the task definition (e.g., each welding process could be composed by four welding sub-tasks, one for each car door). Finally a sub-task is composed by a set of basic processes (called PATH) possibly performed in parallel execution, where each PATH represents the minimum processing block assignable to a worker entity. The whole assigned task can be built as a succession of four basic steps, in which the “working” step represents the execution of a particular PATH.

3.1.2 Spatial Constraints Graph

The SCG defines the links between entities that are able to interact (in a physical sense) in order to carry out a task; only the entities that can have a spatial constraint belong to the *Buffers*, *Positioners* and *Workers* classes. The relationship between entities is defined by using both rough input information and their elaboration provided by a devoted Environmental Constraints block; possible links could then be defined on the basis of different information, e.g., the existence of

common working areas, or the manipulability indices of the involved workers. The spatial links are defined on the basis of some rules:

1. **Worker** entities can be linked only to **Positioner** entities
2. **Buffers** entities can be linked only to **Positioner** entities
3. A pair of entities can be linked only if the spatial constraints are satisfied
4. **Positioners** can be linked with others **Positioners** only if explicitly required

By the application of such rules, the SCG of a generic robotic cell can be built; a possible example of a SCG is shown in Figure 3.6, where the nodes corresponding to entities of the model have been divided on the basis of the belonging class, so highlighting the allowed connections.

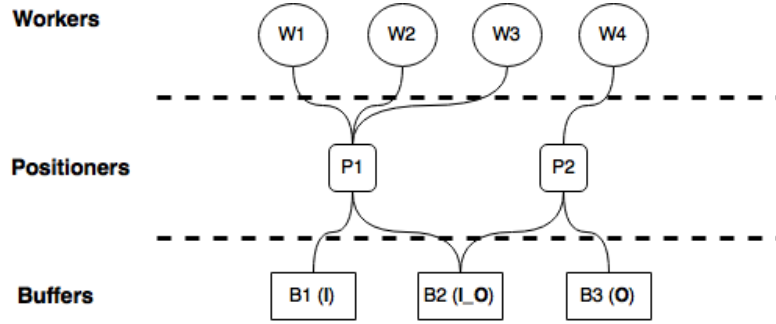


Figure 3.6: Example of definition of a Spatial-Constraints-Graph

The SCG basically defines the entities which are able to interact with each other, but taking into account their role in the model; for this reason some specific connection must be forbidden:

1. between a **Worker** and a **Buffer**: a worker cannot take an object but it can simply perform a process on it;
2. between two **Worker** entities: two workers cannot directly interact each other, they can only work on a work-piece connected to a positioner, so that they are allowed to interact with a **Positioner** entity;
3. between two **Buffers**: buffers are passive entities which simply contain (or keep) an object, so that they cannot interact each other.

The only entity allowed to be connected to any type of entity is the **Positioner**; in fact it interacts with the buffers in order to pick and place the objects, and also with the workers to keep the work-piece during processing. The connection between positioners is also allowed; such a link defines a possible flying exchange of the work-piece between two positioners without using any intermediate buffer.

3.1.3 Functional Link Graph

The Functional Link Graph defines the relation between a PATH and the available workers, so to have a connection between the PATH and each worker having the characteristics required to perform the corresponding task. For each object entity a FLG is then built by using the related set of PATH and worker entities (see Figure 3.7).

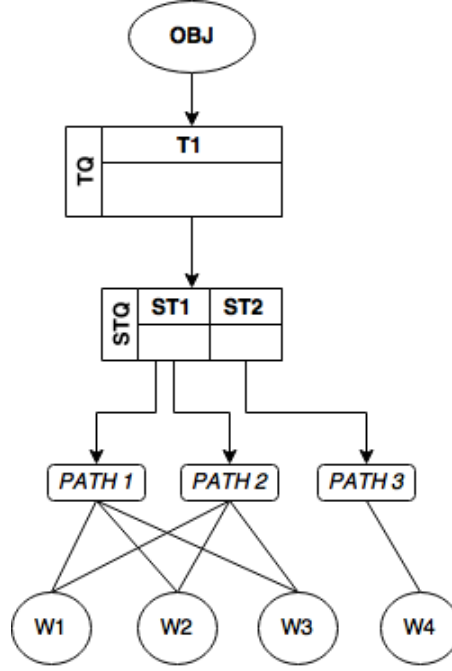


Figure 3.7: Possible example of a Functional Link Graph

The FLG takes into account different scenarios that could be actually present in a real robotic cell: i) several workers could be able to perform a particular task, ii) some tasks (or sub-tasks) could be synchronized and iii) some work-pieces could be physically modified during the process.

The first scenario was addressed by allowing the connection of a path with multiple workers; e.g., in the FLG in Figure 3.7 PATH1 is connected to W1, W2 and W3, corresponding to the set of available workers able to perform the task defined by PATH1.

The second scenario involves the possibility to synchronize Tasks, Sub-Tasks or PATHs belonging to different objects. Such a problem has been managed by introducing a proper link (i.e., *sync_link*), which is used to connect the tasks needing synchronization with the **V_SYNC** entity (Figure 3.8). The **V_SYNC** then groups the set of actions that have to be synchronized, so that some constraints must be respected in order to preserve the correctness of the whole model: 1) each Task, Sub-Task or PATH can have only one *sync_link* (tasks that are executed at

different time instances cannot be synchronized), 2) the *sync_link* must be defined so to avoid deadlocks in the process. A general rule is to avoid the synchronization of operations whose time sequence has been already constrained, e.g., two Sub-Tasks of the same object could not be synchronized because they are performed in serial execution by definition. A further example is shown in Figure 3.9, where the proposed set of *sync_link* defines the synchronization between PATH1.1 and PATH2.2 and between PATH2.1 and PATH1.2. Despite the synchronized paths belong to different objects, in such a case the proposed cross synchronization implies two conflicting constraints in the temporal execution of the involved paths; in fact the execution of PATH1.1 is subject to synchronization with PATH2.2, which can be executed only after PATH2.1, but the latter is waiting for the execution of PATH1.2 (due to the *sync_link*) that cannot be performed because the execution is stopped waiting for PATH1.1. In order to allow different types of synchronization, the **V_SYNC** includes a table containing for each node whether the operation must be synchronized with respect to its beginning, its end or both, so that it could be possible for instance to synchronize the beginning of a Task with the end of a Sub-Task.

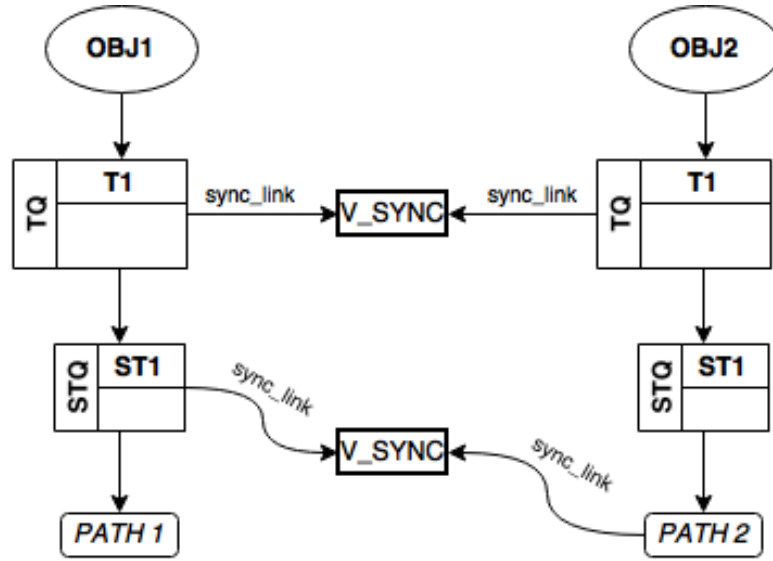


Figure 3.8: Using **V_SYNC** entities in a FLG in order to synchronize different parts of the process

The third scenario takes into consideration possible physical variations of the work-pieces. In fact, while some kinds of mutation do not influence the model of the process, like changes of the color or shape of the work-piece, some others could imply the change of the model, so to maintain its consistency with respect to the process (e.g., during a cutting process). Three possible actions on the work-piece are then introduced and stored in the *action_type* field of the PATH structure: i)

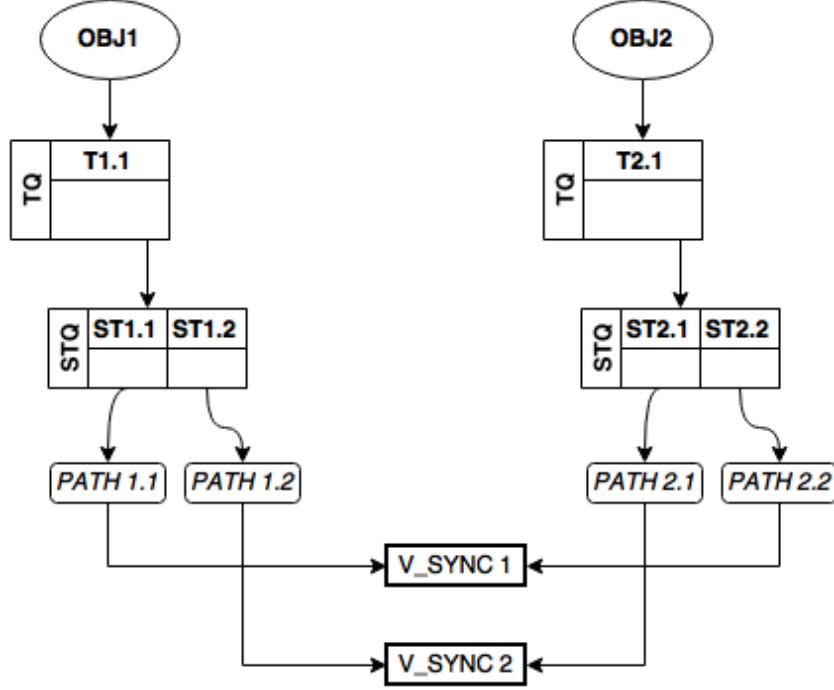


Figure 3.9: Example of a possible deadlock caused by an incorrect usage of the **V_SYNC** entities between two FLG models

joining, ii) splitting, iii) none; specific actions on the model are eventually performed on the basis of the value assumed by *action_type*.

When the action is of *joining* type, a new object entity, which properly merges the features of the starting objects, must be created; such an object inherits the uncompleted Tasks and Sub-Tasks of the starting objects, and could change some physical properties eventually stored for low level operations. On the contrary, when the action is of *splitting* type the starting object could be divided into a number of new objects, depending on the characteristics of the required process and on the number of positioners the work-piece is connected to. However not all the actions of *splitting* type, imply the creation of a new object; two main cases can be taken into account:

- *rejection*: if during a cutting process the cut part of the work-piece is not connected to a positioner, it will be discarded; in such a case the creation of a new object is not required, but a simple update of the physical features of the starting object could be eventually necessary (e.g., shape, weight, inertia). The rejection case can be recognized by analyzing the geometrical characteristics of the required process, the number of positioners connected to the work-piece and the positions of its CF.
- *division*: if during a cutting process the cut part of the work-piece is connected

to a positioner, it continues to be part of the process, so that a new object entity is then required. A division occurs whenever the cut is performed between two CFs actually used by two positioners, so that the cut part of the work-piece is kept by a positioner.

Finally if the process is neither of *joining* nor *splitting* type the starting object entity is kept; the three possible types of processes can then be summarized as in Table 3.2.

Table 3.2: Effects of the three types of actions on the HLM

Action	Effect on the HLM
joining	Decrasing of the number of object entities
splitting(rejection)	Number of object entities unchanged
splitting (division)	Increasing of the number of object entities
neither joining nor splitting	Number of object entities unchanged

Each part of a process is classified using the field *action_type* of the PATH structure; such a field is set up over the pre-elaboration phase of the starting information, during which both the geometrical paths and the tools required to perform the tasks are already known. Such information is important to reconstruct the feasible sequence of operations making the whole process, through a developed algorithm. In fact, the structure of the HLM should be dynamically changed in order to take into account changes on the number of work-pieces during the execution. In particular a task of *joining* type requires the fusion of different objects, so that its features are actually related to each involved object. In order to model such a situation, the connection of a PATH with different objects is allowed (*asbly_ref* > 1), which in practice means that the task affects all the involved work-pieces that must be properly aligned in order to perform the task (e.g., in Figure 3.10 PATH2 belongs to both ST1.2 and ST2.1, and hence to OBJ1 and OBJ2). A devoted algorithm has to properly merge the features of the objects involved in the joining task, so to obtain a final unique object, whose task queue is filled by a combination of the residual task of the involved objects. On the contrary, a task of *splitting* type could require the creation of new object entities; in such a case when the splitting task is analyzed by a proper algorithm, one or more new objects are created, each of which is composed by a set of tasks taken by the starting one on the basis of simple geometrical analysis.

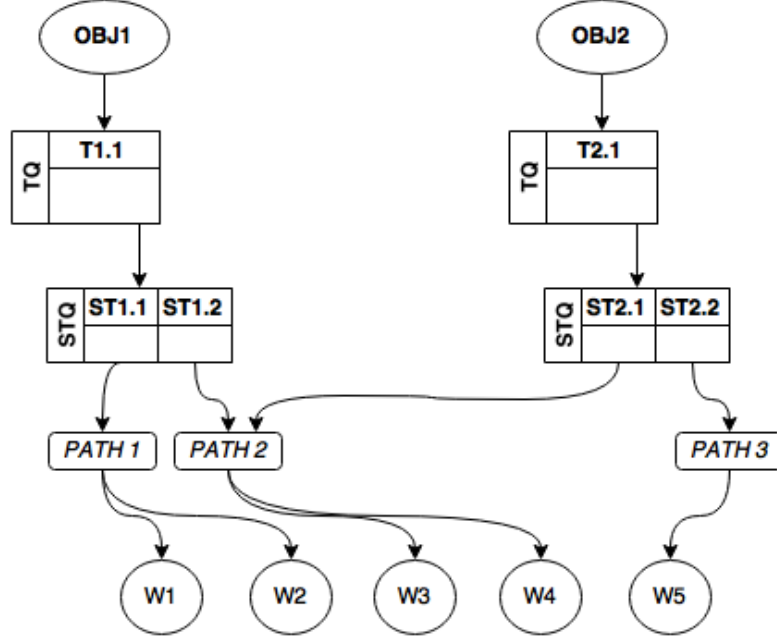


Figure 3.10: Definition of a possible *joining* action between two sub-tasks belonging to different FLG models

3.1.3.1 Complex blocks management

In the task modeling approach developed so far, a punctual correspondence between each real object and a single logical entity was assumed: for example, a Buffer entity was defined for each collection point or box for the work-pieces, as well as a Positioner entity was introduced for each robot able to perform a positioning action.

In some cases, the structure of the process includes complex parts that cannot be modeled by a single entity, or some *external* machinery, which contributes to the execution of the process itself, but that is not part of the robotic cell. Typical examples are given by measurement or quality control stations, which possibly introduce a delay in the process execution, or machinery devoted to specific material processing (e.g., an hydraulic press). Such stations and machinery cannot be considered as simple Buffers, in particular in case of mutually exclusive stations that can alternatively perform the same operation on the work-pieces. In this kind of situation, standard Buffer entities could be used to model such elements, but associating to them also a predefined delay, and giving the user the possibility to explicitly define the exact sequence of the stations.

A general, convenient solution is here proposed, considering such parts of the process as *complex* blocks, which are modeled as complex entities obtained by the connection of some fictitious standard blocks. Figure 3.11 sketches the case of a measurement machinery M1, represented using a dashed line to define it as a special

block; $M1$ is converted into the queue of three (fictitious) standard blocks: the Worker W_{M1} , considered as able to perform the measurement task, the Positioner P_{M1} , for the virtual positioning of the work-piece to be measured, and the Buffer B_{M1} , collecting the workpieces. This solution allows to let the measurement action become a proper task, with the possibility to include priority constraints, if any, as well as to set mutually exclusive constraints in case of more machines, or even constraints about the capability of only some stations to perform specific actions.

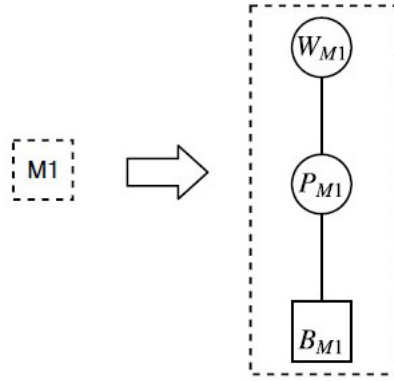


Figure 3.11: Modeling of a *complex* block

If the complex block does not include a working action, only Positioners and Buffers are used, e.g., to represent a net of conveyors in a packaging process.

3.1.4 High Level Model

The High Level Model is a model representing the features of both the process and the robotic cell; it is built after the definition of a SCG model representing physical relations between the entities, and a set of FLG models defining the relation between the tasks of a specific work-piece and the available workers. The HLM is then built by fusing the SCG and the n FLGs (one for each object) so obtaining an overall model, which can be used to define all possible sequences of basic operations that must be performed to implement the required process. As highlighted in Figures 3.6 and 3.7 both the SCG and FLG models include links to **Worker** entities, so that the two independently developed models are actually connected by the set of workers. The final model shown in Figure 3.12 is then completed by adding a link between each object entity and the input buffer of the corresponding process. Such an **action** is necessary because **OBJ** is a passive entity, so that its own reference frames are relative to those of the entity it is connected to (i.e., a **Buffer** or a **Positioner**). A possible algorithm during the building of the overall process should then change properly the link of the object entities in order to keep the consistency with the current situation in the real cell, e.g., when the work-piece must be picked

by a particular positioner the link must be placed to the corresponding entity. The scrolling algorithm (that will be described in Section 3.2.1) represents a very important part of the generation of all possible work-flows defining the required process, which can be built by exploiting some feature of the HLM. In particular its most important property is that each transition of the graph can be directly mapped into one of the four steps defined in Section 3.1.1, each of which corresponds to a real action performed by a specific machine (represented by the corresponding entity).

Remark 2 At this level of abstraction the goal is to find a general structure that allows to build a task by the correct sequencing of a set of basic steps. For this reason extensive details about robot features, motion and timings as well as possible ways to define the optimization problem, are not considered here. Just some basic information about Cartesian movements, frames and working-areas necessary to define physical constraints (e.g., to define the workers able to perform the task) are included in the model (e.g., the PATH structure includes the cartesian path in which the process must be performed). However a sort of time relation between tasks, sub-tasks and PATHs has been introduced by means of the synchronization node, which imposes some tasks, sub-tasks or PATHs to wait for others.

3.2 Work Flow Modeling

The work-flows given by different sub-tasks can be modeled using a set of predefined blocks, which impose the cause-effect condition between the sub-tasks. Given a generic work-flow, a possible model should take into account at least two kinds of relations: i) tasks carried out in series, and ii) tasks carried out in parallel. It is also important to introduce a synchronization relation taking into account possible interdependence between different sub-tasks. In order to include also the possibility to model different alternative ways to perform the same task, it is necessary to introduce a further type of relation, modeling sub-tasks carried out in mutual exclusion. The proposed model, called Work Flow Model (WFM), is based on five basic blocks:

- AND_Split: defines sub-tasks carried out in parallel
- OR_Split: defines sub-tasks carried out in mutual exclusion (alternative ways to define the same task)
- AND_Join: imposes the synchronization between parallel tasks
- OR_Join: allows to join alternative work-flows

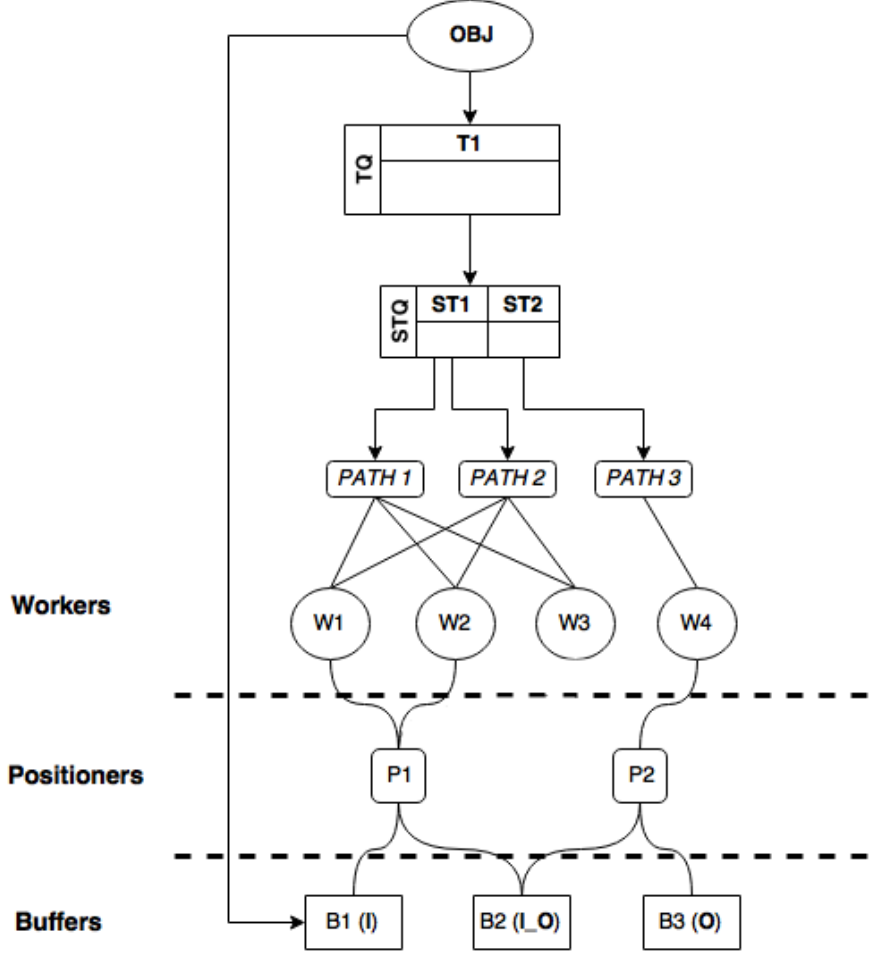


Figure 3.12: Overall HLM model

- **TASK**: defines a recursive block that can both include other basic blocks or a basic task.

A basic task corresponds to the workings (i.e., a real action) carried out by a specific machinery; it includes the sequence of instructions specific for the target machinery (e.g., path planning of a manipulator), and hence it must be defined on the basis of the actual available machineries and of the specific application to be modelled. For the proposed application, devoted to the programming of robotic cells mainly constituted by industrial manipulators, four basic tasks are defined:

- **TASK_pick(P, B, OBJ)**: defines the picking action of the work-piece (OBJ)
- **TASK_place(P, B, OBJ)**: defines the placing action of the work-piece (OBJ)
- **TASK_exec(P, W, PATH, OBJ)**: defines the actions carried out by the Worker in order to execute the sub-task described by a specific path (PATH)

- TASK_flypass(Pi, Pj, OBJ): defines the passage of the work-piece between two Positioners (Pi and Pj)

More complex blocks can be obtained composing the basic blocks of the WFM. In order to improve the level of recursivity of the model, a set of second level TASK blocks are then introduced:

- TASK_parallel: is defined by one AND_Split block and one AND_Join block, both connected to a set of n TASK blocks
- TASK_mutex: is defined by one OR_Split block and one OR_Join block, both connected to a set of n TASK blocks
- TASK_st_exec: is defined by one TASK block followed by n OR_Join blocks

The meaning of TASK_parallel and TASK_mutex is quite evident. TASK_st_exec is included to define the beginning of a new sub-task, as shown in Figure 3.13 where a possible example is reported.

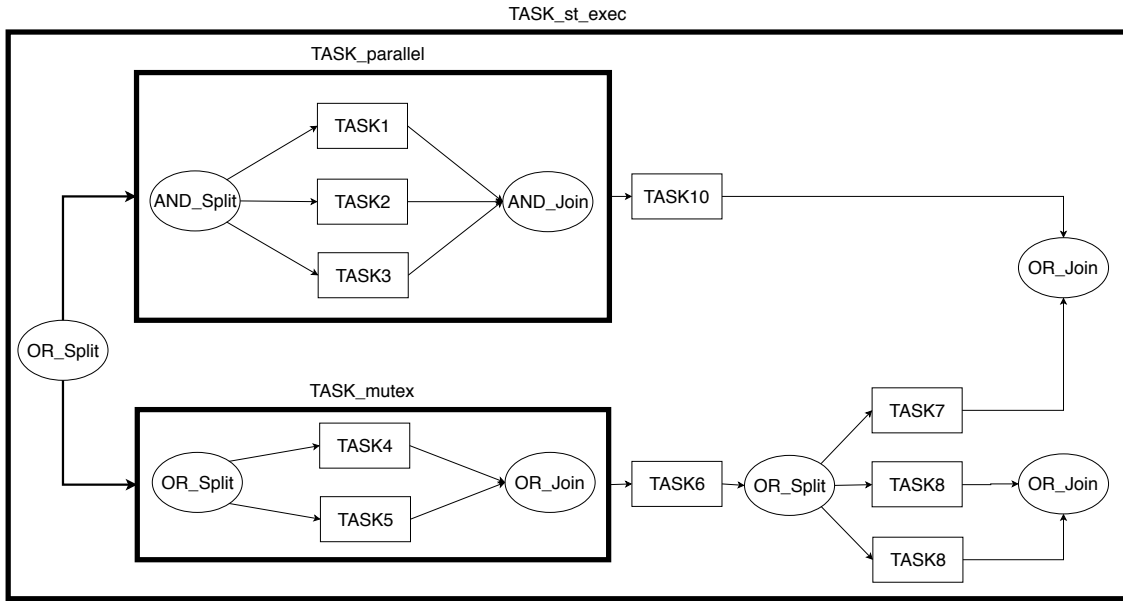


Figure 3.13: Example of a WFM composed by both basic and complex blocks

3.2.1 Automatic conversion HLM → WFM

The building of the WFM can be automatically obtained by the application of a proper algorithm to the HLM. The algorithm allows to scroll the HLM and to translate each transition into a corresponding block of the WFM using a set of

rules. For sake of simplicity the section firstly describes the scrolling algorithm and then the translation rules.

The algorithm here proposed is applicable to HLMs including one OBJ entity.

The pseudo-code illustrated in Algorithm 1 defines the main passages of the proposed scrolling algorithm.

Algorithm 1 HML_search(HML)

```

1: function HML_SEARCH(HML)( $wfm_h$ ,  $stack$ )
2:   while exists next sub-task do
3:     subtask  $\leftarrow$  get_NextSubTask()
4:     path_marking(subtask)
5:     for all starting_state  $\in$  current_states do
6:       DFS(G, starting_state, final_states)
7:       append(next_current_states, final_states)
8:     end for
9:     current_states  $\leftarrow$  next_current_states
10:    clear(next_current_states);
11:  end while
12: end function

```

Figure 3.14 shows the HLM already presented in Figure 3.12, with the addition of some passages of the algorithm highlighted in different colors. From the figure it is evident that the algorithm is based on two different flows, executed alternately, represented in blue and red, respectively.

The blue flow starts from the OBJ entity, and runs always in the upper part of the graph (i.e., the one highlighted in the dashed blue box) corresponding to the Functional Link Graph (FLG) of the HLM; the red flow starts from a specific position, corresponding to the current state of the robotic cell (i.e., the current entity which the OBJ is connected to), and runs always in the lower part of the graph (i.e., the one highlighted in the dashed red box), corresponding to the Spatial Constraints Graph (SCG) of the HLM. At each step the blue flow gets the next sub-task, extracts the corresponding PATH elements and marks them properly (corresponding to the functions get_NextSubTask and path_marking in Algorithm 1, respectively); in practice the first flow defines the goals of the second one. In the example of Figure 3.14 the blue arrows show that the flow firstly extracts the task T1 from the task queue and then the sub-task ST1 from the sub-task queue; subsequently PATH1 and PATH2 are marked.

The red flow starts from the current state (during the first cycle it corresponds to the input Buffer of the task) and applies a Depth First Search algorithm (DFS) in order to find the PATH elements marked during the last iteration of the blue

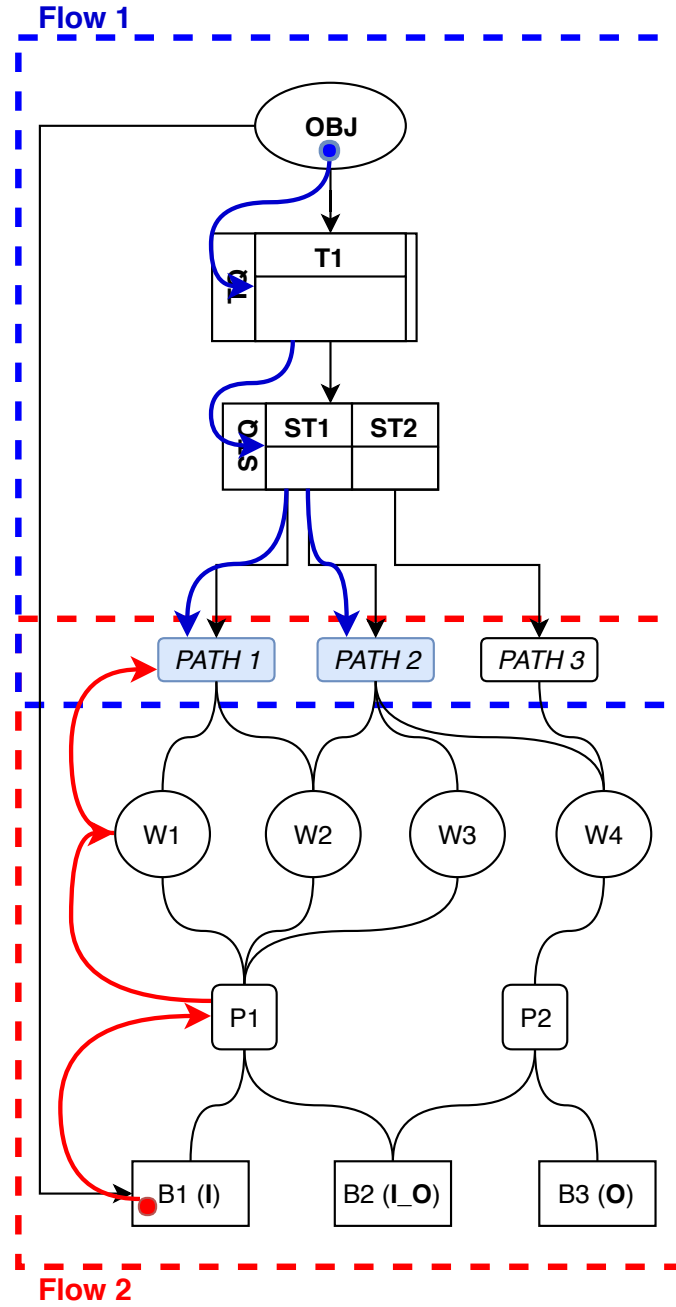


Figure 3.14: Example of HLM with some steps of the scrolling algorithm

flow. Since, depending on the graph topology, different ways to find the marked PATHs can exist (i.e., there are different ways to carry out the sub-task), the search algorithm tests all the possible graph paths; once all attempts are carried out, the red flow finishes its iteration and the blue one can make the next iteration by

marking the PATH elements belonging to the subsequent sub-task.

The example of Figure 3.14 shows that the red flow, starting from the input Buffer B1, scrolls the graph reaching the Worker W1, which is connected to PATH1. From a conceptual point of view, the transitions $B1 \rightarrow P1$ and $P1 \rightarrow W1$ allow to carry out the working defined by PATH1. Next steps (not reported in Figure 3.14) allows to reach W2 and hence to carry out the working defined by both PATH1 and PATH2.

Such an algorithmic cycle is repeated until all the sub-tasks are tested; if another task exists the procedure is repeated, otherwise a last cycle is carried out, during which the output Buffer must be found, if it is possible. In the proposed example, when the red flow finishes the iteration related to ST2 (i.e., the one that aims at finding PATH3), since ST2 is the last sub-task and since there are not further tasks (only task T1 has been defined), the blue flow triggers the last iteration. In such a phase the red flow, starting from the current state, scrolls the graph until the output buffer B3 is reached.

During the running of the red flow, the current state of the robotic cell is updated whenever the flow visits a Buffer or a Positioner entity (i.e., the only ones able to grip the work-piece); when the solution of the current iteration is found, the last state is stored (append function in Algorithm 1), and it will be used in the subsequent iteration as starting point.

Remark 3 It is important to notice that the red flow provides a solution of the i -th iteration only when it finds the set of transitions that allows to visit all the marked PATH elements. Since the PATH elements can be connected only to Workers and Workers only to Positioners (as discussed in Section 3.1.2), when the algorithm performs a transition from a Positioner to a Worker and then to a PATH element, the OBJ (and hence the work-piece) is still connected to the Positioner entity. For this reason when the red flow finds the last marked PATH, the current state is always given by a Positioner entity. The only exception is when the algorithm performs the last iteration, which terminates when the output Buffer is reached.

The translation of the HLM into the equivalent WFM is obtained by the application of a set of rules (see Table 3.3) applied during the transitions between the nodes of the HLM. For each row of the table a brief description is given:

1. When the procedure starts, it is necessary to define different parallel work-flows, one for each OBJ entity of the HLM.
2. When the first flow extracts a new sub-task, a corresponding TASK_st_exec block must be included in the WFM in order to take into account possible multiple ways to perform the sub-task.

3. The transition Buffer \rightarrow Positioner corresponds to the picking of the work-piece from the buffering position.
4. The transition Positioner \rightarrow Buffer corresponds to the placing of the work-piece into the buffering position.
5. The transition Positioner \rightarrow Positioner corresponds to the passage of the work-piece between two different positioners.
6. When the second flow visits a buffer, different possible work-flows can arise (it depends on the graph topology), so that an OR_Split block is introduced in the WFM.
7. When the second flow visits a Positioner, all the marked PATH elements directly reachable (i.e., those connected to the Workers linked to the current Positioner) must be executed in parallel. A TASK_parallel block is then added to the WFM, where the TASK_parallel has a number of parallel flows equal to the number of reachable marked PATH. For each parallel flow a TASK_mutex is also included to take into account the possibility that different Workers could be able to perform the same PATH.
 - (a) If not all the marked PATH elements are reachable passing from the current Positioner, it is necessary to test new paths in order to complete the sub-task (i.e., to find the remaining marked PATHs). An OR_Split block is then included in the WFM.
8. The transition Worker \rightarrow PATH element corresponds to the execution of a specific task described by the PATH element itself. A TASK_exec block is then properly included in the WFM.

Table 3.3: Translation rules

#	Transition/condition HLM	WFM block
1	Start of the procedure	AND_Split
2	Entry to a Sub-Task	TASK_st_exec
3	Transition Buffer \rightarrow Positioner	TASK_pick
4	Transition Positioner \rightarrow Buffer	TASK_place
5	Transition Positioner \rightarrow Positioner	TASK_flypass
6	Entry to a Buffer	OR_Split
7	Entry to a Positioner	TASK_parallel n TASK_mutex
7.a	If not all the PATH element can be executed	OR_Split
8	Transition Worker \rightarrow PATH	TASK_exec

Because of the presence of the OR_Split blocks, the translation of the HLM into the equivalent WFM can provide a number of work-flows that grows exponentially as the number of Buffer and Positioner entities increases. However Remark 3 states that, except for the last iteration, only Positioner entities can be used as final states of the red flow. Since the red flow provides all possible work-flows defining the sub-task, Remark 3 can be reinterpreted as: each sub-task may have different final states (one for each alternative work-flow), which in the worst case are equal to the number of Positioners included in the HLM. For this reason in the transition between two consecutive sub-tasks, there may be more then one starting state of the red flow, and all of them must be tested. Even if this involves an exponential growth of the WFM, thanks to the property that the possible final states are always the same (i.e., the Positioner entities), the exponential explosion can be at least limited as shown in Figure 3.15.

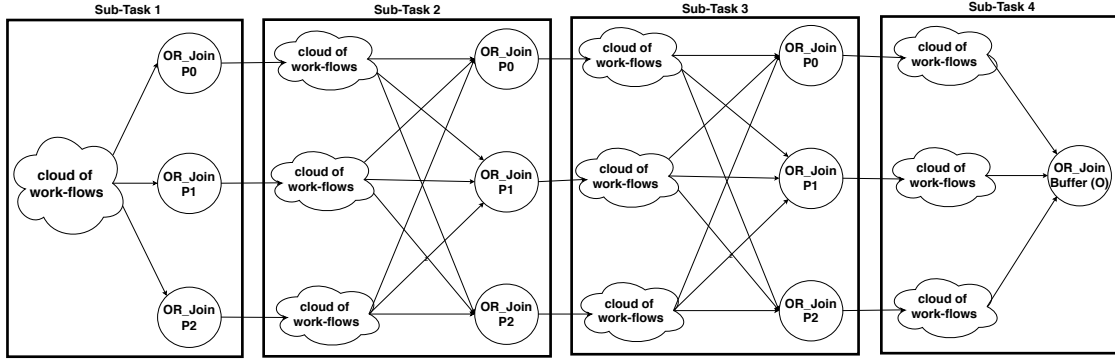


Figure 3.15: Sketch of a WFM obtained from a HLM with three Positioners called P0, P1 and P2

Complexity Analysis

A simple analysis, related to the scrolling algorithm only, has been carried out to approximately compute the asymptotic complexity in the worst case. Since the core of the proposed algorithm includes two nested algorithmic flows, and the internal one (i.e., the red one) is the most expensive one, the worst condition occurs when the internal flow is repeated the highest number of times. Since such a value corresponds to the number of the involved sub-tasks, the worst case is actually achieved when the number of sub-tasks is the maximum allowed for a specific set of PATH elements, and hence when each sub-task is composed by exactly one PATH

element. In such a case the external flow (i.e., the blue one), responsible for marking the PATH elements, has a constant complexity $O(1)$, since only one PATH element is included in the sub-task (the management of the task and sub-task queues, both included in the function `get_NextSubTask` of Algorithm 1, is neglected, since at each iteration only one extraction is performed). The red flow scrolls only the part of the HLM corresponding to the SCG in order to find the marked PATHs. Leaving out the particular composition of the SCG (given by Buffers, Positioners and Workers), it always corresponds to a not oriented connected graph $G(V, E)$ (V denotes the vertices and E the edges, as in Figure 3.16); possible vertices not connected to the main graph are simply ignored by the algorithm, since they correspond to machineries or other objects of the robotic cell that cannot physically interact whit the other ones.

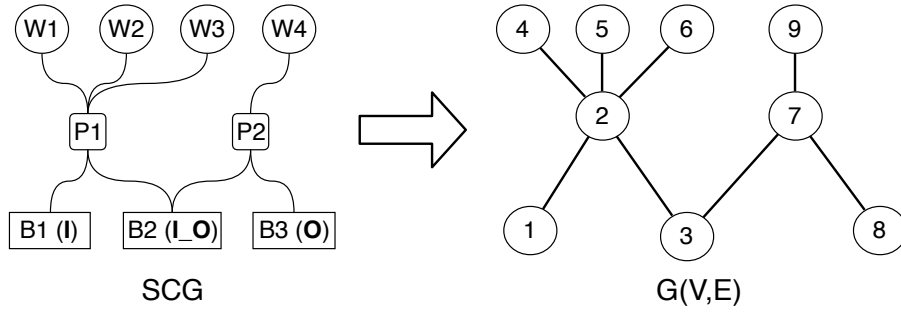


Figure 3.16: Graph $G(V, E)$ corresponding to the SPG of the proposed example

During each iteration, the DFS algorithm is applied in order to find the marked PATH elements; in general the complexity of a DFS is given by $O(n + m)$ when adjacency lists are adopted, where $n = |V|$ and $m = |E|$. The DFS is repeated for each starting state (given by Positioner entities only), so that the complexity can be rewritten as $O(n) \cdot O(n + m)$, supposing a very restrictive (and in general not realistic) condition in which all the nodes are Positioners. For each iteration, the two algorithmic flows are executed in series, so that the total complexity is given by $O(1) + O(n) \cdot O(n + m)$. Each iteration is also repeated for each sub-task included in all the sub-task queues; since the worst case occurs when the total number of sub-tasks is equal to the number of PATH elements k , the overall asymptotic complexity is then given by $O(k) + O(k) \cdot O(n) \cdot O(n + m)$, that can be written as $O(k + k \cdot n^2 + k \cdot n \cdot m)$. In general, in a not oriented graph the number m of edges is between $n - 1$ and $n \cdot (n - 1)/2$, so that, in the worst case, the overall complexity can be approximated as $O(k \cdot n \cdot m)$.

3.3 Case Study

A possible case study is herein presented in order to show the whole flow that allows to map a real robotic cell into the corresponding HLM model, and then computing the WFM. The proposed application involving the welding of car doors is composed by two phases: i) the car door taken from the collection point #1 must be welded according to a set of user-defined paths and then placed in the collection point #2, ii) the car door taken from the collection point #2 must be welded (using a different tool with respect to the first welding process) according to a further set of paths and placed in the collection point #3. The whole process can be defined by a mix of information about the process and the features of the robotic cell, which must be both provided by a high-level layer (e.g., a CAD software) which also provides all the geometrical features of the robots, the environment, the work-piece and the paths. The proposed robotic cell is composed by three collection points and six robots; in particular there are four robots equipped with a welding tool and two robots with a specific gripper for car doors; all the elements (i.e., robots and collection points) can be remapped into the corresponding entities as listed in Table 3.4, and then used to define the SCG model.

Table 3.4: Mapping of the the real objects into the corresponding model entities

Real object	Model entity
Collection point #1	Buffer B1
Collection point #2	Buffer B2
Collection point #3	Buffer B3
Robot 1	Worker W1
Robot 2	Worker W2
Robot 3	Worker W3
Robot 4	Worker W4
Robot 5	Positioner P1
Robot 6	Positioner P2

Figure 3.17 represents a sketch of the proposed robotic cell, where the colored regions define the working-areas of each robot. As highlighted in the figure, some robots cannot interact with each other; the building of the SCG model must then be performed avoiding their connection so preserving the actual set of feasible interactions. In the current example a simplified set of spacial constraints are taken into account, so that the SCG is made up using the building rules reported in Section 3.1.2 with the intersection between the working-areas as unique spatial constraint.

The set of constraints imply that: i) **P1** can be connected only to **B1**, **B2**, **W1**, **W2** and **W3**, ii) **P2** can be linked to **B2**, **B3** and **W4**. According to the

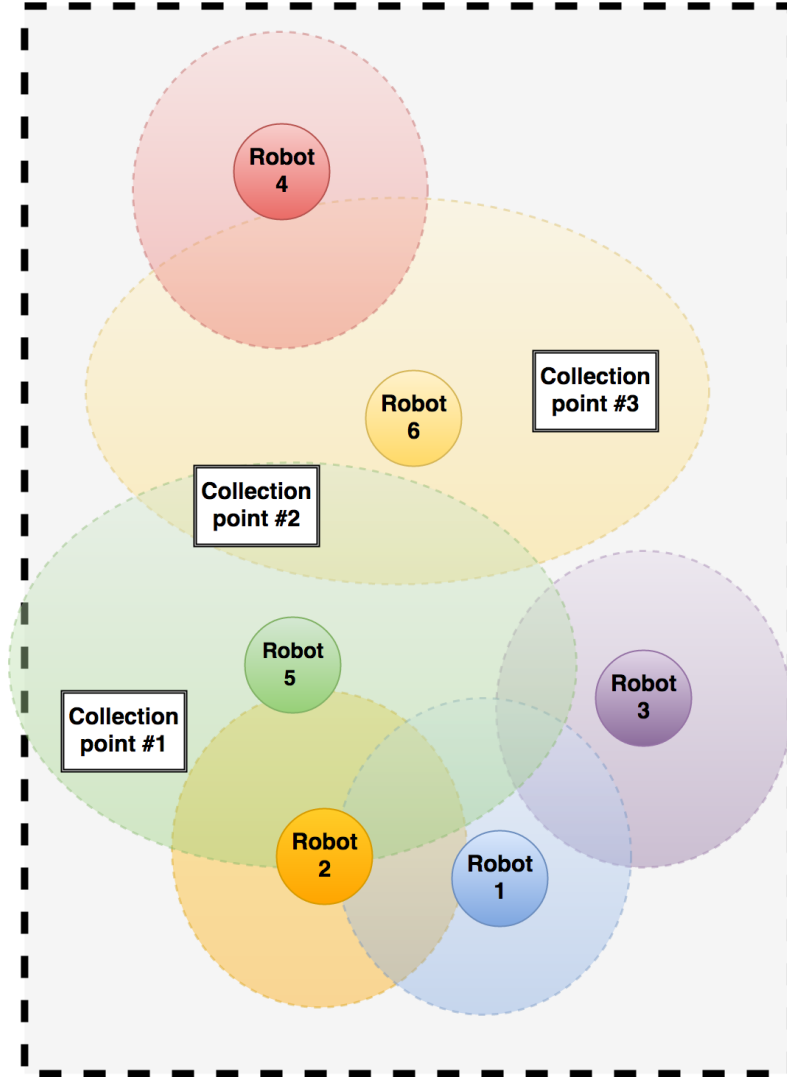


Figure 3.17: Sketch of the robotic cell of the case study

adopted spatial constraint **P1** and **P2** could be connected with each other as well, but it is supposed that their connection is not explicitly required (fourth rule for the building of the SCG). The SCG model representing the proposed scenario is shown in Figure 3.6, in which the buffer entity representing the input (**I**) for the process and the one representing the output (**O**) must be specified in the input information, whereas further buffers present in the model are imposed to be of type **I__O**; their actual usage for the definition of the whole work-flow is however mandatory only if explicitly requested.

For the realization of the unique FLG model (in fact only one work-piece is present), the sequence of tasks forming the whole process must be defined; such information must then be passed as input to the task model in order to create the task queue

of the object entity. In the proposed example the first sub-process is composed by two welds, respectively defined in PATH1 and PATH2, whereas the second one is composed by one weld whose features are stored in PATH3. The corresponding FLG is then defined as shown in Figure 3.7, in which the task connected to the only object entity is composed by two sub-tasks, where ST1 is connected to PATH1 and PATH2, whereas ST2 is linked to PATH3. The FLG is then completed by adding the connection between each path and the proper workers; in particular PATH1 and PATH2 can be performed only by **W1**, **W2** and **W3** (i.e., the only workers with the proper welding tool), while PATH3 can be carried out by **W4**, which owns the welding tool of the other type. The final HLM model is then shown in Figure 3.12 where the additional link connecting **OBJ** to the input buffer **B1** has been included.

Applying now Algorithm 1, the first flow extracts the sub-task ST1 given by PATH1 and PATH2, so that a new TASK_st_exec is defined; the second flow, starting from the input Buffer, scrolls the HLM in order to find the way to carry out PATH1 and PATH2. At the end of the first iteration the WFM in Figure 3.18 is obtained, where the final state is given by the positioner P1; no further states are found.

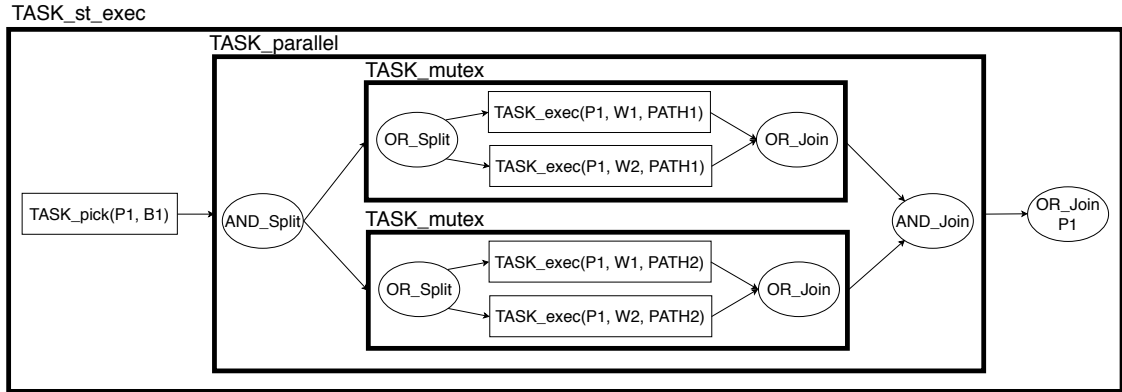


Figure 3.18: TASK_st_exec block corresponding to the ST1

The second iteration starts by extracting the PATH elements belonging to ST2 (i.e., PATH3) and defining a further TASK_st_exec. The second flow starts from P1 (the current state) and scrolls the HLM to find a Worker connected to PATH3. The WFM resulting from such iteration is represented in Figure 3.19, where unnecessary blocks have been neglected to preserve the clarity of the figure (e.g., the TASK_parallel automatically generated during the transition from B2 to P2 is omitted since it includes only one work-flow).

Once all the sub-tasks and tasks have been extracted, a third TASK_st_exec is defined and the last iteration is triggered. During such a phase the algorithm, starting from the current state P2, looks for the output Buffer. The WFM block

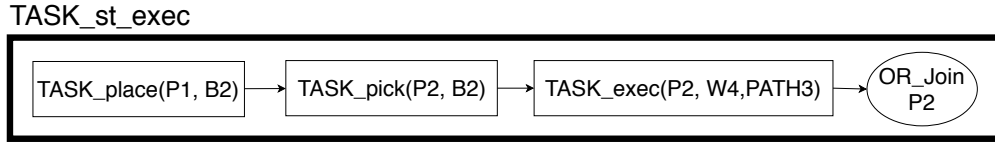


Figure 3.19: TASK_st_exec block corresponding to the ST2

corresponding to such a last iteration is represented in Figure 3.20.

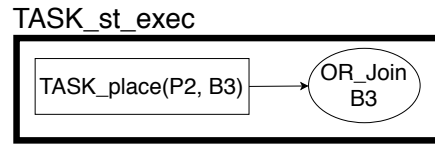


Figure 3.20: TASK_st_exec block defining the last iteration

The three TASK_st_exec blocks of Figures 3.18–3.20 must be intended as connected to each other, in order to preserve the actual cause-effect relations.

Chapter 4

The optimization phase

At the end of the third phase of the proposed Automatic Programming methodology (see Figure 3.4), the WFM of whole the process is provided, containing all the feasible, and alternative, work flows carrying out the process itself. The goal of the fourth phase (i.e., the Optimization one) is to find the best work flow according to some criteria, e.g., the cycle time or the energy consumption. The WFM, as shown in Figure 3.13, is provided in a form that is equivalent to an AND/OR graph. Such kind of notation, which is quite known, allows to represent sub work flows executed both in parallel and in mutual exclusion. In particular the OR nodes branch the work flows, providing alternative ways to carry out the same sub process. AND/OR graph is very known in the state of the art, and various algorithms to efficiently solve such an optimization problem can be found (e.g., AO* algorithm [1]). In order to actually apply such a kind of algorithm it is needed to define a set of performance indices, to be used to weight the graph. A way to obtain them is to use a robot simulator, e.g., the ORL robot simulator [66], [11], for COMAU robots. The ORL simulation environment allows to load a specific COMAU robot and carry out the motion simulation by applying a specific position target; it is also possible to change the motion performance of the robot by acting on a specific variable, called override (OVR), which defines the applied velocity as a percentage of the maximum allowed. The obtained simulation is very accurate, since it exploits the same motion algorithms used by the COMAU controller C5G. Using such an environment it is possible to simulate the tasks, and computing the corresponding (theoretical) values of cycle time or of energy consumption; such values can be then used within the optimization process to weight the graph. After that the optimization algorithm can be applied, providing the best work flow.

A second possibility is to translate the WFM into a corresponding Petri Net (PN). PN is a very suitable tool to model and to plan the task sequence of robots [81], [80], and in general of manufacturing systems. Such approach is feasible, in fact other works in the state of the art build a PN starting from an AND/OR graph[19]; synthesis procedure able to build a PN starting from a graph [18] can

be found, as well. The Activity-Oriented Petri Net proposed in [25] also shows that PN can be adopted to model the process as a set of tasks (activities) properly connected (as in the WFM), taking into account constraints given by possible shared resources; furthermore there is a lot of known theories to analyze the PN, e.g., in order to avoid dead locks of the process [39]. The possibility to optimize a PN is also feasible; different works can be found in the state of the art exploiting PN to solve optimization problems like Assembly Line Balancing Problems (ALBP) e.g., in [53], [54], and in [26], where Activity-Oriented Petri Net are adopted.

The optimization of the robotic cell has been treated to far as it were the only element of a production system (local optimization). Nevertheless, a different approach can include further elements in the optimization process, concerning the efficiency of whole the production line (and not only of the portion of which has been programmed using the proposed approach), in order to find an overall solution reducing possible bottlenecks of the plant. The possibility of integrating the proposed Automatic Programming methodology with a production efficiency tool allows to manage such kind of situations, making the robotic line part of a larger production process, so obtaining a final solution that optimizes the process with a global vision of the production line.

4.1 Integration with a production efficiency tool

This section shows how to include a production efficiency tool in the proposed programming methodology. Such tool is used in this context to optimize the overall process in a broader sense; it gives important information about the process itself, that the user can exploit to eventually modify the tasks execution in order to improve the performance of the process, minimizing possible bottlenecks. The section gives the basics about the adopted production efficiency tool, and then shows how to obtain an overall architecture of the system.

4.1.1 Basics of scalable production efficiency tool

The association of the WFM to a performance improvement method allows to approach the new visions in robotics that consider planning and acting as an inseparable, continuous and multiscale problem [35]. The present methodology gives a viable technique to cope with the new issues in robotic production planning and scheduling. The key of the method relies in a proper abstraction of the basic Overall Equipment Effectiveness (OEE) definition as a result of three efficiency factors

$$OEE = A_{eff} \times P_{eff} \times Q_{eff} \quad (4.1)$$

where: A_{eff} is the availability efficiency that describes the deleterious effects of production stoppages, breakdowns, setups and adjustments; P_{eff} is the performance efficiency that describes the productivity loss caused by reduced speed, minor stoppages, and idling; Q_{eff} is the quality efficiency, which takes into account the losses of production due to defects and rework. By considering the OEE of a productive unit as the rating of the achievement of a goal, the three factors capture separately the major aggregated aspect of even complex phenomena, through a suitable mapping with a cause-effect model of the physical evolution. The better the accuracy of the mapping, the best the strength of this OEE key performance indicator (KPI) is. At this point, several units can be coordinated to collaborate to a common superior parent goal, which is associated to an abstract entity that sees the units as its children. The four fundamental structures in which the children are coordinated (according to [68],[69]) are *series*, *parallel*, *assembly*, and *expansion*, as shown in Figure 4.1.

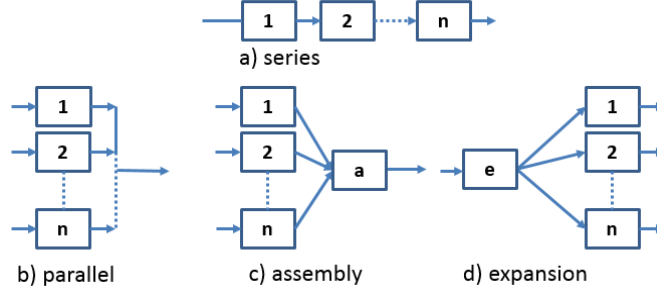


Figure 4.1: The four fundamental structures

The performance of any of the four coordinated structures of the children is in turn assessed by the OTE KPI that is obtained from a function of the OEE and the Q_{eff} of any of the children (see for example [15], [16] for complete expressions). The new parent abstraction can then be in turn a descendant of a similar structure at a higher level. In this case the OTE of the superior entity will depend recursively from the OTE of the descendant. This modelling and abstraction continues open-ended in a bottom-up and top-down manner whilst recursively matching the granularity of the production problem depending on the knowledge context. The overall production goal is decomposed into a hierarchy of goals and subgoals. At each level of the hierarchy a parent entity aggregates and monitors the achievement of the goals of its children. The kind of aggregation of the information from the parent depends on the attribution of one of the 4 fundamental structures in Figure 4.1, as a communication topology between the children and their assigned sub-goals. This constitutes a typical holonic vision, in which the entities are hierarchically participating into a holarchy, a semi-autonomous organization that aims at a same shared global goal [65]. The organization of the children connection at

each level of the hierarchy has been defined as *internal coordination* [65] or *implicit coordination* [31] in the multiagent systems context. Furthermore, the use of recursive relationships in the functional structuring of the holarchy helps in the design and development of the computing agents that are usually associated to the holons [38],[17]. As discussed in [78], the recursive properties enable the adoption of a fractal scheme of computing that is fundamental when the number of the entities and the levels of the tree is huge and with limited computing resources. The holarchy of a production process can be represented as a hierarchical tree as in Figure 4.2.

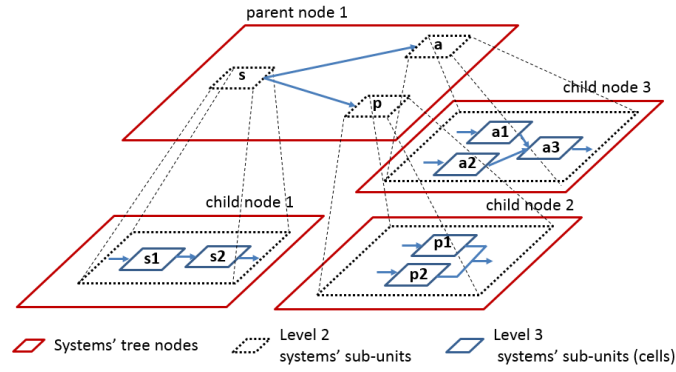


Figure 4.2: An example of tree-based relationship between parent and children structures

Having obtained such a tree structure, the iteration of the computing of the process bottlenecks programmes a list of possible step-by-step actions to be applied on the leaf nodes of the tree, towards a sub-optimal monotonic improvement of the system's performance (see [15] and [16]). Only one improvement action at time must be performed on the system, depending on a knowledge-based policy that takes into account human or artificial reasoning about the actual effects on the physical system. Note that the improvement actions programmed by the algorithm might not produce the theoretical expected effect in the environment, if the OEE model is not accurate enough. Nevertheless, the rate between the expected improvement and the actual one can be used as a measure for the triggering of the refinement of selected parts of the model or of the whole hierarchy. The overall advantage with respect to other methods is the minimum amount of refinement actions that render the system barely viable.

4.1.2 The proposed architecture

The WFM defined in Section 3.2 is a convenient tool for the planning and scheduling of a production. The status and the progress of a production process can be monitored, assessed or replanned basing on the quality, rates of accomplishment,

contingency states of the actions in the work flow. In this sense, WFM constitutes an opportunity for abstraction and aggregation of the production system's information, which reduces and simplifies the high dimensional and multivariate search space for the otherwise computationally hard problem of planning and scheduling. As described in Section 3.2, a task in the WFM is defined with a bottom-up approach from the HLM of the task, which contains informational details on the actual performance possibilities of the task, when planned in time, due to the instantaneous availability of resources and other physical and timing constraints. In this representation, there is the opportunity for a task in the workflow to be associated to a set of aggregate indicators that can be used from a planner and a scheduler to decide the contribution and the role to the overall performance of the task. If the task is a sub-task of some other composite task, made of tasks, split, and join elements of the flow model, then the problem becomes recursive and can be handled with the OTE recursive methodology. Such a methodology is applied seamlessly to the control of the workflow model of the robotic production process when the WFM topology is easily associable to the structures of Figure 4.1; the mapping and the interpretation of the task structures and the abstract system tree is in this case natural and straightforward.

Many concurrent interpretations and topologies can be simultaneously attributed to a WFM. In general the problem of the mapping between WFM and recursive OTE is open and manifold. Nonetheless, as in [69], where the topology and the interpretation of the production layout was well-determined and fixed, also here the well-determinate topology of a WFM can be usually exploited to establish an automatic generation of the OTE systems' tree. In the following Sections 4.1.3 and 4.1.4 we will describe how a simple WFM can be interpreted and managed in association to the recursive OTE methodology to obtain an effective control of the WFM, while in Section 4.1.5 the automatic algorithm generating the OTE systems' tree starting from a WFM is presented.

4.1.3 Computation of the efficiency parameters

The efficiency parameters P_{eff} , Q_{eff} , A_{eff} , introduced in Section 4.1.1 are now interpreted and computed in the scenario of a generic robotic cell; their definitions will be given in a general case applicable both in a completely off-line (simulated) context and in a possible implementation with feedback from the real plant.

The performance efficiency P_{eff} is considered as related to the cycle time required to execute the given task; the parameter has to be defined so that:

- its values are between 0 and 1;
- the unitary value corresponds to the ideal (not necessarily reachable in practice) minimum cycle time, whilst the zero value to an extremely high (undesirable) cycle time.

The quality efficiency Q_{eff} is used to take into account the energy consumption associated to the way the task is executed, with the aim to balance the search for a high performance efficiency (which would let the robot move at very high velocities) with the necessity of limiting the energy consumption, if possible, while guaranteeing anyway satisfying performances for the whole process. As for P_{eff} , the values of Q_{eff} must be between 0 and 1, with the unitary value denoting the *best* situation, i.e., the one corresponding to a very low energy consumption.

The availability efficiency A_{eff} is finally related to the expected number of working hours of the machinery, or equivalently to the number of cycles, carried out without any interruption due to faults or planned maintenance interventions. The values of A_{eff} , too, have to be between 0 and 1, with the unitary value corresponding again to the *best* condition, in which a very high number of cycles is performed without interruption.

Different mathematical expressions can be considered for the three efficiency parameters so to guarantee the desired behavior for them. The adopted solution is based on the use of a sigmoid function, whose properties are briefly recalled hereafter, before the formal definition and computation of P_{eff} , Q_{eff} , and A_{eff} .

4.1.3.1 Sigmoid function

The sigmoid function is real-valued, monotonic, and constrained by two horizontal asymptotes, which limit its values in the (0,1) range, making it suitable for our purposes. Its basic expression

$$sig(x) = \frac{1}{1 + e^{-x}} \quad (4.2)$$

can be generalized, introducing four parameters that allow to scale, translate and reverse its shape, if necessary, by re-defining it as:

$$sig(x) = \frac{a}{1 + b e^{-cx}} + d \quad (4.3)$$

where a determines the amplitude of the function, d its minimum value, and b a possible horizontal translation; the introduction of c has a twofold objective: (i) to vary the slope of the function by changing the module of c , and (ii) to make the function increasing or decreasing as $x \rightarrow \infty$, by changing its sign.

In order to avoid that the efficiency parameters values *collapse* to 1 or 0 in very good or bad situations, only a portion of the sigmoid function is actually employed in their computations, eliminating the initial and final parts almost coincident with the asymptotes, i.e., only a proper percentage $p\%$ of the function is used (e.g., $p\% = 90\%$), denoting with x_{min} and x_{max} the limit values of x corresponding to such a portion.

The four parameters a , b , c , d have to be set to suitably define P_{eff} , Q_{eff} , and A_{eff} . Since all the three efficiency parameters vary between 0 and 1, it follows that

$a = 1$ and $d = 0$ must necessarily hold for all of them. The other two coefficients, b and c , are instead used to adapt the sigmoid function to the specific behavior of each efficiency parameter, according to some guidelines that allow to determine general expressions for them.

A linear translation along the x axis is achieved by imposing:

$$b = e^{cx_c} \quad (4.4)$$

where x_c is such that $\text{sig}(x_c) = 0.5$, and is simply given by $x_c = (x_{\max} + x_{\min})/2$.

The sign of c must be discussed and imposed with reference to the specific behavior of each efficiency parameter as a function of the independent variable chosen to model it (i.e., what is x for such a parameter and if it increases or decreases as $x \rightarrow \infty$).

The absolute value of c , defining the function slope, must be computed so to normalize the function itself between 0 and 1 in the adopted range corresponding to $x_{\min} \leq x \leq x_{\max}$. This property is guaranteed imposing:

$$\begin{cases} \text{sig}(x_{\min}) < d + a - \Delta \\ \text{sig}(x_{\max}) > d + \Delta \end{cases} \quad (4.5)$$

where Δ indicates the difference between the asymptotic value and the adopted bound (see Figure 4.3), given by:

$$\Delta = a \left(\frac{1 - p\% \cdot 0.01}{2} \right) \quad (4.6)$$

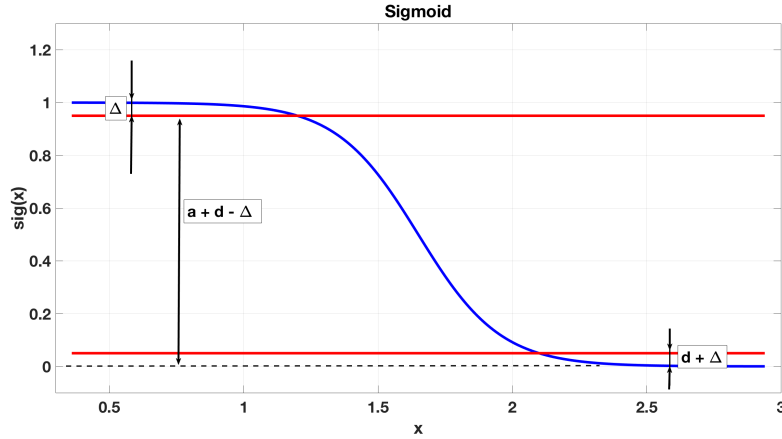


Figure 4.3: The adopted sigmoid function

Equations (4.5) provide an upper and a lower bound for c , given by:

$$c_{\max} := \frac{\ln(\frac{a}{\Delta} - 1)}{x_c - x_{\max}} \quad c_{\min} := \frac{\ln(\frac{a}{a-\Delta} - 1)}{x_c - x_{\min}} \quad (4.7)$$

However, since x_c was chosen as $(x_{max} + x_{min})/2$, the values of c_{max} and c_{min} are always equal to each other for every pair of vales x_{max} and x_{min} . Any of the expressions in (4.7) can be then used to compute c .

4.1.3.2 Computation of P_{eff}

The cycle time values associated to the tasks connected to Assembly and Expansion blocks are assumed to be available. The performance efficiency P_{eff} is then modeled by using the sigmoid function (4.3):

- considering the cycle time as independent variable x ,
- defining a proper usage percentage $p\%$ of the sigmoid (e.g., 90%),
- setting x_{min} and x_{max} equal to the minimum and maximum cycle times values, respectively,
- imposing $c < 0$, so to let P_{eff} tend to 1 for decreasing values of the cycle time.

4.1.3.3 Computation of Q_{eff}

The energy consumption values associated to the tasks connected to Assembly and Expansion blocks are assumed to be available. The quality efficiency Q_{eff} is then modeled by using the sigmoid function (4.3):

- considering the energy consumption as independent variable x ,
- defining a proper usage percentage $p\%$ of the sigmoid (e.g., the same adopted for P_{eff} , i.e., 90%, or a different one),
- setting x_{min} and x_{max} equal to the minimum and maximum energy consumption values, respectively,
- imposing $c < 0$, so to let Q_{eff} tend to 1 for decreasing values of the energy consumption.

4.1.3.4 Computation of A_{eff}

Information about the probability of machinery breaking or fault with reference to the number of working hours is assumed to be available, as well as the cycle time values associated to the tasks connected to Assembly and Expansion blocks (as for P_{eff}). The expected number of cycles without interruption, n_{cycles} , can then be computed in millions of cycles as:

$$n_{cycles} = 10^{-6} \left(\frac{brk_{hours}}{cycle_{time}} \right) \quad (4.8)$$

where brk_{hours} is the expected number of hours without breakages or faults, and $cycle_{time}$ is the cycle time expressed in working hours of the machinery.

The availability efficiency A_{eff} is then modeled by using the sigmoid function (4.3):

- considering n_{cycles} as independent variable x ,
- defining a proper usage percentage $p\%$ of the sigmoid (e.g., the same adopted for P_{eff} , i.e., 90%, or a different one),
- setting x_{min} and x_{max} equal, respectively, to the minimum and maximum number of cycles without breakages or faults,
- imposing $c > 0$, so to let A_{eff} tend to 1 for increasing values of n_{cycles} .

4.1.4 Case Study

A case study is presented in order to evaluate the performance of the proposed architecture. The case study involves three main steps: i) the automatic generation of the WFM starting from some input information, including the definition of the robotic cell and a formal description of the required process, ii) the translation of the WFM into a corresponding OTE systems' tree, and iii) the usage of the recursive OTE methodology on the OTE systems' tree.

The proposed application is a typical pick&place task carried out inside the robotic cell showed in Figure 4.4.

The cell is composed by two COMAU industrial robots, both equipped with a proper gripper: i) a Racer 7 - 1.4, and ii) a NJ4 110 - 2.2, which have been intentionally chosen with very different characteristics, so to empathize the work of the OTE methodology. As highlighted in Table 4.1, the Racer 7 is a small size robot with a low load capacity, but a very high performance in terms of velocity; on the contrary NJ4 110 - 2.2 has a maximum payload about fifteen times higher, but it is slower and has a higher energy consumption, due to its greater joints torque.

Inside the robotic cell there are also: a ball, which is the only work-piece for the given case study, and two boxes (that are used as collection points for the work-piece), the first one containing the work-piece at the beginning, and the second one in which the work-piece must be placed at the end of the application.

In order to build the HML corresponding to the required application, information about the robots (including the tools), the robotic cell, the work-piece, and the process are needed. Simple XML files (created specifically for such an application), containing information about the robots, the ball and the boxes, are used. The definition of a generic pick&place application is quite simple in the domain of the HML, since it does not require any machining on the work-piece, but just to move it. The task is then described by three pieces of information: 1) the work-piece

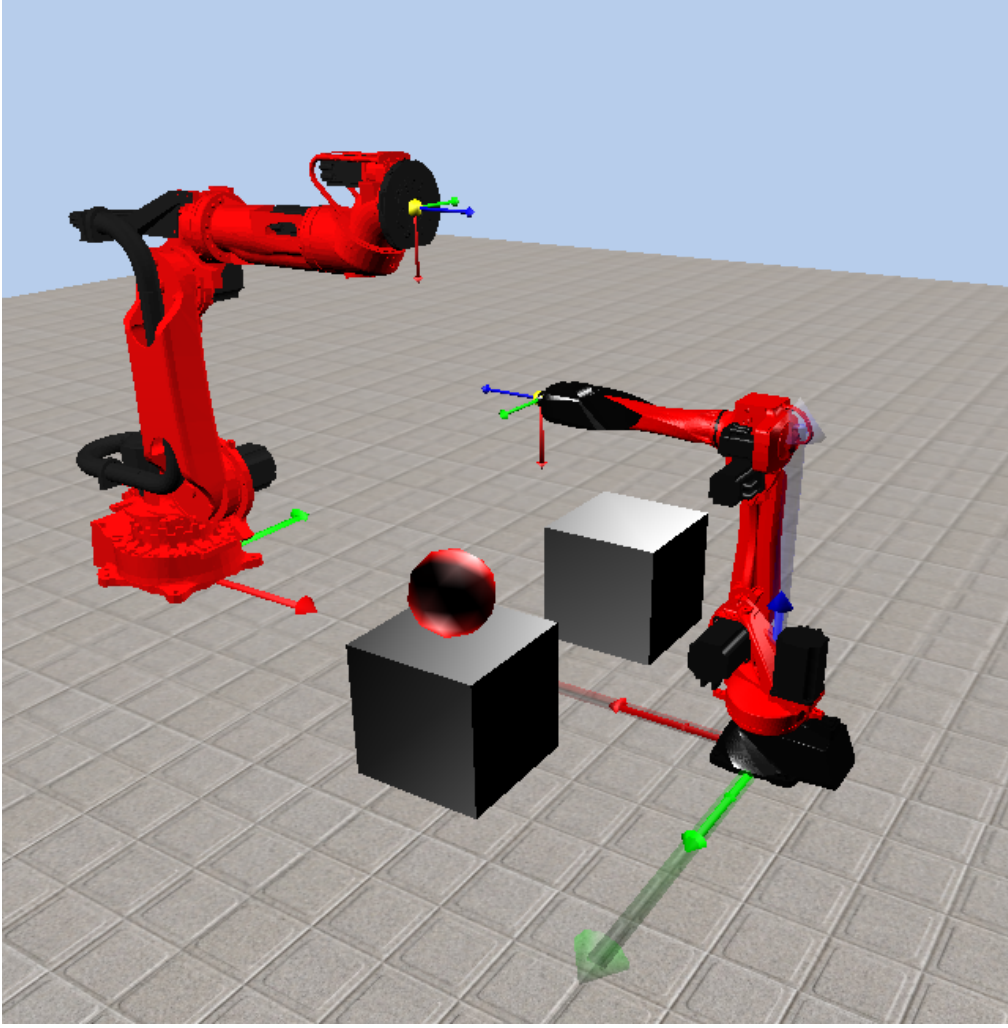


Figure 4.4: Representation of the proposed robotic cell obtained using Blender

involved in the process, 2) the collection point containing the work-piece at the beginning of the process, and 3) the collection point in which the work-piece must be placed; for the proposed case study the process is then described by the ball, box1 and box2. Using the rules described in Section 3.1.3, the real objects imported from the XML files are mapped into the corresponding HML entities, as reported in Table 4.2.

Physical constraints are computed starting from the input information, and used to build the HML showed in Figure 4.5. The WFM is then obtained by the application of the algorithm described in Section 3.2.1. As shown in the upper part of Figure 4.6, two types of basic TASKs are used in the WFM (i.e., TASK_pick and TASK_place), describing the actions of picking and placing of the work-piece, including the planning of the Cartesian trajectory to be followed.

Specification	Racer 7 - 1.4	NJ4 110 - 2.2
Number of axes	6	6
Maximum wrist payload (kg)	7	110
Maximum horizontal reach (mm)	1436	2210
Torque on axis 4 (Nm)	13	796
Torque on axis 5 (Nm)	13	609
Torque on axis 6 (Nm)	7.5	284
Speed Axis 1	220°/s	170°/s
Speed Axis 2	250°/s	125°/s
Speed Axis 3	300°/s	165°/s
Speed Axis 4	550°/s	200°/s
Speed Axis 5	550°/s	165°/s
Speed Axis 6	600°/s	265°/s
Robot weight (kg)	180	685

Table 4.1: Comparison between some technical specifications of a COMAU Racer 7 - 1.4 and of a COMAU NJ4 110 - 2.2

Cell objects	HML entity	HML symbol
ball	Object1	OBJ1
box1	Buffer1 (input)	B1
box2	Buffer1 (output)	B2
Racer 7 - 1.4	Positioner1	P1
NJ4 110 - 2.2	Positioner2	P2

Table 4.2: Mapping between real object and HLM entities

The TASK blocks also include some parameters defining the elements involved in the task, e.g., TASK_pick(OBJ1, P1, B1) means that Positioner1 picks the Object1 from the Buffer1.

The WFM is translated into a OTE systems' tree by converting the OR_Split block and the connected TASKs into an expansion structure, and converting the OR_Join and the connected TASKs into an assembly structure. The expansion and the assembly structures are then included into a series. The lower part of Figure 4.6 shows the so obtained OTE systems' tree; the corresponding mapping between WFM blocks and sub-systems is reported in Table 4.3.

The three efficiency parameters (P_{eff} , Q_{eff} , A_{eff}) required by the recursive OTE performance improvement method are computed through sigmoid functions as detailed in Section 4.1.3. The information necessary for setting x_{min} and x_{max} for each of them is collected by using the simulator ORL already introduced in Section 4.

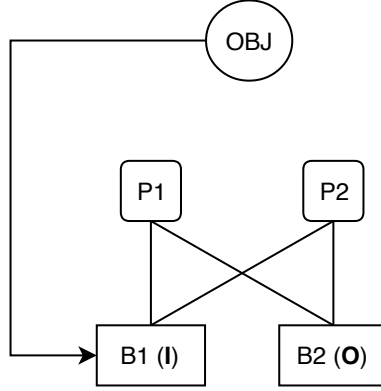


Figure 4.5: HLM corresponding to the proposed case study

WFM block	Sub-system
OR_Split	e1
TASK_pick(OBJ1, P1, B1	e2
TASK_pick(OBJ1, P2, B1)	e3
OR_Join	a1
TASK_place(OBJ1, P1, B1)	a2
TASK_place(OBJ1, P2, B1)	a3

Table 4.3: Mapping between WFM blocks and sub-systems

In order to compute the parameters x_{min} and x_{max} , for each sub-system corresponding to a TASK block (i.e., e2, e3, a2, a3) the planned Cartesian trajectory is simulated for different override values, and two performance indices, i.e., the cycle times and the energy consumptions, are computed and stored for each OVR value. The sub-systems belonging to the same expansion or assembly structure are regrouped, and the values of their efficiency parameters are normalized with respect to the performance indices of all the sub-systems belonging to the same structure. For example P_{eff} is computed for e2 and e3 taking into account the cycle times of both of them, since they belong to the same expansion structure; the parametrization of the sigmoid function is then obtained by considering the values of cycle times related to both e2 and e3, and imposing x_{min} equal to the overall minimum value, and x_{max} to the maximum one; $p\%$ is chosen equal to 90, whereas $sign(c)$ is set negative (see Section 4.1.3.2).

The procedure to parameterize P_{eff} for a2 and a3 is similarly carried out, since they belong to the same assembly structure, as well as the computation of Q_{eff} for the two groups of subsystems, i.e., e2–e3 and a2–a3, on the basis of the energy consumption values, as detailed in Sections 4.1.3.2 and 4.1.3.3.

Also the third efficiency parameter, A_{eff} , is computed and similarly normalized

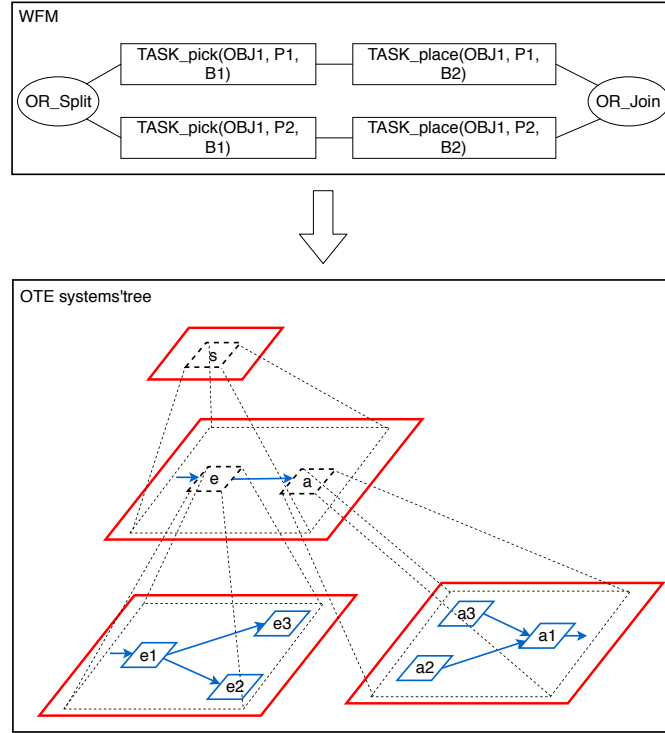


Figure 4.6: Conversion of the WFM into a corresponding OTE systems' tree

for the two groups of subsystems, on the basis of the expected number of cycles without interruption given in (4.8), with brk_{hours} set to tens of thousands of hours, corresponding to several years of work without failures on average (see Section 4.1.3.4).

By the physical mapping interpretation obtained for the three parameters P_{eff} , Q_{eff} , A_{eff} through the sigmoid functions (see section 4.1.3), a relationship between OEE and one controlling aggregate OVR parameter is obtained. In Table 4.4 are shown as an example the mappings that allow to associate a physical meaning to a desired level of the OEE. These relations are then used to implement the actions that lead to improvements in the robotic cell.

Using the proposed parametrization, each efficiency parameter provides values between 0 and 1, thus evaluating the performance of a specific sub-system in relation to the other sub-system belonging to the same structure.

Note that, purposely for the specific example, it has been adopted a modeling of the $OEE(A_{eff}, P_{eff}, Q_{eff})$ mapping in a non separable nor closed form as in the ideal case of (4.1). The separability of the variables depends mostly on the interpretation. When the modeling using a simple interpretation is not accurate enough, a new lower level can be spawn (new nested structure of sub-tasks) and so the granularity of the variables in the modeling grows; the granularity trades off

with the amount of information needed to each variable to accurately model the underlying physical phenomena. In general, the suggested approach is lazy; lazy modeling is to specify a coarse but viable model at first and then refine only if the improvement process identifies the need to improve the model's accuracy. In spite of the residual complexity hidden in the first viable model, using the aggregate information provided by OTE/OEE performance improvement method, it is still possible to apply corrective actions that improve the global performance.

OVR	OEE	Peff	Qeff	Aeff
5	0.002373	0.050000	0.949396	0.050000
10	0.032889	0.520679	0.947217	0.066686
15	0.062474	0.748820	0.943398	0.088436
20	0.090746	0.831579	0.937623	0.116385
25	0.122705	0.869900	0.929369	0.151777
30	0.159908	0.891087	0.917938	0.195495
35	0.202648	0.904352	0.902234	0.248362
40	0.249161	0.913284	0.880994	0.309672
45	0.296687	0.919710	0.852103	0.378578
50	0.340411	0.924528	0.813456	0.452636
55	0.374676	0.928322	0.761521	0.529999
60	0.391369	0.931306	0.694636	0.604974
65	0.386457	0.933745	0.612748	0.675447
70	0.355854	0.935776	0.514637	0.738923
75	0.305617	0.937458	0.411205	0.792808
80	0.241367	0.938928	0.306432	0.838903
85	0.175596	0.940198	0.213112	0.876370
90	0.119554	0.941334	0.140056	0.906815
95	0.074960	0.942260	0.085648	0.928832
100	0.044663	0.943146	0.050000	0.947112

Table 4.4: Mapping between OVR , OEE , and the A_{eff} , P_{eff} , and Q_{eff} factors composing the OEE for the specific example interpretation of the picking task of the NJ110 - 2.2 unit, corresponding to the $e2$ subsystem.

With the modeling here adopted, the technique consists in simulating all the sub-systems (using the ORL simulator) by setting at the first step the robot override to 50%, and then using the corrective actions (always in terms of velocities) computed and suggested by OTE performance improvement method, to adequately modify the override value and then perform a new monitoring of the status. Such an approach is iterated until the OTE recursive methodology finds its best solution. Figure 4.7 shows the evolution of the OTE with respect to improvement iterations for two different policies, favoring higher P_{eff} and A_{eff} , or a higher Q_{eff} . Actually,

by referring again to Table 4.4, it can be seen that a new target OEE can be reached by different combinations of A_{eff} , P_{eff} , and Q_{eff} . The selection is then driven by preferring an OEE obtained with higher Q_{eff} (so the focus is on energy efficiency), or one of the other parameters (P_{eff} and A_{eff}). In our case, the choice of A_{eff} or P_{eff} remains indistinguishable. An example of the actions selection biased towards Q_{eff} is in Table 4.5. The *Selected* column shows the choice made at each iteration with respect to the possibly multiple alternatives. The sequence of the selected actions produces the improvement of Q_{eff} , as shown by the red line in Figure 4.7.

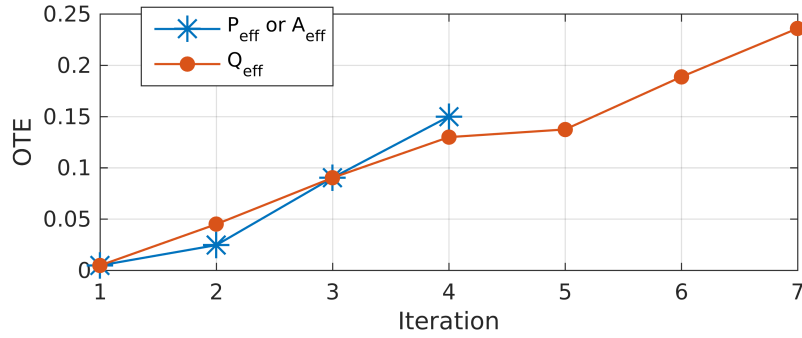


Figure 4.7: OTE evolution using actions that favor higher P_{eff} and A_{eff} (in blue), or higher Q_{eff} (in red)

As better explained in [15] and [16], the improvement can rely on other degrees of freedom beyond OEE.

Iteration	Sub-system	Aeff	Peff	Qeff	OVR	Selected
1	a3	0.091562	0.75169	0.94343	15	1
1	a3	0.93347	0.9434	0.085098	95	0
2	e2	0.53673	0.94309	0.93955	55	1
3	a2	0.083577	0.86661	0.94828	15	1
3	a2	0.90868	0.95	0.76458	100	0
4	a3	0.12064	0.834	0.9376	20	1
4	a3	0.93347	0.9434	0.085098	95	0
5	a2	0.13645	0.91498	0.94469	25	1
5	a2	0.90868	0.95	0.76458	100	0
6	a3	0.15733	0.87191	0.92931	25	1
6	a3	0.91174	0.94245	0.13894	90	0

Table 4.5: Selection of implemented actions, for the higher Q_{eff} preference case.

4.1.5 Automatic conversion of the WFM into a subsystems' tree


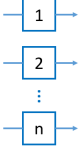
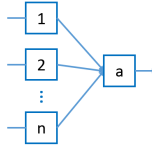
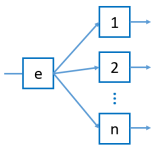
In Section 4.1.4 a case study has been presented, in which the conversion between WFM and subsystems' tree (Table 4.3) was performed by hand, so leading to an overall architecture not fully automatized. An automatic conversion is then described hereafter based on a set of translation rules. In the first part of this section some propaedeutic definitions and notations are presented; then, the topological structure of the WFM is analyzed in order to find a set of features and recurrent structures that can be exploited to define the translation rules. Such a topological view of the WFM allows to define the translation rules adopted by the algorithm, whose features are detailed in the last part of the section.

4.1.5.1 Propaedeutic definitions and notations

The necessary definitions and features of the complete and rich notation introduced in [13] are recalled hereafter; the reader is invited to refer to [13] for details and some extended examples.

The four fundamental system structures collected in Table 4.6 express the topology of the *structural relation* between entities that are siblings in the Holonic Management Tree (HMT), which is topologically equivalent to a tree. The number n of cells in the structures is defined as the *degree* of the structure. In the case of *assembly* and *expansion* structures the cells marked as a and e are called *head* cells.

Table 4.6: n^{th} degree fundamental structures

Series	Parallel	Assembly	Expansion
			

In order to conveniently manage the expressions of the HMT, it is useful to have a tool notation that lets feasible the handling of possibly large trees of systems of systems in shorthand. A system in the HMT is denoted by $S_{X,tag}^{(l)}(n)$, where l is the level of the HMT (and hence the position of the system in the tree), n is the degree of the structure associated to the system, and X is a place-holder for one of the following symbols S , P , A , and E , to denote respectively the four kind of structures, namely *series*, *parallel*, *assembly*, and *expansion*. The subscript field *tag* can be optionally added to denote and conventionally identify the system. If $n = 1$ the corresponding system is said *ground* system. It corresponds to a leaf of

the HMT. If $n > 1$ the system is said *composite*, and it is itself a sub-tree. The degree of a HMT is equal to the maximum degree of its systems.

To represent the trees of systems, with adding denotation of the *structural relation*, the Newick format is used [74]. A slight extension of the original Newick notation is adopted to better denote the intra-level relation in the case of *assembly* or *expansion* structures. With such a format, the HMT can be represented by a sequence of printable characters instead of graphs. The best way to explain this notation is by a few examples. The first example is the tree of an *expansion* structure in which a machine A is feeding production material to three production lines B, C, and D with a proportion of 15, 30 and 55 percent, respectively. A is the head cell of this *expansion*. B, C, and D are the other three cells of this 3rd degree structure that can be associated to a tree in Newick notation as: $A(B : 0.15, C : 0.3, D : 0.55)$. If the structure were an *assembly*, having A as head cell, the notation would have been the following: $(B : 0.15, C : 0.3, D : 0.55)A$. If the system were a *series*, or a *parallel*, the notation would have been simpler, as no information is needed for the edges and the *head*, and it would have resulted in (B, C, D) . Note that the parentheses in this notation enclose the sub-tree of a tree node. Figure 4.8 shows the basic structure included in a generic work flow, as will be detailed in Section 4.1.5.2.

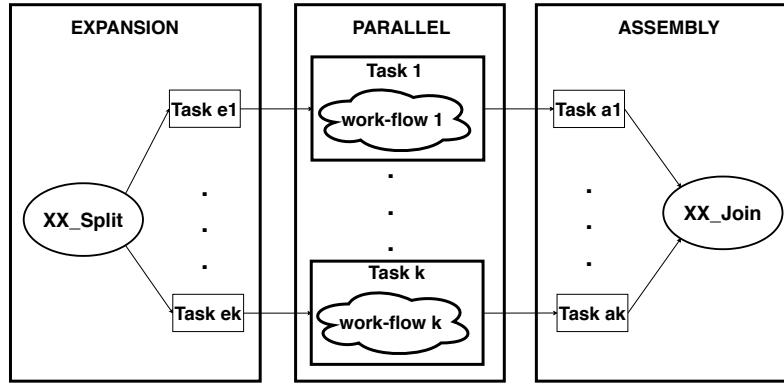


Figure 4.8: Basic structure included in the work flows

Supposing for sake of simplicity that all the task nodes are basic tasks, in such a structure, the WFM can be instantiated with a HMT having the following expression:

$$\begin{aligned}
 S_S^{(l)}(3) &\triangleq \left(S_E^{(l+1)}(k), S_P^{(l+1)}(k), S_A^{(l+1)}(k) \right) \\
 &\triangleq \left(S_{XX_Split}^{(l+2)}(1) \left(S_{Task_{e_1}}^{(l+2)}(1), \dots, S_{Task_{e_k}}^{(l+2)}(1) \right), \right. \\
 &\quad \left(S_{Task_1}^{(l+2)}(1), \dots, S_{Task_k}^{(l+2)}(1) \right), \\
 &\quad \left. \left(S_{Task_{a_1}}^{(l+2)}(1), \dots, S_{Task_{a_k}}^{(l+2)}(1) \right) S_{XX_Join}^{(l+2)}(1) \right)
 \end{aligned} \tag{4.9}$$

In (4.9) the HMT has been arbitrarily chosen of k^{th} degree and supposed starting at level l^{th} of a bigger tree. Note that three levels of detail of the tree are expressed in (4.9). In Figure 4.9, the same tree of (4.9) is graphically represented.

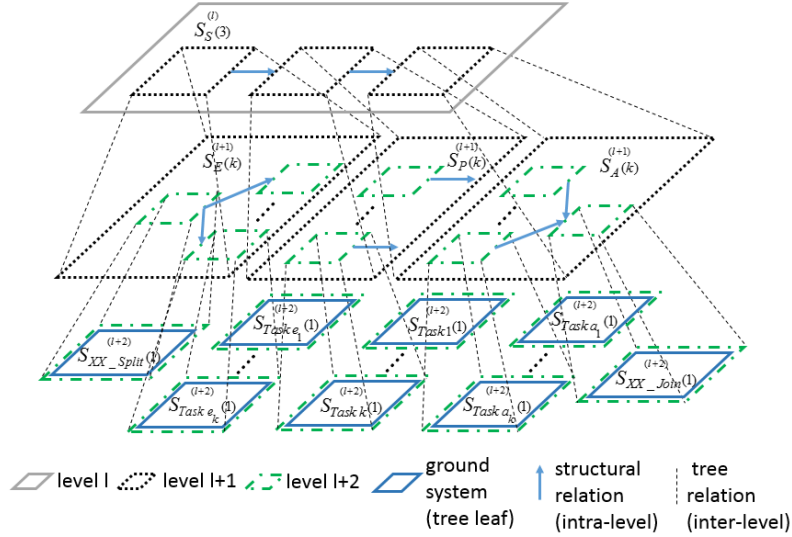


Figure 4.9: HMT for the *basic structure*

4.1.5.2 Conversion algorithm

Analyzing the topological properties of a generic WFM, five main features can be highlighted in order to define the translation rules:

1. The WFM is composed by a sequence of “sub-WFMs”, called Sub-Tasks, properly connected as in Figure 4.10.
2. Each Sub-Task is characterized by a specific number of incoming and outgoing flows, and denoted as $m : n$, where m is the number of incoming flows, and n is the number of outgoing flows. Three main types of Sub-Task can be then distinguished in the general structure of a WFM reported in Figure 4.10:

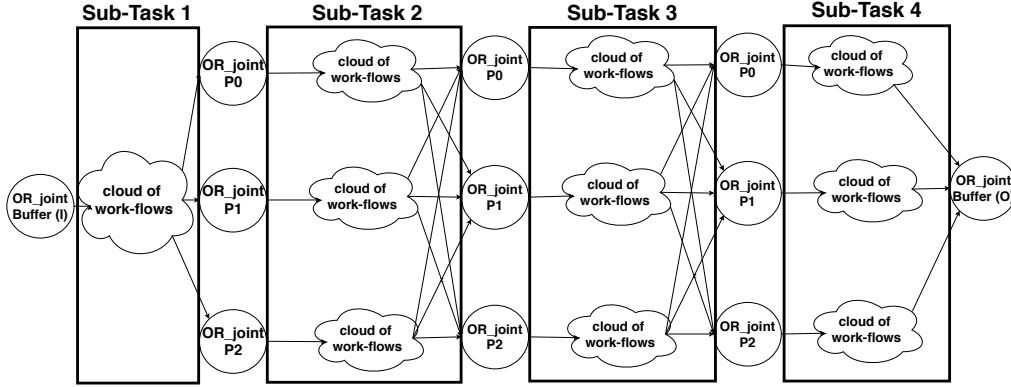


Figure 4.10: General structure of the WFM

- $1 : N$, as the first Sub-Task
- $N : N$, as the Sub-Tasks in the middle of the series
- $N : 1$, as the last Sub-Task.

The special case $1 : 1$ occurs when just one Sub-Task is present in the WFM.

3. A Sub-Task includes one or more work flows (denoted as *cloud of work-flows* in Figure 4.10), each one composed by a sub-net of WFM blocks. Such a sub-net is in turn defined by a set of *Basic Structures* properly connected by Split-type blocks (see Figure 4.11). Split-type blocks branch the work flows, so introducing a new way to perform the Sub-Task, as highlighted in Figure 4.11, in which the i -th cloud of work-flows starts with one incoming work flow and ends with three outgoing flows.
4. The *Basic Structure* is defined by: one Split-type block, from which several work flows (denoted by the Task blocks in Figure 4.8) can start, and one Join-type block, where the work flows join again.
5. A set of OR_Join blocks included in the WFM are used as connection points between the i -th Sub-Task and the $(i + 1)$ -th one. All the outgoing flows of the i -th Sub-Task converge into such OR_Join blocks, and the incoming flows of the $(i + 1)$ -th Sub-Task start from the same OR_Join blocks. They will be then denoted as *initial nodes* and *final nodes*, respectively.

The automatic conversion process, together with its translation rules and functions, is developed on the basis of such features. First of all, thanks to features 1) and 2), a fixed translation rule can be adopted to map the main structure of the WFM into the corresponding subsystems' tree.

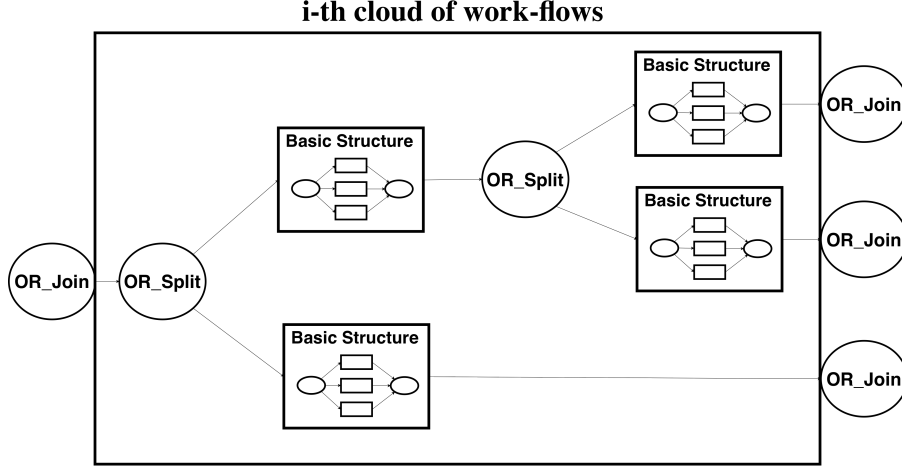


Figure 4.11: General structure of the work flow

Let $wfm(n, m_1, \dots, m_n)$ be the WFM composed by n Sub-Tasks, each one composed by m_i work flows (where m_i corresponds to the number of *initial nodes* of the i -th Sub-Task). The sequence of Sub-Tasks can be translated into a *series* of n subsystems, corresponding to: (i) 1 *expansion* for the $1 : N$ Sub-Task, (ii) 1 *assembly* for the $N : 1$ Sub-Task, and (iii) $n - 2$ *parallels*, covering the $N : N$ Sub-Tasks in the middle of the sequence (e.g., two *parallel* subsystems would be generated for the WFM in Figure 4.10). Each *parallel* also includes m_i *series*, one for each work flow composing the i -th Sub-Task. The definition of the *expansion* and the *assembly* subsystems at the zero level of the subsystems' tree can be however redundant, since they also arise when the translation rules are applied to the internal work flows of the Sub-Tasks (as it will be discussed in the remaining of the Section). The adopted translation rule replaces then the *expansion* and the *assembly* subsystems with two *series*.

Translation rule 1. *Given the WFM $wfm(n, m_1, \dots, m_n)$, the level zero of the subsystems' tree is obtained as:*

$$S_S^{(0,0)}(n) \triangleq \left(S_S^{(1,0)}(3), S_P^{(1,0)}(m_2), \dots, S_P^{(1,0)}(m_{n-1}), S_S^{(1,0)}(3) \right) \quad (4.10)$$

where the parallel subsystem of the i -th Sub-Task is given by:

$$S_P^{(1,0)}(m_i) \triangleq \left(S_S^{(2,0)}(3), \dots, S_S^{(2,0)}(3) \right)$$

Feature 4) highlights that in a generic WFM there exists a fixed structure properly repeated in the model (see Figure 4.8); such a characteristic has been exploited

to design a translation process based on the usage of a fixed translation rule, properly applied whenever such a recurrent structure is found in the WFM. Let $bs(k)$ be a basic structure composed by one Split-type block, one Join-type block, and k complex Task blocks (each one recursively defining a generic work flow); the Split-type block can be mapped into an *expansion*, the Join-type block into an *assembly*, and the Tasks blocks into k *series*, which are all included into a *parallel* subsystem (see Figure 4.8).

Translation rule 2. *Given the Basic Structure $bs(k)$ the corresponding subsystems' tree is obtained as:*

$$S_S^{(l,0)}(3) \triangleq \left(S_E^{(l+1,0)}(k), S_P^{(l+1,0)}(k), S_A^{(l+1,0)}(k) \right) \quad (4.11)$$

where

$$\begin{cases} S_E^{(l+1,0)}(k) & \triangleq \left(S_{e_0}^{(l+2,0)}(1) \left(S_{e_1}^{(l+2,0)}(1), \dots, S_{e_k}^{(l+2,0)}(1), S_{e_k}^{(l+2,0)}(1) \right) \right) \\ S_A^{(l+1,0)}(k) & \triangleq \left(\left(S_{a_1}^{(l+2,0)}(1) \dots, S_{a_k}^{(l+2,0)}(1), S_{a_k}^{(l+2,0)}(1) \right) S_{a_0}^{(l+2,0)}(1) \right) \end{cases}$$

in which the optional tag has not been used, but the following mapping between each WFM block and the corresponding subsystem is adopted:

WFM block	Subsystem
XX_Split	$S_{e_0}^{(l+2,0)}(1)$
XX_Join	$S_{a_0}^{(l+2,0)}(1)$
Task _{ei}	$S_{e_i}^{(l+2,0)}(1)$
Task _{ai}	$S_{a_i}^{(l+2,0)}(1)$

whereas

$$S_P^{(l+1,0)}(k) \triangleq \left(S_S^{(l+2,0)}(\cdot) \dots, S_S^{(l+2,0)}(\cdot) \right)$$

in which the degrees of the series structures are not known (symbol (\cdot) is adopted to represent this situation), since they refer to a complex Task block not explored yet. As in the case of the Translation rule 1, the unknown structures will be defined in the subsequent phases.

Application example: scrolling the WFM in Figure 4.8, the Translation rule 2 is applied when the XX_Split block is found. In this phase, the structure of the

subsequent complex Tasks (i.e., Task_i) is not known yet, so that also in this case the *parallel* structure is defined as:

$$S_P^{(l+1,0)}(k) \triangleq \left(S_S^{(l+2,0)}(\cdot) \dots, S_S^{(l+2,0)}(\cdot) \right) \quad (4.12)$$

A third type of translation is defined and applied when a basic Task block (i.e., a Task block representing a basic action) is found. In such a case the following rule is adopted:

Translation rule 3. *Given a basic Task, the corresponding subsystems' tree is obtained as:*

$$S_X^{(l,0)}(1)$$

where X , i.e., the structure type, is inherited from the the parent subsystem.

Application example: if a basic Task is found while scrolling Task_1 , the *parallel* structure (4.12), is properly updated as:

$$S_P^{(l+1,0)}(k) \triangleq \left(S_S^{(l+2,0)}(1), \dots, S_S^{(l+2,0)}(\cdot) \right)$$

where

$$S_S^{(l+2,0)}(1) \triangleq S_{s_1}^{(l+3,0)}(1)$$

Feature 3) is well sketched in Figure 4.11, in which both $bs(k)$ structures and branching of work flows are present. In order to generalize the translation process, whenever a Split-type block is found in the WFM, each outgoing flows is analyzed, and the *Translation rule 2* is applied when a basic structure is found. The *assembly* subsystem in (4.11) will be actually exploited when the Split-type block belongs to a $bs(k)$ structure, whereas it will be a pending node when a corresponding Split-type block does not exist.

The overall automatic conversion process is then composed of three main phases:

1. **Initialization:** in the main algorithm (whose pseudo-code is reported in Algorithm 2), the *Translation rule 1* is applied in order to define the structure of the subsystems' tree. Subsystems already defined at this level (i.e., $l = 0$) do not represent the leafs of the tree; they will be properly defined in the subsequent phase.
2. **Translating:** in Algorithm 3 each Sub-Task is scrolled starting from its *initial nodes*. The *Translation rule 2* is applied whenever a Split-type block is found, whereas *Translation rule 3* is adopted when a Task block is found (see the pseudo-code in Algorithms 4–6). Translation rules are not applied to the *Final nodes*, which are just used to understand when the work flow of a given Sub-Task ends.

3. **Cleaning:** Pending nodes are finally removed in the final phase, in which the specific cleaning Algorithm 7 is applied.

The important goal of the translation algorithm is to associate each block of the WFM with the correct location in the subsystems' tree. This is achieved by using a memory stack approach, where the current location in the subsystems' tree is kept by a proper sequence of push and pop operations in the stack. Also the algorithm manages some special cases in which translation rules could not be applicable as they are, e.g., when in a *Basic Block* one of the flows outgoing the starting XX_Split node has less than three Task blocks (as in Figure 4.8, if the Task *a1* were not present); in such a case it should be not possible to correctly create the expansion subsystem. The algorithm manages such a situation by adding a dummy node that is called NOP.

The notations adopted in the Algorithms 2 - 7 are reported in Table 4.7, while the main functions included in them are summarized in Table 4.8.

Table 4.7: Description of the notations used in the translation algorithm

Notation	Description
wfm	denotes the nodes belonging to a WFM (e.g., wfm_h)
$next$	represents the child block of a WFM block, having only one output (e.g., $wfm \rightarrow next$)
$next[i]$	represents the child blocks of a WFM block, having several outputs (e.g., $wfm_h \rightarrow next[i]$); i corresponds to the i -th child
hmt	denotes the nodes belonging to a HMT (e.g., hmt_{bs})
$child_a$	represents the assembly child subsystem of the current HMT (e.g., $hmt_{bs} \rightarrow child_a$)
$child_e$	represents the expansion child subsystem of the current HMT (e.g., $hmt_{bs} \rightarrow child_e$)
$child_p[i]$	selects one child of a parallel subsystem (e.g., $hmt_{bs} \rightarrow child_p[i]$)

Table 4.8: Description of the functions adopted in the translation algorithm

Function	Description
InitStack() : Stack Handle	Provides a new stack
GetStack() : HMT Handle	Provides the first element of the stack
Push()	Inserts a new element (handle to a node of the HMT) on the top of the stack
Pop()	Removes the first element from the stack
InitHmt() : HMT Handle	Provides a void HMT
ConnectHmt (Dst, Src)	Connects the source HMT to the destination one
Rule2Hmt() : HMT Handle	Provides an empty HMT
SubSystemHmt (Type, N) : HMT Handle	Provides the handle to a new subsystem of the type defined in the first parameter (e.g., <i>serial</i> , <i>parallel</i> , <i>assembly</i> , <i>expansion</i> , <i>cell</i> , <i>NOP</i>); the second parameter defines the system degree. When the subsystem is of type <i>cell</i> or <i>NOP</i> , the second parameter is always 1, since they are leafs of the HMT
GetNextHmt (HMT Handle) : HMT Handle	Provides the next subsystem of the current HMT node
RemoveNodeHmt (HMT Handle)	Removes the HMT node provided as input, preserving the integrity of the overall HMT
GetSubTask (WFM Handle, i) : Sub-Task Handle	Provides the handle to the <i>i</i> -th Sub-Task of the given WFM
GetDimention (ST Handle) : Integer	Provides the dimension (i.e., the number of clouds of work flows) of the given SubTask
ColorNode (WFM Handle)	Colors the WFM node provided as input

Algorithm 2 Conversion Algorithm

```

1: function TRANSLATE(wfm, n)
2:   stack = INITSTACK;
3:   hmth = INITHMT(void);                                ▷ hmth = head
4:   hmtc = SUBSYSTEMHMT('series', 1);                    ▷ hmtc = current
5:   CONNECTHMT(hmth, hmtc);
6:   if n > 2 then
7:     hmtp = SUBSYSTEMHMT('parallel', n - 2);
8:     CONNECTHMT(hmtc, hmtp);
9:   end if
10:  for i := 1 to n do
11:    hmtc = hmth
12:    st = GETSUBTASK(wfm, i);
13:    mi = GETDIMENTION(st);
14:    if (1 < i < n) then
15:      hmtn = SUBSYSTEMHMT('series', mi);
16:      CONNECTHMT(hmtp → child[i], hmtn);
17:      hmtc = hmtp
18:    end if
19:    for j := 1 to mi do
20:      if hmtc == hmth then
21:        head = hmtc;
22:      else
23:        head = hmtc → child[i];
24:      end if
25:      PUSH(stack, head);
26:      TRANSLATEHMT(st → initNode[i], stack);
27:    end for
28:  end for
29:  CLEANING(hmth)
30: end function

```

Algorithm 3 Translating

```
1: function TRANSLATEHMT( $wfm_h$ ,  $stack$ )
2:   if ( $wfm_h$  is a final node) then
3:     return ;
4:   end if
5:   if ( $wfm_h$  is a Split-type block) then
6:     MANAGESPLIT( $wfm_h$ ,  $stack$ )
7:   end if
8:   if ( $wfm_h$  is a Join-type block and is not colored) then
9:     MANAGEJOIN( $wfm_h$ ,  $stack$ )
10:  end if
11:  if ( $wfm_h$  is a Task-type block) then
12:    MANAGETASK( $wfm_h$ ,  $stack$ )
13:  end if
14:  return ;
15: end function
```

Algorithm 4 ManageSplit

```
1: function MANAGESPLIT( $wfm_h$ ,  $stack$ )
2:    $hmt_h = \text{GETSTACK}(stack)$ ;
3:    $hmt_{bs} = \text{RULE2HMT}$ ;
4:    $\text{CONNECTHMT}(hmt_h, hmt_{bs})$ ;
5:    $hmt_c = \text{SUBSYSTEMHMT}('cell', 1)$ ;
6:    $\text{CONNECTHMT}(hmt_{bs} \rightarrow child_e, hmt_c)$ ;
7:   if ( $wfm_h$  is colored) then
8:      $\text{POP}(stack)$ 
9:   end if
10:   $\text{COLORNODE}(wfm_h)$ 
11:  for  $i := 1$  to number of children of  $wfm_h$  do
12:     $\text{COLORNODE}(wfm_h \rightarrow next[i])$ 
13:     $\text{PUSH}(stack, hmt_{bs} \rightarrow child_a)$ 
14:     $\text{PUSH}(stack, hmt_{bs} \rightarrow child_p[i])$ 
15:     $\text{PUSH}(stack, hmt_{bs} \rightarrow child_e)$ 
16:     $\text{TRANSLATEHMT}(wfm_h \rightarrow next[i], stack)$ 
17:     $\text{POP}(stack)$ 
18:     $\text{POP}(stack)$ 
19:     $\text{POP}(stack)$ 
20:  end for
21:  if ( $wfm_h$  is colored) then
22:     $\text{PUSH}(stack, hmt_h)$ 
23:  end if
24: end function
```

Algorithm 5 ManageJoin

```
1: function MANAGEJOIN( $wfm_h$ ,  $stack$ )
2:    $hmt_h = \text{GETSTACK}(stack)$ ;
3:    $hmt_c = \text{SUBSYSTEMHMT}('cell', 1)$ ;
4:    $\text{CONNECTHMT}(hmt_h, hmt_c)$ ;
5:    $\text{POP}(stack)$ 
6:    $\text{COLORNODE}(wfm_h)$ 
7:    $\text{TRANSLATEHMT}(wfm_h \rightarrow next, stack)$ 
8:    $\text{PUSH}(stack, hmt_h)$ 
9: end function
```

Algorithm 6 ManageTask

```

1: function MANagetASK( $wfm_h$ ,  $stack$ )
2:   if ( $wfm_h$  is colored) then
3:      $hmt_{h1} = \text{GETSTACK}(stack)$ ;
4:      $hmt_c = \text{SUBSYSTEMHMT}('cell', 1)$ ;
5:      $\text{CONNECTHMT}(hmt_{h1}, hmt_c)$ ;
6:      $\text{POP}(stack)$ 
7:     if ( $wfm_h \rightarrow next$  is a Task-type block) then
8:        $hmt_{h2} = \text{GETSTACK}(stack)$ ;
9:        $\text{POP}(stack)$ 
10:      if ( $wfm_h \rightarrow next$  is not a final node) then
11:         $hmt_{h3} = \text{GETSTACK}(stack)$ ;
12:         $hmt_c = \text{SUBSYSTEMHMT}('NOP', 1)$ ;
13:         $\text{CONNECTHMT}(hmt_{h3}, hmt_c)$ ;
14:      end if
15:    end if
16:     $\text{COLORNODE}(wfm_h \rightarrow next)$ 
17:     $\text{TRANSLATEHMT}(wfm_h \rightarrow next, stack)$ 
18:    if ( $wfm_h \rightarrow next$  is not a Task-type block) then
19:       $\text{PUSH}(stack, hmt_{h2})$ 
20:    end if
21:     $\text{PUSH}(stack, hmt_{h1})$ 
22:  else
23:     $hmt_{h1} = \text{GETSTACK}(stack)$ ;
24:    if ( $wfm_h \rightarrow next$  is a Join-type)
25:    && ( $wfm_h \rightarrow next$  is not a final node) then
26:       $\text{POP}(stack)$ 
27:    end if
28:     $hmt_{h2} = \text{GETSTACK}(stack)$ ;
29:     $hmt_c = \text{SUBSYSTEMHMT}('cell', 1)$ ;
30:     $\text{CONNECTHMT}(hmt_{h2}, hmt_c)$ ;
31:    if ( $wfm_h \rightarrow next$  is a final node) then
32:       $\text{POP}(stack)$ 
33:    end if
34:     $\text{COLORNODE}(wfm_h \rightarrow next)$ 
35:     $\text{TRANSLATEHMT}(wfm_h \rightarrow next, stack)$ 
36:    if ( $wfm_h \rightarrow next$  is Join-type but not final node)
37:    || ( $wfm_h \rightarrow next$  is a final node) then
38:       $\text{PUSH}(stack, hmt_{h1})$ 
39:    end if
40:  end if
41: end function

```

Algorithm 7 Cleaning

```
1: function CLEANING( $hmt_h$ )
2:   if  $hmt_h == \text{NULL}$  then
3:     return 0;
4:   end if
5:    $n\_childs = 0$ ;
6:   while  $child_i = \text{GETNEXTHMT}(hmt_h)$  do
7:      $n\_childs ++$ ;
8:      $N = \text{CLEANING}(child_i)$ 
9:     if  $N == 1$  &&  $hmt_h$  is not a leaf then
10:       $child_{child_i} = \text{GETNEXTHMT}(child_i)$ 
11:       $\text{CONNECTHMT}(hmt_h, child_{child_i})$ 
12:       $\text{REMOVENODE}(child_i)$ 
13:    end if
14:  end while
15:  return  $n\_childs$  ;
16: end function
```

Part II

Advanced Robotic Service Algorithms

Chapter 5

Service algorithms for industrial robots

Industrial robots are usually provided with the minimum equipment necessary to perform the typical industrial applications, like arc/spot welding, painting, manipulation, pick and place etc., for which a good position control (both in the joint and the Cartesian space) is only required. For this reason the only available real sensors are usually encoders and current sensors. Implementing service algorithms in the robot controller is possible despite the limited number of information (sensors readings) available; studying the possibility of using such a few information to design new service algorithms, actually implementable in an industrial controller, is then very important to add new features to the robot, or even to convert standard industrial manipulators into robot suitable for the usage in other contexts, like smart factories or collaborative applications. In order to achieve such a purpose, service algorithms can adopt physical models of the robotic manipulators to compute the required physical quantities, e.g., the robot dynamic model can be adopted to compute the theoretical torques applied on each robot joint. The performance of such algorithms are logically conditioned by the quality of the adopted physical models, so that the usage of accurate ones becomes of considerable importance. Several works can be found in the state of the art concerning the development of sensor-less algorithms, where the term sensor-less in such a context is used to highlight that the procedure uses only information provided by the proprioceptive sensors of the robot (i.e., without the insertion of "external" sensors, like force/torque ones or cameras).

Collision detection and post-collision management are different issues, which can be properly combined to achieve interesting robotic applications, e.g., manual guidance, which is a good way to define user-friendly robot programming approaches (like programming by demonstration [55]), and collision reaction strategies that can avoid/reduce possible mechanical damage to the robot.

Collision Detection (CD) algorithms are usually based on the idea of applying

a threshold to a signal (the collision signal) that varies according to the external forces applied on the robot. The threshold can be constant or time varying, but in any case the main issue is to obtain a proper collision signal. Specific sensors (that are usually not included in standard industrial robots), like force sensors, can be used to obtain a measure of the force applied on a specific point of the robot structure, or torque sensors to measure the torques applied on the robot joints. Accurate physical models of the robot are usually exploited to clean the signal from the force/torque components due to the dynamics of the robot, so obtaining a collision signal actually corresponding to the applied external forces/torques. The sensors already included in the robot, like the motor encoders and the current sensors, can be alternatively used to obtain an estimate of the torques applied on the robot joints, or in general a signal that varies according to the external forces applied on the robot. The Collision Detection and Manual Guidance approaches based on this kind of solution have the advantage that no extra sensors are required for their implementation. Various techniques can be found in literature. Initially CD methods were designed to detect collisions in production systems, where robots could accidentally hit other objects/robots, due to programming errors or to the presence of unforeseen objects. In [91] a collision detection scheme based on a disturbance observer system has been presented, in which collisions are detected by setting up a set of thresholds for every joint. In [44] the authors presented a collision detection scheme with adaptive characteristics, based on a Finite State Machine (FSM) (whose states are a priori labeled as safe or unsafe), processing both real and dynamically modeled joint currents. Once an unsafe state is detected, a time varying threshold is applied to detect a collision. Statistical time series methods have also been developed to achieve detection and identification of faults in an aircraft skeleton structure [85]. The main advantage of those methods lies in their ability to use data to build mathematical models that represent the true dynamical system. Even if applied to a different context, such methodologies have been exploited later to develop robot collision detection procedures based on fuzzy identification [28].

Some studies [90] introduced the notion of human pain tolerance to set an acceptable pain level for a human; such level can be used as a threshold for CD algorithms to reduce the impact force, so allowing their adoption in a Human-Robot Collaboration (HRC) context. Different methodologies have been then developed with the purpose to be suitable for HRC applications. In [90] the difference between the dynamic model torques and the actual motor torques is used to obtain a reliable detection scheme. A more general approach was presented in [58], where the robot generalized momentum is exploited to define two functions: $\sigma(t)$ and $r(t)$, being $\sigma(t)$ the collision detection signal, whose value raises when a collision occurs and rapidly returns to zero when the contact is lost. Information about the force direction or the link on which the collision took place is provided by $r(t)$, called *collision identification signal*. A closed control architecture was proposed in [34],

using only motor currents and joint positions in order to define suitable thresholds for the detection scheme, while in [48] authors tried to refine such approach, by preventing or greatly reducing the probability of false alarms using an appropriate band pass filter with a changing frequency window, so to facilitate the distinction between collisions and false alarms. Computationally efficient methods based on fuzzy identification and time series modeling [28] can also be found, whose adoption does not require the explicit knowledge of the robot dynamic model. A training phase is however necessary, but offline training procedures are available.

Several works can be found on robot Manual Guidance (MG) methodologies, as well. In [64] the authors proposed an approach based on the adoption of force/torque sensors to implement a control scheme that imposes a specific velocity profile according to the sensor measures. A lot of classical control schemes can also be found, like force control [87], impedance control [41] and admittance control [9], [30], as well as more advanced methodologies like adaptive admittance control schemes [92] and variable impedance control schemes [40]. A further interesting approach based on the adoption of vision systems is presented in [62]; in this case the robot motion is obtained using the images provided by a camera. Sensor-less methodologies (i.e., that do not use force/torque sensors) can also be mentioned, like the ones based on the adoption of an observer of external forces [58] to achieve manual guidance.

Other interesting service algorithm are related to the identification of the payload parameters [52][20]. Such a functionality is very important to let the robot control work properly, thanks to correct and accurate information provided by the manipulator dynamic model. The Payload Check functionality, through a payload identification, may alert the user if the declared payload is different from the real one.

In Chapters 6, 7 and 8 three different sensor-less service algorithms are proposed: 1) a Collision Detection procedure, 2) a post-collision reaction/manual guidance algorithm, and 3) a Payload Check functionality. All the proposed service algorithms exploit the values provided by the robot dynamic model, and then benefit from the adoption of an accurate model.

The robot dynamic model can be expressed in the following form:

$$M(q)\ddot{q} + C(q, \dot{q})\dot{q} + \tau_f(\dot{q}) + g(q) = \tau \quad (5.1)$$

where $M(q)$ is the inertia matrix, $C(q, \dot{q})$ includes the centrifugal and Coriolis effects, $g(q)$ is the gravity torque vector, $\tau_f(\dot{q})$ is the joint friction torque vector, and τ is the vector of the applied joint torques. Supposing that the model structure is correct, the main source of errors is due to the estimated values of the model parameters; while some parameters can be estimated very well, since they can be directly measured (e.g., the mass of the links) or can be computed using FEM methods, some others, like the friction parameters, cannot be directly measured. Friction modeling is a very well-known problem in robotics, especially at low velocity [12],

and different kinds of friction models, of different complexities, can be found, each of which providing different performances in terms of friction torque estimation. In the industrial context, friction modeling could be critical, since custom solutions for the specific robot are generally avoided; the possibility of developing a general scheme for friction modeling and identification can be then very important in such a scenario.

Several researches can be found in literature proposing different kinds of friction models, both dynamic and static (see e.g., [7], [60]; the influence of particular elements like load and temperature have been also analyzed [50], [88], but the friction identification process is generally handled with reference to a specific robot and/or in a laboratory context. Some adaptive solutions have been proposed [63] or based on friction observers [82], but unfortunately their experimental application is often limited to simple systems and/or in a laboratory context. In the industrial scenario it would be very useful to have a general friction estimation tool, able to perform the off-line estimation of the friction values using a standard set of data acquisitions, so that the operator can easily update the friction model when necessary.

An important part of the work about service algorithms was then devoted to improve the friction model, and to provide a set of generic procedures to carry out both data acquisition and parameter identification phases. Chapter 9 provides the main results of the work carried out about friction modeling.

Chapter 6

Collision Detection

The goal was the development of a virtual collision sensor, easily applicable to various types of manipulators (i.e., both low-payload and high-payload robots, and/or with different kinematic structures), neither requiring specific customizations, nor inserting further, *ad hoc* sensors beyond the standard ones generally equipping any industrial robot. The virtual collision sensor must contribute to guarantee the mechanical integrity of the robot and the cell, and the correct execution of the working process, avoiding false collision alarms that would stop the production cycle. Moreover, it should correctly work without the necessity of a too long warm up phase, before reaching a good level of reliability in detecting an actual collision. In such a case, in fact, any stop of the normal activity of the robot could lead to a loss of accuracy and the need to warm up again.

DC-motors are considered as actuators, and the only available real sensors are supposed to be:

- the encoders, mounted on the motor shafts
- the current sensors, providing the currents absorbed by the motors.

The information provided by the encoders will not be actually employed by the virtual sensor: it would be useful, in fact, only to estimate the joint torques on the basis of the robot dynamic model, which is a solution that has been discarded just to avoid any dependence of the collision detection procedure on the characteristics of the specific robot.

The only actual assumption, which the proposed collision detection approach will rely on, is the availability of the estimates of the motor currents, used inside the original robot controller. Such estimates are computed on the basis of an internal dynamic model of the manipulator, whose structure and parameters are not known. This assumption is quite realistic, independently of the particular control scheme adopted. Let $I(t) = [I_i(t)]$, $i = 1, \dots, n$, be the vector of the measured currents of the robot motors, where n is the number of joints, and let $I_{DM}(t) = [I_{DM,i}(t)]$,

$i = 1, \dots, n$, be the vector of the currents estimated by the robot dynamic model. Such vectors are assumed to be available to the virtual collision sensor; on the basis of this information only, the virtual sensor will have to continuously update the n -dimension *Collision* vector (as in the scheme reported in Figure 6.1), collecting a logical variable for each joint; when at least one of such variables becomes *TRUE*, a collision is detected, and a proper stopping procedure will be immediately applied to the robot.

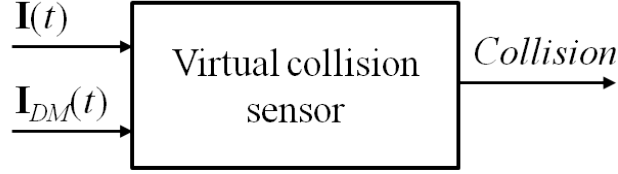


Figure 6.1: Virtual collision sensor scheme.

6.1 The proposed approach for collision detection

The virtual collision sensor action is based on the computation of the current residue vector $R(t) = [R_i(t)]$, $i = 1, \dots, n$, given by:

$$R(t) = I(t) - I_{DM}(t) \quad (6.1)$$

where the estimated current vector $I_{DM}(t)$ is assumed to have been computed by the internal robot dynamic model in absence of any collision, i.e., the complete expression of $I_{DM}(t)$, if available, would be of the following type:

$$I_{DM}(t) = K_I^{-1}(\hat{M}(q_d)\ddot{q}_d + \hat{n}(q_d, \dot{q}_d)) \quad (6.2)$$

where $q_d(t)$, $\dot{q}_d(t)$, and $\ddot{q}_d(t)$ are the reference joint position, velocity and acceleration vectors, respectively, $\hat{M}(\cdot)$ is an estimate of the robot inertia matrix, $\hat{n}(\cdot)$ includes the estimates of the torques due to centrifugal and Coriolis effects, friction and gravity, and K_I is the diagonal matrix of the conversion coefficients $K_{I,i}$ of the motors (from current to torque).

In the ideal case, i.e., if the internal dynamic model were able to *exactly* replicate the behavior of the robot, the current residue $R(t)$ would be zero in absence of any collision. In a real case, some model error is always present, so that the current residue is expected to be small, but never identically zero. When a collision occurs, the current residue immediately grows, because in this case the measured motor currents include also the effects of the torques applied to the joints due to the collision forces.

The working mode of the collision detection procedure is based on the comparison of the current residue $R(t)$ with a proper *smart* threshold, positive-value, vector function $S(t)$ (including a varying threshold $S_i(t)$ for each joint), according to the following collision detection conditions:

$$\begin{cases} \text{If } |R_i(t)| > S_i(t) & \text{then } Collision_i = TRUE \\ \text{If } |R_i(t)| \leq S_i(t) & \text{then } Collision_i = FALSE \end{cases} \quad (6.3)$$

where the threshold varying function is defined as:

$$S(t) = \hat{m}_{err}(t) + Coll_{bias}(t) \quad (6.4)$$

The first term, $\hat{m}_{err}(t)$, represents an estimate of the absolute value of the model error in absence of collisions., which is determined using the current residue $R(t)$ computed inside the virtual collision sensor as in (6.1), while the second term in (6.4), $Coll_{bias}(t)$, represents the sensitivity of the virtual sensor; its entries are positive and are given by the current values corresponding to the minimum collision torque that the virtual sensor should be able to detect on each joint.

The ability of the proposed virtual sensor of detecting collisions, avoiding false alarms, does not depend on the actual quality of the robot dynamic model (which is unknown), but on the capacity of the virtual sensor itself of computing a reliable estimate $\hat{m}_{err}(t)$ of the model error. Considering the adopted expression (6.4) for the threshold function $S(t)$, the absence of any collision at time t is correctly detected on the i -th joint by the second condition in (6.3) if

$$\Delta m_{err,i}(t) \leq Coll_{bias,i}(t) \quad (6.5)$$

where

$$\Delta m_{err,i}(t) = |R_i(t)| - \hat{m}_{err,i}(t) \quad (6.6)$$

Inequality (6.5) shows that small values can be adopted for $Coll_{bias,i}$, i.e., a fine sensitivity of the sensor can be achieved, if $\Delta m_{err,i}(t)$ is sufficiently small, otherwise $Coll_{bias,i}$ must be increased to avoid false collision alarms.

Different approaches can be followed to define $Coll_{bias}(t)$, e.g., in [42] its values were assumed to be constant for all the joints, leaving the user the possibility of changing it, if necessary, while in [44] an automatic learning and adaptation process has been applied to enhance the robustness of the procedure and the speed in detecting a collision, as well as to cope with possible *slow* variations of the robot behavior. Details about the automatic adaptation of the sensor sensitivity can be found in [44] and recalled in Section 6.2.

6.1.1 Model Error Estimation

By the direct analysis of the behavior of the motor currents during any movement of a robot, independently of the specific manipulator and the specific (unknown) internal dynamic model providing $I_{DM}(t)$, it experimentally results that:

- in the motion phases in which $I_{DM}(t)$ is almost constant (or it is varying very slowly), the residue $R(t)$ shows a similar behavior, i.e., it is almost constant or slowly varying;
- when $I_{DM}(t)$ rapidly varies, also the residue does, possibly reaching very high values for some joint.

The two situations indicate that when the current is almost constant, i.e., when the corresponding torque applied to the joint is almost constant, the dynamic behavior of the system is intrinsically easier to be reconstructed by the internal dynamic model, whatever it is. On the contrary, when the robot is in an acceleration or deceleration phase, i.e., when the currents are rapidly varying, the model error automatically tends to grow, because more complex dynamic effects are acting on the robot, and their estimation is typically and reasonably more difficult. Such observations lead to the opportunity of estimating the model error using different algorithms in the two situations, which will be denoted as *steady* state and *unsteady* state, respectively. Figure 6.2 shows a small portion of a real work cycle carried out by a NJ4 110 robot employed in an automotive production line, highlighting the different behavior of the current residue during the *steady* state (white background) and the *unsteady* one (blue background).

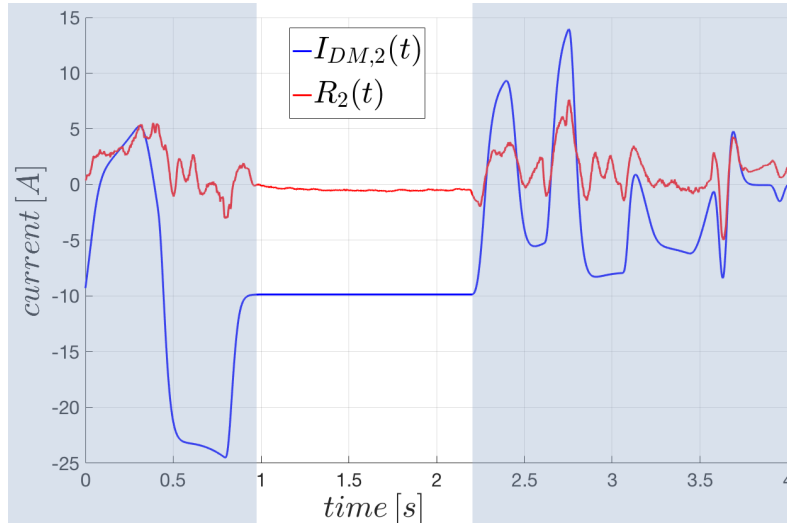


Figure 6.2: Behavior of the current residue of the second joint of a NJ4 110 in the *steady* (white background) and *unsteady* (blue background) states

Steady state During the steady state, the model error for the i -th joint can be simply estimated on the basis of an average process, considering the maximum values reached (in absolute value) by the residue during such a steady state time interval, denoted by T_{ss} , from the time instant t_{ss} , at which the FSM enters the steady state, up to the current time instant t . The process starting at time t_{ss} can be expressed as:

$$\overline{ERR}_{ss,i}(t) = \frac{1}{N+1} \left(|R_i(t_{ss})| + \sum_{\tau \in T_{ss}} \overline{err}_{ss,i}(\tau) \right) \quad (6.7)$$

where $|R_i(t_{ss})|$ is the last residue value obtained before the transition to the steady state, $\overline{err}_{ss,i}(\cdot)$ is the function containing the maximum values reached by $|R_i(t)|$ during the steady state until the current time instant t , whilst N is the number of samples of the function in the same interval. The usage of $|R_i(t_{ss})|$ has been introduced in order to allow a more rapid settling of the values provided by the average process, so to avoid too small values of $\overline{ERR}_{ss,i}(t)$ at the very beginning of the steady state.

The model error for the i -th joint is then computed by sampling $\overline{ERR}_{ss,i}(t)$ with a proper sampling time T_s , so obtaining a model error defined as:

$$\hat{m}_{err,i}(t) = \begin{cases} \overline{ERR}_{ss,i}(t) & t = k T_s \\ \overline{ERR}_{ss,i}((k-1)T_s) & t \in [(k-1)T_s, k T_s) \end{cases} \quad (6.8)$$

As shown in (6.8), $\hat{m}_{err,i}(t)$ is actually updated only at time instants $t = k T_s$. A proper choice of T_s is necessary to allow the correct detection of collision. In particular T_s cannot be too small, otherwise it could not be possible to properly distinguish the residue and the model error during the steady state, in which both $I_{DM,i}(t)$ and $I_i(t)$ change very slowly (and the residue as well) in absence of collisions. On the contrary T_s cannot be too high, otherwise the actual variation of the residue could not be correctly captured. From a data driven analysis of the collision timing, T_s has been set equal to 0.2 s.

Unsteady state During an unsteady state, two functions, computed during *every* unsteady state, are combined to estimate the model error for the i -th joint as

$$\hat{m}_{err,i}(t) = \delta \text{est}_{err,i}(t) + (1 - \delta) \overline{err}_{us,i}(t) \quad (6.9)$$

where $\overline{err}_{us,i}(t)$ is the function containing the maximum values reached by $|R_i(t)|$ during an *unsteady* state, $\text{est}_{err,i}(t)$ is computed on a predefined number of residue samples, saved in a buffer, and $\delta \in [0, 1]$ weighs the contributions of the two terms. In particular, $\text{est}_{err,i}(t)$ is given by:

$$est_{err,i}(t) = \overline{ERR}_{us,i}(t) + 3 \cdot \sqrt{\frac{1}{N} \sum_{\tau \in T_{us}} \left(\overline{ERR}_{us,i}(\tau) - \overline{err}_{us,i}(\tau) \right)^2} \quad (6.10)$$

where T_{us} is the time set in which the currents are in the unsteady state, and

$$\overline{ERR}_{us,i}(t) = \frac{1}{N} \sum_{\tau \in T_{us}} \overline{err}_{us,i}(\tau) \quad (6.11)$$

Figure 6.3 shows the behavior of the two model errors for the first joint of the NJ4 110. The phases in which $I_{DM}(t)$ is in *steady* state are highlighted by a cyan background. The black line, corresponding to the estimate of the model error in the steady state, is applied only in such cyan phases, whereas the model error in blue is applied in the rest of time; such a solution allows to considerably improve the sensitivity of the algorithm, which can adapt its behavior on the basis of the actual working conditions.

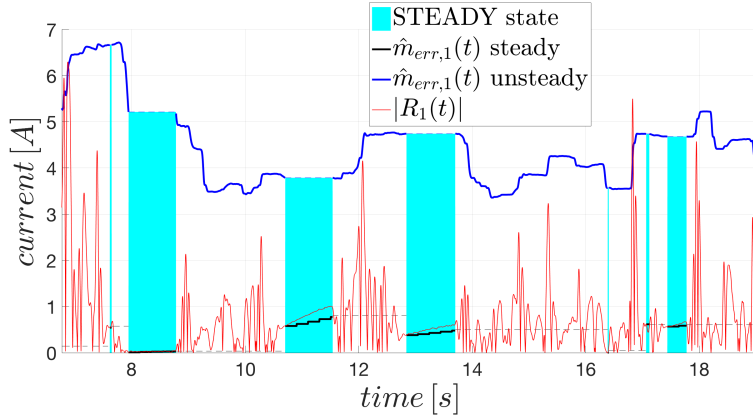


Figure 6.3: Comparison between $\hat{m}_{err,1}(t)$ computed for *steady* state and for *unsteady* ones.

The change of the parameter δ modifies the behavior of $\hat{m}_{err,i}(t)$ for the unsteady states. When its value is decreased, the model error is more influenced by the maximum value reached during the whole elaboration, so obtaining a procedure more reliable in terms of false collisions but less rapid to adapt its behavior to the new trend of the residue (i.e., the blue line in Figure 6.3 remains close to the maximum reached value). On the contrary, when δ is increased, the model error estimation becomes more reactive so improving the sensibility of the algorithm (i.e., the blue line in Figure 6.3 remains close to the residual values). In the experimental tests δ was set to 0.5.

The transition between the steady state estimation and the unsteady one is

performed without any particular management; despite this choice leads to discontinuities in the computation of the threshold (6.4), such an approach does not cause any functional anomaly in practice, so that it can be adopted to keep the computational burden low in the real time implementation of the virtual sensor.

6.1.2 Monitoring of the currents behavior through a FSM

The trend of the currents $I(t)$ and $I_{DM}(t)$, i.e., the inputs of the virtual sensor, are monitored to distinguish:

- steady and unsteady states, to apply the most suitable model error estimation algorithm;
- unsafe and safe states, i.e., situations in which collisions might actually occur or not, so to perform the collision detection test only in the unsafe ones, thus enhancing the efficiency of the virtual sensor implementation.

The monitoring action is performed implementing a five-states FSM for each joint, after having applied a proper filtering action to both $I(t)$ and $I_{DM}(t)$. Such a filtering action, which is mandatory for the measured current, is applied as-it-is to the estimated one, too, so to avoid any time delay between them. A low-pass filter with 10 – 20 Hz bandwidth can represent a satisfactory solution in general.

The five states of the FSM of the i -th joint, which is reported in Figure 6.4 with a sketch of the currents behavior in the various states ($I_i(t)$ in red and $I_{DM,i}(t)$ in blue) are:

- *Steady* state, in which the estimated current $I_{DM,i}(t)$ is almost constant or very slowly varying (as in the phases having white background in Figure 6.2); this state is recognized by computing the first and the second order time derivatives of $I_{DM,i}(t)$, which must tend both to zero. It is worth to be noted that $I_{DM,i}(t)$ is not affected by noise, since it is provided by the internal robot dynamic model on the basis of the reference joint trajectory, so that the time derivative computation can be made without any numerical problem.
- *Moving* state, in which the current values of $I_i(t)$ and $I_{DM,i}(t)$ vary rapidly but remaining synchronous, as in Figure 6.5; this state is distinguished by the previous one monitoring the time derivative of both $I_i(t)$ and $I_{DM,i}(t)$, denoted as $dp_I_i(t)$ and $dp_I_{DM,i}(t)$, respectively: when they start to increase in absolute value, the steady state is abandoned, and the FSM switches to the *moving* state. The synchronicity of the currents is detected by comparing $dp_I_i(t)$ and $dp_I_{DM,i}(t)$, as detailed in Subsection 6.1.3.
- *Reversing* and *Reversing_DM* states, in which only one of the two currents, $I_i(t)$ or $I_{DM,i}(t)$, changes its trend, i.e., the sign of its time derivative changes;

two states of *reversing* type are used to distinguish the two possible cases, i.e., which of the two currents is changing its trend, as sketched in Fig. 6.4.

- *Impulse* state, in which sudden impulses of $I_i(t)$, which are not present in $I_{DM,i}(t)$ occur (see Fig. 6.6); this state is recognized by monitoring the error between the time derivatives of the currents.

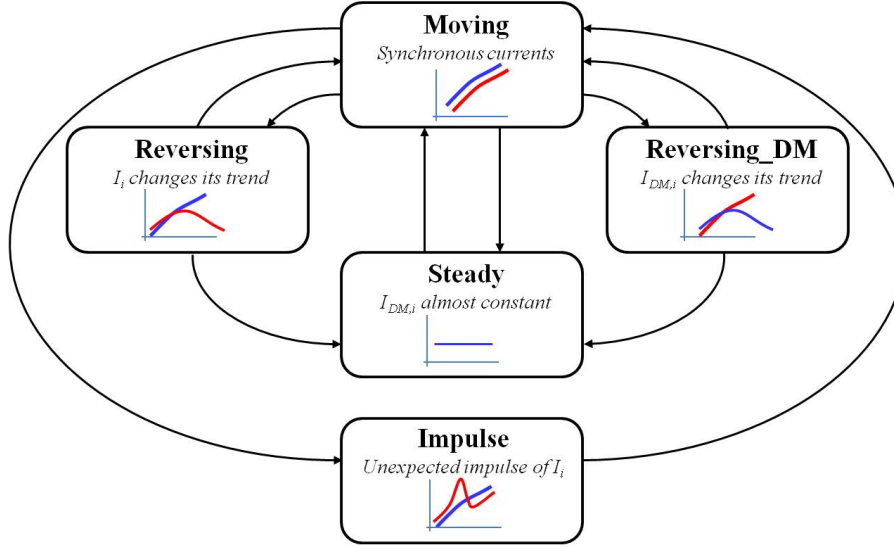
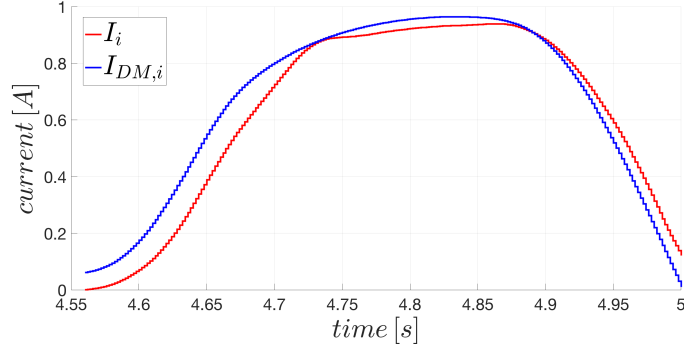
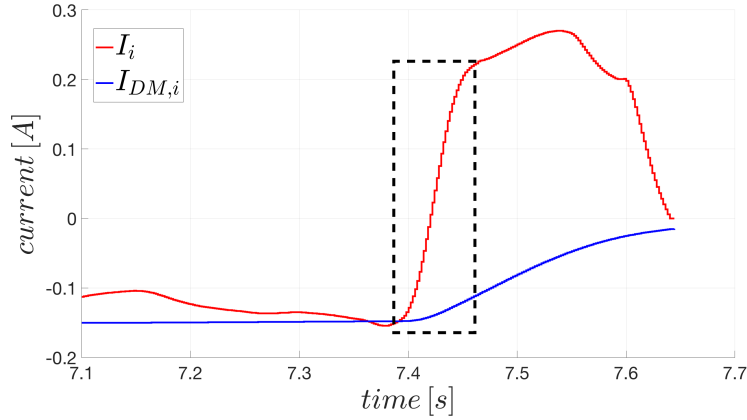


Figure 6.4: FSM scheme with sketch of the currents behavior in the various states (I_i in red and $I_{DM,i}$ in blue).

The two *reversing* states and the *impulse* one, in which the measured and the estimated currents are not accordingly varying, are surely *unsafe* states, in which the anomalous currents behavior may be due to a collision. The *moving* and the *steady* states should correspond to the standard working conditions of the robot, in which acceleration/deceleration phases alternate with constant velocity ones, but only the *moving* state can be surely considered as *safe*; in fact, the *steady* one is recognized on the basis of the estimated current behavior only, so that for the sake of robustness it is convenient to handle it as a potentially unsafe state.

6.1.3 Monitoring of the currents time derivatives

In all the states but the steady one, the time derivatives of *both* the measured and the estimated currents have to be computed and monitored. The noise that inevitably affects the measured current lets the pure numerical computation of its derivative unsuitable for our purposes. The insertion of a filter with a narrow or very narrow band has to be avoided, because it could lead to unacceptable delays in detecting changes of the current signal trend, and hence in detecting a


 Figure 6.5: Examples of the behavior of $I_i(t)$ and $I_{DM,i}(t)$ during the moving state.

 Figure 6.6: Examples of the behavior of $I_i(t)$ and $I_{DM,i}(t)$ during the impulse state.

possible collision. The adopted solution is based on the dynamical estimation of the noise affecting $dp_I_i(t)$, via the statistical computation of the error between the time derivatives of the two currents (since $I_{DM,i}(t)$ is not affected by noise, also its time derivative is not). Such a result is used to define an upper bound Th_{max} and a lower bound Th_{min} of $dp_I_i(t)$, as shown in Figures 6.7 - 6.10, where the solid red lines indicate the Th_{max} and Th_{min} bounds; the green line and the blue one represent $dp_I_i(t)$ and $I_{DM,i}(t)$, respectively, and the solid black line their difference, computed as $dp_I_i(t) - dp_I_{DM,i}(t)$.

In particular, the FSM is in moving state if one of the two following situation occurs: i) the difference between $dp_I_i(t)$ and $dp_I_{DM,i}(t)$ is within the noise limits (see Fig. 6.7), ii) $dp_I_i(t)$ and $dp_I_{DM,i}(t)$ are both over the limits but they have the same sign (see Fig. 6.8). However, when the first case occurs, if a very rapid impulse of the current values brings the error to overcome a further much higher bound (dashed red lines in Fig. 6.9), the FSM changes its state into impulse. As in the second case of the moving state, for the reversing and the reversing_DM states,

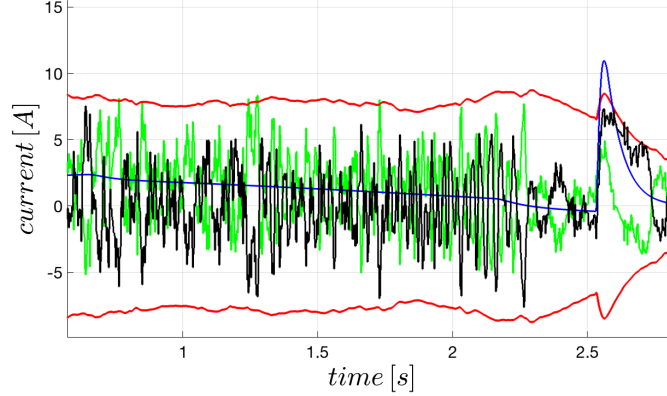


Figure 6.7: Moving state: the time derivative error is within the noise limits.

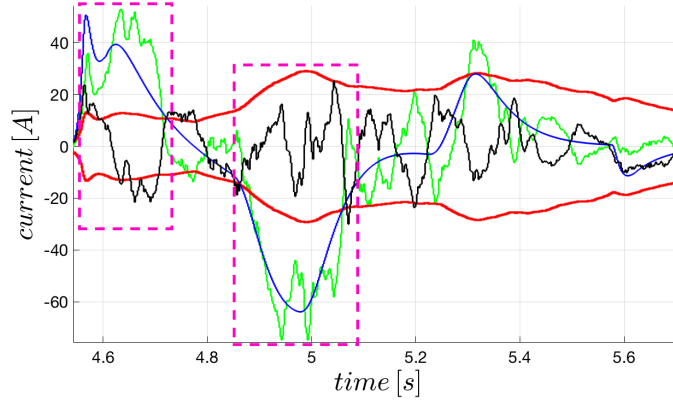


Figure 6.8: Moving state: both the time derivatives are outside the limits but having the same sign.

the error between $dp_I_i(t)$ and $dp_I_{DM,i}(t)$ and their signs are both monitored, but in this case the change of the FSM occurs when the error is over the limits and the signs of the time derivatives are different (see Fig. 6.10).

Remark 1. The insertion of the filtering action on the measured and estimated currents, and the estimation of the model error through average processes would determine an initial, transient phase in the computation of $S(t)$, in which the collision detection could be not fully reliable. This is not a problem in practice, since the duration of such a time interval is generally smaller than the waiting phase that is usually set by the robot constructors after the launch of the "drive-on" state, in which the motors are active and the manipulator is ready to perform the assigned task. In the COMAU case the duration of this phase is of some ms; such a time interval is more than sufficient to achieve a reliable value of the threshold function, so that the virtual collision sensor will be properly working also at the beginning

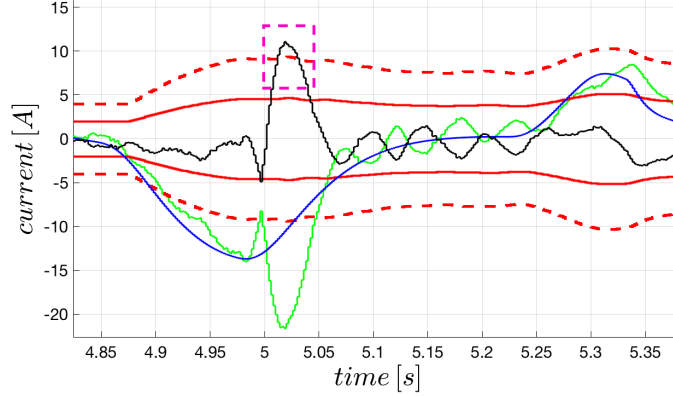


Figure 6.9: Impulse state. The highest bounds (dashed red lines) are overcome by the time derivative error.

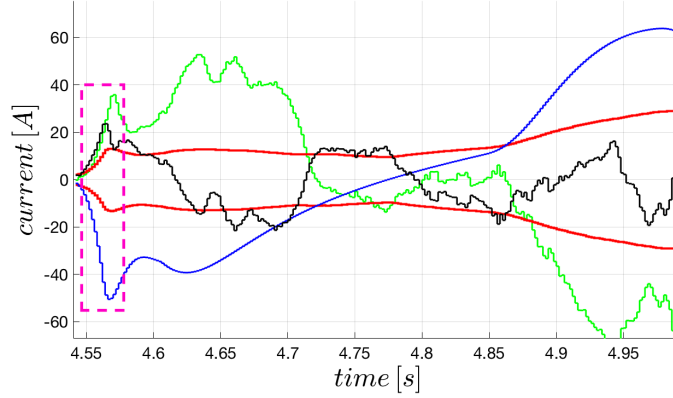


Figure 6.10: Reversing state. In the highlighted region (dashed magenta line) the time derivatives have different signs

of the robot motion.

6.2 Automatic learning and adaptation of the sensor sensitivity to collisions

The adoption of a constant vector $Coll_{bias}$ in the definition (6.4) of the threshold function has allowed satisfactory results for a wide class of manipulators, employed in different robotic applications, simply keeping the same values (heuristically determined) in all the implementations; some results are reported and discussed in Section 6.4. Despite this, significant differences have been noted with reference to the actual ability of *all* the joints of detecting a collision and/or to the speed

in detecting it. The adoption of a unique, constant $Coll_{bias}$ vector can result in quite different levels of sensor sensibility with respect to the specific behavior of each robot, with no possibility of taking into account possible slow variations in the robot behavior as time goes by.

An automatic learning and adaptation process of the sensor sensitivity has been developed to cope with such problems, under the assumption that the whole motion process of the robot is cyclic, as in any industrial application. The goal is to determine the “best” $Coll_{bias}$ term for the specific robot application through a learning phase, and to subsequently apply it enabling a *slow* adaptation phase, in which further small variations of the robot behavior are automatically taken into account. In this context the learning process is aimed at automatically finding a customized value for the $Coll_{bias}$ term for the specific installation of the virtual sensor, while the subsequent adaptation process introduces small or very small corrections to such a term, in order to avoid any false collision detection caused by slow changes of the residue values, e.g., due to temperature variations.

In the proposed learning and adaptation process, an initial, constant $Coll_{bias_0}$ vector is assumed to be available (somehow heuristically determined), and employed as $Coll_{bias}$ in the threshold function $S(t)$ used in the collision test, if the user does not request to adapt it. The entries of such a vector are generally sufficiently high to limit/avoid the risk of false collision detection during the standard, correct execution of the robot moving cycle. A learning *Bias Estimation* block is introduced (and kept *always* active), which executes a parallel collision test, still defined as in (6.3), but adopting a different threshold function, denoted as $S_{Ident}(t)$, whose i -th entry is defined as:

$$S_{Ident,i}(t) = \hat{m}_{err,i}(t) + Coll_{Ident,i}(t) \quad (6.12)$$

in which $\hat{m}_{err,i}(t)$ is still computed as in (6.8) and (6.9) in the steady and unsteady states, respectively, while $Coll_{Ident,i}(t)$ is going to be updated as in the activity diagram reported in Figure 6.11, starting from $Coll_{Ident,i}(0) = 0$. This initial choice intentionally leads to a virtual (false) collision detection by the *Bias Estimation* block, when the collision condition $|R_i(t)| > S_{Ident,i}(t)$ holds for the i -th joint. No collision actually occurs, but such a condition is used to update $Coll_{Ident,i}(t)$ imposing

$$Coll_{Ident,i}(t) = |R_i(t)| - \hat{m}_{err,i}(t) \quad (6.13)$$

i.e., equal to the minimum value which would allow to avoid a false collision detection, if used in the main collision test. The minimum duration of the learning process is set accordingly to the characteristics and duration of the whole motion that the robot cyclically repeats (e.g., a pick-and-place cycle). At least an entire cycle must be monitored during the learning phase to obtain a reliable estimate of the minimum $Coll_{Ident,i}(t)$ (denoted as $\overline{Coll}_{Ident,i}^{(0)}$) that should be adopted, but a

longer learning time can be imposed for the sake of robustness; for example in the experimental tests reported and discussed in Section 6.4 a learning time of three cycles is considered.

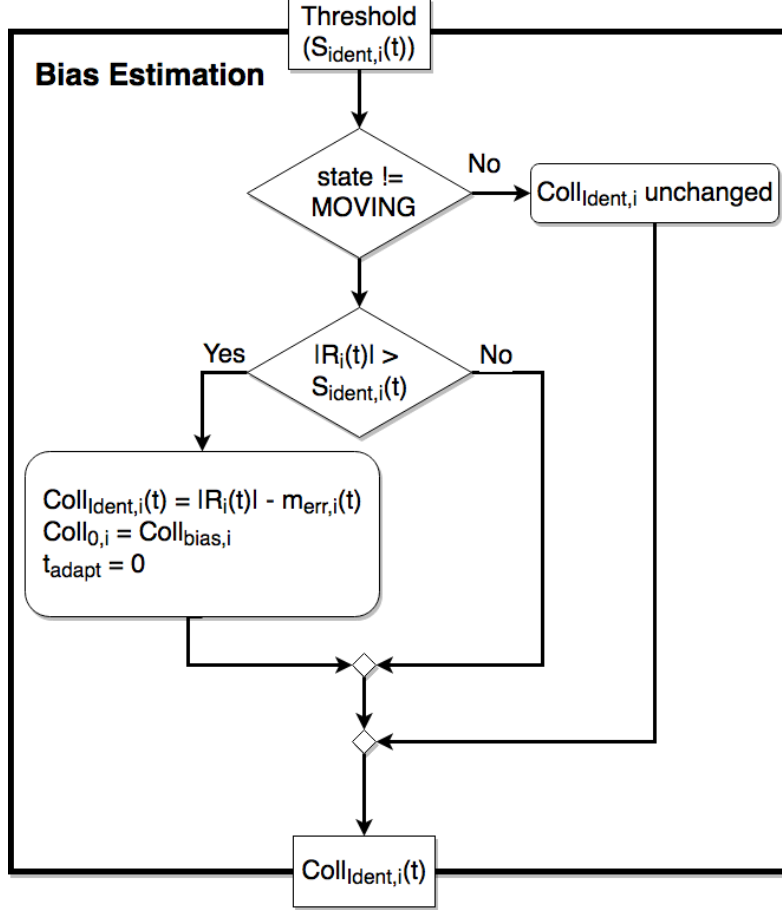


Figure 6.11: Activity diagram of the *Bias Estimation* block for the i -th joint

Further actions are performed by the *Bias Estimation* block, when a virtual (false) collision is detected, to initialize a possible subsequent adaptation process, which actually starts only if and when the user requests it. It must be underlined that even if the *Bias Estimation* block is always active, and hence $Coll_{Ident,i}$ is continuously updated, no change is introduced in (6.4) in the main collision test until the user's request. Such a request determines the immediate application of the new bias value (as soon as the minimum learning time has passed), and the start of a slow adaptation of $Coll_{bias,i}$ by defining:

$$Coll_{bias,i}(t_{adapt}) = \overline{Coll}_{Ident,i}^{(k)} + e^{(-t_{adapt}/\tau_a)} \left(Coll_{0,i} - \overline{Coll}_{Ident,i}^{(k)} \right) \quad (6.14)$$

where t_{adapt} is a time variable that is set to zero by the *Bias Estimation* block each time it detects a virtual (false) collision, $\overline{Coll}_{Ident,i}^{(k)}$ indicates the value of $Coll_{ident,i}$ updated for the k -th time by such a block *after* the start of the adaptation process ($k = 0$ corresponds to the value that directly substitutes the original $Coll_{bias0,i}$), and τ_a is the time constant of the adaptation process. τ_a must be much greater than the typical values of collision detection times, so to avoid a too rapid increase of the bias term that could prevent the correct detection of a real collision; since the collision detecting times are expected to be of the order of some tens of ms, τ_a must be chosen so to have a rise time of $Coll_{bias,i}$ of some minutes or tens of minutes. The $Coll_{0,i}$ parameter in (6.14) is used to force the application of the new bias term at the end of the learning phase, and to define the initial condition of any further adaptation process. In particular, the immediate application (after the user's request) of the first bias term $\overline{Coll}_{Ident,i}^{(0)}$, provided at the end of the learning phase, is simply achieved by imposing:

$$Coll_{0,i} = \overline{Coll}_{Ident,i}^{(0)} \quad (6.15)$$

This implies that the adaptation function (6.14) will actually change the $Coll_{bias,i}$ value only when the *Bias Estimation* block provides a new, updated estimate $\overline{Coll}_{Ident,i}^{(k)}$, with $k > 0$. Each time this happens, this new value is automatically used in (6.14), while the *Bias Estimation* block imposes:

$$\begin{aligned} Coll_{0,i} &= Coll_{bias,i} \\ t_{adapt} &= 0 \end{aligned} \quad (6.16)$$

as indicated in the activity diagram reported in Figure 6.11. These assignments make the adaptation process (6.14) restart from the current $Coll_{bias,i}$ value and let it tend to the new $\overline{Coll}_{Ident,i}^{(k)}$. Such a value will be actually, slowly reached, according to the settling time imposed by τ_a , only if in the meanwhile no further updated value $\overline{Coll}_{Ident,i}^{(k)}$ is provided by the *Bias Estimation* block, otherwise the adaptation process will restart again, imposing the convergence of $Coll_{bias,i}$ to such a new value.

Remark 3. If the user never asks for the adaptation of the $Coll_{bias}$ term, t_{adapt} remains always equal to zero and the original $Coll_{bias0}$ vector is indefinitely maintained in the threshold function (6.4) used in the collision test (6.3).

A FSM included in the virtual collision sensor handles the entire learning and adaptation process, guaranteeing in particular that the adaptive expression (6.14) of $Coll_{bias,i}(t)$ is actually used in the collision test only if a sufficient learning time has passed (corresponding to one process cycle or more, according to the user preferences, as previously discussed). This condition is ensured simply maintaining $t_{adapt} = 0$ until the established learning time has passed and a first reliable $\overline{Coll}_{Ident,i}^{(0)}$

value has been computed. The FSM (sketched in Figure 6.12) uses four states to manage the learning and adaptation process through three main services:

- *Init*, corresponding to the user's request of a new learning and adaptation process of $Coll_{bias}$. It sets to zero all the entries of $Coll_{ident}$ and the time variable t_{adapt} ; such a variable will remain locked to zero until the beginning of the adaptation phase, enabled by the subsequent *Adapt* service.
- *Set*: It allows the direct application of the new $\overline{Coll}_{Ident}^{(0)}$ vector estimated during the learning phase, from which the slow adaptation process will start.
- *Adapt*: it lets the adaptation process (6.14) start, unlocking the time variable t_{adapt} , so that the vector $Coll_{bias}$ in (6.4) becomes a slow function of time.

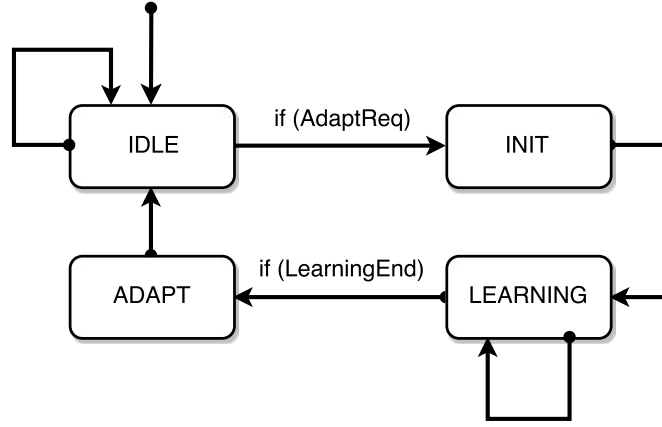


Figure 6.12: FSM for the sensor sensitivity adaptation.

The sequence of operations performed in the four states of the FSM can then be summarized as follows:

1. **IDLE**: The FSM remains in the IDLE state until an adaptation request is received. The values of the $Coll_{Ident}$ vector are continuously updated by the *Bias Estimation* block, but their values do not affect $Coll_{bias}$ and the threshold function (6.4) actually used in the collision test. The user can send a request (*AdaptReq*) using a specific instruction to be inserted in the user program.
2. **INIT**: The FSM launches the *Init* service, so that the *Bias Estimation* process restarts from $Coll_{Ident} = 0$ (any previous value of $Coll_{Ident}$ is discarded).
3. **LEARNING**: The FSM remains in the LEARNING state until the imposed learning time has passed and a reliable $\overline{Coll}_{Ident}^{(0)}$ vector has been determined by the *Bias Estimation* block. When such a waiting phase is over (*LearningEnd*), while leaving the LEARNING state the *Set* service directly applies the new $\overline{Coll}_{Ident}^{(0)}$ vector.

4. **ADAPT**: The FSM launches the *Adapt* service and then comes back to IDLE, leaving the *Bias Estimation* block and the adaptation law (6.14) both active.

A complete cycle involving the initialization (init), learning, set and adapting phases is shown in Figure 6.13; it highlights the great difference in behavior of the threshold function before and after the set instant. The figure compares the current residue (in absolute value) $R_i(t)$, the identified threshold function $S_{Ident,i}(t)$ and the threshold function actually applied in the collision test, defined as $\alpha S_i(t)$, where the $S_i(t)$ function given in (6.4) is multiplied by a factor α , slightly greater than 1, to avoid possible problems in the practical implementation, as discussed in the next section.

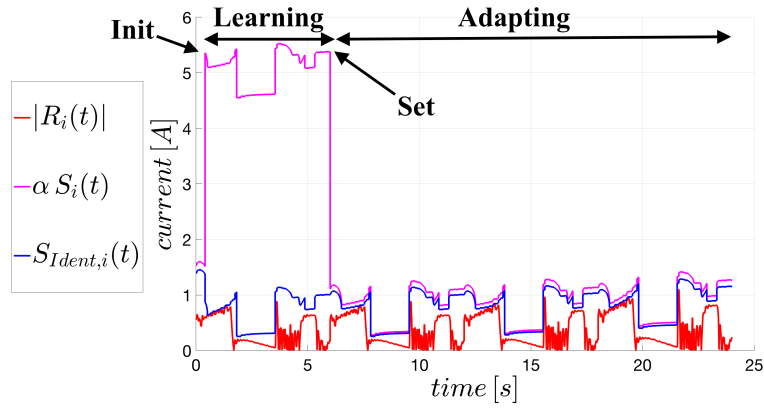


Figure 6.13: Example of a cycle involving init, learning, set and adapting phases.

6.3 Whole structure of the virtual collision sensor

The virtual collision sensor is not aware of the robotic structure of the whole system, so that it simply works checking the current values of each joint one by one. The global virtual sensor is then composed by a cycle in which each joint is tested by the *Virtual collision sensor* block; when a collision occurs a collision event is raised, so that a properly post-collision handling can be carried out. As shown in Figure 6.14, the call of the *Virtual collision sensor* block is preceded by an initialization phase, in which the parameters related to the activation and the timing of the adaptation phase are read and used for the subsequent update of the memory (*Update Memory* block).

During such a phase the object called *Collision Detection State*, containing the state of the algorithm, is used together with the new input values (e.g., current

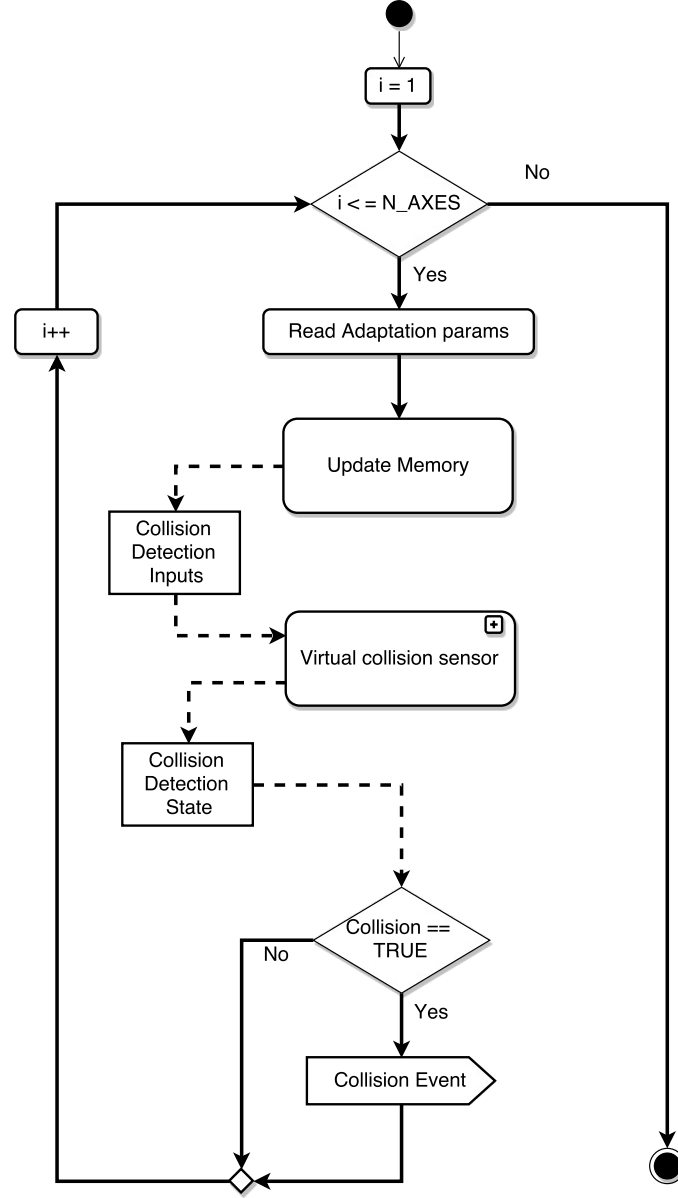


Figure 6.14: General activity diagram of the collision detection procedure

values and adaptation parameters) in order to define the inputs for the collision detection procedure (i.e., the object called *Collision Detection Inputs*).

The flowchart representing the *Virtual collision sensor* block was already presented in [42] but here some modifications are introduced in order to implement the learning and adaptation of the sensor sensitivity. The work-flow of the new *Virtual collision sensor* block (Figure 6.15) is presented using the graphical representation provided by the activity diagrams, which allows to define additional characteristics

like the parameters involved in the activities and the sets of activities that can be executed in parallel (fork/joint statement).

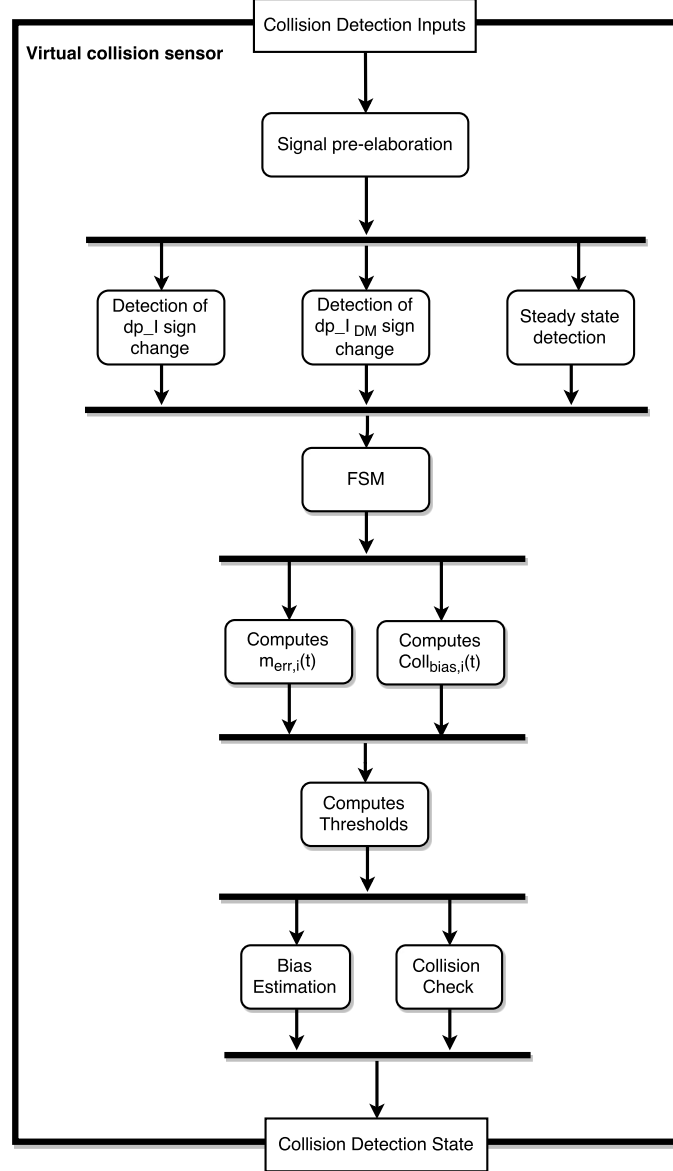


Figure 6.15: Activity diagram of the *virtual collision sensor* block

The first part of the activity flow (i.e., from the starting block to the FSM) is not changed from a conceptual point of view; the input values are pre-elaborated (e.g., applying a filtering action on the input signals) and the first and second order time derivatives of I_i and $I_{DM,i}$ are computed. The time derivatives are then used by the subsequent three blocks (which can be executed in parallel) to evaluate if I_i and $I_{DM,i}$ change their trend and to detect when $I_{DM,i}$ enters in a steady condition.

The FSM works as shown in Figure 6.4, monitoring the behavior of the current signals.

The rest of the activity diagram has been slightly changed to introduce the new features; in particular a specific block (called *Computes Coll_{bias}*) computing $Coll_{bias,i}(t)$ using (6.14) is introduced; such an activity is performed in parallel execution with respect to the computation of the model errors (carried out by the so called *Computes $\hat{m}_{err}(t)$* block, reported in Figure 6.16). The *Computes Thresholds* block performs the computation of the two thresholds $S(t)$ and $S_{Ident}(t)$, using respectively the equations (6.4) and (6.12).

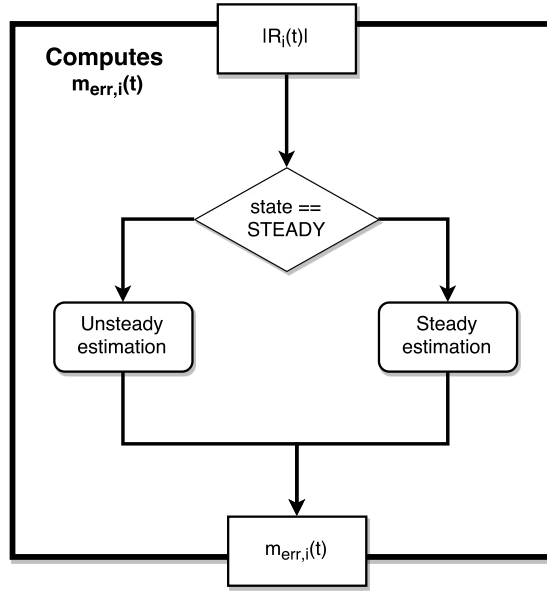
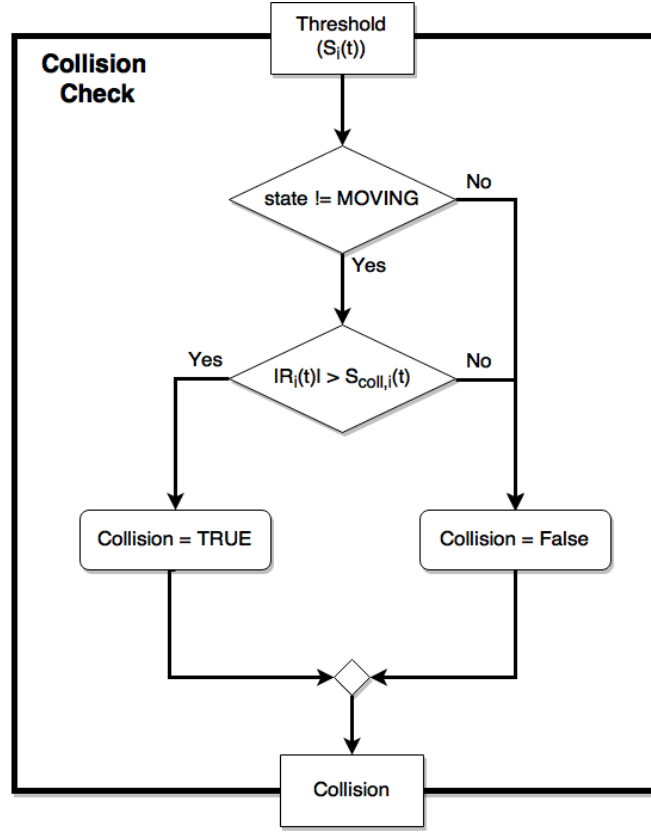


Figure 6.16: Activity diagram of the *Computes $m_{err,i}(t)$* block

The last two activities are performed in parallel; the first one computes $Coll_{Ident,i}$ as shown in the activity diagram in Figure 6.11, whereas the second one carries out the collision test (*Collision check* block, see Fig. 6.17) with small differences with respect to the basic version proposed in [42]. The adaptation phase is based on a parallel updating of $Coll_{Ident,i}(t)$ without stopping the robot.

It must be noted that the procedure properly works only if $S_i(t)$ is always greater than $S_{Ident,i}(t)$, in particular after the end of the learning phase when the subsequent *set* action is performed; in fact if such a condition does not hold, whenever the *Bias estimation* block detects a (false) collision, the *Collision Check* block would detect it as well, because in practice they would perform the same test with the same threshold. In order to avoid this kind of problem, the threshold really applied in the collision test (6.3) is slightly increased (see the example reported in Fig. 6.18) substituting $S_i(t)$ with:

Figure 6.17: Activity diagram of the *Collision Check* block

$$S_{coll,i}(t) = \alpha S_i(t) \quad (6.17)$$

where α is slightly greater than 1, just to let the threshold function used in the *Collision Check* block be always different from the one used in the *Bias Estimation* block.

Remark 4. The usage of the coefficient α does not lead the procedure to become insensitive to real collisions; in fact, after the application of the new bias through the *Set* service, the threshold function decreases drastically with respect to its initial value, so that the α coefficient cannot produce in practice an increase of the threshold function sufficient to let it reach values greater than the initial ones (see Fig. 6.13). In the worst case in which $Coll_{Ident}$ is equal to $Coll_{bias,0}$ the system would have a worsening of its sensibility of about $(\alpha - 1)\%$ with respect to the basic version.

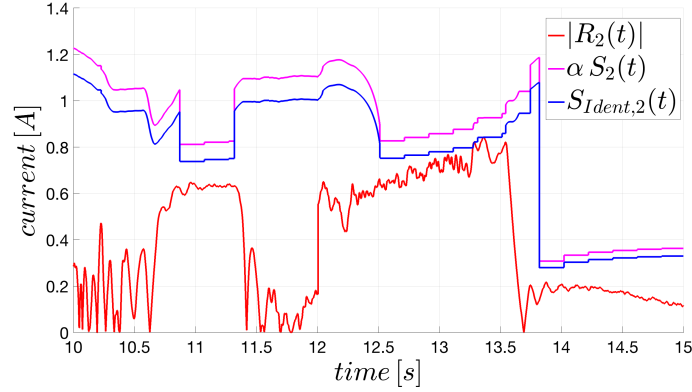


Figure 6.18: Behavior of threshold functions and current residue during the adaptation phase for the second joint of a NS12 manipulator

6.4 Experimental results

Experimental tests are carried out in order to compare the performances of the basic version of the procedure proposed in [42] with the one including the sensitivity learning and adaptation process. The experiments are performed on a COMAU NS 12 by imposing real collisions in some predefined positions of the work-space. The movements are defined using the programming language (i.e., the PDL2) of the COMAU control system (called C5G), through which four different programs has been created, that repeat the movement between a starting and a final point and vice versa, cyclically. The collisions tests are carried out for different types of movement, i.e., when the robot is moving linearly in a plane parallel to the floor (left \rightarrow right), and when the robot is moving linearly along a line perpendicular to the floor (top \rightarrow bottom). For both cases the collisions are imposed in different points of the workspace by placing an obstacle (a cardboard box of about 15 kg) along the line of movement of the robot just during the motion. In order to highlight the behavior in different conditions some of the collision points are chosen in the central part of the workspace, i.e., near the robot base (denoted as NR in Tables 6.1 and 6.2), whereas some others are taken close to the frontier of the workspace (EOS in Tables 6.1 and 6.2). Figure 6.19 shows the Cartesian paths of the tool center point of the robot, highlighting the trajectory in four cases (i.e., top \rightarrow bottom EOS, top \rightarrow bottom NR and left \rightarrow right EOS, left \rightarrow right NR), where the Cartesian point (0,0,0) denotes the base of the robot. Figure 6.20 points out the Cartesian velocity profile, underling that the robot moves cyclically between the starting and final point until the cardboard box is placed in the trajectory, so causing the subsequent collision, and hence the stop of the robot. For the sake of completeness, the joints velocities are also reported in Figure 6.21.

For each point of collision two tests are performed: i) using the basic version of

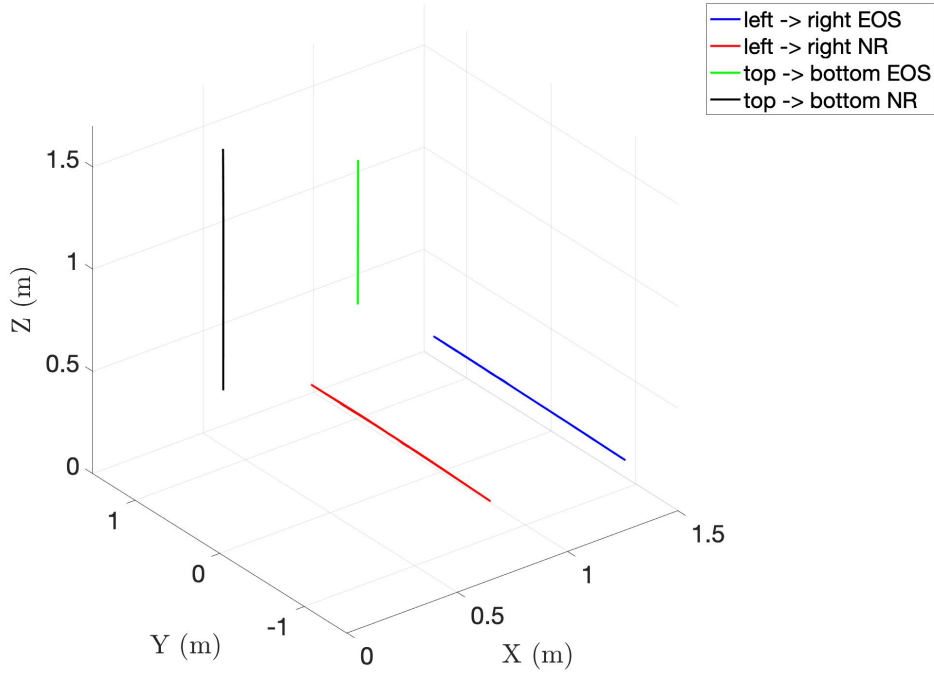


Figure 6.19: Cartesian path of the robot tool center point in the four considered cases.

Table 6.1: Comparison between the collision Detection Times (DT) of the basic version and the adaptive one

Performed tests	DT Adapt	DT Basic	Average reduction %
top \rightarrow bottom EOS	0,026	0,106	75,5
top \rightarrow bottom NR	0,024	0,186	87,1
left \rightarrow right EOS	0,010	0,044	77,3
left \rightarrow right NR	0,010	0,046	78,3

the algorithm proposed in [42], ii) using the adaptive virtual sensor; in the case of the basic version, the collision detection is enabled with the standard thresholds, whereas for the adaptive virtual sensor an initial learning phase of three cycles is performed before the collision. The obtained results show a very large decrease of the time required to detect the collision when the adaptive version is used, in particular the detection time of the basic version can be reduced between 56% and 87% (see Table 6.1). A second important improvement is related to the number of axes able to detect the collision; as shown in Table 6.2 for this particular experiment, in

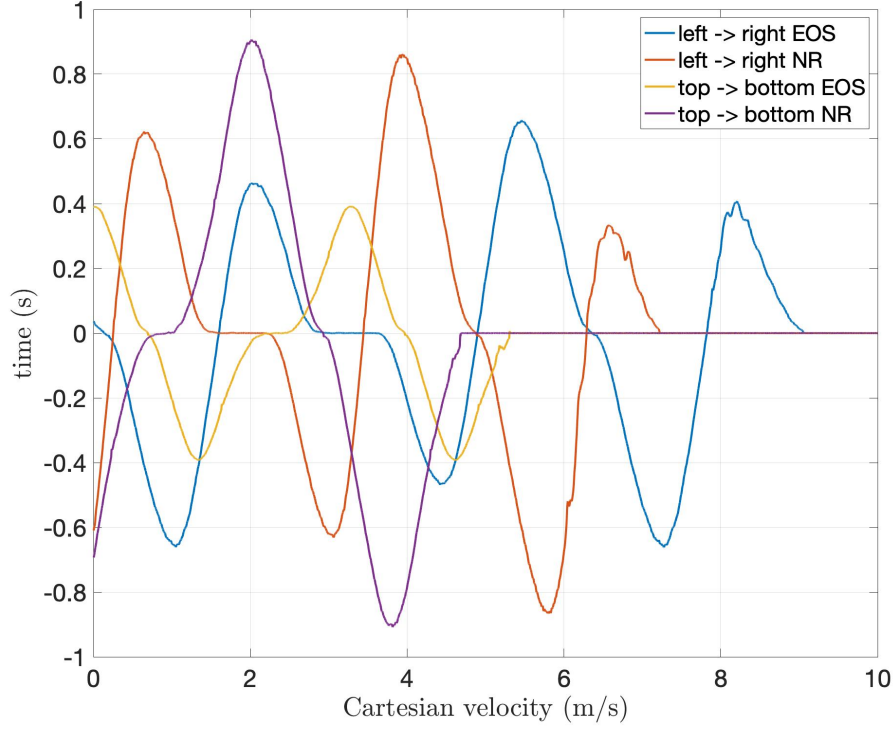


Figure 6.20: Cartesian velocity of the robot tool center point in the four considered cases.

Table 6.2: Comparison of the number of axes which are able to detect the collision using the basic collision detection procedure and the proposed virtual sensor, including the learning and adaptive functionalities

		Ax					
		1	2	3	4	5	6
Basic	Performed tests						
	top → bottom EOS	—	•	—	—	—	—
	top → bottom NR	—	—	—	—	•	•
	left → right EOS	•	—	—	—	—	—
Adaptive	left → right NR	•	—	—	•	—	—
	top → bottom EOS	—	•	•	•	•	•
	top → bottom NR	—	•	•	—	•	•
	left → right EOS	•	—	•	•	•	•
	left → right NR	•	•	•	•	•	•

which the collision with a cardboard box could be difficult to be detected because of its low stiffness, the basic algorithm is able to detect it with no more than two

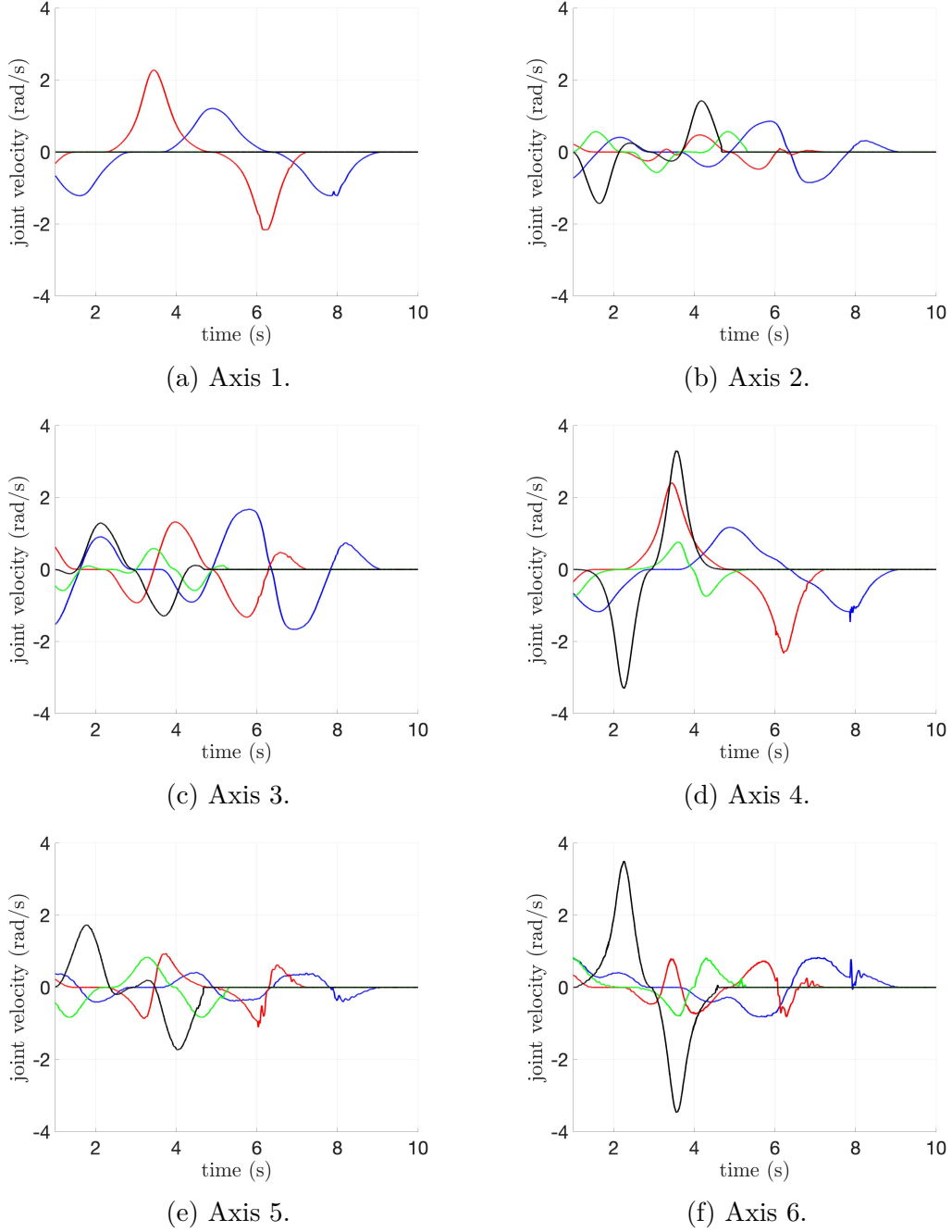


Figure 6.21: Velocity of all the axes of a Comau NS-12 in the four considered cases: 1) left \rightarrow right EOS (blue line), 2) left \rightarrow right NR (red line), 3) top \rightarrow bottom EOS (green line), and 4) top \rightarrow bottom NR (black line)

axes, whereas the adaptive version detects the collision with almost all joints (and in some cases just with all of them), thus enhancing the robustness of the collision

detection process.

Further tests have been carried out, with the adaptation process disabled (i.e., in the configuration that is commonly used in all the plants in which COMAU robots are used all over the world), on two kinds of robots having quite different characteristics: i) COMAU SMART Arc4 (Figure 6.22) with maximum payload of 12 kg and ii) COMAU NJ4 110 (Figure 6.23) with payload up to 110 kg.

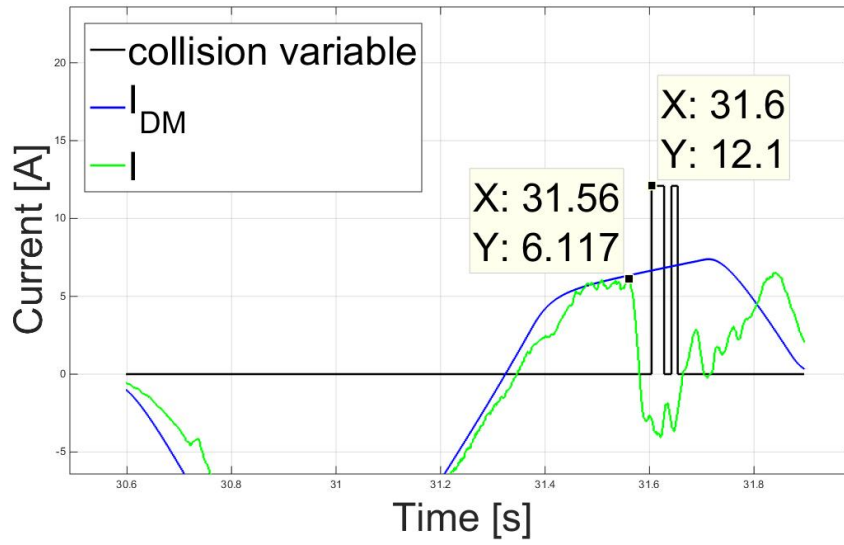


Figure 6.22: *SMART5 Arc 4*

The experiments made using the SMART Arc4 are carried out moving the robot at its maximum velocity, and programming it to firstly perform some critical movements (involving all the joints), and then to finally impact a cardboard box using the first joint only. As a result of the test, we observed that the collision detection system did not detect false collision during the first part of the experiment, while the collision imposed during the final movement is correctly detected in about 40 ms, as shown in Figure 6.24, and highlighted to the user through a red *Alarm* warning displayed in the Teach Pendant screen (Figure 6.25).

A video of this test is available in [22]; the behavior of the manipulator is showed both with and without the obstacle, highlighting the effect of the collision detection system. In particular, when the collision occurs, the manipulator stops immediately and returns in a safe position.

The heavyweight robot is then used in the subsequent tests, applying the same collision detection procedure *without any modification*. In particular, the NJ4 110 in Figure 6.23 with a payload of 90 Kg is used to perform various collision tests. In the first part of the test, the robot is monitored during some typical work-cycles, like spot-welding, and no false collisions are detected. The collision test is implemented using a wooden pallet; collisions are performed both at high velocity and at low

Figure 6.23: *SMART5 NJ4 110*Figure 6.24: Behaviour of current I compared with I_{DM} related to the first joint of a SMART Arc4 during a collision.

velocity, and during both the acceleration phase and the deceleration one (a brief video of such a test is available in [23]).



Figure 6.25: Collision detection warning.

In particular, one of the collision experiments implemented using the NJ4 110 is made during the acceleration phase, limiting the joints velocity to 40% of the maximum value allowed. The movement is programmed in order to involve all the joints and to hit the wooden pallet with the test-mass mounted on the tip of the manipulator. In this case, the collision is detected by all the joints but with different collision times, as reported in Table 6.3; in particular the most reactive has been the fifth joint, which has detected the collision in 46 ms, as highlighted in Figure 6.26.

Table 6.3: Timetables for collision detection during different experiments

Model	joint 1	joint 2	joint 3	joint 4	joint 5	joint 6
SMART5 Arc 4	40 ms	-	-	-	-	-
SMART5 NJ110	82 ms	66 ms	60 ms	50 ms	46 ms	48 ms

In general, the collision tests give different results from the detection time point of view, if they are performed in different conditions (high/low velocity, acceleration/deceleration phase, different compliance of the obstacle used), so that different

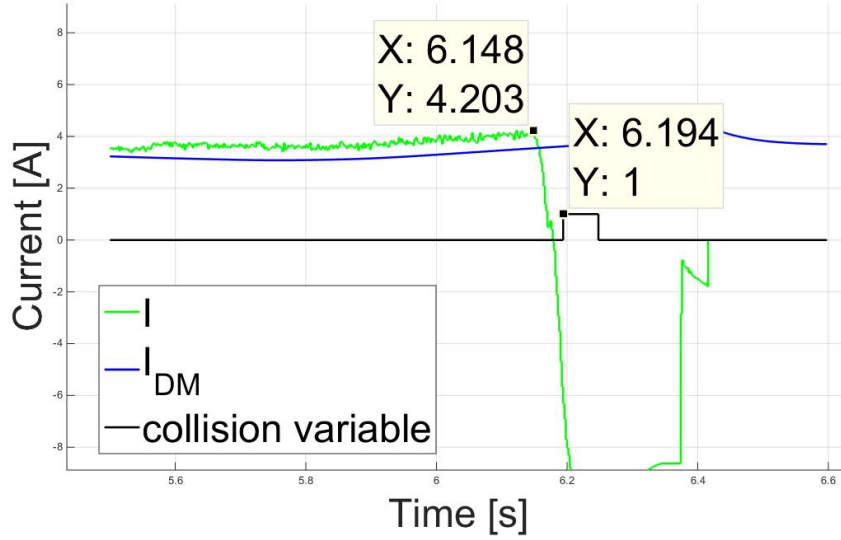


Figure 6.26: Behaviour of current I compared with I_{DM} related to the fifth joint of a NJ4 110 during a collision.

collision detection times can be obtained. For instance, if during a collision the manipulator is moving at very low velocity, the residue value grows up very slowly and the time detection increases, whereas it decreases if the robot is moving at high velocity, thanks to the quick growth of the residue value in this case. This intrinsic asymmetric behavior increases the time detection during those phases that are less dangerous (at low velocity), but allows to improve the time detection during the critical phases (at high velocity). Therefore, the timing reported in Table 6.3, concerning different experimental conditions cannot be directly compared, since different work conditions are involved in the tests.

During the experimental tests a wide range of manipulators are used, both heavyweight and lightweight, but only the most representative results are reported in the thesis for space reasons. It is important to underline that the collision detection procedure is implemented to all the manipulators without any modification: no variation of any parameter (e.g., $Coll_{bias}$ or the filter bandwidth) or in the FSM was introduced; the corresponding software was written only once and used for all the robots.

Chapter 7

Payload Check

The Payload Check algorithm aims at detecting a wrong declaration of the payload by the user. In order to perform such test, it should be necessary to have a measure of the error between the payload actually mounted on the robot and the declared one. In order to obtain the requested virtual measure, a cascade of three virtual sensors is needed: i) a joint torque sensor, ii) a force sensor and iii) a payload balance sensor. In such a context the term virtual is used to highlights that the system does not have any specific sensor measuring the payload of the robot; a set of virtual sensors are then exploited, which provides an estimate of the required measures by applying specific algorithms and/or physical models of the robotic system.

Joint torques sensor The real torque $\tau_{I,i}$ applied to each joint, in order to perform a particular movement, is related to the current absorbed by the corresponding motor. Such a relationship, in the simplest case, can be expressed through a linear model, in which the constant parameter corresponds to the motor constant. In this case a torque measure can be easily computed in the control system as:

$$\tau_I = K_I \cdot I(t) \quad (7.1)$$

where τ_I is the computed applied torque vector, I is the vector containing the motor currents provided by the real sensors, and K_I is the diagonal matrix of the motor constant values, as in (6.2).

Force sensor The force sensor will provide an estimate of the vector F of external forces applied on the End-Effector (E-E), starting from the torques τ_F correspondingly applied to the joints, using the well-known static equation of a robot:

$$\tau_F = J^T(q) F \quad (7.2)$$

where q is the joint position vector, $J(q)$ is the robot Jacobian matrix, and $F = [f_x \ f_y \ f_z \ N_x \ N_y \ N_z]^T$ is the generalized force vector, defined with respect to an inertial reference frame, having the z -axis orthogonal to the floor.

As already discussed in Section 5, in absence of any external force, the dynamic model of a manipulator can be expressed as in (5.1), in which some model parameters, like the matrices $M(q)$ and $C(q, \dot{q})$, can be estimated very well, while some others, e.g., the payload parameters, could be wrongly defined since they are usually provided by the user, so leading to following situation:

$$g(q) \neq \hat{g}(q) \quad (7.3)$$

In such a case it is possible to highlight the consequences of the wrong payload declaration considering:

$$g(q) = \hat{g}(q) + g_{err}(q) \quad (7.4)$$

Equation (7.4) implies that a wrong declaration of the payload introduces a permanent error between the real torque τ required by the robot to perform a specific movement and the corresponding torque $\hat{\tau}$ computed using the dynamic model of the manipulator. If no model error is present, the difference between τ and $\hat{\tau}$, defined as the residual torque τ_{res} , is simply equal to the error introduced during the (wrong) definition of the payload:

$$\tau_{res}(q) = g(q) - \hat{g}(q) := g_{err}(q) \quad (7.5)$$

Such residual torques can be computed as:

$$\tau_{res} = \tau_I - \hat{\tau} \quad (7.6)$$

with $\hat{\tau} = \hat{M}(q)\ddot{q} + \hat{C}(q, \dot{q})\dot{q} + \hat{\tau}_f(\dot{q}) + \hat{g}(q)$ and τ_I provided by the joint torque sensor as in (7.1).

If the robot is not in a singular configuration, the virtual force sensor computes the generalized forces F_{res} virtually corresponding to the residual torques τ_{res} from (7.2) as:

$$F_{res} = \left(J^T(q)\right)^{-1} \tau_{res} \quad (7.7)$$

This result would be correct if the only errors in the computation of $\hat{\tau}$ were due to the gravity term because of a wrong declaration of the payload. In practice the estimates of all the robot dynamic model matrices and of the friction torques differ from their true values, (i.e., $M(q) \neq \hat{M}(q)$, $C(q, \dot{q}) \neq \hat{C}(q, \dot{q})$ and $\tau_f(\dot{q}) \neq \hat{\tau}_f(\dot{q})$) thus leading to an overall torque model error vector, and consequently to a residual torque τ_{res} given by:

$$\tau_{res}(q, \dot{q}, \ddot{q}) = \tau_{model_err}(q, \dot{q}, \ddot{q}) + g_{err}(q) \quad (7.8)$$

where

$$\tau_{model_err} = (M(q) - \hat{M}(q))\ddot{q} + (C(q, \dot{q}) - \hat{C}(q, \dot{q}))\dot{q} + (\tau_f(\dot{q}) - \hat{\tau}_f(\dot{q})) \quad (7.9)$$

Such a result implies a worsening in the quality of the estimation of the forces applied on the E-E, which are affected by an error that changes on the basis of the operational condition, since τ_{model_err} is a function of q , \dot{q} and \ddot{q} .

Moreover, the force estimate provided by the virtual force sensor can be considered as feasible only if it is relative to a robot configuration sufficiently far from any singularity. In fact, in a neighborhood of a singular configuration, numerical problems may lead to a force vector whose components tend to become too high, so destroying the actual information content of the measure. Such a problem is addressed introducing a threshold (to be experimentally evaluated) on the reciprocal of the conditioning number of the Jacobian matrix, which is given by:

$$index_{cond} = \frac{\lambda_{min}}{\lambda_{max}} \quad (7.10)$$

where λ_{min} and λ_{max} are respectively the minimum and the maximum eigenvalue of J^T . If the robot configuration is such that this requirement is not satisfied, the computed force value is discarded.

Payload Balance The Payload Balance and the Payload Check algorithms are very simple and could be merged into a single algorithm; they were separated in our implementation in order to make the output of the Payload Balance available for different purposes, too.

The Payload Balance uses the output of the force virtual sensor to obtain a “measure” of the forces F applied on the E-E. The third component of such a vector contains the force along the z -axis (i.e., the force component aligned with the force of gravity), so that the payload error can be easily computed as:

$$payload_error = f_z/g \quad (7.11)$$

where g is the gravity acceleration. The value returned by the Payload Balance block is just the *payload_error* without any further elaboration; the choice to return rough values as output is justifiable because at this level the final application is still unknown.

7.1 The proposed Payload Check algorithm

The main goal of the Payload Check algorithm is to provide a fault indicator which alerts the user that the payload is not properly declared. To obtain such a service, it uses the *payload_error* in (7.11), but such a value is rough, so that a cleaning phase is required. In fact, *payload_error* should be ideally constant, but, as shown in (7.8), the τ_{model_err} term introduces further non constant errors that degrade the final estimation; in order to find the correct value, the DC component

should be then computed. However, such a computation involves the usage of a filter with a bandwidth very tight, which is difficult to implement in a real controller, because it would lead to numerical problems, also with a too high raise time. Fortunately, the requirements of any real application specify a maximum time within which the result must be available; a proper filtering action can then be introduced in order to provide a stable result within an acceptable time, which is one hour from the start of the procedure in the worst case, according to COMAU requirements. From the theoretical point of view a low-pass filter with a bandwidth of about 0.002 Hz would be a good solution, but such a filter implemented in the COMAU controller unfortunately may give numerical problems, caused by too small values of the filter parameters. For this reason the actually adopted solution is based on an average with a very high number of samples N (about 250000), which behaves in a very similar way to the proposed filter. The results showed in Figures 7.1 and 7.2 are referred to a COMAU NJ4 175, but very similar results have been obtained for all the robots used in the experimental tests.

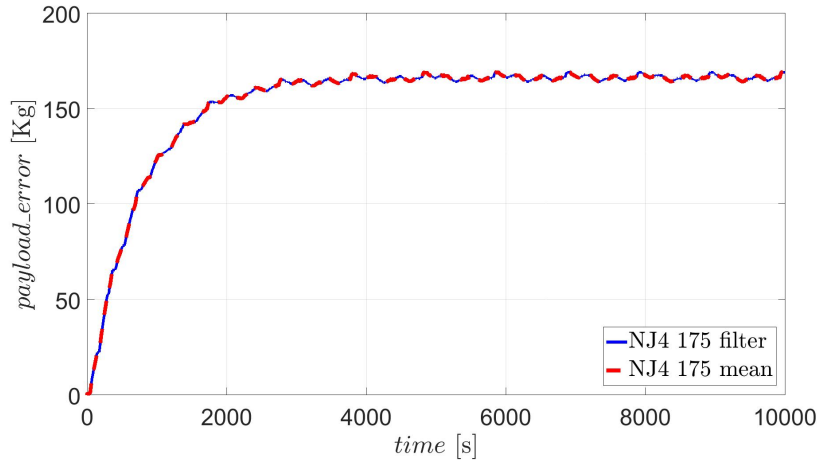


Figure 7.1: Estimation of the payload error using the lowpass filter (blue, solid line) and as average (red, dashed line)

Figure 7.1 compares the behavior of the filter and the average solution, showing their equivalence; such a result is further highlighted in Figure 7.2, which shows that the difference between the values computed using the two methods has an order of magnitude of 10^{-3} . The final result is still affected from model errors, but their effects are quite low, thanks to the properties of the proposed filter (or average); further analysis will be presented in Section 7.2.

In order to limit the usage of too much memory, necessary to keep a very large buffer of N elements, the computation of the average was implemented as:

$$\overline{\text{payload_error}} = \frac{\overline{\text{payload_error}} \cdot N + \text{payload_error}}{N + 1} \quad (7.12)$$

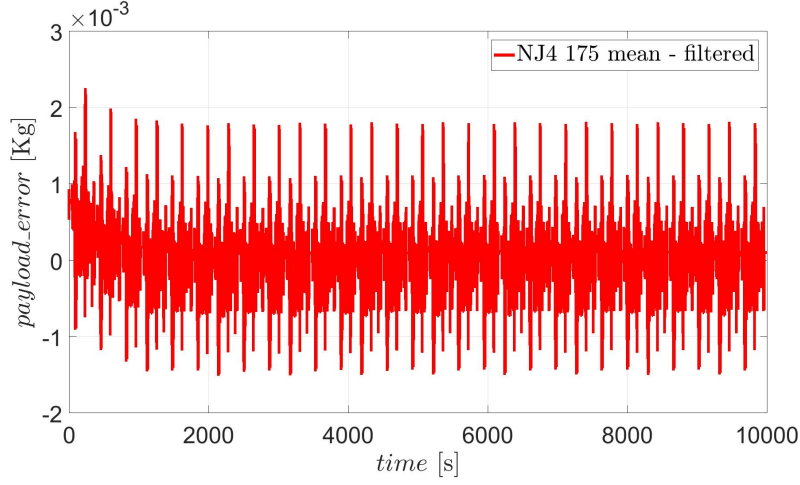


Figure 7.2: Difference between the values returned by the filter and as average.

so that only one memory location is necessary for the calculation.

The payload check algorithm applies a threshold to $\overline{payload_error}$ in order to detect the fault situation; the check can be formalized as:

$$\begin{cases} \overline{payload_error} > threshold, & \text{fault detected} \\ \overline{payload_error} \leq threshold, & \text{payload ok} \end{cases} \quad (7.13)$$

with

$$threshold = P \cdot payload_{max} \quad (7.14)$$

where P is a percentage and $payload_{max}$ is the maximum payload applicable to the robot. In order to avoid false detections in the worst case, the percentage was experimentally set to 50%.

When the faulty situation is detected, the Payload Check generates an event, which can be caught by a fault handler that carries out a specific management.

7.2 Experimental results

The Payload Check procedure was tested on a wide range of COMAU manipulators (see www.Comau.com), starting from lightweight robots with a maximum payload of few kg up to heavyweight ones, carrying payloads up to 470 kg. The procedure was also tested on manipulators with a various number of Degrees of Freedom (DoF) and different kinematic chains, possibly with single or double closed loop chains, or having the standard anthropomorphic structure with a spherical or hollow wrist.

The tests were performed using the standard program realized to carry out the “run-in” of new robots. Such a program moves the robot for four hours imposing both single joint movements and complex movements involving all the joints. The experiments led to good results if related to the proposed application; the procedure was able to find an estimation of the payload error within one hour as shown in Figure 7.3, where it is reported the behaviour of the *payload_error* over a period of 10000 seconds for a COMAU NJ 130 with a payload of 130 kg mounted on the flange and a declared payload equal to 0 kg. For this experiment the procedure

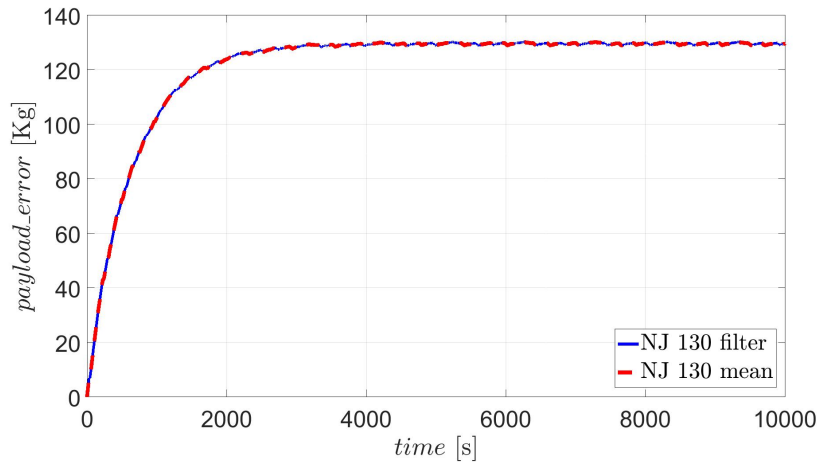


Figure 7.3: Behavior of the filtering action and the average during the transient.

should reach 130 kg of error, and the fault must be detected when *payload_error* overcomes 65 kg; thanks to the accuracy of the dynamic model of this manipulator, the fault condition is reached after about 414 seconds, much earlier than the maximum time required. Figure 7.4 highlights the steady state behavior of the payload error; due to the model errors, it shows an oscillatory component with a mean value of 129.6 kg and a standard deviation of 0.31 kg, i.e., there is a relative mean error of 0.31% and the returned values are placed around the mean value with an error of about ± 1 kg.

The presented results are valid, as in the case of the NJ 130, for manipulators

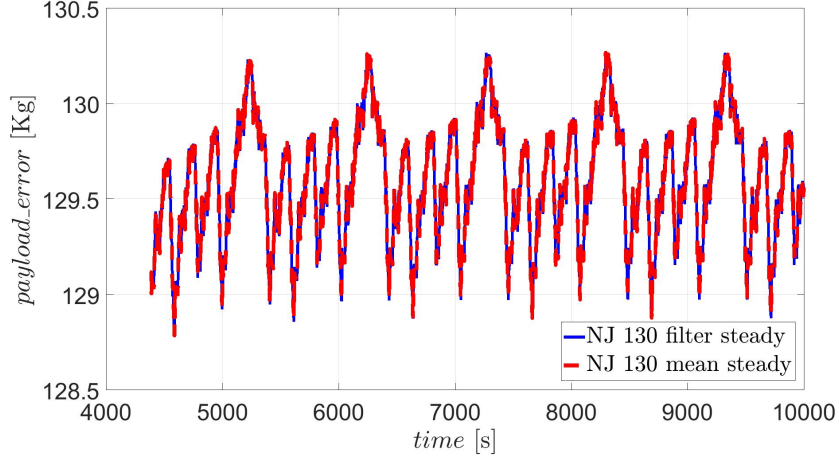


Figure 7.4: Behavior of the filtering action and the average after the transient.

whose dynamic model fits very well the real dynamic of the manipulator; in other cases worse results were obtained, but in general the relative mean error was always under the 6.5%. It is also fundamental that the movement imposed by the robot run-in program keeps it often quite far from singular configurations, otherwise too many *payload_error* values generated during the motion should have to be discarded, with a consequent worsening of the time necessary to detect the actual wrong payload declaration.

Chapter 8

Post-collision reaction and Manual Guidance

In order to give to the human operator the possibility to directly guide the motion of a non-collaborative manipulator during the programming phase, a manual guidance approach has been developed. After an overall analysis of the available state-of-the-art solutions for collision detection and manual guidance, the proposed approach is illustrated in this chapter, reporting some experimental results confirming its validity.

8.1 Sensor-less approach to Manual Guidance

A sensor-less methodology implementing virtual sensors to manage both collision detection and manual guidance sessions is proposed. Such an approach is defined as sensor-less since no external sensors are required; only information provided by the proprioceptive sensors, typically included in industrial robots, is used, i.e., position of the joints and current absorbed by the motors. The methodology includes: (i) a monitoring phase, which detects if a collision occurred, and distinguishes if it was due to an accidental impact with the environment, during a non collaborative application, or determined by an intended human-robot contact, and (ii) a post-impact phase, which imposes an appropriate reaction strategy: a MG algorithm when an intended human-robot contact is detected, or a CD reaction when an accidental collision occurs. The proposed procedure requires heuristic approaches to choose suitable values for some of the involved parameters, which must be customized for the specific robot to which it is applied. Experimental application of the procedure was made using the COMAU Racer 7 - 1.4; the choices about parameters, even if experimentally developed in a particular case, can provide useful guidelines and suggestions for the application to other manipulators.

The finite state machine shown in Figure [8.1](#) manages all the phases of the

developed procedure; it is composed of the following four states:

1. **Monitoring**: where the currents are monitored and decision parameters are updated. No orders are given to the robot for the position adjustment in this state.
2. **Manual Guidance**: where a manual guidance contact is detected and movement corrections are accordingly sent to the robot.
3. **Collision Reaction**: where a collision is detected and a reaction strategy is adopted to stop or move the robot back to a safe position.
4. **Waiting**: void state imposing the waiting of 1 s.

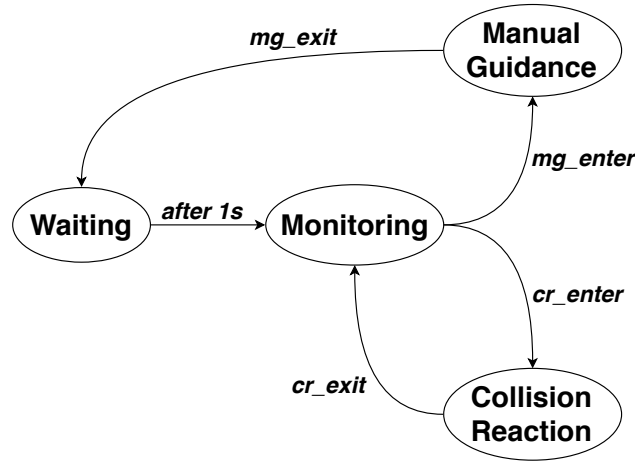


Figure 8.1: Basic state diagram for the state machine.

The algorithm starts from the **Monitoring** state, which is referred to the normal operation of the robot during which no interactions occur. When a MG interaction is detected (i.e., condition *mg_enter* is satisfied), the system moves to the **Manual Guidance** state, and the MG interaction is enabled until condition *mg_exit* is met. At this point, the system moves to the **Waiting** state, where a 1 s delay is imposed before returning to the **Monitoring** state. The same operation occurs in the case of collision detection. Whenever condition *cr_enter* is met, the system moves to the **Collision Reaction** state and performs the programmed reaction strategy, before returning to the **Monitoring** state when condition *cr_exit* is satisfied.

8.1.1 Monitoring state

The goal of the **Monitoring** state is to detect whether a collision occurred, and to distinguish if it is due to the interaction between the user and the robot

or to an accidental collision, i.e., to verify if one of the conditions mg_enter and cr_enter is satisfied. In a sensor-less context, such a goal can be reached exploiting the information included in the residual current vector (6.1). Using the matrix K_I of conversion coefficients of the motors (see equation (6.2)), it is possible to obtain the vector of residual torques as:

$$\tau_{res}(t) = K_I \cdot R(t) \quad (8.1)$$

In the absence of external forces applied to the robot, (8.1) highlights the error between the real measured torque and the value provided by the adopted dynamic model. Even if such an error cannot be canceled, it can be sufficiently reduced using an accurate robot dynamic model, as well as constraining the movements of the robot within the validity region of the model, e.g., applying low accelerations when a rigid body model is adopted. It must be underlined that, in a Manual Guidance session, accelerations are kept low for safety reasons, and hence the assumption must not be considered as restrictive in such conditions, in which equation (8.1) then provides a good estimate of the torques applied by motors to counteract the external forces applied on the robot.

When the state machine in Figure 8.1 is in the **Monitoring** state, $R(t)$ and $\tau_{res}(t)$ are computed, and both conditions mg_enter and cr_enter are evaluated at the same time, in order to deduce the nature of the interaction between the robot and the environment.

8.1.1.1 Condition mg_enter

The detection of the condition mg_enter is achieved by comparing: (i) the vector of the estimated Cartesian forces applied on the end-effector $F_{mg}(t)$ with a pair of varying threshold vectors $Th1_H(t)$ and $Th1_L(t)$, and (ii) the vector $F_{mg,s}(t)$ of the estimated Cartesian forces slopes with a constant threshold vector $Th1_s$. In practice, such conditions are based on the assumptions that when a physical interaction is underway, both the Cartesian forces detected on the end-effector and their slopes increase in absolute value; such a double check approach allows to make the detection procedure more reliable, so avoiding possible false detections, leading the robot to move in an arbitrary and haphazard manner.

In such a sensor-less context, the vector of Cartesian space forces $F_{res}(t)$, applied on the end-effector, could be obtained as:

$$F_{res}(t) = \left(J(q)^T\right)^{-1} \tau_{res}(t) \quad (8.2)$$

where J is the Jacobian matrix, and q is the joint position vector. In order to detect only MG-type interactions, and to clean up the signal as well, a proper filtering action is required for $\tau_{res}(t)$. Observing the behavior of the residual torques while MG and CD interactions are experimentally imposed (e.g., intentionally hitting the end effector, throwing an object on it, pulling the manipulator in a smooth

way or guiding it rapidly), it can be noticed that unexpected collisions produce significantly higher frequency spikes than in the manual guidance case. A low-pass filter can then be properly designed, analyzing the frequency spectrum of the force signal while MG and CD interactions are applied to the robot. Such a filter is then applied to each component of the residual current vector in (6.1), so obtaining the low-pass filtered residual current vector $I_{mg}(t)$, which is used to compute the estimate of the residual torques as:

$$\tau_{mg}(t) = K_I \cdot I_{mg}(t) \quad (8.3)$$

Using relation (8.2), replacing $\tau_{res}(t)$ with $\tau_{mg}(t)$ given in (8.3), it is possible to obtain the vector $F_{mg}(t)$, representing a good estimate of the external Cartesian forces on the end-effector as:

$$F_{mg}(t) = \left(J(q)^T \right)^{-1} \tau_{mg}(t) \quad (8.4)$$

The slope of $F_{mg}(t)$ is then computed as:

$$F_{mg,s}(t_k) = \frac{F_{mg}(t_k) - F_{mg}(t_{k-1})}{T} \quad (8.5)$$

where T is the adopted sampling time, and t_k is the time instant defined as $k \cdot T$. Vectors $F_{mg}(t)$ and $F_{mg,s}(t)$ are both used in the following set of conditions that are simultaneously tested to detect the *mg_enter* condition:

$$\begin{cases} \exists ax \in [1, 3] : F_{mg(ax)}(t) > Th1_{H(ax)}(t) \vee F_{mg(ax)}(t) < Th1_{L(ax)}(t) \\ \exists ax \in [1, 3] : |F_{mg,s(ax)}(t)| > Th1_{s(ax)} \end{cases} \quad (8.6)$$

where ax is chosen between 1 and 3, since only forces along x , y and z axes are taken into account, while vectors $Th1_H(t)$ and $Th1_L(t)$ are defined as:

$$\begin{cases} Th1_{H(ax)}(t) = \bar{F}_{res(ax)}^N(t) + 4\sigma_{F(ax)}(t) \\ Th1_{L(ax)}(t) = \bar{F}_{res(ax)}^N(t) - 4\sigma_{F(ax)}(t) \end{cases} \quad (8.7)$$

where $\bar{F}_{res}^N(t)$ is the mean of the last N samples of $F_{res}(t)$ and $\sigma_F(t)$ is its standard deviation. The term $4\sigma_F(t)$ was inserted in the thresholds definition to include 99.99% of the data inside them.

8.1.1.2 Condition *cr_enter*

The detection of the condition *cr_enter* is obtained by comparing the vector of filtered residual currents $I_{cd}(t)$ with a varying threshold vector $Th_{cd}(t)$, as detailed hereafter. Thresholds for collision detection are chosen time-varying in order to avoid false detections or missed collisions.

As highlighted before, CD and MG interactions produce a quite different behavior of the force signal, which can be analyzed using its frequency spectrum. In case of unintended collisions between the robot and the environment, high frequency spikes of force are produced, which can be detected and separated from MG signals, using a proper high-pass filter. The filter is then applied to each component of the residual current vector in (6.1), so obtaining the high-pass filtered residual current vector $I_{cd}(t)$.

Condition cr_enter is then defined as:

$$\exists j \in [1, n] : I_{cd(j)}(t) > Th_{cd(j)}(t) \quad (8.8)$$

in which n is the number of joints of the robot, $I_{cd(j)}(t)$ is the high pass filtered residual current for the j -th joint, whereas $Th_{cd(j)}(t)$ is a variable threshold defined as proposed in [34], in which the following structure has been adopted:

$$Th_{cd(j)}(t) = k_{cd_{c(j)}} + k_{cd_{v(j)}} \frac{|\dot{q}_j(t)|}{\dot{q}_{j,max}} + k_{cd_{a(j)}} \frac{|\ddot{q}_j(t)|}{\ddot{q}_{j,max}} \quad (8.9)$$

where \dot{q}_j and \ddot{q}_j denote the velocity and the acceleration of the j -th joint, respectively; the positive coefficient $k_{cd_{c(j)}}$ is chosen heuristically for each joint, to cover the high-pass filtered currents in no-motion conditions ($\dot{q} = 0$, $\ddot{q} = 0$), while $k_{cd_{v(j)}}$ and $k_{cd_{a(j)}}$ are chosen imposing, to the j -th joint, the maximum velocity ($\dot{q}_{j,max}$) and the maximum acceleration ($\ddot{q}_{j,max}$), so to set $Th_{cd(j)}(t)$ as an upper bound of the motor currents with some margin. In this way, during the standard operation of the robot, the motor currents cannot overpass the thresholds.

8.1.2 Manual Guidance state

The goal of this state is the imposition of the Cartesian movement of the end-effector defined by the human operator through the forces he/she applies on the robot. The MG session finishes when condition mg_exit is satisfied. The conversion of the Cartesian forces F_{mg} into the corresponding Cartesian positions is achieved through a proportional estimation using the theory of elasticity (only translations are taken into account) as:

$$\Delta p_{mg} = K_{mg}^{-1} \cdot F_{mg} \quad (8.10)$$

where K_{mg} is a diagonal matrix containing the stiffness parameters for the three main directions. Such compliance matrix is properly set up to impose a specific behavior as a reaction; its values depend on the maximum safe speed of the robot as well.

8.1.2.1 Condition mg_exit

Condition mg_exit is verified through a checking process in which: (i) the estimated vector of the Cartesian forces $F_{mg}(t)$ is compared with a pair of time varying threshold vectors $Th2_H(t)$ and $Th2_L(t)$, and (ii) the mean of the estimated vector of Cartesian force slopes $F_{mg,s}(t)$ is compared with a constant threshold vector C_{flat} . In practice condition mg_exit is satisfied when force signals are within the bounds and flattened.

The flatness of the signal is monitored considering a window of N_{mg} samples, in which the average of $F_{mg,s}(t)$ should be lower than a pre-determined constant vector C_{flat} in order to fulfill the flatness condition. The absolute value is also used here to include the cases of a negative slope. The values of C_{flat} are critical to be determined; a high value would cancel the flatness condition, while low values would keep the system in the **Manual Guidance** state.

Condition mg_exit is satisfied when:

$$\begin{cases} Th2_{L(ax)}(t) < F_{mg(ax)}(t) < Th2_{H(ax)}(t) & \forall ax \in [1, 3] \\ \left| \frac{1}{N_{mg}} \sum_{k=1}^{N_{mg}} (F_{mg,s(ax)}(t - k - 1)) \right| < C_{flat(ax)} & \forall ax \in [1, 3] \end{cases} \quad (8.11)$$

where $Th2_{H(ax)}(t)$ and $Th2_{L(ax)}(t)$ depend on the magnitude of the force applied to the TCP, and they are computed as:

$$\begin{cases} Th2_{H(ax)}(t) = Th1_{H(ax)}(t) + c_{th2} \cdot |\Delta F_{mH(ax)}| & for \quad \Delta F_{mH(ax)} > 0 \\ Th2_{H(ax)}(t) = Th1_{H(ax)}(t) & for \quad \Delta F_{mH(ax)} \leq 0 \end{cases} \quad (8.12)$$

$$\begin{cases} Th2_{L(ax)}(t) = Th1_{L(ax)}(t) + c_{th2} \cdot |\Delta F_{mL(ax)}| & for \quad \Delta F_{mL(ax)} < 0 \\ Th2_{L(ax)}(t) = Th1_{L(ax)}(t) & for \quad \Delta F_{mL(ax)} \geq 0 \end{cases} \quad (8.13)$$

in which c_{th2} is a pre-determined coefficient between 0 and 1, properly chosen to adjust the exit from the **Manual Guidance** state. $\Delta F_{mH(ax)}(t)$ and $\Delta F_{mL(ax)}(t)$ represent the difference between the maximum force increase after the collision and the thresholds $Th1_{H(ax)}(t)$ and $Th1_{L(ax)}(t)$, respectively, and are defined as:

$$\Delta F_{mH(ax)}(t) = \max(F_{mg(ax)}(t)) - Th1_{H(ax)}(t) \quad \forall t \geq t_c \quad (8.14)$$

$$\Delta F_{mL(ax)}(t) = \max(F_{mg(ax)}(t)) - Th1_{L(ax)}(t) \quad \forall t \geq t_c \quad (8.15)$$

where t_c is the instant at which the interaction is detected.

In practice, when the MG session is not started yet, both upper and lower levels of $Th2$ are equal to the upper and lower levels of $Th1$, respectively, since

$F_{mg}(t) < Th1_H(t)$ and $F_{mg}(t) > Th1_L(t)$. When the MG session starts, only one between $F_{mg}(t) > Th1_H(t)$ and $F_{mg}(t) < Th1_L(t)$ is satisfied depending on the direction of the change; as a consequence, only one between $Th2_H(t)$ and $Th2_L(t)$ is updated in (8.12) and (8.13), while the other one is kept at the same level. In this way, moving the arm in a certain direction will not automatically produce the same sensitivity in the opposite direction.

8.1.3 Collision Reaction state

In this state the system imposes a proper reaction strategy in order to move the TCP back, following the same direction of the collision force with a proportional magnitude. The collision peak can be isolated by applying a proper low-pass filter to the external force vector $F_{res}(t)$ given in (8.2). The so obtained low-pass filtered force vector $F_{cr}(t)$ is then used to monitor the behavior of the force peak and to compute a proportional displacement in the Cartesian space using the theory of elasticity, previously exploited for the MG algorithm. The displacements to be imposed along the x , y and z axes, collected in vector $\Delta p_{cr}(t)$, are computed as:

$$\Delta p_{cr}(t) = K_{cr}^{-1} \cdot F_{cr}(t) \quad (8.16)$$

Experimental tests were carried out on different robots to analyze the force signal during collisions. They showed that in all types of collisions the peak after an impact is attained in the subsequent 40 ms, so a good solution can be to extract the information about the direction and magnitude of the impact from the filtered force vector during this time interval.

The algorithm starts reacting directly after entering the **Collision Reaction** state. It attenuates the effect of the impact by applying a velocity profile composed of three phases:

- Phase 1: the robot position is changed according to (8.16). Such a management is applied in the first 40 ms, acquiring the following information: (i) the time instant t_1 in which the collision is detected, (ii) the force applied in t_1 , i.e., $F_{mg}(t_1)$, (iii) the time instant t_2 in which the force peak is reached, and (iv) the force applied in t_2 , i.e., $F_{mg}(t_2)$. Using such information, the time interval between the collision detection and the force peak $\Delta t = t_2 - t_1$ and the difference between the collision force and the force peak $\Delta F_{cr} = F_{mg}(t_2) - F_{mg}(t_1)$ are computed.
- Phase 2: a constant-speed is applied for a predefined time interval (160 ms was chosen for our implementation), by imposing at each time instant the displacement obtained using relation (8.16), replacing $F_{cr}(t)$ with $F_{mg}(t_2)$.

- Phase 3: the stop strategy is applied. Each motor is stopped applying a deceleration profile of N_{dec} samples, computed as:

$$N_{dec} = a_s P_{cr} + b_s \quad (8.17)$$

where a_s and b_s are two parameters defining the type of linear relation between the deceleration time and the variable P_{cr} , whereas P_{cr} can be defined in different ways; in particular three alternative strategies have been tested:

- **Strategy 1:** $P_{cr} = (\|K_{cr}^{-1} \Delta F_{cr}\| / \Delta t)$ (i.e., the stop interval N_{dec} is a linear function of the slope of the impact force)
- **Strategy 2:** $P_{cr} = \|K_{cr}^{-1} \Delta F_{cr}\|$ (i.e., the stop interval N_{dec} is a linear function of the impact force)
- **Strategy 3:** $P_{cr} = 0$ (i.e., the stop interval N_{dec} is constant and equal to b_s)

8.1.3.1 Condition cr_exit

The exit condition is automatically satisfied once the collision reaction management is ended, i.e., the robot has been stopped.

8.1.4 Waiting state

The system imposes a 1 s wait before returning to the **Monitoring** state. In this state no movement is imposed to the robot to stabilize motor currents and to calculate decision parameters based on a no-motion situation.

8.2 Experimental results

The proposed methodology has been implemented in the real industrial controller C5G of the COMAU robots, and several MG sessions and collision reaction tests have been carried out. The monitor functionality of the C5G controller has been used to perform data acquisition during MG and CD sessions. The following results are relative to a MG session carried out using a COMAU Racer 7 - 1.4; the data are directly provided by the C5G controller within a proper log file. The following experimental choices were made.

Monitoring State The matrix K_I of conversion coefficients of the motors, as well as the other robot information necessary to compute the Jacobian matrix $J(q)$ (Section 8.1.1) are taken from the characterization file of the Racer 7 - 1.4.

Condition mg_enter : The current vector $I_{mg}(t)$ suitable to check for MG condition are obtained by filtering the residual current using a low-pass filter (Section 8.1.1.1). An IIR low-pass filter with cutting frequency at 2 Hz has been then chosen for each joint. The structure of the adopted filter for the j -th joint is given by:

$$I_{mg(j)}(t_k) = a_1 \cdot I_{mg(j)}(t_{k-1}) + a_2 \cdot I_{mg(j)}(t_{k-2}) + b_1 \cdot R_{(j)}(t_k)$$

where the following values have been adopted for the three parameters: $a_1 = -1.95083889$, $a_2 = 0.95145506$, $b_1 = 0.00061616$. Thresholds $Th1_H(t)$ and $Th1_L(t)$ in (8.7) have been computed with $N = 500$, whereas $Th1_s$ in (8.6) has been heuristically set to 40 N/s, in order to remove false detections while leaving an acceptable level of sensitivity.

Condition cr_enter : The currents values $I_{cd}(t)$ for CD are obtained by applying a high-pass filter to the residual currents (Section 8.1.1.2). The digital Chebyshev filter with a cutting frequency of 10 Hz proposed in [34] has been adopted in this case, for each joint. The structure of the adopted filter for the j -th joint is given by:

$$I_{cd(j)}(t_k) = c_1 \cdot I_{res(j)}(t_k) + c_2 \cdot I_{res(j)}(t_{k-1}) + c_3 \cdot I_{res(j)}(t_{k-2}) + c_4 \cdot R_{(j)}(t_{k-3}) \quad (8.18)$$

where the following values have been adopted for the four parameters: $c_1 = -0.239207$, $c_2 = -0.6262528$, $c_3 = 0.6262528$, $c_4 = 0.239207$. The thresholds $Th_{cd(j)}(t)$ in (8.9) have been obtained using the values reported in Table 8.1.

Parameter	Joint					
	1	2	3	4	5	6
$k_{cd,c}(A)$	0.45	0.4	0.25	0.05	0.05	0.05
$k_{cd,v}(A)$	0.2	0.2	0.1	0.04	0.03	0.02
$k_{cd,a}(A)$	0.02	0.04	0.03	0.02	0.01	0.01
$\dot{q}_{max}(rad/s)$	0.05	0.06	0.09	0.32	0.32	0.36
$\ddot{q}_{max}(rad/s^2)$	0.003	0.004	0.0045	0.01	0.01	0.01

Table 8.1: Parameters of the varying threshold function $Th_{cd}(t)$ in the experimental implementation.

Manual Guidance state The compliant matrix K_{mg} in (8.10) has been heuristically set up, in order to provide an adequate feedback to the human operator during MG sessions. After multiple tests the value 300N/mm has been adopted for the stiffness along every axis.

Condition mg_exit : The condition to exit from **Manual Guidance** state is defined in (8.12), (8.13) (8.11), for which the best experimental results were obtained using the following parameters: $N_{mg} = 50$ samples, $c_{th2} = 0.7$ and $C_{flat} = 120N/s$ for each axis.

An IIR low-pass filter at 25 Hz has been adopted to filter external forces. The structure of the adopted filter for the ax-th Cartesian axis is given by:

$$F_{cr(ax)}(t_k) = d_1 F_{res(ax)}(t_k) + e_1 F_{cr(ax)}(t_{k-1}) + e_2 F_{cr(ax)}(t_{k-2}) \quad (8.19)$$

where the following values have been adopted for the three parameters: $d_1 = 0.07288762$, $e_1 = 1.46396303$, $e_2 = -0.532685065$.

Collision Reaction state For the computation of the robot displacements in (8.16), feasible results have been obtained using the value 120N/mm (or greater) for all the components of the stiffness matrix K_{cr} . The three reaction strategies have been implemented using the values reported in Table 8.2.

	Strategy 1	Strategy 2	Strategy 3
a_s	7500	281	any
b_s	-250	-62	500

Table 8.2: Values of a_s and b_s for the three reaction strategies experimentally applied.

As illustrated in the previous section, the collaboration session starts when at least one of the forces is greater than its threshold and when one of the force slopes achieves a predefined level.

Figures 8.2 and 8.3 show a manual guidance session, predominately on the x direction. The force $F_{mg(1)}(t)$ (i.e., the external force along the x direction) starts decreasing at time $t = 15.8$ s, and the slope along x overpasses its threshold of 40N/s; however, the detection happens when the $F_{mg(1)}(t)$ overpasses $Th1_{L(1)}$ at around $t = 16.1$ s. The plot in magenta is relative to a Flag showing the transition between states; it is 0 if the system is in the **Monitoring** state, and different from 0 if it is in the **Manual Guidance** state.

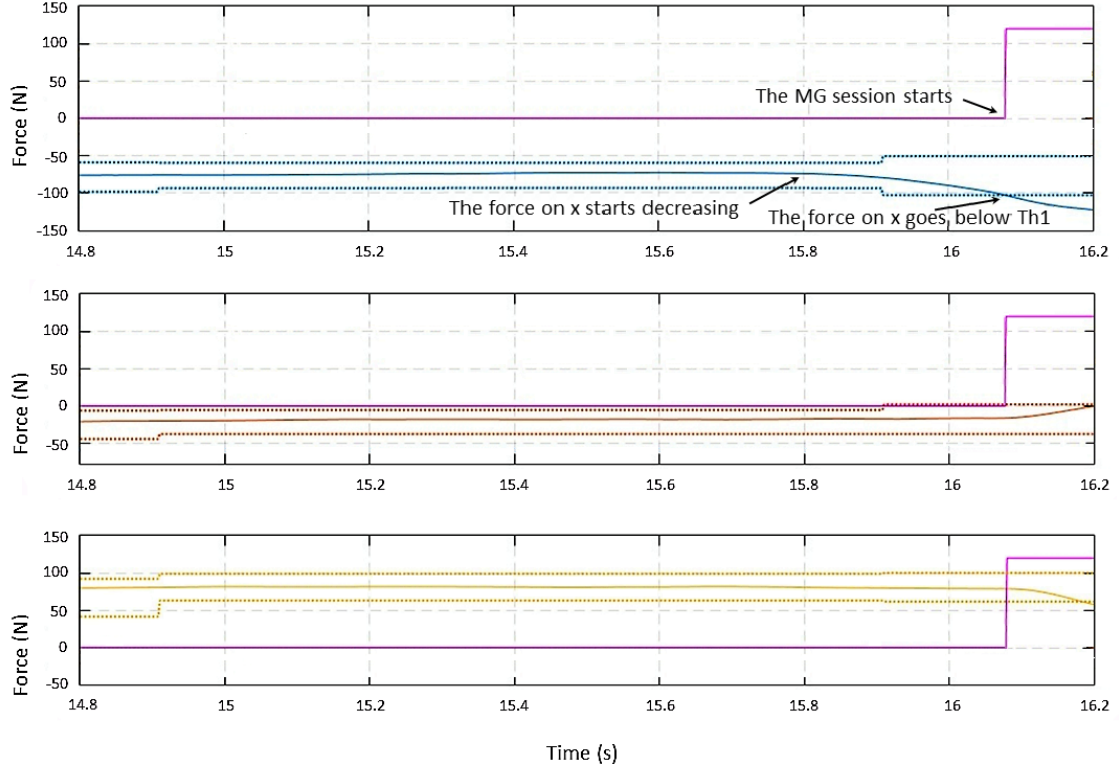


Figure 8.2: First plot from the top: filtered forces (solid blue) and $Th1$ on the x-axis (dotted blue); middle plot: filtered forces (solid orange) and $Th1$ on the y-axis (dotted orange); lowest plot: filtered forces (solid yellow) and $Th1$ on the z-axis (dotted yellow). All the three plots include the transition Flag (magenta).

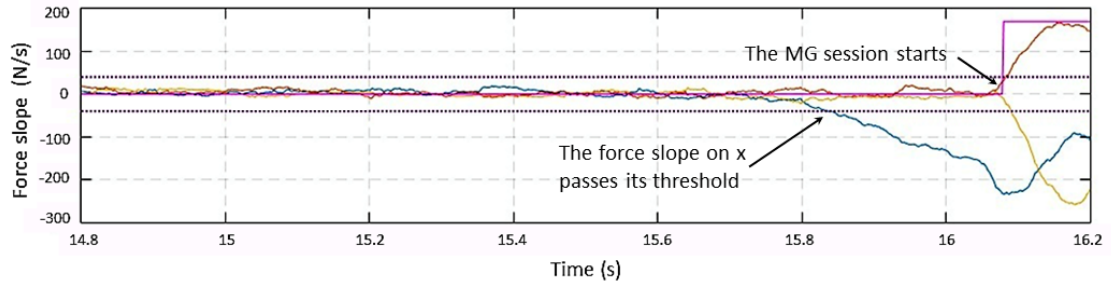


Figure 8.3: Forces slope on the x-axis (blue), forces slope on the y-axis (orange), forces slope on the z-axis (yellow), transition Flag (magenta), slope threshold (dotted violet).

The behavior of $Th2(t)$, given in (8.12) and (8.13), is shown in Figure 8.4 after entering in the **Manual Guidance** state. A force is detected on the xz plane.

When the force signals along axes x and z overpass their $Th1(t)$ thresholds, $Th2(t)$ starts updating its value following closely the change in the force signal. Once the maximum of the force signal is reached, $Th2(t)$ conserves its value at the same level until the end of the MG session. On the y-axis the force signal does not overpass $Th1(t)$, so $Th2(t)$ remains equal to $Th1(t)$.

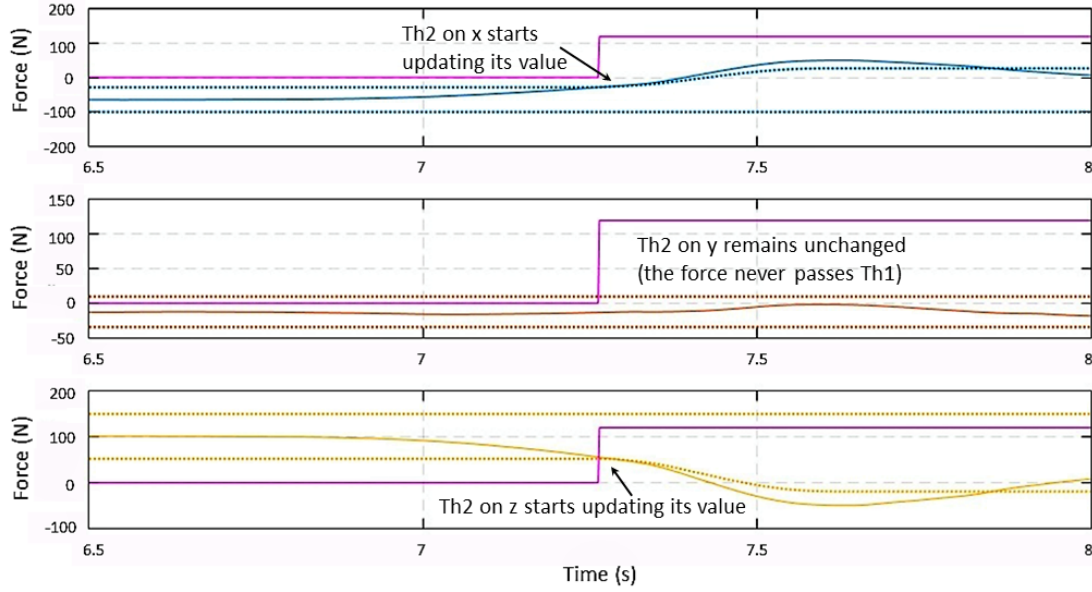


Figure 8.4: First plot from the top: filtered forces (solid blue) and $Th2$ on the x-axis (dotted blue); middle plot: filtered forces (solid orange) and $Th2$ on the y-axis (dotted orange); lowest plot: filtered forces (solid yellow) and $Th2$ on the z-axis (dotted yellow). All the three plots include the transition Flag (magenta).

After force signals go below their $Th2(t)$ thresholds and become flat, the system returns to the **Monitoring** state. Figure 8.5 shows the force behavior at the end of the collaboration session. Force signals on the x-axis and on the z-axis go between their $Th2$ limits at 7.85 s. After such time instant, all the three signals are smaller than their thresholds but the system does not exit MG session until the three force slopes go below the slopes threshold at $t = 8.1$ s, as shown in Figure 8.6.

The video in [61] shows the behavior of the Racer 7 - 1.4 during CD and MG sessions. In the first part of the video, a post collision reaction is shown, while Strategy 2 with stiffness $K = 120\text{N/mm}$ is adopted.

Different collisions are applied on the end-effector and on random points of links 4 and 5, obtaining acceptable reactions of the robot. It can be also noticed that the reaction was very good in terms of direction and safe moving away from the collision position.

The second part of the video shows a MG session. In particular (from 00:44 to 00:55), the ability of the low-pass filter to discard all high frequency components

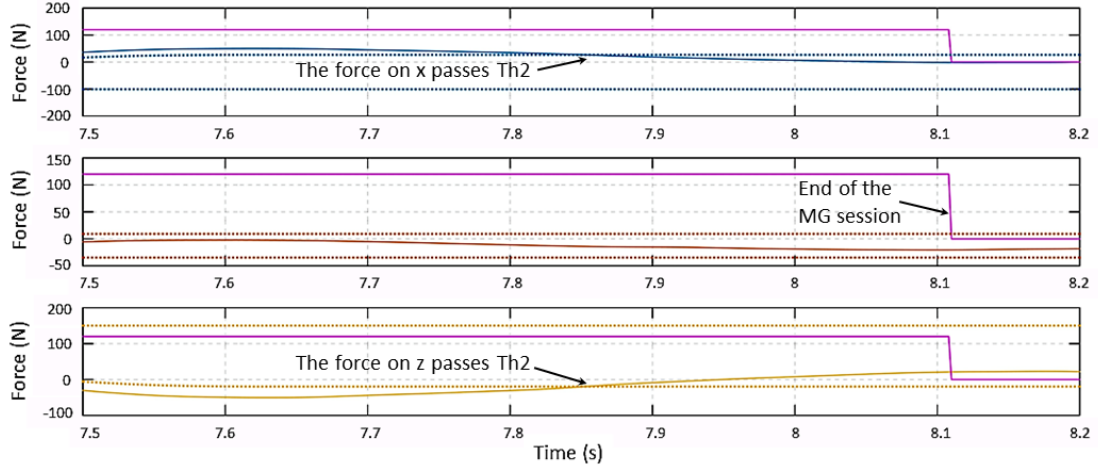


Figure 8.5: First plot from the top: filtered forces (solid blue) and $Th2$ on the x-axis (dotted blue); middle plot: filtered forces (solid orange) and $Th2$ on the y-axis (dotted orange); lowest plot: filtered forces (solid yellow) and $Th2$ on the z-axis (dotted yellow). All the three plots include the transition Flag (magenta).

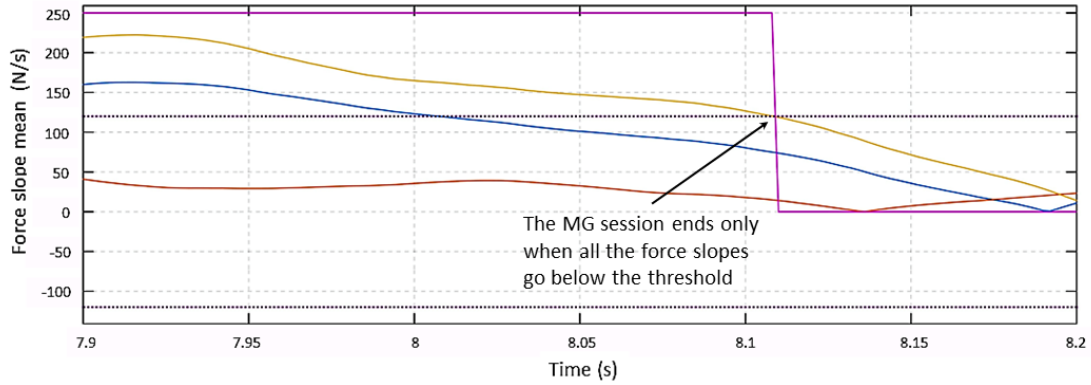


Figure 8.6: Forces slope on the x-axis (blue), forces slope on the y-axis (orange), forces slope on the z-axis (yellow), transition Flag (magenta), slope threshold (dotted violet).

resulting from a fast impact and keeping all low frequency signals resulting from a normal MG interaction is highlighted.

In the third part, a second MG session is shown. In this case, at the end of the session (at 02:05) a force on link 2 is applied, showing a slightly greater difficulty in moving the robot since a smaller leverage is applied.

In the final part of the video, a series of MG sessions are tested with a delay of 1 s between each couple of them. Forces are applied not only to the end-effector, but also to different parts of the robot. Although applying forces to the end-effector is

more accurate, collisions applied on other parts of the robot gave very good results in terms of movement direction, exiting the MG session and stopping the robot after the dissipation of the applied force.

Chapter 9

Friction modeling

Some upgrades and extensions have been developed to the results of [45], where the authors proposed two possible friction models (static and dynamic), within a general framework for the identification of friction for industrial manipulators to be integrated in the robot software architecture. The insertion of pre-computed friction values for compensation in the robot control scheme, on the basis of a previously identified model, is the simplest and lightest solution from the computational point of view, but it is obviously not able to deal with possible variations. The aim of the considered work is twofold: (*i*) the development of a general friction identification framework, including both data acquisition and identification procedures; (*ii*) the insertion of the so estimated friction model for compensation purposes in the robot control scheme. The developed friction estimation framework is general enough to be used for different manipulators without a specific customization, keeping the computational burden as low as possible, and providing a good friction reconstruction at low and very low velocity in view of possible applications in a HMI scenario. The static model proposed in [45] is used as starting point for the proposed framework, with the insertion of a rough approximation of the hysteretic behavior of friction directly in the implementation of the software module.

The developed procedure improves the basic identification solution given in [45], exploiting sub-optimal research methods to handle the nonlinear parameters of the model. Strong points of the developed framework are: (*i*) the capacity of automatically recognize friction behaviors representable by a simpler model (e.g., in absence of the Stribeck effect), and (*ii*) the automatic definition of a proper validity range of the model, so to correctly handle what happens for velocities very close to zero, avoiding an undesirable friction overcompensation in the robot control scheme.

The adopted static model [45], describes the friction of a robot joint as the sum

of four mathematical functions of the joint velocity v :

$$\begin{aligned} \tau_f(v) = & \tau_s \frac{2}{\pi} \arctan(v K_v) + \tau_{sc} \frac{2}{\pi} \arctan(v \delta) \\ & + \tau_v v + (\tau_{nlv} v^2) \frac{2}{\pi} \arctan(v K_v) \end{aligned} \quad (9.1)$$

where τ_s is the static torque, τ_{sc} is the difference between the Coulomb friction torque and the static one, τ_v is the coefficient of the viscous friction, the operator $\frac{2}{\pi} \arctan(v K_v)$ is an approximation, continuous in time, of the *sign* function, whereas the operator $\frac{2}{\pi} \arctan(v \delta)$ defines the nonlinear characteristic of the Stribeck effect through parameter δ .

Analyzing equation (9.1) it is clear that, if the parameters of the nonlinear terms (K_v, δ) are fixed, the identification becomes a pure Linear In Parameters (LIP) problem, which can be solved using the Least Squares (LS) method as:

$$\theta = (\Phi \Phi^T)^{-1} \Phi T_f \quad (9.2)$$

where

$$\begin{aligned} \Phi &= \left[\frac{2}{\pi} \arctan(v K_v), \frac{2}{\pi} \arctan(v \delta), v, v^2 \frac{2}{\pi} \arctan(v K_v) \right] \\ \theta &= [\tau_s, \tau_{sc}, \tau_v, \tau_{nlv}]^T \end{aligned} \quad (9.3)$$

and T_f and v are two column vectors belonging to R^m , the first containing the experimental friction measurements at the corresponding velocity values listed in the second one:

$$\begin{aligned} T_f &= [\tau_{f,1}, \tau_{f,2}, \dots, \tau_{f,m}]^T \\ v &= [v_1, v_2, \dots, v_m]^T \end{aligned} \quad (9.4)$$

In [45], the authors proposed a very simple method to derive a rough estimation of the parameter δ , and also suggested to simply choose a sufficiently high value for the compress factor K_v , in order to achieve a good representation of the *sign* function. Hence a more robust and general procedure based on two main phases *Rading and data pre-processing* and *Parameters identification* is developed (Figure 9.1), starting from the acquired data of T_f and v [43].

In real applications the friction model (9.1), as proposed in [45], should not be applied as it is for compensation purposes, because the range of model validity is not verified. For the simulation tests and the experimental implementation, the standard model is then converted into a new one, which properly handles friction at velocity values outside the validity range of the identified model. A limit value is used in the definition of the following limiting function, which deals with the

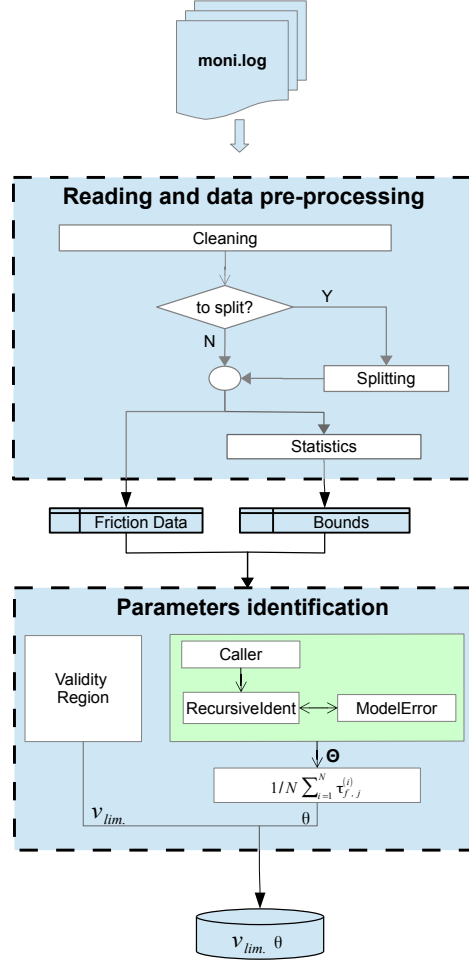


Figure 9.1: Overall scheme of the friction identification procedure.

computation of friction values for velocities smaller then v_{lim} in absolute value, leaving unchanged the torque values identified in the validity range:

$$f_{lim}(v) = \begin{cases} 1 & \text{if } |v| \geq v_{lim} \\ \left| \frac{v}{v_{lim}} \right| & \text{if } |v| < v_{lim} \end{cases} \quad (9.5)$$

Such a function (9.5) is applied to the part of (9.1) representing the friction trend at low velocity, so that the model for the real implementation becomes:

$$\begin{aligned} \tau_{fa}(v) = f_{lim}(v) & \left(\tau_s \frac{2}{\pi} \arctan(v K_v) + \tau_{sc} \frac{2}{\pi} \arctan(v \delta) \right) \\ & + \tau_v v + \left(\tau_{nlv} v^2 \right) \frac{2}{\pi} \arctan(v K_v) \end{aligned} \quad (9.6)$$

The proposed model cannot catch the dynamic behavior of friction, since it is purely static; however, small arrangements can allow to capture the main dynamic aspects. The measurement of friction torques can differ appreciably if acquired during the deceleration phase or in the acceleration one; in fact, when the motor starts the movement, a higher friction peak occurs that is no more present in the stopping phase. Such an effect, mainly due to the hysteretic behavior of friction, can be roughly reproduced using two different friction models: one for the acceleration phase and another for the deceleration one. During the acceleration of the motor, friction is represented using the standard model (9.6), but during the deceleration the initial friction peak is cut down changing the effect of the nonlinear parameter δ , which is responsible for such a behavior. In the deceleration model (9.7), δ is multiplied by a factor which steps it up of three orders of magnitude, thus removing the peak and consequently the Stribeck effect:

$$\begin{aligned} \tau_{fd}(v) = f_{lim}(v) & \left(\tau_s \frac{2}{\pi} \arctan(v K_v) \right. \\ & + \tau_{sc} \frac{2}{\pi} \arctan(v (1000\delta)) \Big) \\ & + \tau_v v + (\tau_{nlv} v^2) \frac{2}{\pi} \arctan(v K_v) \end{aligned} \quad (9.7)$$

A proper procedure allows the correct switch between the two models, avoiding any discontinuity.

Experimental tests were carried out on the COMAU Smart NS12, available in the lab of Politecnico di Torino, integrating the software modules of the developed procedure in the standard C5G controller. The goal was to compare the original COAMU friction model to the new model in terms of current reconstruction. Results reported in [43] show that the quality of the current reconstruction is enhanced for all the joints, apart from the fourth one, for which the adopted performance indices remain practically the same. This is justified by the fact that friction does not show any Stribeck effect for such a joint, so that the original Coulomb + viscous model was already suitable. For the other joints, showing a significant Stribeck effect, substantial improvements are achieved for both indices; in particular the greatest enhancements are obtained for the third joint in the first test, with a reduction of 39% for RMSE and of 28% for the mean value. Such improvements have been of considerable importance in order to obtain satisfying results from service algorithms based on the usage of the robot dynamic model, like the ones discussed in Chapters 6, 7 and 8. Reducing the model error, and hence the residual current, allowed to obtain a more reliable Collision Detection procedure, free from possible false collision warnings, as well as a better identification of the payload parameters in the Payload Check algorithm. The Manual Guidance and the Post-collision reaction algorithms benefit from such improvements, as well, providing better results in terms of accuracy of the measurement of the external forces applied to the robot.

Part III

Conclusions

The research activity carried out during the PhD, from a broader point of view, was focused on the conversion of standard robotic lines into Smart Factories, reducing as much as possible the introduction of new components or machinery. The goal was then the adoption of new software features directly applicable to the available machinery. Even if such a concept could be applicable also to new plants, where other kinds of ad-hoc choices can be made, it is very convenient for old production lines, in order to avoid any expensive stop of the production to convert the line to the Smart Factories standards. The possibility to convert the line without expensive replacements of machinery is obviously preferred. On the basis of such an idea, the research activity was then focused on two main aspects: i) the programming approach of complex robotic lines, and ii) innovative service algorithms implementable in standard industrial robots.

About the first topic, an automatic task-based programming approach has been developed, which exploits a high level layer (e.g., 3D CAD software) to define the information about both the robotic cell and the required tasks. The work was mainly devoted to the definition of a proper task model taking into account physical and functional constraints of the application, and to the automation of the programming process itself; as a final result a four phases methodology was developed, providing as output the sequence of tasks performing the required application. Such an approach, requiring as main skill the ability to model the robotic cell and the tasks using the high level layer (CAD software), allows even soft skilled programmers to program complex robotic cells. A second important feature of the proposed methodology is the faster programming speed, which allows to reprogram the robotic cell on the basis of new business requirements, if necessary. This feature is very important for old plants, where the introduction of Flexible Manufacturing Systems is not feasible; in fact, despite the proposed approach is less responsive than FMS, it allows to introduce a good level of flexibility even in this kind of production systems. In this sense, also possible machine failures can be easily managed, by fast reprogramming the robotic cell on the basis of the machinery actually available (e.g., robots) after the failure.

The second research topic was about the introduction of service algorithms into standard industrial robots without using external sensors, but only employing the ones already included in the standard robot setup. All the developed algorithms need the estimate of the motor torques, which is computed using the robot dynamic model. An important part of the research activity was then devoted to the definition of a friction model having specific characteristics to be a good trade off between the model accuracy and the constraints given by its implementation into a real-time industrial controller. A standardized machine independent methodology for data acquisition and parameter identification has been also developed, in order to make the selected friction model actually usable in the industrial context. On the basis of such a more accurate robot dynamic model, three main algorithms were developed for: i) collision detection, ii) payload identification, and iii) manual guidance and

post collision reaction.

The collision detection algorithms are very important in industrial context, also for not collaborative applications. They are in fact usually employed to detect collisions between the robot and the environment, in order to stop the robot before suffering mechanical damages. However such kind of service algorithms can be used also to detect other types of anomalous conditions, whose effects are similar to those of collisions. A typical example can be found in arc welding applications, where sometimes the tip of gun sticks to the work-piece; in such a case the collision detection algorithm can detect the anomalous situation and stop the robot before damaging the tip of the gun or the work-piece itself. The main problem with the collision detection algorithms is due to the possibility to detect false collisions, provoking the unnecessary stop of the overall robotic line. The developed collision detection algorithm exploits its internal finite state machine to improve the detection rate of real collisions; furthermore, the usage of a set of time variable thresholds allowed to keep a good accuracy in detecting real collisions, so obtaining good values of detection time.

The online payload identification procedure has been developed to be an important tool for the user to evaluate the goodness of the payload mass parameter declared in the control software. Payload parameters are, in fact, usually set up by the user, which could set wrong values by mistake, so leading to different kinds of problems, e.g., a worse dynamic behavior of the robot, and the deterioration of the accuracy of the robot dynamic model. The possibility to warn the user when an anomalous payload parameter has been set up can be very important to avoid malfunctions of the robot or of other service algorithms.

The algorithm for manual guidance and post collision reaction provides two different services: i) the possibility to move the robot by hand, making the robot compliant with the external forces applied to it, and ii) the introduction of a proper management of the post collision phase in order to reduce possible risks. From a practical point of view such services can be exploited by the user to have an alternative programming methodology (based on hand guidance) and to reduce the impact force during the collision, also avoiding that the robot continues to apply a force to the environment after the collision.

All the developed service algorithms were actually implemented in a real industrial controller and tested in a realistic environment. The basic version of the Collision Detection procedure and the Payload Check were fully inserted in the control software of Comau robots and applied to factories and production lines all over the world. The results obtained with the Collision Detection procedure led to the inclusion in the list of the five finalists of the 2017 euRobotics Technology Transfer Award at the European Robotics Forum, held in Edinburgh in March 2017.

Table 9.1 reports the full list of my publications

Table 9.1: List of publications

M. Indri, L. Lachello, I. Lazzero, F. Sibona and S. Trapani. "Smart Sensors Applications for a New Paradigm of a Production Line". In: *Sensors* 19.3 - 650 (2019).

M. Indri and S. Trapani. "Using Virtual Sensors in Industrial Manipulators for Service Algorithms Like Payload Checking". In: *Advances in Service and Industrial Robotics*. Cham: Springer International Publishing, 2018, pp. 138–146. isbn: 978-3-319-61276-8.

M. Indri and S. Trapani. "Programming robot work flows with a task modeling approach". In: *IECON 2018 - 44th Annual Conference of the IEEE Industrial Electronics Society*. Oct. 2018, pp. 2619–2624. doi: 10.1109/IECON.2018.8591629.

M. Indri, S. Trapani, A. Bonci and M. Pirani. "Integration of a Production Efficiency Tool with a General Robot Task Modeling Approach". In: *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*. Vol. 1. Sept. 2018, pp. 1273–1280. doi: 10.1109/ETFA.2018.8502666.

S. Trapani and M. Indri. "Task modeling for task-oriented robot programming". In: *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. Sept. 2017, pp. 1–8. doi: 10.1109/ETFA.2017.8247650.

M. Indri, S. Trapani, and I. Lazzero. "Development of a virtual collision sensor for industrial robots". In: *Sensors* 17.5 (2017), p. 1148.

M. Indri, S. Trapani, and I. Lazzero. "Development of a general friction identification framework for industrial manipulators". In: *IECON 2016 - 42nd Annual Conference of the IEEE Industrial Electronics Society*. Oct. 2016, pp. 6859–6866. doi: 10.1109/IECON.2016.7794017.

M. Indri, S. Trapani, and I. Lazzero. "A general procedure for collision detection between an industrial robot and the environment". In: *2015 IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA)*. Sept. 2015, pp. 1–8. doi: 10.1109/ETFA.2015.7301539.

Bibliography

- [1] Eric A. Hansen and Shlomo Zilberstein. “Heuristic Search in Cyclic AND/OR Graphs”. In: *Proceedings of the National Conference on Artificial Intelligence* (June 2003).
- [2] W. M. P. Van der Aalst. “Three good reasons for using a Petri-net-based workflow management system”. In: *Information and Process Integration in Enterprises*. Springer, 1998, pp. 161–182.
- [3] W. M. P. Van der Aalst, A. Hirnschall, and H. M. W. Verbeek. “An Alternative Way to Analyze Workflow Graphs”. In: *Advanced Information Systems Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 535–552. ISBN: 978-3-540-47961-1.
- [4] T. Abbas and B. A. MacDonald. “Generalizing topological task graphs from multiple symbolic demonstrations in programming by demonstration (PbD) processes”. In: *2011 IEEE International Conference on Robotics and Automation*. May 2011, pp. 3816–3821. DOI: [10.1109/ICRA.2011.5979848](https://doi.org/10.1109/ICRA.2011.5979848).
- [5] S. Alexandrova et al. “Robot programming by demonstration with interactive action visualizations”. In: *Robotics: science and systems*. Citeseer. 2014.
- [6] B. Argall et al. “A survey of robot learning from demonstration”. In: *Robotics and Autonomous Systems* 57 (May 2009), pp. 469–483. DOI: [10.1016/j.robot.2008.10.024](https://doi.org/10.1016/j.robot.2008.10.024).
- [7] B. Armstorng -Helouvry, P. Dupont, and C. Canudas de Wit. “A Survey of Models, Analysis Tools and Compensation Methods for the Control of Machines with Friction”. In: *Automatica*, vol. 30, no. 7, 1994, pp. 1083–1138.
- [8] J. Backhaus and G. Reinhart. “Digital description of products, processes and resources for task-oriented programming of assembly systems”. In: *Journal of Intelligent Manufacturing* 28 (Mar. 2015). DOI: [10.1007/s10845-015-1063-3](https://doi.org/10.1007/s10845-015-1063-3).
- [9] L. Bascetta et al. “Walk-through programming for robotic manipulators based on admittance control”. In: *Robotica* 31.7 (2013), pp. 1143–1153.

- [10] A. K. Bedaka, J. Vidal, and C. Y. Lin. “Automatic robot path integration using three-dimensional vision and offline programming”. In: *The International Journal of Advanced Manufacturing Technology* (Jan. 2019), pp. 1–16. DOI: [10.1007/s00170-018-03282-w](https://doi.org/10.1007/s00170-018-03282-w).
- [11] A. Bisson. “Development of an interface for intuitive teleoperation of COMAU manipulator robots using RGB-D sensors”. In: (2014).
- [12] B. Bona and M. Indri. “Friction Compensation in Robotics: an Overview”. In: *Proceedings of the 44th IEEE Conference on Decision and Control*. Dec. 2005, pp. 4360–4367. DOI: [10.1109/CDC.2005.1582848](https://doi.org/10.1109/CDC.2005.1582848).
- [13] A. Bonci, S. Longhi, and M. Pirani. “Holonc Management Tree Technique for Performance Improvement over Self-similar System Structures”. In: *In press, Journal of Management Studies* (2019).
- [14] A. Bonci, M. Pirani, and S. Longhi. “An embedded database technology perspective in cyber-physical production systems”. In: *Procedia Manufacturing* 11 (2017), pp. 830–837.
- [15] A. Bonci, M. Pirani, and S. Longhi. “Robotics 4.0: Performance improvement made easy”. In: *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. Sept. 2017, pp. 1–8. DOI: [10.1109/ETFA.2017.8247682](https://doi.org/10.1109/ETFA.2017.8247682).
- [16] A. Bonci et al. “Performance Improvement in CPSs over Self-similar System Structures”. In: *IFAC-PapersOnLine* 51.11 (2018). 16th IFAC Symposium on Information Control Problems in Manufacturing INCOM 2018, pp. 570–575. ISSN: 2405-8963. DOI: <https://doi.org/10.1016/j.ifacol.2018.08.379>. URL: <http://www.sciencedirect.com/science/article/pii/S2405896318315040>.
- [17] M. Calabrese et al. “Hierarchical-granularity holonic modelling”. In: *Journal of Ambient Intelligence and Humanized Computing* 1.3 (2010), pp. 199–209.
- [18] A. Camurri, R. Minciardi, and M. Paolucci. “Synthesis procedures for Petri net modelling of a class of manufacturing systems”. In: *Proceedings of IEEE International Conference on Systems, Man and Cybernetics*. Vol. 2. Oct. 1994, 1989–1994 vol.2. DOI: [10.1109/ICSMC.1994.400144](https://doi.org/10.1109/ICSMC.1994.400144).
- [19] T. Cao and A. C. Sanderson. “Task sequence planning in a robot workcell using AND/OR nets”. In: *Proceedings of the 1991 IEEE International Symposium on Intelligent Control*. Aug. 1991, pp. 239–244. DOI: [10.1109/ISIC.1991.187364](https://doi.org/10.1109/ISIC.1991.187364).
- [20] F. S. Cheng. “Programming advanced control functions for enhanced intelligence of industrial robots”. In: *2010 8th World Congress on Intelligent Control and Automation*. July 2010, pp. 4486–4490. DOI: [10.1109/WCICA.2010.5554085](https://doi.org/10.1109/WCICA.2010.5554085).

- [21] J. W. S. Chong et al. “Robot programming using augmented reality: An interactive method for planning collision-free paths”. In: *Robotics and Computer-Integrated Manufacturing* 25.3 (2009), pp. 689–701.
- [22] *Collision detection system for industrial manipulators 1*. <https://youtu.be/RxoGVW2etiA>.
- [23] *Collision detection system for industrial manipulators 2*. <https://youtu.be/zj62qa3I0as>.
- [24] F. Dai et al. “Robot Assembly Skills Based on Compliant Motion”. In: *Proceedings of ISR 2016: 47th International Symposium on Robotics*. June 2016, pp. 1–6.
- [25] R. Davidrajuh. “ACtivity-Oriented Petri Net for scheduling of resources”. In: *2012 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. Oct. 2012, pp. 1201–1206. DOI: [10.1109/ICSMC.2012.6377895](https://doi.org/10.1109/ICSMC.2012.6377895).
- [26] R. Davidrajuh. “Solving Assembly Line Balancing Problems with Emphasis on Cost Calculations: A Petrinets Based Approach”. In: *2014 European Modelling Symposium*. Oct. 2014, pp. 99–104. DOI: [10.1109/EMS.2014.9](https://doi.org/10.1109/EMS.2014.9).
- [27] S. Deng et al. “Application of robot offline programming in thermal spraying”. In: *Surface and Coatings Technology* 206.19-20 (2012), pp. 3875–3882.
- [28] F. Dimeas et al. “Robot Collision Detection based on Fuzzy Identification and Time Series Modelling”. In: *Proceedings of the RAAD* (2013), pp. 42–48.
- [29] V. Elia, M. G. Gnani, and A. Lanzilotto. “Evaluating the application of augmented reality devices in manufacturing from a process point of view: An AHP based model”. In: *Expert systems with applications* 63 (2016), pp. 187–197.
- [30] G. Ferretti, G. Magnani, and P. Rocco. “Assigning virtual tool dynamics to an industrial robot through an admittance controller”. In: *2009 International Conference on Advanced Robotics*. June 2009, pp. 1–6.
- [31] K. Fischer, M. Schillo, and J. Siekmann. “Holonc Multiagent Systems: A Foundation for the Organisation of Multiagent Systems”. In: *Holonc and Multi-Agent Systems for Manufacturing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 71–80. ISBN: 978-3-540-45185-3.
- [32] K. Foit and G. Ówikła. “The CAD drawing as a source of data for robot programming purposes—a review”. In: *MATEC Web of Conferences*. Vol. 94. EDP Sciences. 2017, p. 05002.
- [33] M. Forbes et al. “Robot Programming by Demonstration with situated spatial language understanding”. In: *2015 IEEE International Conference on Robotics and Automation (ICRA)*. May 2015, pp. 2014–2020. DOI: [10.1109/ICRA.2015.7139462](https://doi.org/10.1109/ICRA.2015.7139462).

- [34] M. Geravand, F. Flacco, and A. De Luca. “Human-robot physical interaction and collaboration using an industrial robot with a closed control architecture”. In: *2013 IEEE International Conference on Robotics and Automation*. May 2013, pp. 4000–4007. DOI: [10.1109/ICRA.2013.6631141](https://doi.org/10.1109/ICRA.2013.6631141).
- [35] M. Ghallab, D. Nau, and P. Traverso. “The view of automated planning and acting: A position paper”. In: *Artificial Intelligence 208* (2014), pp. 1–17.
- [36] J. Haihua, W. Dongyu, and H. Fuwen. “Cooperative motion planning of dual industrial robots via offline programming”. In: *2018 WRC Symposium on Advanced Robotics and Automation (WRC SARA)*. Aug. 2018, pp. 46–51.
- [37] J. Herbst. “A Machine Learning Approach to Workflow Management”. In: *Machine Learning: ECML 2000*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 183–194. ISBN: 978-3-540-45164-8.
- [38] T. T. H. Hoang, M. Ocelllo, and T. B. Nguyen. “Applying Type Theory to Formal Specification of Recursive Multiagent Systems”. In: *2009 IEEE-RIVF International Conference on Computing and Communication Technologies*. July 2009, pp. 1–8. DOI: [10.1109/RIVF.2009.5174624](https://doi.org/10.1109/RIVF.2009.5174624).
- [39] Y. Huang, T. Row, and P. Su. “A graph-based deadlock prevention technique for FMSs Petri nets”. In: *Proceedings of SICE Annual Conference 2010*. Aug. 2010, pp. 2248–2252.
- [40] R. Ikeura and H. Inooka. “Variable impedance control of a robot for cooperation with a human”. In: *Proceedings of 1995 IEEE International Conference on Robotics and Automation*. Vol. 3. May 1995, 3097–3102 vol.3. DOI: [10.1109/ROBOT.1995.525725](https://doi.org/10.1109/ROBOT.1995.525725).
- [41] R. Ikeura, H. Monden, and H. Inooka. “Cooperative motion control of a robot and a human”. In: *Proceedings of 1994 3rd IEEE International Workshop on Robot and Human Communication*. July 1994, pp. 112–117. DOI: [10.1109/ROMAN.1994.365946](https://doi.org/10.1109/ROMAN.1994.365946).
- [42] M. Indri, S. Trapani, and I. Lazzero. “A general procedure for collision detection between an industrial robot and the environment”. In: *2015 IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA)*. Sept. 2015, pp. 1–8. DOI: [10.1109/ETFA.2015.7301539](https://doi.org/10.1109/ETFA.2015.7301539).
- [43] M. Indri, S. Trapani, and I. Lazzero. “Development of a general friction identification framework for industrial manipulators”. In: *IECON 2016 - 42nd Annual Conference of the IEEE Industrial Electronics Society*. Oct. 2016, pp. 6859–6866. DOI: [10.1109/IECON.2016.7794017](https://doi.org/10.1109/IECON.2016.7794017).
- [44] M. Indri, S. Trapani, and I. Lazzero. “Development of a virtual collision sensor for industrial robots”. In: *Sensors* 17.5 (2017), p. 1148.

- [45] M. Indri et al. “Friction modeling and identification for industrial manipulators”. In: *2013 IEEE 18th Conference on Emerging Technologies Factory Automation (ETFA)*. Sept. 2013, pp. 1–8. DOI: [10.1109/ETFA.2013.6647958](https://doi.org/10.1109/ETFA.2013.6647958).
- [46] M. Indri et al. “Integration of a Production Efficiency Tool with a General Robot Task Modeling Approach”. In: *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*. Vol. 1. Sept. 2018, pp. 1273–1280. DOI: [10.1109/ETFA.2018.8502666](https://doi.org/10.1109/ETFA.2018.8502666).
- [47] M. Indri et al. “Smart Sensors Applications for a New Paradigm of a Production Line”. In: *Sensors* 19.3 - 650 (2019).
- [48] B. Jung et al. “Enhanced collision detection method using frequency boundary of dynamic model”. In: *IEEE ISR 2013*. Oct. 2013, pp. 1–3. DOI: [10.1109/ISR.2013.6695700](https://doi.org/10.1109/ISR.2013.6695700).
- [49] A. Kabir et al. “Robotic Finishing of Interior Regions of Geometrically Complex Parts”. In: *ASME 2018 13th International Manufacturing Science and Engineering Conference*. American Society of Mechanical Engineers. 2018, V003T02A005–V003T02A005.
- [50] N. Kammerer and P. Garrec. “Dry friction modeling in dynamic identification for robot manipulators: Theory and experiments”. In: *2013 IEEE International Conference on Mechatronics (ICM)*. Feb. 2013, pp. 422–429. DOI: [10.1109/ICMECH.2013.6518574](https://doi.org/10.1109/ICMECH.2013.6518574).
- [51] C. Kardos, A. Kovács, and J. Váncza. “Towards feature-based human-robot assembly process planning”. In: *Procedia CIRP* 57 (2016), pp. 516–521.
- [52] W. Khalil, M. Gautier, and P. Lemoine. “Identification of the payload inertial parameters of industrial manipulators”. In: *Proceedings 2007 IEEE International Conference on Robotics and Automation*. Apr. 2007, pp. 4943–4948. DOI: [10.1109/ROBOT.2007.364241](https://doi.org/10.1109/ROBOT.2007.364241).
- [53] O. Kilincci. “A Petri net-based heuristic for simple assembly line balancing problem of type 2”. In: *The International Journal of Advanced Manufacturing Technology* 46 (May 2010), pp. 329–338. DOI: [10.1007/s00170-009-2082-z](https://doi.org/10.1007/s00170-009-2082-z).
- [54] O. Kilincci and B. Mirac. “A Petri net approach for simple assembly line balancing problems”. In: *The International Journal of Advanced Manufacturing Technology* 30 (Jan. 2006), pp. 1165–1173. DOI: [10.1007/s00170-005-0154-2](https://doi.org/10.1007/s00170-005-0154-2).
- [55] P. Kormushev, S. Calinon, and D. G. Caldwell. “Imitation Learning of Positional and Force Skills Demonstrated via Kinesthetic Teaching and Haptic Input”. In: *Advanced Robotics* 25 (Mar. 2011). DOI: [10.1163/016918611X558261](https://doi.org/10.1163/016918611X558261).

- [56] D. R. Liu and M. Shen. “Modeling workflows with a process-view approach”. In: *Proceedings Seventh International Conference on Database Systems for Advanced Applications. DASFAA 2001*. Apr. 2001, pp. 260–267. DOI: [10.1109/DASFAA.2001.916386](https://doi.org/10.1109/DASFAA.2001.916386).
- [57] Y. Liu, Q. Tang, and X. Tian. “A discrete method of sphere-pipe intersecting curve for robot welding by offline programming”. In: *Robotics and Computer-Integrated Manufacturing* 57 (2019), pp. 404–411.
- [58] A. D. Luca et al. “Collision Detection and Safe Reaction with the DLR-III Lightweight Manipulator Arm”. In: *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*. Oct. 2006, pp. 1623–1630. DOI: [10.1109/IRoS.2006.282053](https://doi.org/10.1109/IRoS.2006.282053).
- [59] S. Magnenat et al. “Enhancing robot programming with visual feedback and augmented reality”. In: *Proceedings of the 2015 ACM conference on innovation and technology in computer science education*. ACM. 2015, pp. 153–158.
- [60] C. Makkar et al. “A new continuously differentiable friction model for control systems design”. In: *Proceedings, 2005 IEEE/ASME International Conference on Advanced Intelligent Mechatronics*. July 2005, pp. 600–605. DOI: [10.1109/AIM.2005.1511048](https://doi.org/10.1109/AIM.2005.1511048).
- [61] *Manual Guidance and Collision Reaction experiment video*. <https://www.youtube.com/watch?v=1EyI8zNlNpk>.
- [62] P. Marayong, G. D. Hager, and A. M. Okamura. “Control methods for guidance virtual fixtures in compliant human-machine interfaces”. In: *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*. Sept. 2008, pp. 1166–1172. DOI: [10.1109/IRoS.2008.4650838](https://doi.org/10.1109/IRoS.2008.4650838).
- [63] L. Marton and B. Lantos. “Modeling, Identification, and Compensation of Stick-Slip Friction”. In: *IEEE Transactions on Industrial Electronics* 54.1 (Feb. 2007), pp. 511–521. ISSN: 0278-0046. DOI: [10.1109/TIE.2006.888804](https://doi.org/10.1109/TIE.2006.888804).
- [64] D. Massa, M. Callegari, and C. Cristalli. “Manual guidance for industrial robot programming”. In: *Industrial Robot: An International Journal* 42.5 (2015), pp. 457–465.
- [65] P. Mella. “The Holonic Revolution”. In: *Holons, Holarchies and Holonic* (2009).
- [66] S. Michieletto et al. “ROS-I Interface for COMAU Robots”. In: *Simulation, Modeling, and Programming for Autonomous Robots*. Cham: Springer International Publishing, 2014, pp. 243–254. ISBN: 978-3-319-11900-7.

- [67] Y. Mollard et al. “Robot programming from demonstration, feedback and transfer”. In: *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Sept. 2015, pp. 1825–1831. DOI: [10.1109/IROS.2015.7353615](https://doi.org/10.1109/IROS.2015.7353615).
- [68] K. M. N. Muthiah and S. H. Huang. “Overall throughput effectiveness (OTE) metric for factory-level performance monitoring and bottleneck detection”. In: *International Journal of Production Research* 45.20 (2007), pp. 4753–4769. DOI: [10.1080/00207540600786731](https://doi.org/10.1080/00207540600786731).
- [69] K. M. N. Muthiah, S. Huang, and S. Mahadevan. “Automating factory performance diagnostics using overall throughput effectiveness (OTE) metric”. In: *The International Journal of Advanced Manufacturing Technology* 36 (Mar. 2008), pp. 811–824. DOI: [10.1007/s00170-006-0891-x](https://doi.org/10.1007/s00170-006-0891-x).
- [70] P. Neto. “Off-line programming and simulation from CAD drawings: Robot-assisted sheet metal bending”. In: *IECON 2013 - 39th Annual Conference of the IEEE Industrial Electronics Society*. Nov. 2013, pp. 4235–4240.
- [71] P. Neto and N. Mendes. “Direct off-line robot programming via a common CAD package”. In: *Robotics and Autonomous Systems* 61 (Aug. 2013), pp. 896–910. DOI: [10.1016/j.robot.2013.02.005](https://doi.org/10.1016/j.robot.2013.02.005).
- [72] P. Neto, J. N. Pires, and A. P. Moreira. “CAD-based off-line robot programming”. In: *2010 IEEE Conference on Robotics, Automation and Mechatronics*. June 2010, pp. 516–521. DOI: [10.1109/RAMECH.2010.5513141](https://doi.org/10.1109/RAMECH.2010.5513141).
- [73] P. Neto et al. “CAD-based robot programming: The role of Fuzzy-PI force control in unstructured environments”. In: *2010 IEEE International Conference on Automation Science and Engineering*. Aug. 2010, pp. 362–367. DOI: [10.1109/COASE.2010.5584058](https://doi.org/10.1109/COASE.2010.5584058).
- [74] G. Olsen. “Newick’s 8: 45”. In: *Tree format standard* (1990).
- [75] Y. S. Pai, H.J. Yap, and R. Singh. “Augmented reality-based programming, planning and simulation of a robotic work cell”. In: *Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture* 229.6 (2015), pp. 1029–1045.
- [76] Z. Pan et al. “Recent Progress on Programming Methods for Industrial Robots”. In: *ISR 2010 (41st International Symposium on Robotics) and ROBOTIK 2010 (6th German Conference on Robotics)*. June 2010, pp. 1–8.
- [77] A.C. Perzylo et al. “Toward Efficient Robot Teach-in and Semantic Process Descriptions for Small Lot Sizes”. In: *RSS 2015*. 2015.
- [78] M. Pirani, A. Bonci, and S. Longhi. “A scalable production efficiency tool for the robotic cloud in the fractal factory”. In: *IECON 2016 - 42nd Annual Conference of the IEEE Industrial Electronics Society*. Oct. 2016, pp. 6847–6852. DOI: [10.1109/IECON.2016.7793536](https://doi.org/10.1109/IECON.2016.7793536).

- [79] J. N. Pires, T. Godinho, and P. Ferreira. “CAD interface for automatic robot welding programming”. In: *Industrial Robot: An International Journal* 31.1 (2004), pp. 71–76.
- [80] J. Rosell. “Assembly and task planning using Petri nets: A survey”. In: *Proceedings of The Institution of Mechanical Engineers Part B-Journal of Engineering Manufacture - PROC INST MECH ENG B-J ENG MA* 218 (Aug. 2004), pp. 987–994. DOI: [10.1243/0954405041486019](https://doi.org/10.1243/0954405041486019).
- [81] J. Rosell, N. Munoz, and A. Gambin. “Robot tasks sequence planning using Petri nets”. In: *Proceedings of the IEEE International Symposium on Assembly and Task Planning, 2003*. July 2003, pp. 24–29. DOI: [10.1109/ISATP.2003.1217182](https://doi.org/10.1109/ISATP.2003.1217182).
- [82] M. Ruderman and M. Iwasaki. “Observer of Nonlinear Friction Dynamics for Motion Control”. In: *IEEE Transactions on Industrial Electronics* 62.9 (Sept. 2015), pp. 5941–5949. ISSN: 0278-0046. DOI: [10.1109/TIE.2015.2435002](https://doi.org/10.1109/TIE.2015.2435002).
- [83] W. Sadiq and M. Orlowska. “On Correctness Issues in Conceptual Modeling of Workflows”. In: *In Proceedings of the 5th European Conference on Information Systems (ECIS 97*. Citeseer. 1997.
- [84] W. Sadiq and M. E. Orlowska. “Analyzing process models using graph reduction techniques”. In: *Information Systems* 25.2 (2000). The 11th International Conference on Advanced Information System Engineering, pp. 117–134. ISSN: 0306-4379. DOI: [https://doi.org/10.1016/S0306-4379\(00\)00012-0](https://doi.org/10.1016/S0306-4379(00)00012-0). URL: <http://www.sciencedirect.com/science/article/pii/S0306437900000120>.
- [85] J. S. Sakellariou and S.D. Fassois. “Vibration based fault detection and identification in an aircraft skeleton structure via a stochastic functional model based method”. In: *Mechanical Systems and Signal Processing* 22.3 (2008), pp. 557–573. ISSN: 0888-3270. DOI: <https://doi.org/10.1016/j.ymssp.2007.09.002>. URL: <http://www.sciencedirect.com/science/article/pii/S0888327007001793>.
- [86] K. Salimifard and M. Wright. “Petri net-based modelling of workflow systems: An overview”. In: *European journal of operational research* 134.3 (2001), pp. 664–676.
- [87] B. Siciliano and L. Villani. *Robot force control*. Vol. 540. Springer Science & Business Media, 2012.
- [88] L. Simoni et al. “Friction modeling with temperature effects for industrial robot manipulators”. In: *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Sept. 2015, pp. 3524–3529.

- [89] S. Stadler et al. "Augmented reality for industrial robot programmers: Workload analysis for task-based, augmented reality-supported robot control". In: *2016 25th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN)*. Aug. 2016, pp. 179–184. DOI: [10.1109/ROMAN.2016.7745108](https://doi.org/10.1109/ROMAN.2016.7745108).
- [90] K. Suita et al. "A failure-to-safety "Kyozon" system with simple contact detection and stop capabilities for safe human-autonomous robot coexistence". In: *Proceedings of 1995 IEEE International Conference on Robotics and Automation*. Vol. 3. May 1995, 3089–3096 vol.3. DOI: [10.1109/ROBOT.1995.525724](https://doi.org/10.1109/ROBOT.1995.525724).
- [91] S. Takakura, T. Murakami, and K. Ohnishi. "An approach to collision detection and recovery motion in industrial robot". In: *15th Annual Conference of IEEE Industrial Electronics Society*. Nov. 1989, 421–426 vol.2. DOI: [10.1109/IECON.1989.69669](https://doi.org/10.1109/IECON.1989.69669).
- [92] K. P. Tee, R. Yan, and H. Li. "Adaptive admittance control of a robot manipulator under task space constraint". In: *2010 IEEE International Conference on Robotics and Automation*. May 2010, pp. 5181–5186. DOI: [10.1109/ROBOT.2010.5509874](https://doi.org/10.1109/ROBOT.2010.5509874).
- [93] G. J. Tsinarakis, K. P. Valavanis, and N. C. Tsourveloudis. "Modular Petri net based modeling, analysis and synthesis of dedicated production systems". In: *2003 IEEE International Conference on Robotics and Automation (Cat. No.03CH37422)*. Vol. 3. Sept. 2003, 3559–3564 vol.3. DOI: [10.1109/ROBOT.2003.1242141](https://doi.org/10.1109/ROBOT.2003.1242141).
- [94] Y. Wang et al. "Hierarchical Task Planning for Multiarm Robot with Multi-constraint". In: *Mathematical Problems in Engineering* 2016 (2016).
- [95] R. Zoellner et al. "Towards Cognitive Robots: Building Hierarchical Task Representations of Manipulations from Human Demonstration". In: vol. 2005. May 2005, pp. 1535–1540. DOI: [10.1109/ROBOT.2005.1570332](https://doi.org/10.1109/ROBOT.2005.1570332).

This Ph.D. thesis has been typeset by means of the \TeX -system facilities. The typesetting engine was \pdfL\TeX . The document class was `toptesi`, by Claudio Beccari, with option `tipotesi=scudo`. This class is available in every up-to-date and complete \TeX -system installation.