

POLITECNICO DI MILANO
SCUOLA DI INGEGNERIA INDUSTRIALE E DELL'INFORMAZIONE
CORSO DI LAUREA MAGISTRALE IN AUTOMATION AND CONTROL
ENGINEERING - INGEGNERIA DELL'AUTOMAZIONE



POLITECNICO
MILANO 1863

**HRC-TEAM: A MODEL-DRIVEN APPROACH TO
FORMAL VERIFICATION AND DEPLOYMENT OF
COLLABORATIVE ROBOTIC APPLICATIONS**

RELATORE

PROF. MATTEO GIOVANNI ROSSI

CORRELATORI

ING. MEHRNOOSH ASKARPOUR

ING. NICCOLÒ IANNACCI

ING. FEDERICO VICENTINI

TESI DI LAUREA MAGISTRALE DI
LIVIA LESTINGI, MATR. 836691
SAMUELE LONGONI, MATR. 837500

A.A. 2016/2017

Ringraziamenti

Desidero ringraziare, in primo luogo, i miei genitori per il sostegno costante durante tutti questi anni, per avermi incoraggiata nell'inseguimento dei miei obiettivi, senza mai dubitare delle mie capacità anche quando l'opinione più critica di tutte era la mia. Un grande ringraziamento va anche a Stefano, per il suo supporto durante l'intera esperienza universitaria, per la pazienza inesauribile e per tutto l'amore donatomi durante i periodi di buio e i periodi di luce: grazie di essere rimasto con me *fin proprio alla fine*. Ringrazio di cuore il mio preziosissimo collega Sem, per l'impegno e la dedizione nella congiunta realizzazione di questo progetto, ma, soprattutto, per l'amicizia e l'affetto con cui abbiamo condiviso i mesi di lavoro. Nel ribadire a voi la mia riconoscenza per avermi aiutata a raggiungere questo atteso traguardo, estendo il sentimento a tutti gli amici, vecchi e nuovi, e a chiunque mi sia stato accanto.

Livia

Il primo ringraziamento spetta ai miei genitori, capaci di supportarmi da anni nei miei studi e nella mia crescita personale, un aiuto incondizionato e impareggiabile. A Greta, che più di ogni altro mi ha ascoltato e sostenuto senza tregua, sempre, durante questi mesi di alti e bassi, di gioie e di ansie: a lei va il mio Grazie più importante e profondo. Ho avuto la fortuna di condividere questa Tesi con Livia, che non potrò mai ringraziare abbastanza per essere stata una collega, ma ancora di più un'amica, esemplare ed insostituibile. Non posso poi non essere riconoscente anche a tutte le persone che mi hanno accompagnato in questi anni: dagli amici di lunga (e lunghissima) data ai più recenti, ma non per questo meno determinanti. Infine, dedico questo lavoro ai miei nonni, guida costante ed esempio da imitare ogni giorno.

Samuele

Un ringraziamento corale va al Relatore Prof. Rossi e al Correlatore Ing. Vicentini per la disponibilità, i preziosi consigli dispensatici e per la guida attenta e costruttiva durante la realizzazione e stesura di questo progetto di Tesi. Ringraziamo anche Mehrnoosh e Niccolò per la pazienza dimostrata nel chiarire i nostri dubbi e la proficua collaborazione durante questi mesi di lavoro.

Infine, ringraziamo tutti gli amici e i colleghi di Ing. dell'Automazione e del DEIB, in particolare del Lab. di Ingegneria del Software, per la compagnia e il sostegno morale reciproco durante gli anni di università e per aver contribuito con affetto al raggiungimento di questo traguardo.

Livia e Samuele

Contents

Sommario	xiii
Abstract	xv
1 Introduction	1
1.1 Goal and Motivations	1
1.2 Thesis Outline	5
2 State of the Art	7
2.1 Workflow Modeling Methodologies	7
2.2 Domain-Specific Modeling Languages (DSMLs)	12
2.3 UML for Workflow Modeling and Domain Representation	17
2.4 Discussion Conclusions	19
3 Background	21
3.1 Formal Methods for Safety Verification	21
3.1.1 Risk Assessment	22
3.1.2 TRIO and ZOT	23
3.1.3 SAFER-HRC	26
3.2 Distributed Industrial Applications Design and Deployment	31
3.2.1 International Standard IEC 61499	31
3.2.2 Robot Operating System (ROS)	34
3.2.3 IEC 61499-ROS Hybrid Architecture	36
3.2.4 Software Tools: 4DIAC-IDE and FORTE	39
4 HRC-TEAM Profile	43
4.1 UML Profile Definition	43

4.2	Class Diagram	46
4.2.1	Resources	47
4.2.2	Layout	55
4.3	Component Diagram	56
4.3.1	Agents	56
4.3.2	Layout	58
4.4	Activity Diagram	59
4.4.1	Actions	60
4.4.2	Workflow Elements	61
4.5	Case Study	65
4.5.1	Setting Description	65
4.5.2	Class Diagram Instances	69
4.5.3	Component Diagrams	69
4.5.4	Activity Diagrams	72
5	Formal Verification-oriented Model Transformation	77
5.1	Generation of SAFER-HRC Models	77
5.1.1	Operator, Robot and Layout Models	78
5.1.2	Task Model	79
6	Deployment-oriented Model Transformation	85
6.1	Translation Procedure	85
6.1.1	Configuration Files Generation	86
6.1.2	Application Generation Mechanism	88
7	Tools and Experimental Validations	95
7.1	Software Tools	95
7.1.1	Modeling Tool	95
7.1.2	Transformation Tool	97
7.2	Validation of the Formal Verification Process	99
7.3	Validation of the Deployment Process	106
8	Conclusion and Future Developments	109
8.1	Achievements	109
8.2	Future Work	110

Appendix A HRC-TEAM Models	113
A.1 Class Diagram	113
A.2 Component Diagrams	113
A.3 Activity Diagrams	113
Appendix B SAFER-HRC Models	119
B.1 Java Code	119
B.2 ZOT Input	121
B.3 Parsed ZOT Output	124
Appendix C IEC 61499 Application	127
C.1 Java Code	127
C.2 TaskGen-FB Root C++ Code	129
C.3 Simulation Log	131
Bibliography	132

List of Figures

1.1	Initial Toolchain Scheme	4
2.1	YAWL Task Example	9
2.2	CTT Example	10
2.3	EOFM Task Model Example	11
2.4	SmartTCL Architecture	13
2.5	V ³ CMM Schematic Representation	15
2.6	RobotML Packages	16
3.1	SAFER-HRC procedure scheme	27
3.2	Default Action Internal Finite-State-Machine	29
3.3	Function Block Structure Simplified Scheme	33
3.4	Example of Function Block Internal ECC	33
3.5	ROS Architectural Scheme	35
3.6	IEC 61499-ROS Layered Architecture Scheme	37
3.7	Task-FB Internal Structure	37
3.8	Sample of IEC 61499-ROS Application Branch	38
3.9	Sample of Complete IEC 61499-ROS Application	39
3.10	4DIAC-RTE Interface Screenshot	40
4.1	Class Diagram template	47
4.2	Class Diagram Close-up: Operator class	48
4.3	Class Diagram Close-up: Robot class	51
4.4	Class Diagram Close-up: Type Enumerations	52
4.5	Class Diagram Close-up: End-Effector class	53
4.6	Class Diagram Close-up: Layout class	55
4.7	Component Diagram n.1: Operator for Case A and Case B	57

4.8	Component Diagram n.2: Robotic System for Case A	57
4.9	Component Diagram n.3: Robotic System for Case B	58
4.10	UML Opaque Action Template	60
4.11	Activity Diagram Segment	63
4.12	Set of resources for Case A	66
4.13	Set of resources for Case B	66
4.14	Instance Specification Set n.1: Case A resources	68
4.15	Instance Specification Set n.2: Case B resources	68
4.16	Layout scheme for Case A	70
4.17	Component Diagram n.4: Layout for Case A	70
4.18	Layout scheme for Case B	71
4.19	Component Diagram n.5: Layout for Case B	71
4.20	Case A Informal Activity Diagram	73
4.21	HRC-TEAM Activity Diagram for Case A, V1	73
4.22	Case B Informal Activity Diagram	76
4.23	Partial HRC-TEAM Activity Diagram for Case B	76
5.1	Action to FSM Conversion	80
5.2	Opaque Actions to Formal Model Actions Transformation	81
5.3	Stereotype-related Pre-Condition Example	82
5.4	Workflow-related Pre-Condition Example	83
6.1	HRC-TEAM Diagrams to FB App Transformation Procedure	86
6.2	Action to Task-FB Conversion	86
6.3	Application-Generating Network	90
6.4	TaskGen-FB Structure	91
6.5	Multithread Execution Scheme	93
7.1	Customized Papyrus Palette	96
7.2	Profile Refinement Options	96
7.3	GUI for Formal Model Generation	98
7.4	GUI for Configuration Files Generation	98
7.5	Case A Experiment Hazard n.1	101
7.6	Case A Experiment Hazard n.2	102
7.7	Case A Experiment Hazard n.3	102

7.8	Case B Experiment Hazard n.1	103
7.9	Case B Experiment Hazard n.2	103
7.10	Case B Experiment Hazard n.3	104
7.11	ROS Nodes Feedback Segment	106
8.1	ConverTEAM Final Scheme	110
A.1	Full Class Diagram	114
A.2	Operator Component Diagram	115
A.3	Case A Robotic System Component Diagram	115
A.4	Case B Robotic System Component Diagram	115
A.5	Case A Layout Component Diagram	116
A.6	Case B Layout Component Diagram	116
A.7	Complete Case A Activity Diagram	117
A.8	Complete Case B Activity Diagram	118

List of Tables

2.1	Surveyed languages features	20
3.1	List of derived TRIO Operators	24
4.1	Supported logic connectors	64
5.1	Connectors logic operator counterparts	84
6.1	Connectors Function Block counterparts	87

Listings

6.1	ArrowFile	88
6.2	LogicFile	88
6.3	OpActFile	88
B.1	Portion of Total Action List (TAL) creating function	119
B.2	Portion of Pre-Conditions list creating function	120
B.3	Generated operator body constraints for Case A/O.lisp	121
B.4	Generated robot structure constraints for Case B/R.lisp	122
B.5	Generated layout structure for Case A/L.lisp	122
B.6	Generated layout adjacency constraints for Case B/L.lisp	123
B.7	Generated decision variable constraints for Case A/T.lisp	123
B.8	Examples of action block configuration constants in Case A/T1.lisp	123
B.9	Examples of action block configuration constants in Case B/T1.lisp	124
B.10	Sample of parsed ZOT output for Case A experiment	124
C.1	Portion of Total Edge List (TEL) creating function	127
C.2	Function run by the internal thread to scan configuration files	129
C.3	Function run by the internal thread to process commands	129
C.4	Main TaskGen-FB switch case managing response to input events	130
C.5	ROS resource nodes feedback during task simulation	131

Sommario

Grazie al fenomeno conosciuto come Industria 4.0, la robotica collaborativa sta subendo una notevole diffusione come tecnologia fulcro delle applicazioni industriali. Tale evoluzione, per quanto fondamentale nell'incrementare la flessibilità delle linee di produzione, solleva questioni relative alla sicurezza a causa dello stretto contatto tra umani e robot. Tuttavia, le attuali tecniche per la valutazione del rischio richiedono tempi di realizzazione tendenzialmente lunghi e una preparazione di carattere scientifico che spesso manca agli operatori.

Questa Tesi introduce un approccio basato su modelli astratti e supportato da un tool automatico, il quale, a partire da una notazione grafica intuitiva, genera modelli di logica formale alla base della metodologia SAFER-HRC. I risultati del processo di verifica formale forniscono un quadro delle condizioni di sicurezza della task, in base al quale l'utente può modificare il modello fino a riportare il livello di rischio sotto la soglia desiderata. Lo strumento permette anche la traduzione del modello in un'architettura ibrida basata sullo standard IEC61499 e ROS. L'applicazione di Function Block così generata può essere, quindi, eseguita su risorse fisiche o simulate in ambiente virtuale. Per ottenere questi risultati, all'utente è richiesta la creazione di diagrammi UML, costituenti il profilo HRC-TEAM, tramite il software di modellazione Papyrus. Questo include Class Diagram per la definizione delle risorse disponibili e Component Diagram per una rappresentazione dettagliata delle relative strutture interne. Una volta definito il contesto operativo, gli Activity Diagram sono impiegati per definire il workflow dell'applicazione. Al fine di soddisfare le esigenze di modellazione dovute al contesto specifico, la semantica dell'UML, ove eccessivamente generica, è stata estesa tramite stereotipi da applicare dinamicamente ai modelli in fase di elaborazione.

La procedura illustrata assiste l'utente durante l'intero processo di progettazione, testing ed esecuzione dell'applicazione, consentendo un progressivo affinamento del modello. Inoltre, la natura visiva della notazione e l'alto livello di automazione la rendono accessibile a un bacino di utenti più vasto e con diverso grado di esperienza. Infine, l'approccio è stato testato su casi di studio realistici per valutarne in modo più concreto efficacia e facilità d'uso.

Abstract

Collaborative robotic applications are becoming increasingly widespread as a result of the Industry 4.0 phenomenon. This innovation, albeit instrumental in improving production lines flexibility, raises safety issues due to humans and robots working in close proximity. Current risk assessment methods, though, are either overly time-consuming or require a strong scientific background which practitioners usually lack.

This Thesis introduces a tool-supported model-driven approach facing these issues. The proposal provides an intuitive graphical notation from which formal logic models, at the heart of the SAFER-HRC methodology, are automatically generated. The results of the formal verification process provide insight into the task's overall safety conditions, on the basis of which the user can modify the model in order to bring the estimated risk level below a certain threshold. The tool is also able to translate the model into a hybrid IEC 61499 - ROS architecture. The generated Function Block application can either be deployed on physical or emulated resources. In order to achieve this result, the user is required to create a set of UML diagrams, constituting the custom HRC-TEAM Profile, through the Papyrus modeling tool. This comprises Class Diagrams for the definition of available resources and Component Diagrams for a detailed description of their inner architectures. Once the operational environment is laid out, Activity Diagrams are used to define the application's workflow. Standard UML semantics, often overly generic to meet domain-specific demands, has been extended through stereotypes which need to be dynamically applied to the produced models.

The illustrated toolchain supports users throughout the whole application design, testing and execution process, enabling successive refinements thanks to its automated nature. Moreover, its visual notation and high automation degree make it accessible to a wider public with varying levels of expertise. Finally, the approach has been tested against realistic case studies in order to evaluate its efficiency and assess its usability.

Chapter 1

Introduction

1.1 Goal and Motivations

Since the beginning of industrialization, significant technological leaps, from mechanization to digitalization, have led to the occurrence of shifts which have been ex-post labeled as *industrial revolutions* [1]. Nowadays, factories are undergoing a new profound revision due to the introduction of Internet-related and future-oriented technologies. This is prompting the conversion into *smart* objects, or –more specifically– *smart factories*. For this reason the expression *Industry 4.0* has been coined to encompass the bundle of advancements currently under development.

The triggers behind the fourth industrial revolutions have a social, economic and political connotation. Firstly, manufacturing industry poses increasingly exigent demands that need to be fulfilled. Production processes will require shorter *development periods* in order to favor a higher *innovation rate*. In the same way, lines are pressed to hit new *flexibility* standards and be able to withstand higher *customization* levels and a faster *adaptation pace*. On the other hand, buyers also expect to be able to define the conditions of the trade and be granted *product individualization*, to the extent that the used expression is *batch size one*. A certain deal of attention also starts being given to the matter of *resource efficiency* and *sustainability*, so that improvements from an economic-efficiency point of view may take place without undermining key ecological goals.

The transformation process is concretized by means of a series of approaches. Firstly, *autonomous* manufacturing cells have ever higher levels of *automation* and *mechanization*, which provide *physical aid* and support for the aforementioned *versatility* demands [2]. Another major theme in this context is that of *digitalization* of manufacturing and supporting tools. This allows the creation of a network of de-

vices cooperating for the registration of actor- and sensor-data in support of control and analysis functions. Concurrently, work is underway in the direction of device *miniaturization* to broaden their applications field, especially towards production and logistics.

In contrast to previous shifts, though, this time the core tendency involves active integration of human-workers into *cyber-physical* systems with the intention of highlighting and exploiting their individual skills and talents [3]. It is also possible to predict that workers will have a wider set of responsibilities, ranging from *supervision* to working out *production strategies*, and problems to solve. This contributes to the rise of needs for technological upgrades concerning the collection and representation of data. In particular, in order to be properly assisted with the carrying of his/her monitoring activities, the user will have to be able to rely on easily *comprehensible* visualizations of data and processes and standardized interfaces. A series of innovative technologies is currently being introduced in factories to support these requirements, such as *Virtual Reality* (VR) and *Augmented Reality* (AR).

Another aspect that is currently under deep revision is that of human-robot forms of collaboration. In fact, the most common mode of collaboration so far has been that of using robots as *programmable machines* working in *isolated* cells, often divided from human-frequented areas by physical barriers. The vision is now evolving towards a *teamwork*-like setting [4] in which humans and robots work shoulder-to-shoulder, *sharing* the same workspace and operational tools. In this way, the robot turns into an active *team member* working with the human to pursue the same goal following a previously *agreed-upon* sequence of actions.

First talks about *collaborative robotics* go back to the mid-'90s [5], when such innovative perspective started showing its appeal. Over the years collaborative robots sales have steadily risen, reaching an estimation of about 5.000 units for the year 2015 [6]. Collaboration, indeed, entails a number of well accepted improvements to human operators' working conditions. This is due to the fact that robots can step in to bear activities that would otherwise inflict excessive physical exertion on workers or result in overly dull and repetitive work [7]. Therefore, aside from health-related benefits, this allows the worker to devote his/her time to higher-level duties such as taking decisions about manufacturing processes.

Human-Robot-Collaborative (HRC) applications, due to their very nature, entail operators and robots working in close proximity, hence making the occurrence of physical contacts very likely. For this reason, it is possible to conclude that safety-critical situations, i.e., *hazards*, will never be entirely preventable when collaboration is in the picture. Nevertheless, it is possible to exploit the rigorous foundations of formal methods to run safety assessments of the developed collaborative task. For

this purpose, Askarpour et al. have worked out the SAFER-HRC methodology [8][9] to formally verify the risk level of the application which is going to be implemented. SAFER-HRC is based on the linear temporal logic TRIO notation, which is used to model the operational system's resources and the sequence of actions as sets of logic formulae. These models are fed to an automatic model-checker which simulates the evolution of the system and verifies the satisfiability of user-specified properties. This feature is exploited to apply constraints on the maximum risk level and analyze the resulting execution trace, which may be empty if no traces that carry an unacceptable level of risk exist. Otherwise, the output can be used to visualize critical circumstances that may arise in the current application design and to apply modifications if deemed necessary. On the other hand, producing such formal models might be out of reach for users lacking a strong mathematical background, hence limiting its target audience.

At the same time, developing distributed automation applications imposes additional requirements. As already mentioned, the rising level of automation requires more complex solutions, possessing much higher degrees of *flexibility* and *reconfigurability* [10] to satisfy the aforementioned emerging industrial demands. The IEC 61499 standard [11] takes care of introducing a new application architectural model to cope with these innovative issues. The model is based on the Function Block (FB) unit, which is replicated and assembled in networks to form the final fully-fledged application targeting industrial process measurement and control systems (IPCMC) [12]. This development process envisages three main steps [13]: programming the block's inner code, allocating blocks to actual devices (resources), and mapping the so-constructed application to underlying communication platforms. The proposal by Iannacci et al. [14] extends such methodology so that it fits HRC modeling purposes. This is achieved by implementing a IEC 61499 - ROS (Robot Operating System) hybrid architecture which splits modeling and planning duties into four differently allocated layers. The separation of concerns makes for a simpler and more intuitive programming procedure since the user is only required to model the task workflow by assembling elementary blocks.

This Thesis has been developed in collaboration with CNR-ITIA (Istituto di Tecnologie Industriali e Automazione, Italian National Research Council). The main goal is to build a connection among these two works, in order to achieve a comprehensive modeling toolchain. The produced plan would simultaneously allow verification and deployment of the collaborative task the user needs to carry out. Nevertheless, a *two-way* translation from one formalism to the other cannot be envisaged. As a matter of fact, the FB application requires access to a set of more technical data which the SAFER-HRC models are missing and, viceversa, formal models require insight on resources and environmental features which may be irrelevant for the IEC 61499 standard. In a nutshell, the two pools of data and modeling requirements are

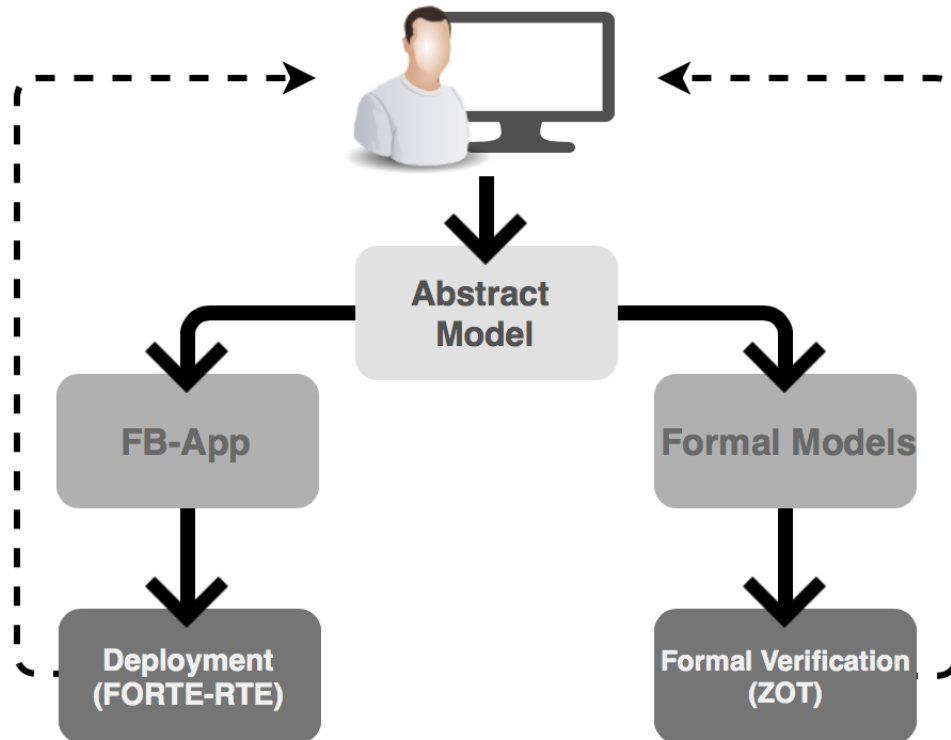


Figure 1.1: Initial toolchain scheme. Each block represents a step of the procedure, starting from the Abstract Model (AM) necessary to model the collaborative task. Such model is then either transformed into a FB application or the formal logic models, respectively deployed using FORTE run-time (see Section 3.2.4) or verified through the model-checker ZOT. The user is then allowed to modify the AM based on previous results and iterate the procedure until deemed necessary.

not identical, hence a direct translation would be unfeasible, whereas extending one of the two environments specifically to solve this issue may lead to efficiency losses. For this reason, the core issue of the project has shifted from the creation of the single translation tool to a complete toolchain including a custom *Abstract Model* (AM). The main goal of the AM is to push the modeling phase to a higher level, in a position fit for covering all the illustrated needs. The produced models compliant with this formalism are then to be translated into the target languages by a custom automated tool, as summed up by the scheme in Figure 1.1.

This work is expected to allow a wide range of users to take advantage of a modeling tool that covers most, if not all, of their needs. The so-resulting collaborative tasks would thus be deployment-ready and more compliant with safety standards. The usability and efficiency of the proposed approach was verified by testing it on real-life case studies. In addition, the application to different examples showed the conformity of the generated models to the ones that were previously manually produced.

1.2 Thesis Outline

The Thesis is structured as follows.

The first important matter under consideration is the selection of the best suited formalism for the Abstract Model. **Chapter 2** comprises a survey of the methodologies already present in literature, together with an evaluation of their strength with respect to our modeling needs. At the end of the chapter, the chosen notation is selected.

Chapter 3 revolves around a detailed presentation of SAFER-HRC and the IEC 61499 - ROS architecture, in order to provide the reader with a comprehensive overview of the project's background and foundations.

Chapter 4 introduces the developed notation, describing in detail each of the implemented elements, discussing their semantics and how this was changed with respect to the standard when necessary in order to fit the specific HRC modeling requirements. Examples are provided of how the model applies to real case studies in order to test its effectiveness.

After the introduction of the *Abstract Model*, it is necessary to examine how it lends itself to the aforementioned translations and how the latter have been put into practice: **Chapter 5** thus presents the translation procedure to automatically obtain the equivalent formal model, whereas **Chapter 6** introduces the FB application generation process starting from the developed models.

In **Chapter 7**, after an overview of the tools involved in the project, the results obtained from the application of the developed approach to case studies are reported to validate the two transformation processes.

Finally, **Chapter 8** draws conclusions about the results obtained with respect to the stated goals. It also presents suggestions about possible future developments to improve and extend the toolchain's potential.

Chapter 2

State of the Art

This chapter presents a review of existing languages and methodologies concerning the creation of an abstract model. Firstly, a collection of workflow-specific languages is analysed in Section 2.1 to highlight the contextually relevant aspects of the matter. Then, the survey is extended in Section 2.2 to more comprehensive modeling languages targeting the robotic domain, and subsequently UML's stance in this scenario is assessed in Section 2.3. Finally, Section 2.4 gathers the conclusions drawn from the analysis.

2.1 Workflow Modeling Methodologies

Over the last twenty years a significant deal of effort has been put in the development of workflow-related languages. The constantly growing demands of industrial production planning have raised the need for an efficient and accessible tool that is also able to encompass all the issues that may come up when dealing with a workflow design. Having access to a language that satisfies such requirements grants remarkable support for the task development process both during the construction and the critical analysis phases.

The definition of the term *workflow* is not unanimously agreed upon, due to the interdisciplinary nature of the matter, but some of them are reported by Georgakopoulos' work [15] about workflow management. For the purpose of industrial robotics applications, the best suited definition seems to be that of a collection of *activities* performed by *actors* involving the manipulation of *objects* (physical or abstract) [15][16], coordinated with each other so that a specific *goal* is achieved by the end of the execution. Each of the key elements, including the synchronization

mechanisms, featured in such definition needs to be precisely identified and tailored to the current task application domain while selecting or crafting the workflow modeling language. In order to reach the goals of the Thesis, it is recommended that the workflow modeling language of choice is able to encompass the following issues:

- Synchronization mechanisms coverage (**SYN**);
- State-based transitions definition (**STB**);
- Action referencing to resources (**RES**);
- Action parameters specification (**PRM**).

There is a vast variety of modeling languages serving this purpose, each of them showing exclusive features and arguably suffering from limitations. Among all of them, the concept of using Petri nets to model workflows is attracting notable attention among researchers [17]. In particular, it is claimed that specific Petri nets extension, such as coloured and timed Petri nets, are well-suited for capturing process and case workflow dimensions [17]. More specifically, based on the standard definition of Petri net as a directed graph consisting of places, transitions and arcs, and referring to a task architecture, an action is represented by a transition, places correspond to pre-conditions or resources and arcs serve as logic relationships regulating the flow [17]. The strength of this approach lies on three main factors [18]. Firstly, the rigorous mathematical foundation of this formalism [17][18], which makes it particularly fit for modeling and verification of models featuring parallelism and synchronization mechanisms. Moreover, the formal and unambiguous nature of this language can also assign it the role of conflict-solving contract among different departments using the same workflow procedure [18]. The second reason resides in the state-based versus event-based infrastructure of Petri nets: according to van der Aalst [18], this allows for a better distinction between different phases of a task deployment, i.e., enablement and execution. Finally, there are a number of already existing and well-established methods for qualitative and quantitative analysis. These respectively aim at verifying correctness properties of the designed net, such as absence of deadlocks and non-conflictive management of shared resources, and the fulfillment of specific requirements, such as the time required to bring the task to completion [17].

Nevertheless, some limitations of using Petri nets emerge when evaluating them based on the Workflow Patterns classifications, as discussed by van der Aalst in [19]. Workflow patterns were originally conceived at the end of the last Century by van der Aalst and ter Hofstede [20][21] in order to precisely identify the points that an all-encompassing workflow modeling language needs to hit and divide them in subcategories. More specifically, Petri nets are weaker when it comes to patterns involving multiple instances, advanced synchronization and cancellation [19]. To make up for Petri nets weaknesses, the same authors have proposed a new modeling language,

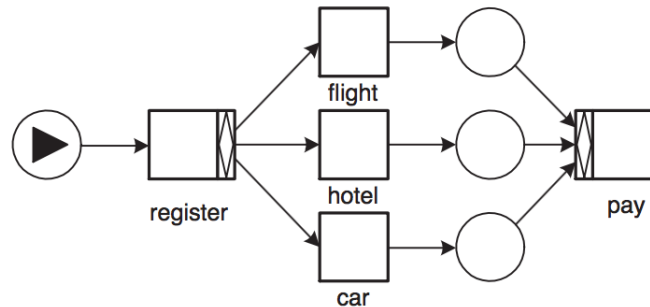


Figure 2.1: Example of Task described through YAWL notation, from [19]. The left-most *play* button corresponds to the initial node, whereas tasks are represented as square blocks (*register*, *pay*, *hotel*, *flight* and *car* in this case). As shown, *register* and *pay* are endowed with connectors (the narrow blocks containing a diamond) since the first one enables the three central ones, whereas the latter can take place only after all three have been completed. The round blocks are needed to collect the three *completion* tokens.

YAWL (Yet Another Workflow Language), which is built upon the same foundation as Petri nets. According to such notation, a workflow specification is composed by tasks, as in Figure 2.1, either simple or composite [19], connected to each other by a variety of operators, which, once combined, allow for a remarkable coverage of the above mentioned control flow patterns. Moreover, a detailed parametrization of the tasks is possible in order to manage, in a compact and efficient fashion, flows featuring multiple instances and dynamic versus static instantiation [19].

The previously provided definition is well embodied by WebWorkFlow, developed by Hemel et al. [16]. This is an object-oriented embedded language extending WebDSL, specific for web applications development [16]. Workflows are formalized through *procedures* definitions, which can either consist of single steps or a combination of procedures, and are further characterized by a series of *clauses*, more specifically *who*, *when*, *view*, *do* and *process* [16]. These respectively specify who is allowed to apply the procedure, the constraints on the ordering of procedures, the user interface, which action is taken when the procedures is applied and the composition of procedures to apply when the containing procedure is invoked [16]. Despite offering some advanced features in terms of dynamic adaptation of the flow, the textual specification is commonly perceived as harder to read and interpret, which is why studies point mostly in the direction of graphical tools [22].

According to Paternò, the crucial steps when first outlining a workflow [22] are the definitions of smaller-scaled elements the task is going to be decomposed into and the temporal relationships in which they are involved. The notation developed by Paternò et al. features a hierarchical tree-like task architecture, as in Figure 2.2, labeled as ConcurTaskTree (CTT). The sub-tasks composing each layer are linked through temporal operators that specify their synchronization mode, as shown in

Figure 2.2. The CTT structure might need refinements during the design phase if found subject to ambiguity. While the temporal unravelment of the task execution steps is exhaustively covered by the CTT notation, the representation of the information about nodes and the objects that they are manipulating is deepened by the work presented by Martinie et al. [23]. In CTT, subroutines are modeled as *actions* manipulating an *object*, which can either be perceivable (such as images or sounds) or internal (i.e., data and their status) [22]. Actions themselves can be differently labeled based on their nature, that is to say whether they are entirely performed by the user or by the system, if they require an interaction among the two or if they are performed at a higher level of abstraction so that they do not fall in any of the previous categories [22].

The work by Martinie et al. extends the CTT architecture by introducing the HAMSTERS notation that includes concepts about knowledge representation inherited from cognitive psychology. According to the latter [23] there are four main types of knowledge the human brain can process and acquire, either through education or through direct experience, which are: declarative knowledge, which deals with the representation of objects and their properties, procedural knowledge, revolving around the way the task is executed in order to reach its goal, and finally situational and strategic knowledge, respectively related to case-based reasoning and planning [23], that is to say taking into consideration and weighting multiple choices. All of the above need to be addressed in some way by the task modeling language of choice, so that all of the concepts which the human brain finds necessary for a complete representation of the task, and not only those falling into one particular category, can be matched by its model and be eventually processed by an artificial intelligence.

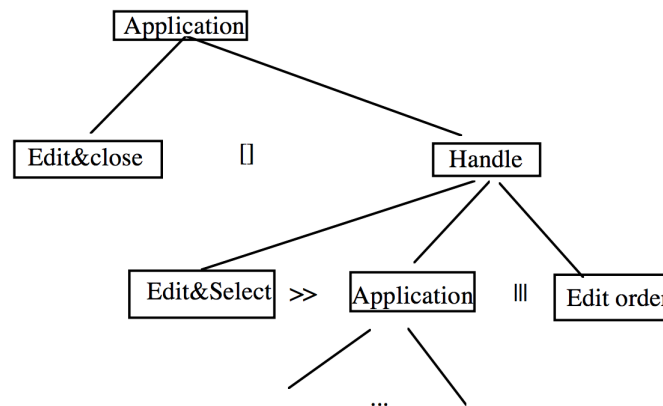


Figure 2.2: CTT task example, as presented in [22]. Sub-tasks are represented as blocks, connected to each other by temporal operators. In this case, *Edit&close* and *Handle* are connected by the *synchronization* operator. *Application* is connected to *Edit&Select* by the *enabling* operator, which means the first one can start when the first one is done, and to *Edit order* by the *interleaving* operator, that is to say the two actions can be performed in any order.

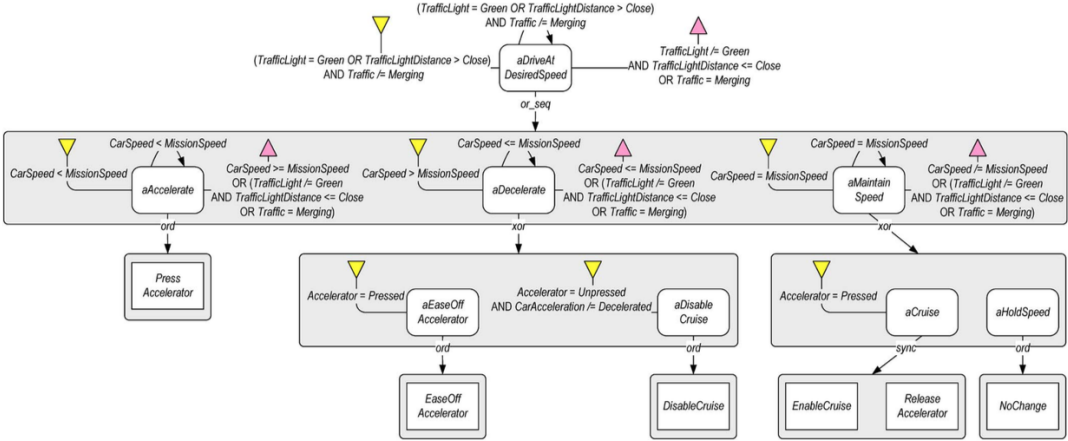


Figure 2.3: Visualization of the EOFM task model for driving at the desired speed, from [24]. Activities are represented as rounded rectangles, actions as unrounded rectangles, preconditions as inverted yellow triangles, and completion conditions as magenta triangles. Boolean expressions in conditions use the syntax supported by transition guards in the SAL.

A significant proposal hinting at operational domain modeling is the EOFM (Enhanced Operator Function Model) language presented by Bolton, Simicenu and Bass [24]. Their work specifically aims at producing a notation, based on a finite-state machine formalism, that models the system behavior and which is at the same time formal-verification ready through an automated model transformation process. The language is centered around the human operator model as an input/output system, able to receive inputs in terms of data or mission goals from interfaces or other operators, and issue outputs in the form of actions. Such actions constitute the basic building block of a EOFM task model, an example of which is shown in Figure 2.3, which, in this case, is meant as the collection of states the system can be in rather than a sequence of activities assembled to achieve a specific goal. The transitions between these states are controlled by conditions, also visible in Figure 2.3, such as *start*, *end* and *repeat* conditions, written as linear temporal logic formulae and featured in the EOFM model through dedicated symbols [24]. The formal verification is performed by translating the EOFM model into a model checking language: for this specific case the language of choice is SAL, which is a framework expressly for the verification of concurrent systems’ properties [24]. The actual translation is carried out automatically by a custom Java script which parses the EOFM’s XML file and converts it into the SAL code [24]. For this context, its main weakness is that most of the core features are already covered by SAFER-HRC, while it does little to fulfill the remaining requirements. As a matter of fact, its operator-centric and –even more importantly– its uni-operator nature does not really fit *collaborative* modeling purposes.

2.2 Domain-Specific Modeling Languages (DSMLs)

When dealing with advanced robotics issues, especially when human cooperation is involved, the sole mission workflow modeling is not sufficient to cover all aspects of an application. It is also necessary to provide the action sequence with a background on the agents involved, their specific skills and the environment they operate in. This modeling step is also essential to provide an automatic tool performing a model transformation procedure with a sufficient amount of information to meet the target framework's requirements. The approach that seems to deal with this issue most efficiently, also according to [25], [26] and [27], is that of *Domain-Specific (Modeling) Languages* (DSMLs) definition. Arguably, in some application fields, such as Artificial Intelligence and Web services, also the concept of *ontology* is of paramount importance. As a matter of fact, such formalization is fundamental for systems design, interoperability and for an unambiguous collaboration among human users or developers [28]. On the other hand, some engineering fields, including robotics, tend to favor the use of DSLs (Domain Specific Languages), which provide a higher level of security and robustness and are better suited to the development of software toolchains [28]. A Domain-Specific Language can be defined as a notation aimed at fulfilling the modeling needs of a restricted domain [29]. The main advantages with adopting a DSL involve ease-of-use, especially when it is formulated as the specification of an already existing general-purpose language (GPL) [25], and a certain degree of flexibility in terms of attainable level of detail and capacity to capture a system from different perspectives [25]. Hence the two main characteristics a good DSL must have, according to [26], are: expressive power sufficiently broad for the target domain and a level of formalism which correctly balances approachability by non-expert users and machine processability. Under these assumptions, among all the existing forms of programming language [30], the natural choice was to survey graphical manual input languages, which are attracting a growing amount of attention among researchers trying to close the flexibility gap with textual languages [30]. With an approach similar to the one used in [26], the surveyed proposals were evaluated based on their capability of abstracting the following real-life concepts, considered relevant for the purposes of the Thesis:

- Robot(s) structure definition (**ROB**)
- Human operator(s) characterization (**HUM**)
- Agent skills definition (**SKL**)
- Operational environment representation (**ENV**)
- Task workflow modeling options (**WKF**)

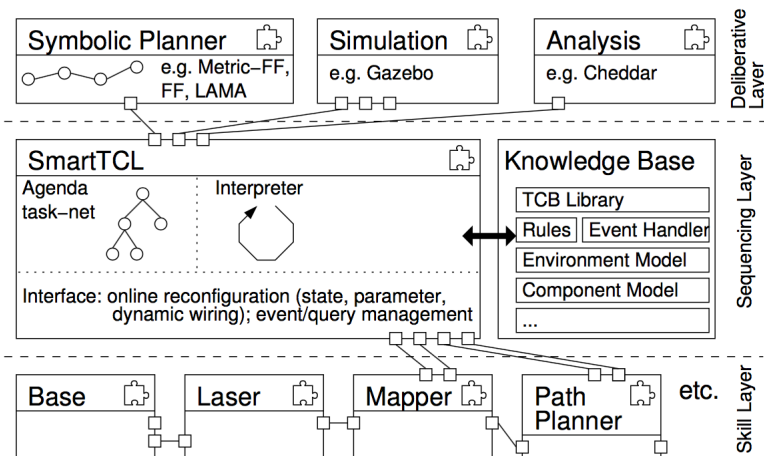


Figure 2.4: SmartTCL three-layer architecture scheme as it is pictured in [31]. The *skill layer* comprises components operating on the level of sensors and actuators. The *sequencing layer* is responsible for the situation-driven task execution. The *deliberative layer* processes deals with symbolic task planning, simulations and system analysis. Each component of the system is represented in the *knowledge base* (KB) with associated information about state, parametrization and constraints.

The languages under examination all share some key features, i.e., they all lean towards the establishment of a toolchain that includes one or more transformations of the main model, but were selected so that a variety of different approaches could be weighted and inspected.

A proposal featuring a degree of complexity appropriate for robotics applications is the one by Schlegel et al. [32] about SmartTCL (Task Coordination Language), which is part of the SmartSoft-MDSD Eclipse-based toolchain. The most important trait about this approach is that it does not only focus on the actual workflow structure for the task development, but it also involves a number of additional components, ranging from planning and analysis tools, to skills and system resources models, ending with simulation tools where possible, all orchestrated by the main sequencer [31]. The main reason behind such architecture, depicted in Figure 2.4 is that the application field requires a broader comprehension of the resources participating to the task and the skills that they possess, besides an insight on the environment they are operating in. This is also related to the concept of task goal achievement appearing in the definition at the beginning of this Section. As a matter of fact, for robotic applications this usually requires further deepening, in terms of additional constraints that must be taken into consideration when evaluating whether the task ended with a success or with a failure. In this particular approach, the selected enhancement is related to variability management [32], that is to say variation points handling thanks to a constraint solver, which uses specific rules to select strategies in response to contingencies and assign them to task blocks [32]. In order to achieve these goals,

the task structure also takes advantage of the aforementioned language architecture, since it often requires access to events from system components or the knowledge base, i.e., to query for a location coordinates or get updates about known objects [32].

The work by Blanc, Delatour and Ziadi exploits the advantages of model-driven engineering (MDE) for the development of software systems for Aibo, which is a *family* of robotic pets developed by Sony [33]. The project is founded on the notion of MDA pyramid, which –in this specific case– is taken into consideration in its four-layered form. The stack is constituted –from top to bottom– by the meta-meta-model, meta-model (such as UML), the model level and the real world [33]. More specifically, three meta-models have been built for Aibo Software development. Firstly, the *Robot* meta-model, aimed at characterizing the single members of the Aibo members, secondly the *Validation* meta-model, that defines which states and transitions are consistent or not, and finally the *Behavior* meta-model, which identifies the software that is going to be built, also ascribable to the programming language category [33]. When the model is complete, it is possible to translate it into code thanks to predefined templates [33]. In particular, the target programming language is the Aibo platform-specific URBI, hence such transformation can be labeled as a model-to-text one, implementing a prearranged set of rules, one for each meta-class of the Behavior meta-model [33].

Similarly, the joint research project BRICS (Best Practice in Robotics), carried on by members of several research groups [34], launched in 2009 and completed in 2013, aims at formalizing a robot development process in its entirety and provide tools to support and accelerate such procedure. Its main objectives, as in similar cases, are interoperability and best practice promotion, and the creation of an integrated development environment, supported by two complementary software packages, BRIDE (BRics Integrated Development Environment) and BROCRE (BRICS Open Code Repository) [34]. The project is structured in seven activities, that is to say seven different research areas. These aim at identifying patterns in robot architecture systems and subsystems for re-usability purposes, and to achieve a *sustainable* [34] software development process also in terms of future developments. They also work on robustness with respect to external unexpected adverse situations, with minimum system flexibility losses [34]. Finally, there are also research branches on the integration of a MDE model into the Eclipse toolchain and the provision of guidelines and suggestion to work in the direction of inter-component harmonized interfaces.

The proposal by Rahman et al. [35] also supports the use of a MDA approach and more specifically argues its strength in terms of software module reusability, which is still a critical issue in the robotic field. Plus, the effectiveness in terms of functional decomposition to suit different development teams' demands is also a desirable asset of the MDA approach [35]. Their proposed meta-model is SysML,

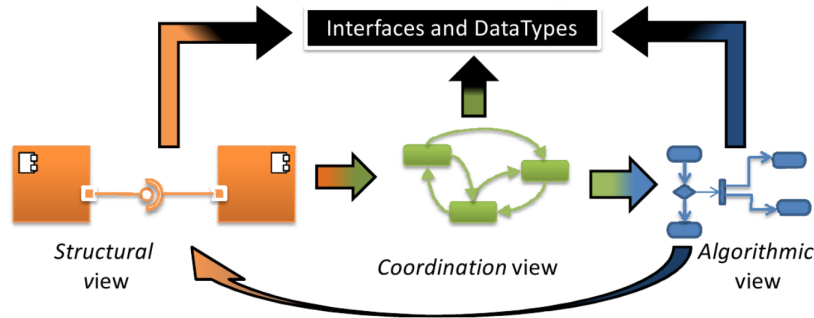


Figure 2.5: Schematic representation of the V^3CMM views, taken from [36], showing the kind of concepts appearing in each view and the loosely coupled relationships existing among them.

which is a general-purpose UML extension, featuring nine different diagrams, some of them modified with respect to their standard UML form, and two new ones. The additional diagrams are the parametric diagram, aimed at representing the mathematical relationships among components of the system that is being designed, and the requirement diagram, which allows the user to model relationships between requirements in various forms and highlight which components satisfy them or not [35]. The proposed modeling process can be split into three main phases. The first one revolves around the actual robotic mission modeling and features context and use case analysis together with functional and non-functional requirement analysis, each of which is covered by a specific type of diagram. Secondly, there is a hardware and software analysis phase, which features functional, structural and behavioral analyses and, finally, a platform-oriented modeling phase with RTC blocks choice and system implementation [35].

The work by Alonso et al. [36] discusses the limitations of current approaches for robot frameworks development. According to them, this is mainly related to the Object-Oriented approach which makes it impossible to verify whether system components are correctly linked and which interaction protocols they will use, whereas components should be modeled as architectural units rather than objects, which are instead essential for the code-generation phase [36]. The core of their proposal is the V^3CMM (3-View Component Meta-Model) modeling language, which adopts concepts from UML but is directly defined using OMG's MOF (Meta-Object Facility), and does not impose any platform-specific requirement [36]. V^3CMM comprises, as the name suggests, three different views, as shown in Figure 2.5, illustrated by three different diagrams. Firstly, the *structural* view, capturing the static structure of simple and complex components, the *coordination* view that describes the event-driven behavior of a component and, finally, the *algorithmic* view which encompasses the functionalities of a component in its current state [36]. They are respectively modeled with representations based on component diagrams, finite-state machines and activ-

ity diagrams. The subsequent model-transformation into code must be preceded by a formal mapping between the generic modeled components and the platform-specific primitives and the completion with application-specific details: more specifically the article covers the transformations from V^3CMM to UML and from UML to Ada [36].

A similar approach is presented in the article by Klotzbücher [37] about BCM, a *minimal* component model for robotic platforms. The term *minimal* is used to highlight the fact that the notation focuses only on elements functional to the model transformation and code generation phases. Such elements can be divided into two classes mirroring the two main design phases this procedure consists of: the component and the system design phases. The first one revolves around the development of a single component, its interaction primitives, platform-independent computations and configurable properties [37]. The second one aims at instantiating, configuring and connecting such components in order to obtain a complete system ready to be deployed [37]. Both phases are established on concepts inherited from UML, in particular *Component Diagrams*, such as components themselves, state-machines, properties and flow ports. The code generation process starts from *semi-opaque* code components which are then parsed and transformed into actual *opaque* code components thanks to a custom-built BCM API and, finally, transformed to the appropriate framework API [37]. Currently, there is a compiler able to apply such steps in order to transform BCM components into working ROS and OROCOS-RTT components [37].

Another attempt at the formulation of a DSL for robotic systems is the PROTEUS project, aimed at supporting the French robotic community’s growth [28]. The ontology defined in this project is able to model tasks and control systems, as

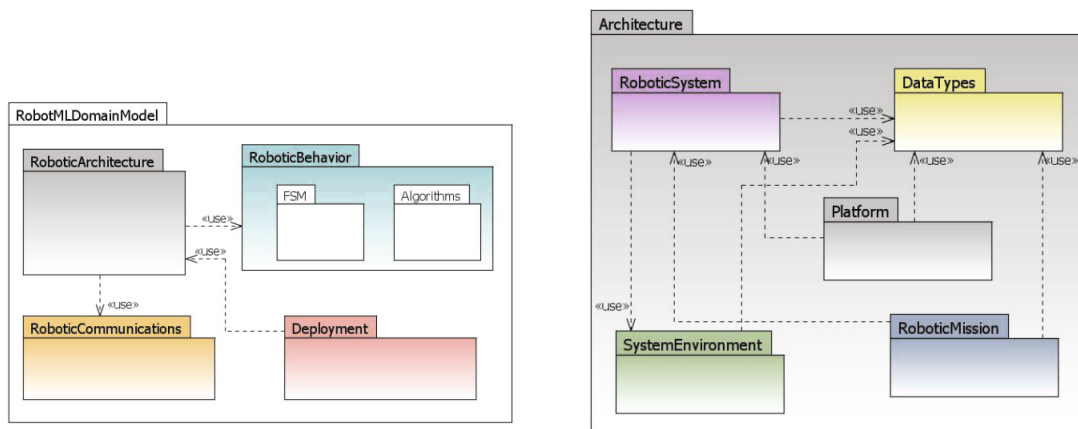


Figure 2.6: As it is illustrated in [27], a general view of the RobotML Domain Model (left) and Architecture (right) packages, with relative sub-packages.

well as component models, mechanical and electrical, and component databases. The resulting tool is simultaneously adequate for testing and validation and linking to hardware for real-life simulation [28]. The quest for all these functionalities justifies why the language allows the user to model both the robot's behavior and skills and the environment it operates in [28].

The PROTEUS project has served as a base for the creation of RobotML, a DSL for robotic applications implemented as a UML profile [27]. The target of this work is to deliver a toolchain that automates the code generation phase, so that the user is not required to deal with lower-level details that call for a stronger background in robotics programming [27]. Their agenda includes eight main requirements, ranging from ease of use to platform interoperability. The Eclipse-based toolchain starts with the scenario modeling, for which the notation contemplates a collection of packages, represented by UML Class Diagrams. The *architecture* package, shown in Figure 2.6, covers concepts regarding the robotic system components, the operational environment components, which data types will be exchanged between components, the operational mission and data about the execution environment, such as robotic middleware and simulators [27]. Additional packages, also featured in Figure 2.6 deal with *communications* details, exploiting the concepts of ports and connectors, and the robotic system's *behavior*, modeled through finite-state machines [27]. In a second phase the user defines a *deployment plan*, whose specific details are also covered by a package, to allocate the model components to the execution platform and eventually reuse already deployed components [27].

2.3 UML for Workflow Modeling and Domain Representation

Within the UML notation, Activity Diagrams are the ones devoted to workflow modeling both for computational and organisational processes [38]. Similarly to the cases illustrated in Section 2.1, in [38] Dumas and ter Hofstede conduct a critical analysis on the suitability of Activity Diagrams as workflow specification languages based on their coverage of advanced constructs from the workflow patterns collection [20][21]. The conclusions they come to is that they present some weaknesses when facing specific kinds of synchronisation such as the *discriminator* or *N-out-of-M* join and do not fully support all *produced-consumer* patterns, but perform particularly well when challenged with *state-based* patterns. The latter are especially relevant when the collaboration between a human operator and a resource is involved and it is necessary to distinguish between different states of an action, i.e., waiting and

processing [38]. Further strong points involve the support of conceptual signal processing and activity decomposition into subactivities [38].

From a more general point of view, they exhibit the same qualities as most UML diagrams in terms of rigorousness and adaptability to numerous settings, also thanks to UML functionalities targeting extensibility such as stereotypes, and have thus been selected, in their standard or extended form, to model workflows for various applications. One example is the C-Wf approach, proposed by Bastos and Ruiz in [39], which consists of an activity diagram extension to improve its usability for production process modeling. The core concept of this proposal is the adaptation of UML standard elements to concepts native to the business field through *stereotyping*. One example of this is the *swimlane* notion which is reinterpreted as the representation of a *domain*, that is to say the organisational unit where an enterprise activity will be executed [39].

A similar approach can be found in the work by Brüning and Gogolla, who aim at enriching activity diagram with dynamic properties [40]. The way this is achieved is by defining execution semantics through OCL invariants. The latter express system states, operations pre and post-conditions, and temporal relationships between existing elements [40]. During the execution, this semantics is interpreted and the flow is modified based on the outcome, i.e., certain branches can become forbidden and hence be deactivated [40]. Such control flow perspective is captured by a meta-model and applied to the case study of a peri surgical emergency process. In such application, another asset of Activity Diagrams which must be exploited and eventually integrated by the meta-model is that of data flow management. As a matter of fact, different types of information must be figured out or accessed during different phases of the process, for example the generalities of the patient during transportation or a database of symptoms when finding a diagnosis [40]. Specularly, the diagram also needs to capture the generation of data during the execution of the process, i.e., the produced documentation regarding medications [40].

As already stated in Section 2.2, it is necessary to capture the system from different perspectives to accomplish the goals established for this thesis, hence it is also imperative to have a comprehensive look at UML as a whole as a modeling language. According to Bruccoleri, La Diega and Perrone [41], the object-oriented approach is the most efficient when it comes to designing complex systems and UML was expressly designed to represent any software system and close the gap between OO design area and OO programming by means of an integrative meta-model [41][42]. Thanks to UML it is possible to specify the notation and the semantics of a model or, more specifically, display its requirements and its use-case realizations plus represent a static structure of the system, its behavior and its physical implementation [41]. The adequacy of UML to model automation and control systems has been specifi-

cally addressed in [43] and [44], both coming to the conclusion that its usage would provide great support in the software development process [44] to control engineers also during the system performance assessment phase, lifting the need for a deep understanding of its inner working [43]. Ritala and Kuikka have further exploited such advantages by defining a UML automation profile [44]. Their work covers all requirements of a control application, starting from the different levels of required technicality, i.e., the local control of a device requires a much lower-level perspective than the high-level supervisory control [44]. Similarly, the proposal profits from the UML profile mechanism to hit other requirements, such as *distribution* of software components, data *exchange* and *simulation*. The result is a language that possesses sufficient expressive power to capture all aspects of an automation application while maintaining an adequate degree of usability. Other examples of how a UML extension can specifically address robotic applications issues [35][28][27] have been illustrated and discussed in Section 2.2.

2.4 Discussion Conclusions

As a conclusion of the discussion carried on so far, Table 2.1 displays how the investigated languages compare to the modeling requirements established in Sections 2.1 and 2.2. As for the workflow-specific languages, the main issue is that the work in this field is mostly focused on improving them so that they can cover as many advanced flow constructs as possible. This is to be expected due to the nature of the problem, but when it comes to collaborative applications the focal point tends to shift from the complexity of the flow, which is clearly still relevant to a certain degree, to the expansion of actions' properties. As illustrated in Section 2.3, a promising path to a resolution of the matter is to consider the adoption of an extended version of UML Activity Diagrams.

Proceeding to DSLs analysis, it is clear that most of the existing languages are predominantly centered around the robotic system's architecture and software components, but lack the capacity to model the human operator(s) and the aspects concerning the collaboration among the two, which is obviously imperative in HRC applications. More in detail they tend to be very proficient in technical aspects, such as lower-level control components, middlewares, and deployment specifics. From a wider perspective, also the toolchains such languages are part of focus on target-platforms requisites and optimizing software reusability and flexibility. All of these qualities are very valuable when developing a procedure aimed at specifying the robot mission and directly interacting with the real-life resource to be deployed, hence requiring low-level technicalities, but lose relevance when it is necessary to

capture more *physical* aspects, such as the positioning of agents inside a layout. As a consequence, trying to employ these languages in a way that falls far beyond their original scope would lead to a heavily under-performing and mediocre result. On the other hand, as discussed in Section 2.3 and also reported in Table 2.1, adapting the semantics of a standard UML element through stereotypes and collecting the latter in a new profile has proved to be an efficient approach to target the modeling needs of a specific domain, which precisely complies with the objectives of this Thesis.

Table 2.1: Surveyed languages features

Language	KWB				SYN	STB	RES	PRM
Petri Nets					✓	✓	✓	
YAWL					✓		✓	
WebWorkFlow					✓	✓	✓	
CTT					✓			
HAMSTERS					✓		✓	
EOFM		✓			✓	✓	✓	
	ROB	HUM	SKL	ENV	WKF			
SmartTCL	✓		✓	✓	✓			
AIBO	✓		✓					
BRICS	✓		✓					
SysML	✓		✓	✓	✓			
V ³ CMM	✓		✓					
BCM	✓		✓					
PROTEUS	✓		✓		✓			
RobotML	✓		✓	✓	✓			
UML Profile	✓	✓	✓	✓	✓	✓	✓	✓

Chapter 3

Background

This Chapter presents the features of the two frameworks that the project of the Thesis aims at connecting. Section 3.1 offers an overview on safety analysis and then it focuses on the TRIO language, the Zot model-checker and the SAFER-HRC methodology that are all exploited by the toolchain to formally verify the designed task. In Section 3.2, the second environment, involving the IEC61499 standard and ROS, is introduced as well as the Function Block element and the related software tools that allow the verified task to be deployed.

3.1 Formal Methods for Safety Verification

Formal verification is a systematic process that mathematically checks whether designed specifications satisfy requirements for implementation. More precisely, it exhaustively explores all possible input values over time: this is the reason why high observability is directly achieved with no need for simulation of multiple scenarios to stimulate the desired design. However, in order to formally verify a project, it must be firstly converted into a simpler *verifiable* mathematical model: it is the approach based on model checking. For this purpose, a linear temporal logic language, TRIO, has been used to describe the properties to be verified. Still, before inspecting the adopted formal verification procedure, it is necessary to introduce the concept of safety analysis and to explain what is meant by risk assessment in human-robot collaborative task in the most recent literature.

3.1.1 Risk Assessment

Control and, possibly, elimination of factors that may cause harm, particularly to people, are issues increasingly considered in the scientific literature related to human-robot collaboration, where close proximity and interactions are unavoidable. In particular, the term *Risk Assessment* identifies a precise procedure that starts with the hazards recognition and the evaluation of their severity and ends with the determination of applicable countermeasures. Rigorous approaches are based on standards, such as the ISO 10218-2 [45] that distinguishes four possible collaborative modes between humans and industrial robots. Our project focuses on Power and Force Limitation (PFL), the one involving actual physical contact which is strictly associated with safety requirements in terms of pressure and force thresholds to limit the effects on the human body. Another standard is the ISO 12100 [46] in which existing hazards and unwanted situations due to misuses or errors of the operator are identified; their consequences are then measured in terms of quantified risk values. As schematically introduced in [9], five stages are the basic structure of a standard iterative risk analysis. First of all, one has to determine the limits of the involved machineries, their regulations and constraints together with the desired task to be accomplished. Secondly, the hazards listed in product-specific standards are identified. Then, for each of them a risk value is measured, usually combining the severity of the harm with its likelihood. Finally, the risks are evaluated understanding their significances and, if one is not negligible, appropriate countermeasures are iteratively introduced to reduce it. When the residual risk value is considered acceptable the refining process is stopped.

[8] underlines the issue of viewing the operator as a proactive factor in safety violations. For this purpose, the behavior of the human can be determined by means of two approaches: the cognitive one involves a formal model for the human cognition, whereas the task-analytic one identifies principles that generate plausible human behaviors and formalized templates from cognitive psychology. However, cognitive approaches are too specific models and they do not address human fallibility and errors; instead, task-analytic approaches focus on all the possible combinations of hazardous situations, regardless of their cognitive reasons, by hierarchical structures of tasks to be then decomposed into smaller functional units. Moreover, traditional formal approaches such as FMEA (Failure Mode and Effects Analysis) and FTA (Fault Tree Analysis) are not well-suited for HRC applications because they do not capture hazards due to human factors, produce false positives and are too dependent on the analyzers team which makes them less generic [47]. Also [48], considering Failure Mode, Effects, and Criticality Analysis (FMECA), states the difficulty of having a comprehensive view of the overall safety achieved by an autonomous robot when it interacts with people. Due to the complexity of human-robot interactions and to the lack of data concerning rate of failures associated with human actions, traditional

risk assessment techniques are inconclusive.

In this project, the approach considered to automatically identify hazards and retrieve adequate countermeasures is based on the SAFER-HRC (Safety Analysis through Formal vERification in HRC applications) methodology. As stated in [9], although it would be impossible to claim that all possible hazards can be discovered, SAFER-HRC provides an exhaustive exploration of the behavior of the target system which should be enough to guarantee that no significant hazardous situations will be left unconsidered. Section 3.1.3 is dedicated to a deeper analysis of this method, even if a brief step-by-step description (as in [8]) is provided here for the sake of completeness of the Risk Assessment review. Firstly, it is necessary to achieve a mutual understanding of the desired task regarding safety requirements, hazards and their treatments, together with a precise definition of the applications that have to guarantee a minimum level of safety. Then, the TRIO metric temporal logic can be used to build a modular model of the task, to which an iterative verification by use of the Zot tool is applied. Finally, a generalization of the model plus an evaluation of the methodology through new tasks should create a proper framework to help safety engineers, both a priori to design systems and, with extensions to runtime, “on the fly” to monitor, detect hazards and introduce suitable risk reduction measures.

3.1.2 TRIO and ZOT

TRIO is a first order logical language that allows the user to express in a precise and formal way temporal properties, which are of the utmost importance for real-time systems. It is also augmented with temporal operators to establish the truth or falsity of propositions at time instants different from the current one, which is left implicit in the formula. On the other hand, it provides limited means to describe the structure of large and complex systems: this is because TRIO specifications are very finely structured whereas the language does not provide powerful abstraction and classification mechanisms and it lacks an intuitive and expressive graphic notation [49]. Regarding the syntax and the semantics of TRIO, a brief introduction is needed to accomplish a complete understanding, whereas detailed and formal definitions may be found in [50]. Like in any first order language, the alphabet of TRIO is composed of variable, function and predicate names, plus a fixed set of quantifiers (\forall and \exists) and primitive or derived propositional connectors such as \neg , \rightarrow , \wedge , \vee or \leftrightarrow .

A necessary analysis of these elements, as it is reported in [49], concerns the time dependence. Indeed, time dependent variables can be distinguished from time independent ones: the former have values that may change with time, the latter present invariant values. The same differentiation can be carried out for formulae and predicates, where independent predicates always express the same relation whereas

a time dependent predicate corresponds to a changeable relation. In order to specify the set of values that may be assumed by a variable, a type or domain has to be declared for it. The Time Domain is the most noteworthy among them: it is a numeric set of instants, equipped with a total order relation plus the usual arithmetic relations and operators, and it represents where a TRIO formula may be evaluated. Another special domain is the Distance Domain, a numeric domain composed of the distances between instants of the Time Domain [51]. Usually, the predicates are assumed to be time independent so that the associated relational operations are applicable to elements of these domains. According to [52], the syntax of TRIO founds the definition of new terms on an inductive methodology: each variable and each n-ary function applied to n terms is defined as a term itself and atomic formulae are adopted as predicate applied to terms of the proper type. More precisely, a formula is inductively defined by 4 clauses:

- Every n-ary predicate, applied to n terms of the appropriate type, is a formula (atomic formula);
- If A and B are formulae, \bar{A} and $A \wedge B$ are formulae;
- If A is a formula and x is a time independent variable, $\forall x \cdot A$ is a formula;
- If A is a formula and t is a term of the temporal type, then $\text{Futr}(A, t)$ and $\text{Past}(A, t)$ (see Table 3.1) are formulae.

Besides the usual propositional operators and the quantifiers, a single basic modal operator, $\text{Dist}(F, t)$, may be used to compose TRIO formulas [53]. It connects the current time, to another time instant: the formula $\text{Dist}(F, t)$, where F is a formula and t a term indicating a time distance, states that F holds at a time instant exactly at t time units from the current one. By means of propositional composition and first order quantification on variables representing time distances, it is possible to derive temporal operators from the basic $\text{Dist}(F, t)$ operator, including all the operators of classical linear temporal logic [51] summed up in Table 3.1.

Table 3.1: List of derived TRIO Operators

TRIO Operator	Definition	Meaning
$\text{Past}(\phi, d)$	$d > 0 \wedge \text{Dist}(\phi, -d)$	ϕ occurred d time units in the past
$\text{Futr}(\phi, d)$	$d > 0 \wedge \text{Dist}(\phi, d)$	ϕ will occur d time units in the future
$\text{futr}(v, d) \sim c$	$d > 0 \wedge \text{Futr}(v \sim c, d)$	$\text{futr}(v, d)$ indicates the value of variable v sometimes d time units in the future
$\text{Alw}(\phi)$	$\forall t(\text{Dist}(\phi, t))$	ϕ always holds
$\text{AlwF}(\phi)$	$\forall t(t > 0 \Rightarrow \text{Dist}(\phi, t))$	ϕ holds always in the future
$\text{AlwP}(\phi)$	$\forall t(t > 0 \Rightarrow \text{Dist}(\phi, -t))$	ϕ holds always in the past

Lasted (ϕ, d)	$\forall t(0 < t \leq d \text{ Dist}(\phi, t))$	ϕ occurs for the past d time units
Since (ϕ, ψ)	$\exists t(\text{Past}(\psi, t) \wedge (\forall t'(0 < t' < d \Rightarrow \text{Dist}(\phi, -t'))))$	ψ occurred in the future and ϕ held since then
Som (ϕ)	$\exists t(\text{Dist}(\phi, t))$	ϕ occurs sometimes
SomF (ϕ)	$\exists t(t > 0 \wedge \text{Dist}(\phi, t))$	ϕ occurs sometimes in the future
SomP (ϕ)	$\exists t(t > 0 \wedge \text{Dist}(\phi, -t))$	ϕ occurs sometimes in the past
Until (ϕ, ψ)	$\exists t(\text{Futr}(\psi, t) \wedge (\forall t'(0 < t' < d) \Rightarrow \text{Dist}(\phi, t')))$	ψ will occur in the future and ϕ will hold until then
Until _w (ϕ, ψ)	$\text{Until}(\phi, \psi) \vee \text{Alw}(\phi)$	weak until: ψ may never occur in the future
WithinF (ϕ, d)	$\exists t(0 < t < d \wedge \text{Dist}(\phi, t))$	ϕ occurs within d time units
WithinP (ϕ, d)	$\exists t(0 < t < d \text{ Dist}(\phi, t))$	ϕ occurs sometimes in the past

Moreover, it is also stated in [51] that the traditional operators of linear temporal logics can be defined as TRIO derived operators. So, since many different logic formalisms can be described as particular cases of TRIO, this argues in favor of its generality.

TRIO formulae have a precise and well defined mathematical meaning in order to favor decidability and computability. Therefore, it grants a solid basis for TRIO executability that allows mechanical and constructive proof of the satisfiability of a TRIO formula (that is, of its consistency as a specification), by means of the generation of a model for it assigning values to variables and relations to predicates [52]. The TRIO formalism can thus become the kernel of a specification environment that provides an abstract characterization of system behavior: an ideal input for programs related to model checking. Indeed, different techniques have been developed in order to furnish a fully automated response, positive or negative, to the satisfiability of the stated property by the designed system. In [53], two kinds of model checking are defined: Automata-theoretic, based on the transformation of temporal logic formulae into automata and Satisfiability-based which follows the converse approach. As a matter of fact, it transforms the operational model S of the system into an equivalent logic formula FS , and then applies logic-based algorithms to a suitable combination of FS and P .

The satisfiability-based model checker delegated to formal verification of the designed project is *Zot*. As explained in [54], *Zot* presents at least two main advantages: it is quite flexible and easily extendible since it is fairly concise and written in Common Lisp; it supports different logic languages and usage modalities by means of plugins¹. *Zot* guarantees these features because of its multilayered approach. Indeed, the core uses Propositional Linear Temporal Logic (for a complete description of PLTL formulae see [55]) and on top of it a decidable predictive fragment of TRIO

¹The adopted plugin *ae2sbvzot* can be downloaded from <https://github.com/fm-polimi/zot> together with the model checker.

is defined. The three available basic usage modalities are the *Bounded satisfiability checking* (BSC), *Bounded model checking* (BMC) and *History checking and completion* (HCC). All of them return a history, a possible execution trace of the specified system. Of course, if the history file is empty it means that it is impossible to satisfy the specification. Even if the user has to set, in the Main file, a bound that is the maximum temporal length of the provided output histories, they may still represent infinite behaviors exploiting the loop selector variables that mark the beginning of the periodic sections. For example, BSC and BMC can be used to check if a property *prop* of the given specification *spec* holds over every periodic behavior with period $\leq k$. In this case, the input file contains $spec \wedge \neg prop$, and, if *prop* indeed holds, then the output history is empty. If this is not the case, the output history is a counterexample, explaining why *prop* does not hold [54]. It is now apparent that the validity of a formula *r* is demonstrated by showing that the opposite is not satisfiable. This operation is computationally intensive, since the model generator tries exhaustively all possible ways of building a structure verifying $\neg r$, and only after all such attempts have failed *r* is declared as valid. Because of how the demonstration is conducted, the prover is also suitable for disproving properties, finding proper counterexamples [52]. This way of combining a TRIO specification together with an automated model checker as Zot, turns out to be useful and effective in the design phase of the human-robot task. As a matter of fact, this toolchain can create a powerful iterative procedure that, based on a trial and error strategy, helps the user with detecting the unattainable conjectures and redesigning the unfeasible constraints.

3.1.3 SAFER-HRC

The Safety Analysis through Formal vERification in HRC applications methodology (SAFER-HRC), as described in [9], provides a technique to identify hazards through the exhaustive exploration of the behavior of the target system. It is precisely focused on operational hazards that are caused by human-robot interactions and violation of safety requirements mentioned in ISO10218 [56]. Due to the impossibility of foreseeing all possible behaviors of the operator, it cannot be claimed that all possible interactions of operator and system are taken into account; nevertheless, an iterative methodology based on formal verification techniques can eventually provide a thorough analysis of all significant ones. At the core of SAFER-HRC lies a safety assessment team (*SATeam*), which includes robotic and formal methods experts with the purpose of studying the limitations of the machinery and the tasks of the target robot, while predicting possible human-robot interactions. They also determine which hazards may occur and evaluate its risk level, based on the list present in ISO 12100 [46]. *SATeam* relies on a formal model of the HRC application to support and systematize these activities: starting from the ideal concept of the task, they build its formal representation through some modular files written in

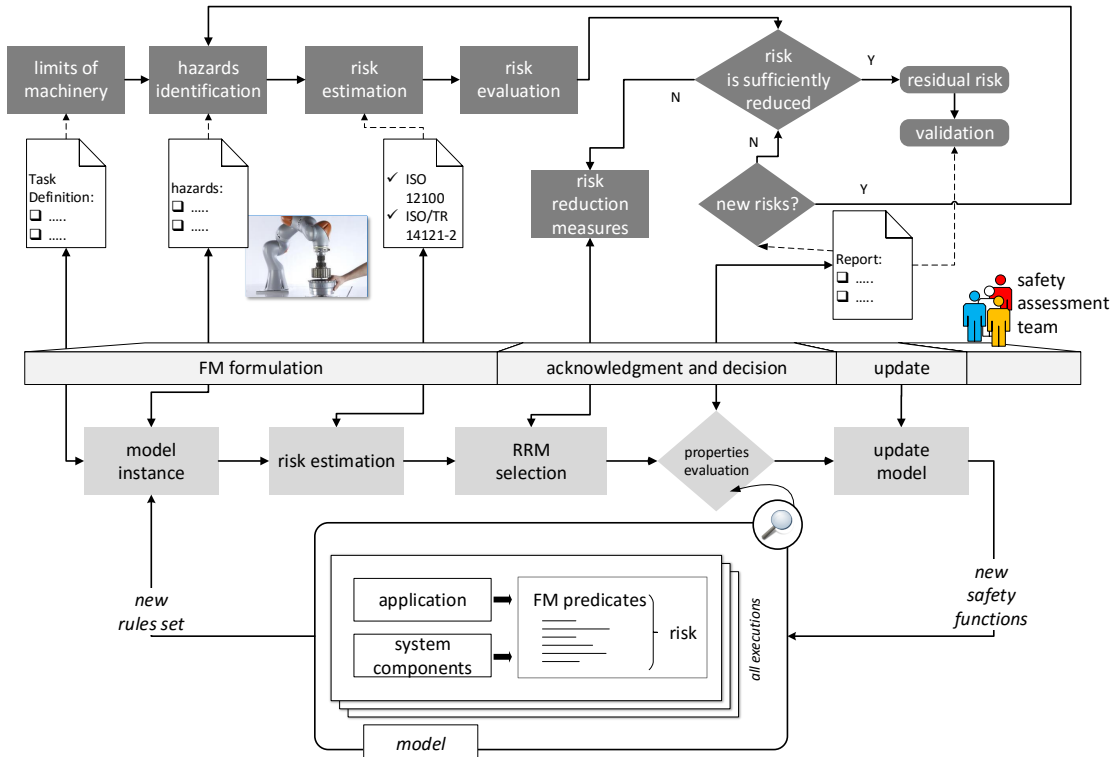


Figure 3.1: Overview of SAFER-HRC. The upper part shows the reference standard risk analysis approach, whilst the scheme on the bottom summarizes the steps of the iterative process based on formal verification defined by SAFER-HRC.

Common Lisp language observing TRIO rules. At each iteration of the process, if the design fails to satisfy the desired safety requirements, it is possible to improve it by activating risk reduction measures or redesigning the task or some of its constraints. The essential aspect is the systematic validation of the constraints and their possible violations at all steps of the application. The thoroughness of the validation ensures that the selected safety strategy is failsafe. SAFER-HRC starts from informal, goal-oriented descriptions of collaborative tasks, and converts them into formal models built upon logical formulae, on which formal verification techniques are applied to check whether the safety requirements are satisfied or not. After the original model has been thoroughly analyzed, it can be modified and re-used to study different scenarios for the HRC application (e.g., combinations of different safety functions or uncommon actions by the operator). The whole procedure is summed up by Figure 3.1.

As stated in [8], the formal model is composed of different files, which are divided in folders: *ORL-Module* includes formal descriptions for operator \mathcal{O} , robot \mathcal{R} and layout \mathcal{L} ; *TaskLib* contains a definition of the task; the main folder, with an arbitrary name, groups the previous two folders and other files that will be introduced in a mo-

ment. With the *ORL-Module*, the dynamics of the system are captured in terms of relationships among the three agents and their descriptions. In particular, \mathcal{L} is used to set rules for the division of the layout into fine-grained regions, their adjacencies and movement boundaries: in this way, movements and positions of operator and robot are expressible; besides, \mathcal{O} includes the list of the considered operator's body parts and their mutual constraints to guarantee a realistic arrangement of the operator in space. Moreover, the forbidden sections which the operator can not occupy due to obstacles or design restrictions are also indexed. \mathcal{R} describes the composition of the chosen robotic arm and how its links are allowed to move: the first link is confined to the *Home* section or an adjacent one, and the other links follow this format recursively. The SAFER-HRC procedure acknowledges a double nature for elements and constraints present in *ORL-Module*: some are common to all HRC applications and therefore are always in the files (the description of the human body parts), whereas others are instantiated depending on the specific HRC application (the features of the robot and the predicates linked to the selected end-effector).

As far as *TaskLib* is concerned, a file \mathcal{T} is compiled to model few task and action parameters. Indeed, the possible agents involved in the activity and the available states of each actions (described in the next paragraph) are listed. Secondly, another file is inserted in the folder, $\mathcal{T}1$, to achieve a complete description of the task. The first part of the file is standard and contains the formalisation of every action state and applicable transition from state to state. Then, before the initialization of each action subject, some properties are defined to guarantee a reasonable model: the robot is forced to remain still if it is not the agent of any executing action, the operator can concurrently perform at most two actions. Finally, a compartment for each operation is generated to enclose specific logic expressions that will be explained in the next paragraph.

The remaining files are contained by the main folder together with the previous two modules. *Main* incorporates the time span for the simulation, the assumptions to set all the actions as not-started at the beginning and a condition to declare the task completely executed. *Hazards* includes the parameters related to severity, risk, location and consequences of dangers that may happen during the execution of the prescribed activity. Two types of hazard are considered, impact and entanglement, together with three macro-areas for the human body, head-shoulders, waist-part and hand-arms. Moreover, the origin can be discriminated depending on three parts of the robot, link1, link2 and end-effector so that five couples type-origin (end-effector can not entangle anything). Thus, 15 hazards are described with proper formulae in the file in order to be identified and reported during the simulation. *REs* stands for Risk Estimator and consists of rules to determine the severity and the risk of each hazard depending on information related to the method reported in [57]. This file is integrated with *REv* that, according to the kind of hazard, defines the countermea-

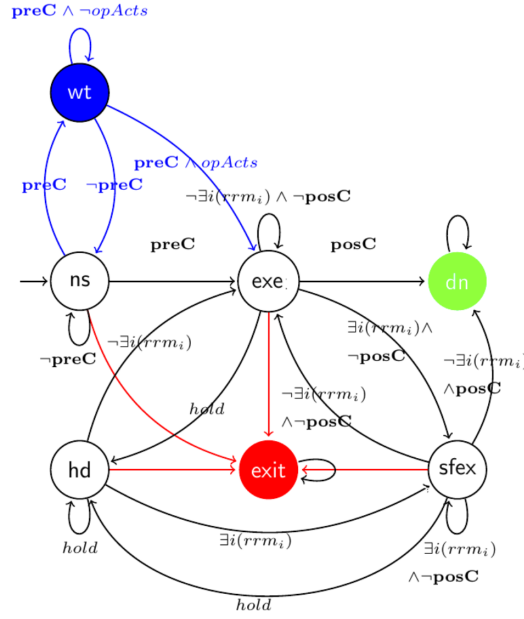


Figure 3.2: Finite-state-machine regulating the inner behavior of all action blocks, as it is presented in [8]. All actions start in the *ns* state and the evolution ends when either the *dn* or *exit* states are reached. Transitions are regulated by guard conditions (*preC* and *posC*), predicates (e.g., *opActs*) and risk-reduction measures activation (see, for instance, $\exists i(rrm_i)$ from *exe* to *sfex*).

asures to be adopted. These solutions are described in *RRM*, stating what concrete risk reduction measure has to be applied during the task execution. In conclusion, at each iteration of the experiment, the full formal model is checked against the properties and, if an hazards occurs, its risk value is computed: if it is not less than an acceptable fixed threshold, then *Zot* tries to apply the associated countermeasure. If the risk value remains non-negligible, then the verification is considered failed and, by means of the output of the tool, it is possible to retrieve the problematic states and to refine the model accordingly.

The smallest possible functional units [9] are the actions that constitute the task description presented in the *TaskLib* paragraph. Breaking down the model to these basic elements grants correct relationships among \mathcal{O} , \mathcal{R} , \mathcal{L} agents and a more precise identification of hazards. SAFER-HRC characterizes each of the elementary actions by three main features, formalized as TRIO formulae: its pre-conditions, post-conditions, and safety properties [9]. Examples of compliant code segments can be found in Listings B.8 and B.9. In order to clearly understand the difference between these terms, it is necessary to analyze how a single action is modeled. According to [8], it can be imagined as a finite state machine, pictured in Figure 3.2, composing the internal structure and giving a precise guideline for the evolution of the represented operation. Indeed, at any given time, each action can be in one of

the following states:

- ns: the action is not started yet;
- wt: the action is not started yet and, being defined only for the operator, it tries to mimic his hesitation to start the execution;
- exe: the action is under execution;
- sfex: the action is under execution and a risk reduction measure is simultaneously active;
- hd: at some point in the execution, the action is paused and temporally stopped;
- dn: the execution is terminated and the action is done.

Now that the single action model has been clarified based on [9], the previously introduced terms can be finally illustrated. Pre-conditions are the requirements that must be true in order to move the action state from *nsto* to *exe* or to *wt*, if it is admissible by the nature of the agent. Post-conditions are the logic expressions that trigger the end of the execution, moving the token of the final state machine into the *dnstate*. Finally, the safety properties are the element to be evaluated during the execution itself, that may cause the action to be paused (and resumed) or to be definitively stopped. Each action has also a property called *priority*, which defines its execution preference over other actions. More precisely, if at a time instant the pre-conditions of multiple actions are satisfied, the one with higher *priority* starts to execute. The current SAFER-HRC version considers that systems operate at their maximum level of parallelism: i.e., all actions that have the highest priority among those that are enabled start executing in parallel.

To retrieve consistent results from the formal verification procedure, the safety experts need to tailor the *ORL-Module* to the desired *HRC* task. Among more technical requirements, it is necessary to set the correct numbers of agents (how many operators and robots) and all the sections in which the layout is divided, to compile the R file accordingly to the appropriate robot type and model, to exclude in the \mathcal{L} file the agents from being in their forbidden sections. Having completed the set-up phase, the procedure can be completed by launching the execution of the Zot tool: the formal verification is able to check whether the model satisfies the desired safety requirements and to find any errors that can cause hazards or any possible incompatibilities between layout and task execution. If the model checker cannot retrieve a feasible execution of the task, a counterexample is produced to highlight the presence of one or more safety property violations in the system. Then, the designer should improve the system model: by adding proper risk reduction measures, which

correspond to TRIO formulae that should avoid the violation; or by including new formulae to capture hazards that were undetected in the previous analysis. Next, a new validation is carried out on the improved model. The model is refined iteratively until no more violations occur. At each iteration, if the design fails to satisfy the desired safety requirements, it is improved by adding new risk reduction measures.

3.2 Distributed Industrial Applications Design and Deployment

After the formal verification has been accomplished and the task is designed in a satisfactory way, it is possible to create a version of the application executable in a real setting with concrete agents. This feature exploits an architecture compliant with the IEC 61499 standard, introduced in Section 3.2.1, and based on the Function Block unit, whose structure and characteristics are extensively explained in Section 3.2.1. Finally, Section 3.2.3 presents the architecture of the FB application which is adopted as framework of the executable task, whereas the supporting tools are displayed in Section 3.2.4.

3.2.1 International Standard IEC 61499

The standard IEC 61499 *Function Blocks for Industrial Process Measurement and Control Systems* [58] specifies an architectural model that serves as a pivotal reference for distributed, modular [59], and flexible applications in industrial processes and control systems. In particular, the standard is dedicated to frameworks that meet the following requirements:

- Portability: software tools ability to accept and interpret correctly library elements produced by other software tools;
- Configurability: the ability of devices and their software components to be configured (selected, assigned locations, interconnected and parameterized) by multiple tools;
- Interoperability: the ability of devices from different vendors operating together to perform the functions specified by one or more distributed application.

The most important concepts of IEC 61499 are an event-driven execution model, the key mechanism enabling transparent modelling of distributed systems, a management interface capable of basic reconfiguration support and an application-centered

modeling methodology. There are also some parts of the standard that are weak and/or not defined in enough detail for specific implementations [60]. The standard provides in principle an ideal starting base for the control architecture and the reference model for distributed industrial process, measurement and control systems. A process represents an independent computational activity with its own set of variables (context) and communication channels with other processes via messages. The event interface is well suited to model message-based inter-process communication [61].

This result is achieved by the use of the Function Block, a basic unit for industrial applications to define robust and re-usable software components [58], which will be analyzed in detail in Section 3.2.1. The architecture of the standard IEC 61499 supports unlimited nesting of composite function block structures, and combination of several diagram types: block-diagrams, state charts, and ladder logic in the same design. The result of the design is a duality of the function block language construct: it guarantees an executable specification of distributed automation systems, including at the same time also models of devices and their network interconnections. On the other hand, a function block may still represent just a piece of code executed within another process. However, the process-like encapsulation mechanism provides strength to this architecture, enabling arbitrary re-allocations of components to distributed execution domains without affecting their functionality. Moreover, strong data encapsulation into components is another provision for portability: not only it has been widely recognized by the software community as one of the pillars of creating safe and re-usable code, but it can also ensure the absence of hidden dependencies between variables of several FBs [61].

Function Block Concept

In object-oriented terms, the Function Block is a class defining the behavior of (possibly) multiple instances: its schematic structure is shown in Figure 3.3. It includes input and output events to guarantee the synchronization in program execution in distributed systems [62]. They can store the software solution for various problems and they have a defined set of input and output parameters, which can be used to connect them to form complete applications [58].

In the IEC 61499 context an application, also called FB network, is a collection of interconnected function blocks [63]: they are connected by event and data flows and can be distributed over multiple resources and devices. The single function block consists of head and body, where the head is connected to the event flow, and the body to the data flow. Its functionality is provided by means of algorithms [64]. An algorithm is a finite set of ordered statements that operate over the ECC variables. Typically, an algorithm consists of loops, branching and update statements, which

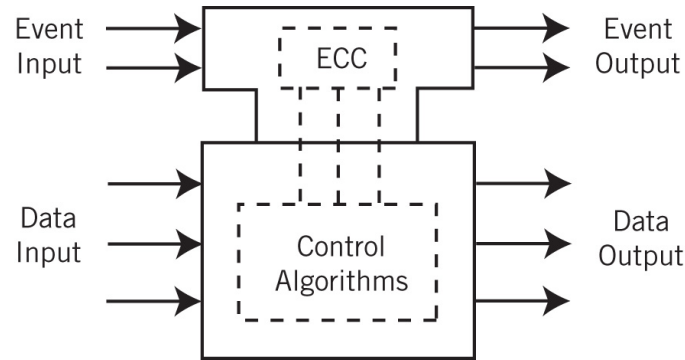


Figure 3.3: Schematic representation of a FB. The Head block wraps the inner ECC (an example is found in Figure 3.4), and collects input and output events. The Body contains the Control Algorithms and stores input and output parameter values.

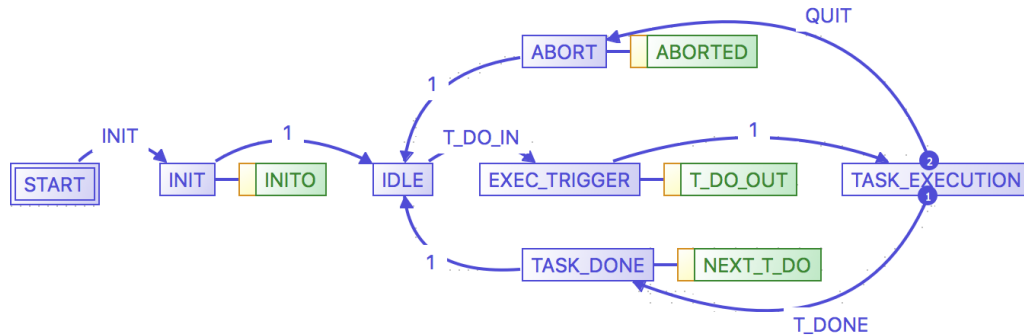


Figure 3.4: Moore-type FSM regulating the inner behavior of a FB. The initial state (START) is featured as a double-line border purple block, whereas all other blocks represent states the FB can be in. Green flags represent output events which are issued when such state is entered. Purple arrows represent transitions among states, labeled with the triggering condition: usually it corresponds to the reception of an input event, e.g., T_DONE between states TASK_EXECUTION and TASK_DONE.

are used to consume inputs and generate outputs. The IEC 61499 standard allows algorithms to be specified in a variety of implementation-dependent languages, such as Structured Text (ST), Java and C [65]. Function blocks of IEC 61499 are event-driven, i.e., they remain idle unless an event is sent to one of their event inputs. The main motivation behind this choice was the desire to make the code independent from the sequence of FB invocations in the PLC scan loop [61].

Different types of FBs are used for modelling tasks and applications [14], the most important of which is the Basic function block (BFB). The behavior of a BFB is expressed as a Moore-type state machine, known as Execution Control Chart (ECC), an example of which is shown in Figure 3.4. The reaction to an event is determined by the evaluation of this ECC together with the possible invocation of algorithms [66]. The execution of an ECC starts from its initial state and progresses by taking

transitions, which are guarded by an input event and an optional Boolean expression over input and/or internal variables. Upon evaluation, a transition is considered to be enabled if the respective guard condition evaluates true. The ECC will then transit to the next state by taking the enabled transition from the source state to the corresponding target state [65]. It turns out that ECC states can be of two types: those where the ECC can stop and wait for incoming input events (let us call them sensitive) and transitional, which are just passed during an execution. Besides, the standard does not provide sufficient information on how to treat event-input variables. However, this ostensible lack of attention is precisely explained by the concept of event-driven invocation of FBs: there is no need to consider event-input variables as real variables since they are used only once [66].

Another relevant category is constituted by the service interface function blocks (SIFB), in which the source code is hidden and the functionality is only described by service sequences. They are useful especially for specific operations that cannot be accomplished by BFBs (e.g., communication). In addition, a third group gathers the composite function blocks (CFB), whose functionality is defined by a function block network. The standard allows it to encapsulate a network of FBs so as to achieve more complex behavior and functionalities by combining multiple components. Finally, adapter interfaces manage several events and data connections within one connection. Thus, they allow the creation of sockets/plugs in CFB interface, simplifying and reducing the number of drawn connections.

3.2.2 Robot Operating System (ROS)

ROS² is a *meta-operating system* providing common functionalities such as hardware abstraction and low-level control [67]. It can be viewed as a collection of *software frameworks* for robot-oriented software development, similarly to other projects like OROCOS, YARP and Microsoft Robotics Studio. Unlike most robotics software platforms, ROS is not a real-time OS but a distributed framework of *processes*. More specifically, these are called *Nodes*, representing executables individually designed and eventually coupled at run-time, which can be grouped into *Packages* or *Stacks* and shared. Through this policy, ROS highly promotes code *reuse* in robotics research and development. ROS runs on Unix-based platforms and can be implemented in Python, C++ and Common Lisp with additional libraries currently under development.

ROS has three levels of concepts (*Filesystem*, *Computation Graph* and *Community*) and two types of names: *Package Resource* and *Graph Resource* Names. The

²Documentation at <http://wiki.ros.org>

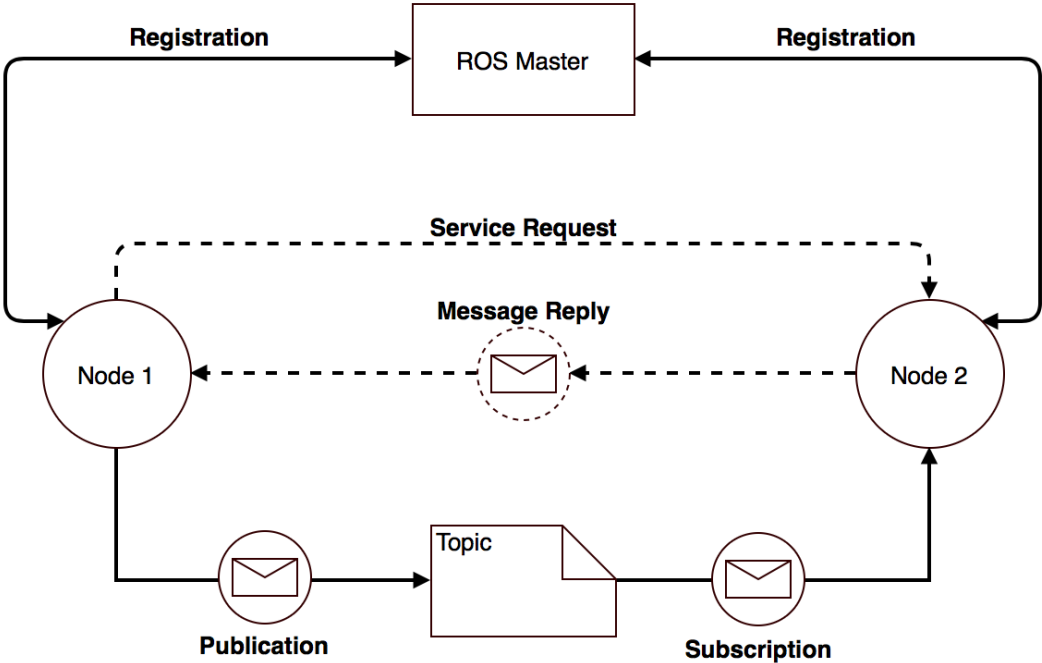


Figure 3.5: Scheme representing ROS architecture. The two nodes can interact with the ROS Master for *registration* purposes. In order to exchange messages they can directly communicate with each other. The publish/subscribe mechanism works as a *message* streaming over a shared bus, defined as *topic*. Services, instead, must be directly requested and lead to the emission of a *message reply*.

Filesystem Level mainly includes ways to arrange related files on disk. The most relevant one are *Packages*, which is the atomic building unit of ROS containing Nodes, libraries, datasets and configuration files, and type-describing files. The latter can be referred to *messages* (*.msg*), i.e., data structures of messages exchanged in ROS, and *service* (*.srv*) files which describe request and response structures of services. The Community level includes resources which enable software and knowledge exchange, such as *repositories* and *distributions*.

The Computation Graph level represents the peer-to-peer network of ROS cooperating processes. The main concept in this case is that of ROS *Node*. Each Node represents an active process and is connected to other units in order to constitute a larger-scale system [67]: for instance, in a single control system, there may be multiple nodes representing localization, path planning and graphical visualization. All nodes have a graph resource name that uniquely identifies them and a *node type*, that is to say a package resource name that simplifies the act of referring to a specific node in the system. The use of this structure provides several benefits especially in terms of *fault tolerance*, since crashes affect single nodes and not the entire system, and *code complexity* reduction. Nodes interact with each other through *topics*, *services* and the *Parameter Server*. The latter handles data storage by key and is part of

the *Master*, i.e., the entity in charge of name registration and graph scouting. Topics are identifiable buses over which nodes exchange *messages* through anonymous publish/subscribe semantics. A node that generated data can *publish* a message to a topic, whereas a node interested in collecting data can *subscribe* to such topic, usually unaware of each other's existence [67]. On the other hand, nodes requiring remote procedure calls, i.e., receiving response to a request, rather than a unidirectional communication streaming, should use *services* instead. A service is defined by a *request/reply* pair of messages, and is mostly used in distributed systems [67]. A service structure is defined by a *.srv* file, a message one by *.msg* files, and both can include standard primitive types such as strings, boolean and floats.

The ROS Master, as already mentioned, works as a nameservice and stores information about topics and services registration. A node can interact with the Master to report registration information and make inquiries about other nodes in order to create connections. The Master can also perform a callback if such information changes. On the other hand, nodes directly interact with each other, e.g., nodes willing to subscribe to a topic will request connections to the publisher and establish communication over an agreed upon protocol. This architecture, summed up by Figure 3.5 allows for operation decoupling, so that nodes can be started, killed and restarted without affecting the system's integrity. *Names* play a major role when building a system, whose complexity can be smoothly increased through dynamic *remapping*.

3.2.3 IEC 61499-ROS Hybrid Architecture

The illustrated traits of IEC 61499 applications perfectly fit current robotics demands. As a matter of fact, *adaptability* requires the representation of flow alternatives which is highly eased by the implementation of re-usable logic units, i.e., FBs. Iannacci et al. have proposed in [14] a joint IEC 61499-ROS architecture specifically aimed at modeling and deploying collaborative robotic applications. The approach exploits usability and readability of standard FB interfaces, which are handily customizable and provides a comprehensible visual workflow representation of the application that is going to be deployed.

The core of the proposal is the layered architecture structure which is composed by levels, as shown in Figure 3.6, with different purposes:

- *Planning* level, which collects factory-level requirements
- *Scheduling* level, aimed at choosing the proper flow alternative based on the current plant state

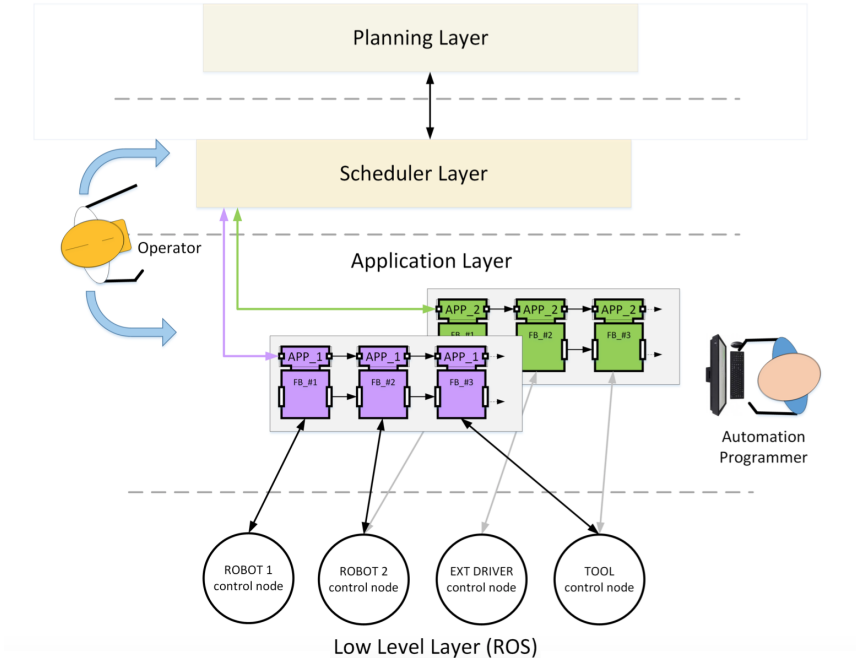


Figure 3.6: Layered control architecture as pictured in [14]: the scheduler launches all the applications which share resources controlled at low-level; operators (left) are able to act on scheduling and build applications by combining FBs; programmer (right) is in charge of preparing the building blocks for later assembly.

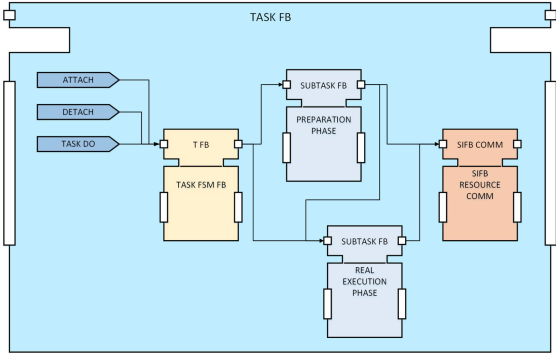


Figure 3.7: Internal structure of the CFB Task-FB, as presented in [14]. The Task FSM FB (whose ECC can be found in Figure 3.4) receives the three input events. The two Subtask FBs are in charge of preparation and execution phases. Finally, the SIFB deals with communication with system resources.

- *Application* level, which contains the instantiated clusters of FBs
- *Low* level, dealing with machine-level behavior, hence closed-loop control, realized through ROS nodes

The designed basic unit is the *Task-FB*, which corresponds to an elementary action in common workflow-specific terms. Task-FBs are combined to each other

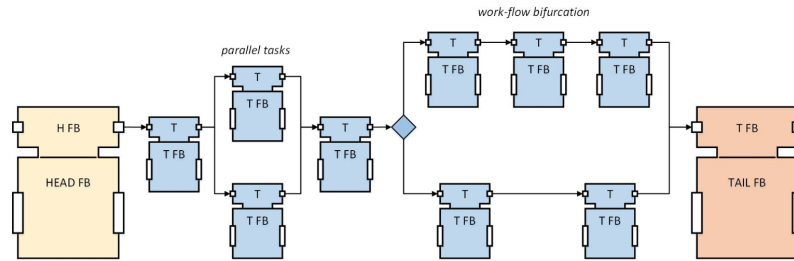


Figure 3.8: [14] furnishes an example of application branch: resource allocation and de-allocation processes are handled by the HEAD and TAIL FBs. The actual action blocks are arranged as to realize more complex structures, such as parallel execution (left) and flow bifurcation (right).

also implementing conditional blocks acting like join and fork nodes to constitute the alternative sequences, hence visually resembling activity diagrams: an example of this kind of network can be found in Figure 3.8. More specifically, it is a CFB encompassing a set of homogeneous actions large enough to be self-consistent but not to the point where it compromises modularity and re-usability [14]. It is subdivided into two separate phases: preparation, which deals with control strategy and settings, and execution. Its internal structure, pictured in Figure 3.7, features various components in charge of fulfilling the stated objectives. Firstly, the *Task-FSM FB* models the internal task finite state behavior, adhering to the notation illustrated in Section 3.2.1, and handles the execution flow. It also contains two *sub-tasks* in charge of setting parameter values for the two aforementioned phases and a SIFB that implements the communication between task and resource abstractions. More specifically, the Task-FB is in charge of establishing an allocation link with the resources, whereas the resource can initiate an allocation/deallocation link from a Task-FB. The activation of a scheduled Task-FB is achieved through the propagation of an *attach* command, modeled as an input event, whereas the actual execution is triggered by the *TASK-DO* event.

According to [68], a fully comprehensive control application must encompass the following sets of components:

- control and automation components
- machine and process interface components
- communication interface components

which are all mapped to different elements of the joint architecture.

The top level is produced by combining the scheduler block, applications and resource abstractions, as in Figure 3.9. The latter consist of blocks capturing the state evolution of resources present in the system. This is needed by the *scheduler*

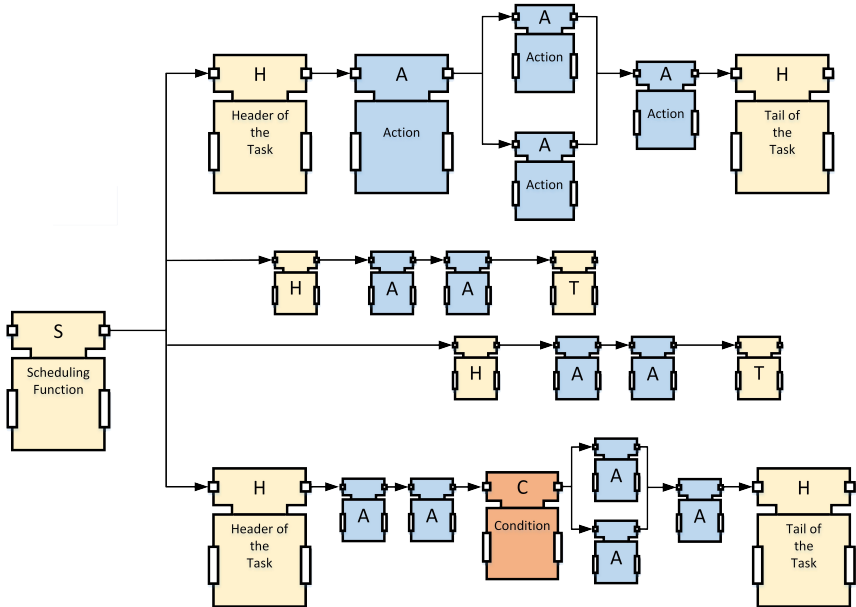


Figure 3.9: Example of complete application, featuring multiple branches managed by the *Scheduler* FB (S).

in order to avoid conflicts and allocate a task only when all resources are actually available. The scheduler itself works as a bridge between the task and the external cell, collecting inputs coming from the environment in order to take planning decisions. Finally, the overall application must also contain HEAD and TAIL FBs. The first is in charge of collecting and transmitting information on resource availability, whereas the second ensures that the allocation has been properly performed.

The lower level is entirely entrusted to ROS nodes, two for each device: one manages the physical control and one the communication with IEC 61499. These nodes are purely *reactive*: they remain in *idle-state* unless an instruction is received. Finally, communication is equally split between IEC 61499 and ROS and realized through instances of publishing/subscribing mechanisms. The framework is exploited by Resource FBs to receive messages from the net of tasks and ROS nodes during the allocation process, and by ROS bridge nodes to collect and transmit data to lower level controllers.

3.2.4 Software Tools: 4DIAC-IDE and FORTE

The spread of the standard has inspired the creation of different supporting tools. The usual implementation toolset includes a workbench for editing function block designs and translating them into executable form, plus a runtime environment that

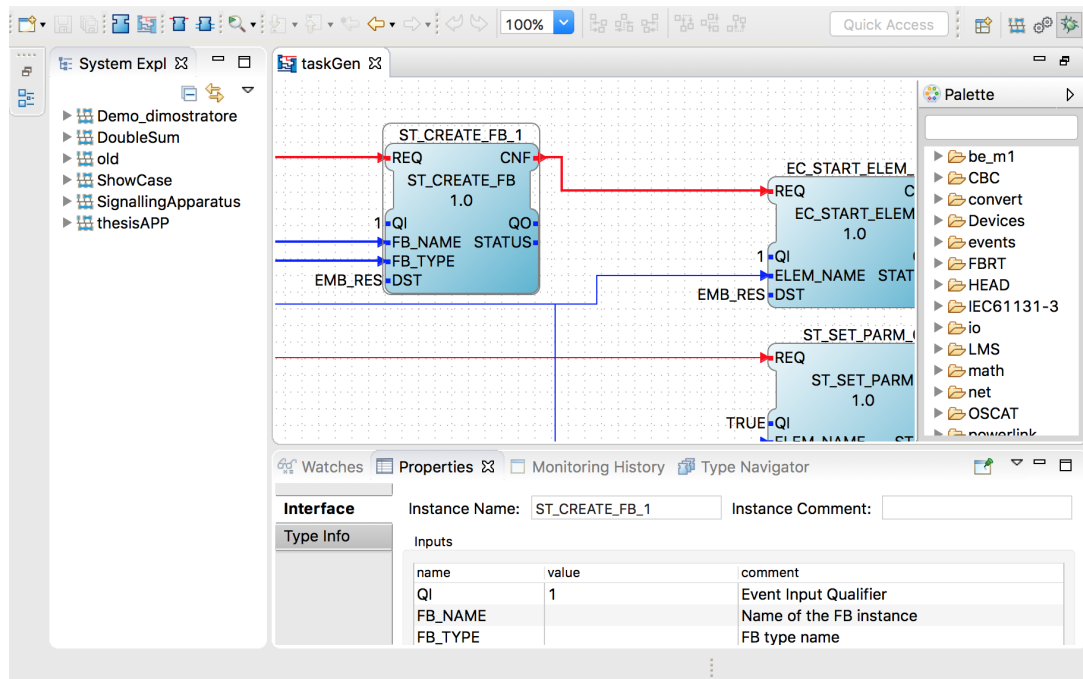


Figure 3.10: 4DIAC-RTE Interface Screenshot. The middle window shows the application currently under construction. The windows on the right and on the left act as model explorer, hence allowing the navigation among developed applications and the choice of FBs out of their libraries. The bottom box displays properties for the selected item.

supports the execution of the executable code. As far as it concerns the present case study, the designated IEC 61499 standard compliant framework to manage the software tools is 4DIAC³, an open-source infrastructure for distributed industrial process measurement and control systems developed by Eclipse [61].

4DIAC-IDE is the dedicated research-oriented workbench that provides an extensible engineering environment for modeling distributed control application, as pictured in Figure 3.10. A hardware capability definition allows the modeling of control hardware and its interconnections through networks [60]. Moreover, in [69] it is established that the tool supports the specification of function block types as well as the development of system configurations including the application model and the device configurations, as well as deployment of the application to distributed devices. It also supports the debugging and testing of distributed control applications via online display, setting and forcing of remote data.

In the same framework, the runtime environment is 4DIAC-RTE, also called FORTE. As illustrated in [60], it is a small portable C++ implementation targeting small embedded control devices (i.e., controllers with 16/32 Bit architecture) that

³The IDE suite can be downloaded from <https://eclipse.org/4diac/>

manages the execution of function-blocks: as a matter of fact, the modeled FB applications can be downloaded to and uploaded from FORTE powered control devices. The execution mechanisms in FORTE allow the real-time constrained execution of IEC 61499 control configurations triggered by external events, where different parts of the configuration can fulfill different real-time constraints and the execution of low priority processes does not disturb the execution of higher priority processes. One of the main advantages is the platform-independency [69], so that is straightforward to target different hardware and operating system platforms.

Chapter 4

HRC-TEAM Profile

This chapter introduces the HRC-TEAM Profile in detail, highlighting innovative aspects and how it fits the needs of the specific domain. First, the selected extension mechanism is illustrated in Section 4.1, pointing out the motivations behind this choice. Each implemented type of diagram is individually presented in Sections 4.2, 4.3 and 4.4, providing the steps required to create abstract models. For each case a great care is given to how the provided semantics differs from the UML standard and how it fits concepts relevant to collaborative task modeling. Finally, real case studies are provided in Section 4.5 to show how the profile performs when tested against a complete application.

4.1 UML Profile Definition

The core of any modeling language is based on three fundamental concepts [70]:

- abstract syntax;
- concrete syntax;
- semantics.

In this context, the expression *abstract syntax* is equivalent to *meta-model*. The latter, as the name suggests, is a higher-level representation of the current model, or better of its core infrastructure. Which elements the model under development should contain and their mutual associations are all information described by the meta-model. *Concrete syntax* refers to the graphical representation of the language, how diagrams should be drawn and what they should look like. Finally, *semantics* encompasses the meaning of each of these elements. More precisely, it indicates

how each featured item and its connections should be interpreted when translating a model into a different language.

UML was initially created as a potential family of languages with an application domain spanning the whole software engineering area [71]. As such it offered a series of extension mechanisms, which were later enhanced in 2005 when its 2.0 version was officially released. These techniques demonstrate their relevance when extending or changing UML vocabulary is brought to the table. This tends to be the case while developing a Domain-Specific Modeling Language, which generally calls for a customization of semantics otherwise overly generic [70].

There are four *standard* applicable extension mechanisms [71]: specifications, common divisions, adornments and extensibility mechanisms. Only the latter actually involve the modification of the semantics, and will thus be more thoroughly analyzed. The term *specifications* in UML approximately matches its equivalent in natural language. Specifying an element means enriching it so that its meaning and behavior are clearer to its user, e.g., a class can be specified by adding a full set of attributes and operations. *Common divisions* are mainly split into two sets to distinguish items that might otherwise be easily mistaken for each other: abstraction vs. manifestation (corresponding to the dichotomy between class and object) and interface vs. implementation, that is to say a contract and a concrete realization of such agreement. *Adornments* are graphical or textual attributes which can be attached to an element to provide more details about its nature, e.g., the multiplicity of an association or generic notes, which -again- have no impact on semantics. *Extensibility mechanisms* actually involve the creation of new blocks and properties with custom semantics that *bend* the language to tailor it to the required domain of application. The standard includes three extensibility mechanisms [25][71]:

- tagged values;
- constraints;
- stereotypes.

The concept of *tagged values* is fairly similar to that of *attributes* but with a fundamental difference: the latter refer to properties and associated values belonging to instances, whereas tagged values provide a *keyword-value* pair for model elements themselves, hence essentially behaving as *meta-data*. They are particularly useful for code generation or configuration management processes, for example to specify the target programming language or the release team of a project [71]. Their graphical syntax is a string enclosed by curly brackets below the model element name.

Constraints, as the term clearly suggests, represent conditions that must hold true for all the involved elements. The chosen language for these expressions is *Object*

Constraint Language (OCL), a formal language specifically intended for queries and constraints implementing a notation that does not necessarily require a strong scientific background [72]. OCL can be used to declare invariants for classes or stereotypes, pre- and post-conditions for operations and methods, constraints, guard-conditions and derivation rules. Such expressions are constituted by four main elements: context, namely the application framework of the condition, property, as the attribute involved in the constraint, an operation manipulating the property and keywords, such as *if*, *and*, *or*, and so forth. This mechanism is helpful when refining the model with additional rules, e.g., concerning budgets or deadlines.

Finally, the word *stereotype* originates from Ancient Greek terms *στερεος* (fixed, rigid) and *τυπος* (class, type), and is usually intended as the impulsive classification of an object, a person, or a situation based on common knowledge rather than a circumstantial thorough assessment. Nonetheless, as a UML extension mechanism, it virtually has the opposite effect, as it extends the semantics of an object instead of oversimplifying it. Stereotyping an element implies modifications of its behavior, meaning and appearance, and allows it to turn into the depiction of a concept native to the specific modeling domain, which does not necessarily match the original UML standards. More formally, in UML 2.0 stereotypes are comparable to special *meta-classes* that allow the creation of new constructs, coherently with the hierarchical meta-modeling architecture established as part of the Meta Object Facility specification (MOF) [70].

For the purposes of this thesis, the implementation of stereotypes was crucial to incorporate concepts inherited from HRC tasks verification- and deployment-oriented descriptions into the UML diagrams infrastructure, whose level of refinement was a convenience to take advantage of, using a well-established and formal technique relying as little as possible on natural language. Furthermore, another valuable asset was the possibility to endow stereotyped elements with custom properties, and set their values while specifying instances, which comes in very handy when dealing with components sporting a broad set of parameters and attributes. Ultimately, such *labeling* system is also essential when converting the task model into its logic formulae equivalent or the executable application since it works as a *decision variable* when picking the appropriate target form.

As argued also in [25], it is possible to collect these conceived stereotypes and standard non-stereotyped elements in a custom UML *profile*. A profile can be defined as a specialized *viewpoint* that can be dynamically applied and un-applied to a model to cast it under a specific perspective. The procedure to define a well-formed profile requires two fundamental steps: the definition of the domain meta-model and its mapping to to the UML framework. The content of the former, in terms of constructs, relationships, syntax and semantics has already been illustrated. As for

the latter, it is necessary to go through the defined set of concepts and find suitable matches in the UML standard. This operation needs to be carried out paying a significant amount of attention to contradictions and conflicts that may arise between one's custom concepts and standard attributes of the selected counterparts. The observance of these requirements leads to a proper and precise notation for the domain of choice. In this specific case, it has led to the creation of the HRC-TEAM (*Human-Robot Collaborative Task Execution- and safety Assessment-oriented Model*) profile, which will be introduced in detail in the following sections.

4.2 Class Diagram

The primary requirement for a robotic application modeling environment, as already mentioned in Section 2.2, is the possibility to render the *scenario* where the task will be executed, intended as the *static snapshot* of the operational environment. Such perspective needs to include elements which are functional to the industrial domain, covering the area as extensively as possible without stretching to a level of abstraction that is out-of-scope. On the other hand the granularity of such characterization must be maintained sufficiently coarse so that it is still practical for users without a strong engineering background. This implies that lower-level and more technical concepts, such as controllers, trajectory planners, sensors and so forth, are not featured by this work, but are covered by other well-established proposals existing in literature (see section 2). The included elements will be individually examined in the following sections.

The main goals of this viewpoint are to serve as reference when drafting tasks and as benchmark for consistency checks. As a matter of fact, having an overview of the *resources* that are currently available, their skills, their requirements and of the *space* where they will be working provides valuable support to plan tasks more realistically and more efficiently. Symmetrically, these models can be automatically scanned and compared to the static depiction to detect trivial modeling mistakes, such as typos, or more severe ones, such as agent-skill mismatches. When formal safety assessment and hazard detection play a major role in the toolchain and are among the fundamental goals, having a stronger foundation and additional verification measures also make for more trustworthy and accurate results.

Having described the subject in abstract terms, it is necessary to select the diagram that best responds to these demands. UML diagrams can be classified in two main categories [71]: *structure* and *behavior* diagrams. In this specific case, the first

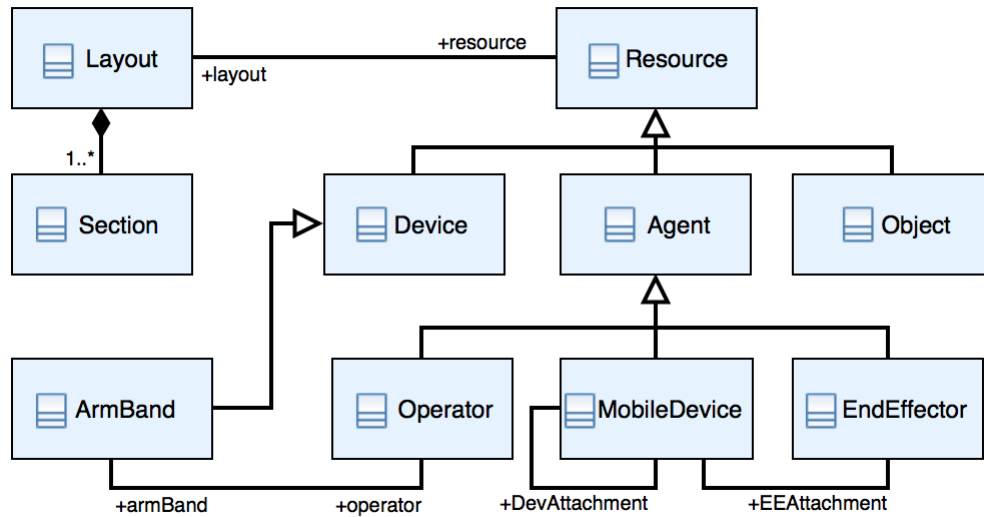


Figure 4.1: Simplified illustration of the notation-specific *Class Diagram*. It features the two main super-classes *Resource* and *Layout*, and the corresponding specifications. The figure also contains the associations connecting such classes.

ones are clearly the best-suited candidates for representing the static structure of a system [73]. The standard type of diagram that serves this purpose is the *Class Diagram*. The approach features a single *Class Diagram* representing the *layout* and the *resources* that can populate it, which fulfills the objectives previously illustrated. A simplified template of this representation is portrayed in Figure 4.1. The user intervention on this diagram should be limited to the creation of instance specifications corresponding to what is actually available in real life, setting specific values for their properties and optionally establishing associations. Further modifications to actual classes may be conceivable but not accessible to users lacking a good programming background, since they would require code writing to be supported by transformation tools and only after an accurate quest for potential conflicts between existing features and the ones that one is willing to include.

4.2.1 Resources

As hinted above, the two main *super-classes* featured in the model represent *resources* and *layouts*, connected by a "*populate*" association, as shown by Figure 4.1. The layout is the selected space representation method, which will be more deeply described in its related section. Resources are intended as components functional to the task goal attainment, further categorized based on their modality of participation. More specifically, they are divided into objects, devices and agents, as in Figure

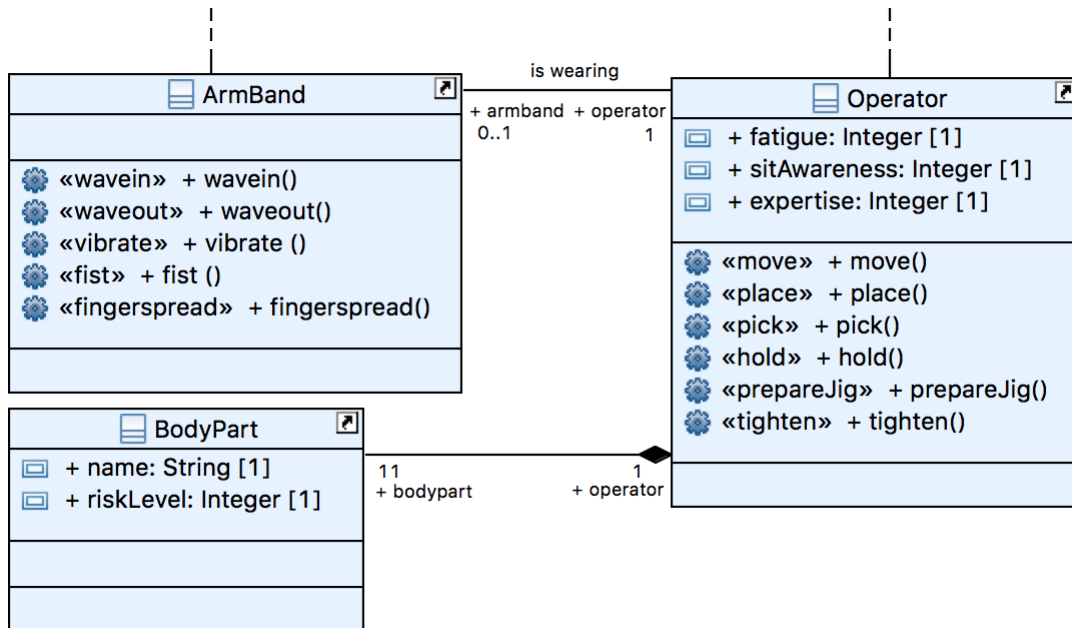


Figure 4.2: Fragment of the *Class Diagram* capturing the Operator and ArmBand classes. The Operator class includes attributes concerning his/her mental and physical state, and methods corresponding to the available skills. Methods owned by the ArmBand class correspond to the types of signal that it is able to send or receive. The figure also features the aggregation relationship between Operator and BodyPart.

4.1. The latter term is used as a synonym of *actor*, as in something or someone who plays an *active* role in the pursuit of the objectives. As such it is possible to further discern into human *Operators* and *robotic* agents (*Mobile Devices*), as also pictured in Figure 4.1, which will be individually analyzed in detail in the following sections.

Human agents

According to the current trends in manufacturing industry, the role of operators in collaborative applications should be centered around activities requiring creativity, imagination, or a high level of meticulousness, e.g., very precise movements involving small objects, which would require a notable investment of resources if assigned to a robot, whereas the latter should handle strength or speed demanding operations [74]. The exact scenario the research is aiming for is that of *teamwork-like* collaboration between humans and machines, where robots are in charge of easing the operators' workloads [74]. Using the term *collaborative* in a broader sense, the operator can also be appointed to a *supervisory* role, which includes tasks such as planning, instructing, monitoring and eventually intervening. The two cases may bring up different characteristics of the human agent that need to be captured and taken into account. In their current state these are modeled as *discretized* attributes

of the class *Operator*, captured by Figure 4.2, which is clearly a basic approximation, but may be enhanced as a future development through the addition of SysML *parametric diagrams* [35] implementing -where possible- analytical models established by engineering psychology researchers.

The first factor to take into consideration for collaborative tasks is the operator's *expertise*, which mainly depends on his/her years of experience, learning rate and skill mastery level. These traits affect the quality standard of the final product, which mostly has economic implications, and the time required to bring an operation to completion, which may instead compromise the synchronization with parallel robot's actions and give rise to hazards. Another critical feature is the *fatigue* assessment [75], which has similar implications related to the operator's *responsiveness*, but is influenced by shorter-term issues. These include physical and mental fatigue, which are, in turn, affected by workload, duty time, periodic rest duration and eventual shifts impact on sleep quality [75], as well as reduced motivation. The latter is linked to the previously discussed matter of reducing the operator's strain which must not go as far as making the task un-challenging to a degree that undermines workers' motives [76].

As for *supervisory* tasks, one of the main issues is the level of *trust* in automation [76]. In natural language the term indicates *reliance* on someone or something, whereas in automation a slightly more precise connotation would be that of *calibration* of the intervention threshold. In fact, the human should take action only when his/her decisions have a higher chance of producing a positive effect than the machine's ones. On the other hand this strictly depends on how accurate the mutual mental models of each other and the environment are [76][77], i.e., if the operator sees the robot as a *team member* rather than a standalone machine he/she is more likely to properly calibrate the value of his/her own decisions with respect to automated solutions [76]. As a final remark, another point considered crucial by engineering psychology experts [76][77] is that of *situation awareness*, especially in human-in-the-loop situations involving multiple simultaneous tasks and goals. According to Endsley [78], situation awareness involves perception of data and environmental elements, comprehension of the current situation and projection of future events, and is a key factor to an efficient and accident-free task execution. Furthermore it may also be affected by the *change blindness* phenomenon, as in the inability to perceive variations in one's surroundings, i.e., when *task switching* occurs without adequate warning cues.

Having presented the selected attributes for the class *Operator*, it is equally indispensable to discuss its methods, or *operations*, which represent the range of skills that can be taken into account when planning an application. Clearly the staple set of operations has been put together with the purpose of covering as many different

situations as possible, since, as already discussed, extending such set should not be within the grasp of standard users. The whole set can also be viewed in Figure 4.2. The featured skill stereotypes are:

- << *move* >>, capturing the displacement of the agent in space;
- << *pick* >>, intended as the combination of -assumably- small arm and hands movements required to grasp an object, whereas the displacement to the target section is covered by << *move* >> actions;
- << *place* >>, which has the same connotation as << *pick* >>;
- << *prepareJig* >>, standing for the setup of a station which will be required by a subsequent operation;
- << *tighten* >>, appropriate when the operator is holding a screwdriver tool and is required to take action with it;
- << *hold* >>, which can either refer to the act of *keeping something still* or *waiting* for a specific occurrence, i.e., in supervisory tasks.

Note that the notation for some of this cases has been chosen to fit better with the specific case studies which will be later examined, i.e., for *prepareJig* and *tighten*. Nevertheless, these could be used under a wider interpretation as *setup* and *useTool*, intended as a generic action performed through some gear.

Finally, Figure 4.2 also displays an aggregation between the class *Operator* and its *BodyParts*. This is a standard feature of the corresponding \mathcal{O} module in the formal model, which manages the positioning of eleven body parts in space. As a future development, support could be added to be able to customize this aspect through the *Class Diagram*, hence modifying the multiplicity of the aggregation and creating as many *BodyPart* instances as desired. This could be useful to adapt the granularity of this trait to the specific case study requirements, since it may be useful to refine the subdivision or make it more simplistic and otherwise invest the available computational power.

Robotic agents

It is equally essential for the *Class Diagram* to capture robots' attributes and skills coherently with the approach taken with human behavior. In this case, though, a further remark is necessary about the criterion for distinction between two agent classes. In fact, the logic that has been selected is to define a separate class for each component powered by a different type of controller and actuation system. Following this rationale it is necessary to further differentiate into *Mobile Devices*, for instance

the robotic manipulator, and *End Effectors*, as in the functional devices mounted on the terminal area of the arm [79], as shown in Figure 4.3.

In more detail, mobile devices are mainly in charge of displacements, the specifics of which are determined by their *architecture*. The latter is declared as an attribute which can take a value out of a list of possible configurations, defined as an *enumeration*, visible in Figure 4.4 and featured as *type* attribute in Figure 4.3. The list contains the most common structure types for industrial applications, such as *anthropomorphic*, *SCARA*, *cartesian* and *dual arm*. The structure can be more deeply detailed by means of the *DoF* (Degree of Freedom) attribute, useful for cases in which it is necessary to declare a standard architecture with a tweaked number of joints, e.g., a conveyor can be instantiated as a cartesian with only one degree of freedom. Furthermore the class Mobile Device is also featured as an aggregation of *joints* (see Figure 4.3), whose type is also defined as an *enumeration* attribute (see Figure 4.4), including *prismatic*, *rotational* and *spheric*. Thanks to the last property it is possible to instantiate a completely customized architecture (see Section 4.3 for details).

The class MD is also endowed with other properties which strengthen the characterization of such devices [80], all displayed as attributes in Figure 4.3. First, it is possible to set the *rated payload* value, as in the maximum weight the device can carry, considering both the object and the end-effector. Violating such constraint gives rise to performance decay and most importantly to security issues, since it

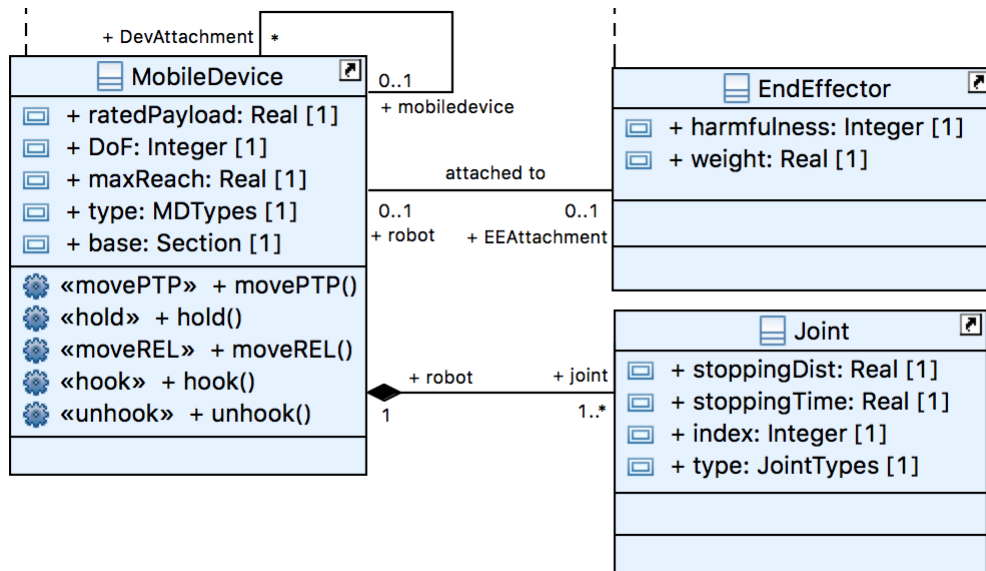


Figure 4.3: Fragment of the *Class Diagram* capturing the Robot and EndEffector classes. Attributes concerning the architecture are also depicted, as well as the aggregation relationship with the Joint class.

may endanger the integrity of the equipment, of the object under concern and also the human operators working in the same layout due to possible impacts. Finally, modeling the device operational workspace coherently with the layout representation is also essential to accurately simulate the robot's movements. This is achieved by means of two attributes: the *base section* and the *maximum reach*, expressed in *meters*, which should be compared to the average section size to determine how many of them the robot can cover at most.

The skills defined for such devices are:

- `<< movePTP >>`, intended as relocation to a precise destination point in the workspace;
- `<< moveREL >>`, which consists of a displacement of a fixed distance along a certain direction;
- `<< hook >>` (and conversely `<< unhook >>`), necessary for tool-changing sequences;
- `<< hold >>`, comparably to the operator's case, corresponding to the act of keeping something in place.

Similarly also end-effectors can be of different types, e.g., grippers and screw-drivers. For this specific case, though, creating an enumeration was not the most efficient solution, since this property does not only involve the structure of the component but it also affects the operations that it can or cannot perform. Hence these were modeled as standalone specification classes, shown in Figure 4.5, each with their own set of methods. As for the attributes of the super-class, they involve the *weight*, whose relevance has already been illustrated, and an estimation of its *harmfulness* level, which impacts the severity of collisions with an operator's body part, i.e., a pointy end-effector may cause more damage than others. The included operations are:

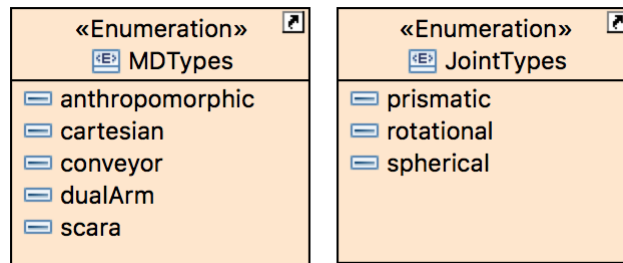


Figure 4.4: Fragment of the *Class Diagram* capturing the *Mobile Device* (MDTypes) and *Joint* (JointTypes) types enumerations: these are selectable when picking the corresponding attributes values for instance specifications.

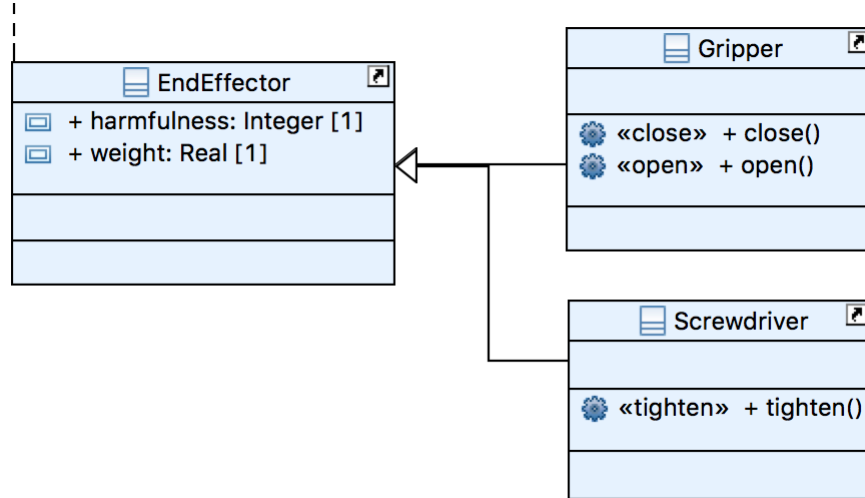


Figure 4.5: Fragment of the *Class Diagram* capturing the End Effector and its specification classes. Each available types is featured as a standalone class, in this case Gripper and Screwdriver, rather than as part of an enumeration in order to better capture the skill set differentiation.

- *<< open >>* and *<< close >>*, which are specific to *gripper*-type end-effectors;
- *<< tighten >>*, which can be performed by a *screwdriver-like* tool.

As a final remark, the *Class Diagram* also features an *EEattachment* association among Mobile Device and End-Effector and a self-association labeled *DevAttachment* on the former, both visible in Figure 4.3, the realizations of which will be discussed in detail in section 4.3.

Signal-emitting devices

A different specialization for the class Resource is that of devices specifically aimed at processing signals. Unlike proper agents, these play a *semi-active* role in the pursue of the task goal, meaning that they do provide additional functionalities but only implementable when associated with an existing agent. This is modeled as an *association-like* relationship, also captured in Figure 4.1 and in Figure 4.2 between *Operator* and *ArmBand*. Essentially they constitute an augmentation of agents' capabilities, which allows the achievement of several goals. They mainly serve as a communication channel between operator and robot at run-time, since verbal or visual command processing is still hardly supported by industrial plants. This works in both directions, as in when the operator needs to issue state-based commands, whose details are explained in section 4.4, and when he/she requires feedback notifications from the environment, e.g., about the completion of an operation or the occurrence of an hazardous event which demands intervention. The kinds of devices

that may play this role are standard signal-processing items, such as *buttons* and *knobs*, and *wearable devices*, e.g., arm bands which are able to map and interpret a human gesture or push notifications, whose corresponding class is captured by Figure 4.2. These are all featured in the *Class Diagram* as specifications of the super-class *SignalEmitter*, visible in Figure 4.1, they do not possess any attribute in the current version and provide a method for every type of signal that they are able to handle. For instance, the list of signal-related stereotypes possessed by the *ArmBand* class, as in Figure 4.2, is now reported:

- << *waveIn* >>;
- << *waveOut* >>;
- << *vibrate* >>;
- << *fist* >>;
- << *fingerSpread* >>.

Cameras in a way represent an exception to this rule, since they may be an extension to an agent, e.g., in *eye-in-hand* vision control systems, but may also work as standalone components. In the latter case, a possible workaround is to consider the *environment* as a fictitious agent the camera is enhancing, also specifying the section in which it is mounted. Note that the current model only features a generic *activate* method and does not contain any further detail about the specific content of the script that is executed as a result of the activation, since it is not necessarily crucial to the formal verification procedure and -as far as deployment goes- the support extent is strongly limited by which blocks have been developed in advance. As a result this issue may be only partially manageable by an automated tool and, when necessary, the user is recommended to thoroughly check the level of coverage.

Objects

Following the same rationale as for the previous Resource specifications, Objects are intended as *passive* entities which are manipulated by agents to bring the task to completion. Such modifications can consist of displacements, assembly sequences, e.g., in the case of fixtures, or manufacturing processes, such as workpieces destined to CNC operations. Due to its nature, this class, shown Figure 4.1 does not possess any method but it owns properties regarding its physical characteristics, such as *weight* and *width*. These are necessary for payload management, as previously explained, and as parameters for certain operations. For instance, the gripper's closure in some cases might require additional details like, indeed, the size of the object that is going to be grasped.

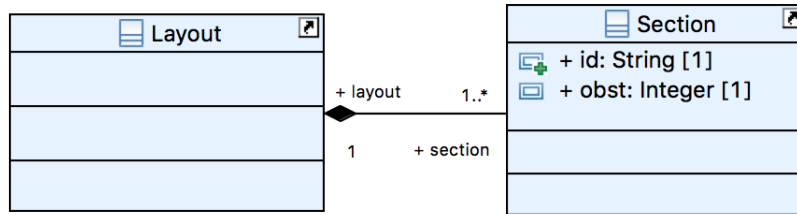


Figure 4.6: Fragment of the *Class Diagram* capturing the *Layout* and *Section* classes. These are connected by an aggregation relationship. The snapshot also captures a portion of the association between *Layout* and *Resource*, which is fully visible in Figure 4.1.

4.2.2 Layout

Concerning industrial applications, a crucial matter is that of layout organization. This comprehensively affects robot’s performance while bringing a task to completion, since a good setup allows the optimization of key productivity factors such as cycle-time, speed and overall traveled distance [81][82]. Furthermore it also influences the plant safety conditions, concerning operator’s security as well as machinery integrity, which might be jeopardized by unforeseen collisions. There are numerous available formal methods to optimize space arrangement [81][83], which must take into account various factors regarding working stations, robots and human workers. In fact, it is imperative to heed machine shapes, where their access and delivery points are located so that they are handily accessible, and consequently space allowance so that no operation is obstructed. Likewise planning must also factor in how many robots will be working on the same layout, their envelope and base mobility degree, as already discussed in their related section, plus their trajectory planning, which may or may not be linear and hence easily computable, and the nature of the application, e.g., a spraying operation requires more free space than an assembly sequence [81]. Finally, when the scenario opens up to collaborative applications, it brings into play aspects also concerning the operator, such as the preservation of his security, how comfortably he can access his workstation and the maximum degree of visual blockage when performing supervisory tasks.

As for the purposes of this thesis, the focus is not on the definition of the optimal layout but on the selection of a proper modeling strategy, that is simultaneously exhaustive but not overly demanding. A survey on existing methods has been conducted by Sadeghpour and Andayesh in [84], and the retrieved techniques were further classified into three main categories. Firstly, the *predetermined locations* set, featuring a fixed array of areas, with fixed sizes, where objects can be positioned. A slightly more complicated, but not as rigid, method is that of dividing the space into *grid* elements, so that objects can be assigned to a specific grid cell. The entry-level

version introduces equally scaled cells, but the method can be extended to support different shapes and sizes [85]. Finally, the most advanced choice is that of modeling the space as a *continuous* set of points, which is clearly the most accurate pick but requires very high computational power. The natural choice for this project was the grid system, which is the most reasonable compromise in the first place and is also the method chosen for the *L* module in SAFER-HRC. Therefore, the *Class Diagram* features the super-class *Layout* which is an aggregation of *Sections* (see Figure 4.1), corresponding to the aforementioned grid cells. The granularity of these units is up to the user's decision based on the available computational power and so is their shape, e.g., the layout can be divided into radial or rectangular sections, or a mixture of the two, based on the circumstantial needs. The class clearly does not own any method, even though the environment can be treated as a fictitious agent, as already discussed, but it features a set of attributes, visible in Figure 4.6 including the chosen *ID* and the discretized *obstruction* level.

4.3 Component Diagram

As discussed in section 4.2, instance specifications defined by the user are able to capture the existence of a resource and its relevant properties, but they do not provide the possibility to specify anything about their internal structure, which, in some cases, might equally be a priority to model the system in its entirety. The most suitable type of diagram to accomplish this instance deepening is the *Component Diagram*, whose specific purpose is to display structural relationships between components of a system [71]. In this specific implementation component diagrams are mostly aimed at providing realizations of inter-resource associations featured only at an abstract level by the *Class Diagram*, which would lead to unfeasible or inaccurate behaviors throughout the simulations. Sample cases of implementation of this diagram authorized by the notation will be analyzed in detail in the following sections.

4.3.1 Agents

As stated in section 4.2.1, devices are meant to extend operator's functionalities when they are associated with each other. A concrete representation of this association can be achieved through the creation of a dedicated component diagram, displayed in Figure 4.7, following the notation introduced in Section 4.3. The super-component, in this case, models the *enhanced* version of the operator's *skill set*. Con-

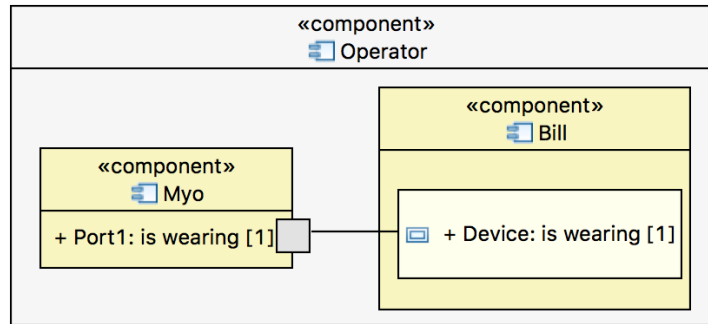


Figure 4.7: Component Diagram representing the *enhanced* agent structure: both the operator and the device are featured as *sub-components*.

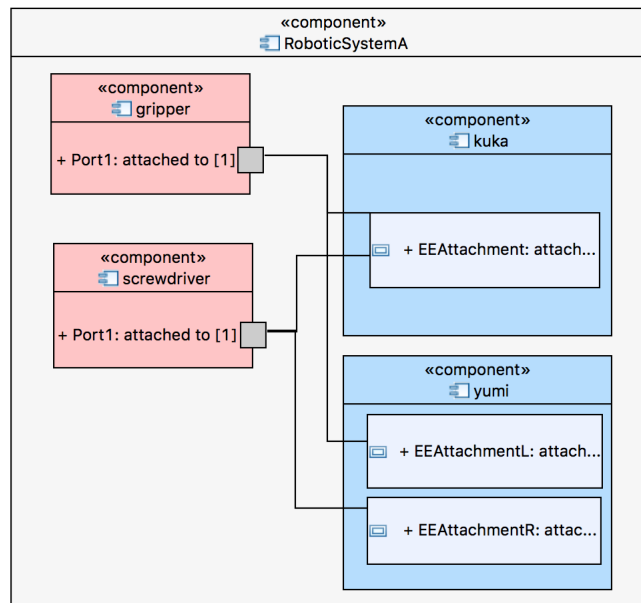


Figure 4.8: Snapshot of the Robotic System architecture for Case A. Note that both arms and end effectors are simultaneously represented, together with the correlated network of connectors.

sequently its internal structure features a sub-component that mirrors the standard operator instance specification and one (or multiple ones) for the selected device(s), corresponding to *Bill* and *Myo* in Figure 4.7. The construct that transposes the class diagram association includes a *property* held by the operator sub-system, whose type corresponds to the association’s name, a *port* on the device component and a *connector* joining the two: respectively *Device: is wearing*, *Port1* and the connector in Figure 4.7. This implies that the standard semantics previously illustrated has been fairly bent to better fulfill the specific domain purposes. As a matter of fact, the connector-port pair is exploited to jointly capture the concepts of *skill-augmentation* and *kinematic constraint*.

The latter holds more significance when dealing with the robotic system. The

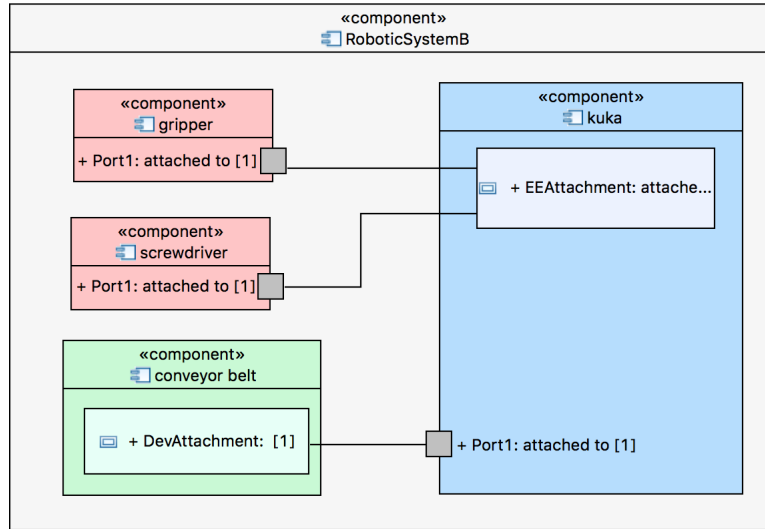


Figure 4.9: Component Diagram representing the Robotic System architecture for Case B. In this case an additional kinematic constraint is defined between the conveyor belt and the robotic arm.

term refers to the overall entity resulting from the attachment of multiple devices or end-effectors, which must all be featured by the related component diagram. Examples can be found in Figures 4.8 and 4.9. In this case the *kinematic constraint* connotation implies that the mobility of the subsystem endowed with the port is partially subordinated to the *dominant* component’s displacement. For instance the association between an end-effector and a robotic arm (see, for example, *gripper* and *kuka* in Figure 4.8) usually entails that a position change of the latter causes the former to move as well, whereas such implication does not hold the other way around. As a final remark, an arguable limitation of this perspective is that it lacks support of the temporal dimension, as in the capability to also model the architectural evolution over a task execution, e.g., in cases where a tool change is envisaged. In this regard the currently adopted solution is to implement the component diagram as an overlap of all the configurations that are going to be enabled amidst the application, shifting the responsibility of accounting for physical constraints to class diagram associations’ multiplicities. Consequently a cross-verification of the two views is due to perform a consistency check. For instance, in Figure 4.9, both end-effectors (*gripper* and *screwdriver*) are simultaneously connected to *kuka*, which would be actually unfeasible as shown by the *attached to* association’s multiplicity 0..1 in Figure 4.3.

4.3.2 Layout

The same component perspective can be applied to model the layout grid subdivision. Therefore, following the same rationale as the previous examples, implementing

the aggregation association present in the class diagram (see Figure 4.1 and 4.6). In this case the chosen convention aims at achieving a good coverage of the modeling objectives without making the diagram creation procedure overly cumbersome. Therefore, the previously introduced notation has been simplified so as to include a *property* instead of a component-property pair for each section, linked to each other by a *connector*-type edge. Figures 4.17 and 4.19 provide related examples. Therefore, also in this case, standard semantics has been altered so that this specific type of connection is interpreted as an *adjacency* constraint, which is fundamental for the formal verification process to return execution traces which are not realistic or an unreasonable *unsatisfiability* notice due to erroneous layout representation. As already stated, in order to keep this specific type of component diagram sufficiently lightweight, further information, e.g., regarding spatial limitations on agents' movements, will be otherwise introduced into the model.

4.4 Activity Diagram

The last part of the HRC-TEAM notation provides a workflow description for the desired task, enriching the model with a *dynamic* perspective that embodies all the aspects that have not been considered yet. As a matter of fact, the previous sections covered the *static* analysis of agents and layout to design what they can do and how they are set up, but no hints were given about the effective strategy to achieve the prefixed objective of the task. Since the project deals with industrial human-robot collaborative activities, the final goal is always the realization of a certain sequence of actions, which is representable through *Activity Diagrams*. Indeed, the purpose of an *Activity Diagram* is to model the procedural flow of actions that are part of a larger activity and the specific use in projects in which use cases are present [71]. It focuses on the order of operation execution and the conditions that trigger or guard them. A legitimate critique to this approach is expressed in [39] and is addressed to the insufficient understanding of requirements and constraints that forces the concrete application of *Activity Diagrams* as an UML tool to remain limited with not many workflow-modeling works. As for this project, the customization of the *Activity Diagram* and the integration with other UML tools succeed in providing a valid and robust implementation.

The UML standard [71] introduces an *Activity* node as core for every project done with *Activity Diagrams*. In the HRC-TEAM notation, it is exploited as a container for everything that refers to the workflow of the task being represented. The contained elements can be split into two categories to distinguish the ones directly modeling the actions and the ones building the framework to provide a meaningful

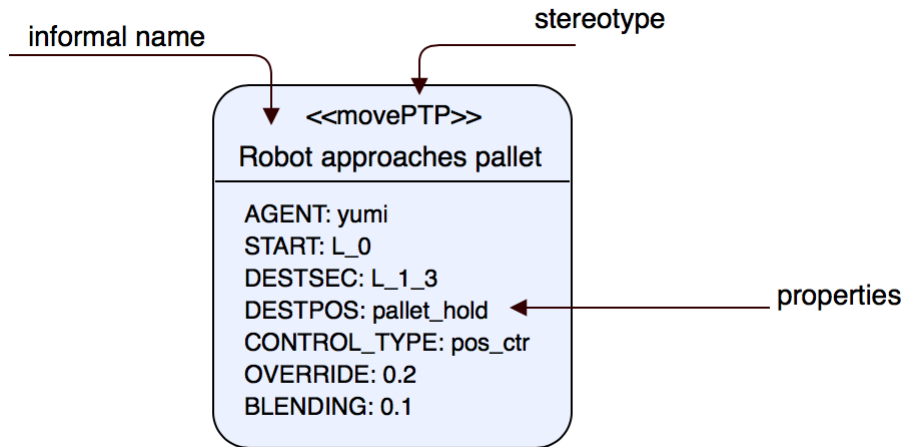


Figure 4.10: Schematic stereotyped *OpaqueAction* example. The top compartment features the stereotype name in guillemets and the informal action’s name. The bottom contains the list of stereotype’s properties with the chosen values.

interpretation to the former: both will be explained in the following sections. What is promptly illustrated is -instead- the connection that the *Activity* has with *Class* and *Component Diagrams*. Besides the informal name that could be assigned to the *Activity*, other properties can be added in terms of *Activity Parameter Nodes* (see left-most small blocks in Figure 4.21 and 4.23). Indeed these elements declare which agents and which layout will be involved in the modeled task: each *Activity Parameter Node* needs to report the name of one of the instances present in the *Class Diagram*.

4.4.1 Actions

In any flowchart there is an atomic unit to represent operations: in the *Activity Diagram* this duty is assigned to *OpaqueActions*, an example of which can be found in Figure 4.10. Its core is represented by the finite state machine visible in Figure 3.2, whose structure of states and transitions is fixed and identical for each action, hence not needing a UML counterpart.

In order to ensure a complete description of the desired task, it is mandatory to apply one and only one stereotype to each *OpaqueAction*. As a matter of fact, by default these nodes offer only a *name* cell, visible in Figure 4.10, that has to be filled with an informal phrase with no other utility than guaranteeing a more readable diagram to the user, whereas stereotypes are exploited to ensure the complete description of each action. Indeed, for each operation executable by an agent, there is a precise stereotype with a list of properties: e.g., for the action *move* some of the

requirements are the agent, the starting point and the destination, as in the example in Figure 4.10. The stereotypes are grouped in different sets depending on the agent that can perform the related operation:

- *robAction* includes the ability of the robotic arm to move, to hold its position and to hook and release the end effector;
- *opAction* provides a catalogue of actions in order to model the behavior of the operator in the most accurate way possible;
- *EEActions* includes the operation executable by the end effector mounted on top of the robotic arm, such as open and close in case of a gripper or tighten for a screwdriver;
- *devAction* includes all the methods by which the signal emitters can communicate with the controller, both automatically (activating a camera) and in a human-driven way (the different signals generated by an armband).

4.4.2 Workflow Elements

Although atomic actions are fully described by the stereotyped *OpaqueAction* nodes, it would still be impossible to generate a complete model without other elements of the *Activity Diagram*. Indeed, they are exploited to provide a meaningful translation towards the UML language for all the workflow properties that identify the desired task: for example connections, their directions and their logic structures. Every workflow element will be described in its proper section, but among them it is reasonable to immediately deal with the two nodes delegated to delimit the *Activity*: the *InitialNode* and the *FinalNode*.

As the name suggest, the *InitialNode* shows the starting point of the diagram. Obviously, it cannot have an incoming arrow and it has one and only one outgoing arrow connecting it to the first element of the task sequence. Similarly, the *FinalNode* is the conventional ending of the model, having only one entering arrow and no departing connections. However, for the HRC-TEAM notation a change in the convention is needed. As a matter of fact, it is fundamental that the very last unit in the *Activity* is an *OpaqueAction*. The reason for this variation is the demand of a fictitious pre-condition for the conclusion of the task, as if the act of declaring concluded the task was itself an action: this will be further explained in Chapter 5. However, this node will remain the only *OpaqueAction* with no stereotypes and, to facilitate its retrieval, with the informal name of *ActivityFinal*.

Arrows

In addition to the atomic actions, the connections between them are also indispensable to define an understandable and effective workflow. Inside the *Activity Diagram* semantics, it is straightforward to apply a *ControlFlow* whenever a transition should occur: indeed, its *arrow-like* graphical representation indicates also the direction from the element that originates the transition to the one that receives it. In order to properly use connections in this approach, some rules have been set. Constraints on the incoming and outgoing arrows of *InitialNode* and *ActivityFinal* have already been explained. Further mandatory requirements are based on the configuration of the finite state machine presented in section 3.1.3 for every basic action. Firstly, the maximum number of arrows coming in and out of an action depends on its possible internal states. This is due to the fact that transitions are triggered when the source action is in a particular state: i.e., for each state the action can be in, there may be an outgoing arrow. Moreover, for each command that an action can receive to change its internal state, there may be an incoming arrow.

The essential consequence of multiple *ControlFlows* attached to a single *OpaqueAction* is uncertainty in giving the appropriate meaning to each transition line. The worked-out solution gets rid of the ambiguity by using stereotypes. In this case, they are applied directly to the arrows, as shown in Figure 4.11, with two key rules:

- Each outgoing arrow must have a state-describing stereotype, in order to identify which state of the source will trigger the transition: *done*, *executing*, *hold*
- Each incoming arrow must have a command-describing stereotype, in order to identify which command will be delivered to the target: *start*, *stop*, *pause*, *resume*.

Since the majority of the transitions between actions are built on the standard pattern of “when source action is *done*, target action *starts*”, it may be possible not to apply any stereotype to arrows representing this kind of link: the default values are *done* as state and *start* as command. Lastly, a further specification related to command stereotypes should be introduced. As a matter of fact, two different cases have been treated in modeling the command an action can receive: one implies a full synchronization, that is an immediate trigger and execution of the command as soon as the source action has entered the interested state; the other one constitutes a softer constraint by stating that the target command must be executed some time after the source state has been reached. The former case is referred to as *strong* command and the latter as *soft* command: the aforementioned command stereotypes refer to synchronous commands, whereas *SOFTstart*, *SOFTstop*, *SOFTpause* and *SOFTresume* refer to the asynchronous counterparts. For instance, the arrows in Figure 4.11 imply that *waveIn* must *start* when *hold* is *executing*, and that *movePTP* must start once *waveIn* is *done*: the latter could have also been omitted, as already

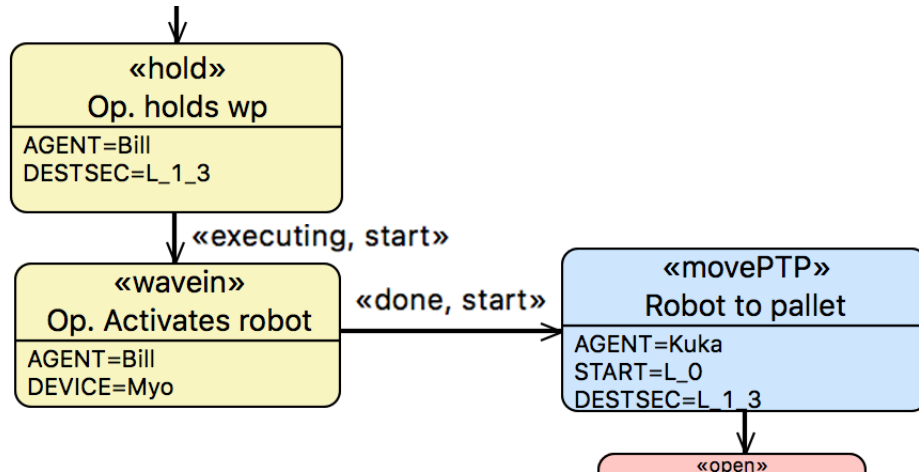


Figure 4.11: Portion of an Extended *Activity Diagram*. All Opaque Actions carry a stereotype to be precisely characterized and its correlated list of properties. They also feature an informal name to guarantee the user a better comprehension of their meaning. The figure also captures some arrow stereotypes that establish the state-based conditions guarding actions execution.

explained.

Logic connectors

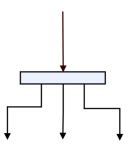
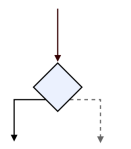
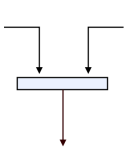
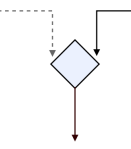
Even in simpler tasks, the demand of non-trivial connections is very common. One may consider, for example, the end of an action that has to trigger the start of two others: despite its simplicity, this condition would be impossible to design strictly following the rules above. As a matter of fact, just one arrow for each state can be drawn from an Opaque Action, whereas two arrows (with the same *done* stereotype) are required to concretize the example. The solution for these semantic obstacles involves standard *Activity Diagram* elements plus a light customization. The problem concerns the realization of complex connections, usually between more than two actions, and it is actually related to the precise representation of logic expressions, e.g., “action A being *done* implies that action B AND action C must *start*” (formally expressed in 4.1). The implications are modeled by means of the direction of the arrows, while the logic operators by nodes of the UML language that are here identified as logic connectors. A complete overview will now be presented, and is additionally summarized in Table 4.1.

$$a_{A,sts} = dn \Rightarrow \text{Futr}(a_{B,sts} = exe \wedge a_{C,sts} = exe, 1) \quad (4.1)$$

In the first place, the *ForkNode* represents the correct instrument to model formula 4.1. It stands for any AND condition that regards the outgoing arrows, since

it manages just one incoming arrow but multiple arrows can depart from it: with just one arrow with a unique state stereotype coming from an *OpaqueAction* (so as to satisfy the semantic requirements) the *ForkNode* can trigger as many elements as required. Secondly, a similar connector but with a reversed approach is the *JoinNode*. The shared feature is that it still regards an AND condition, but the distinction resides in the handling of different incoming arrows and just one outgoing: to launch the command towards the goal action it is necessary that a trigger has arrived from each input action. Considering once more the incoming arrows, the *MergeNode* is in charge of the realization of the OR condition. Therefore, the structure of this element is identical to the one of the *JoinNode*, but to trigger the output action is enough to receive a signal from one of the incoming arrows. The last logic connector is the *DecisionNode*, basically a XOR condition useful to represent a mutually-exclusive choice. In other words, the trigger coming from the unique input is propagated through only one output connection. There are different kinds of decision: purely non-deterministic, autonomously solvable by retrieving data during the actual execution of the task (e.g., for the question "is the end effector attached?" the answer depends on the configuration of the robot at that time instant), or human input-dependent. In principle any question can be modeled with a *DecisionNode* and a proper customization, even if the model checker Zot will simply manage it by non-deterministically choosing one of the available paths and then verifying that particular case (see Chapter 5 for further clarifications).

Table 4.1: Supported logic connectors

Structure				
Element Name	Fork Node	Decision Node	Join Node	Merge Node
Logic Condition	and	xor	and	or
Affected arrows	outgoing	outgoing	incoming	incoming

As a conclusion, it is appropriate to mention that the just presented logic connectors can be combined in complex networks to achieve the modeling of any kind of logic condition. As far as connections are concerned, if an arrow starts from a connector and ends into another connector, no stereotypes must be applied to it because the State and Command values are retrieved from arrows having as source and/or target an *OpaqueAction*.

Loops

When dealing with industrial robots it is very frequent that a set of operations has to be executed several times before the task is completed and this holds true also for collaborative applications. While handling repetitive actions, the main issue is the quest for a compact and rapid way to model them avoiding redundancy and duplicates. According to the UML standard, this is achieved through the *LoopNode* which consists of a container ideal to distinguish the set of *OpaqueActions* to be iterated. For this reason, it has been exploited in the HRC-TEAM notation as a solution for the loops present in the task models. However, to achieve a comprehensive modeling of the desired task, it is necessary to customize the *LoopNode* with some conventions. Firstly, one needs to set how many iterations the loop should have and that is why, also to grant an easy retrieval of the value, the name of the *LoopNode* must correspond to the number of iterations. Then, concerning the external connections, just one arrow is allowed to enter the container and at the same time only one arrow can trespass the contour of the *LoopNode* from the inside to an external element.

4.5 Case Study

4.5.1 Setting Description

The developed notation has been applied to a real case study in order to test its usability and efficiency. The chosen scenario is the *assembly* and *disassembly* procedure of a machine tool pallet to be employed in CNC operations. The pallet is manipulable on both sides, *Face 1* and *Face 2*. The former features twelve *slots* that can fit two rectangular *workpieces* each, and a *fixturing* structure with a central *bolt* that can hold both pieces in place once it has been tightened. The second side is only able to hold two similarly shaped but larger workpieces. In both cases, the pallet design takes into consideration its target application, hence allowing enough space between two adjacent slots so that the machining tool is able to perform all necessary operations in a precise fashion.

For the purposes of this thesis, the analysis will focus only on the setup of *Face 1*, in order to keep the investigation all-encompassing but free from minor redundancies. In this respect, two processes are conceivable: *assembling* in preparation to machining and *disassembling* at the end of the sequence. Two slightly different setups have been devised for the two alternatives in order to test the approach against a wider set of conditions, which, from now on, will be respectively referred to as Case A and

Case A

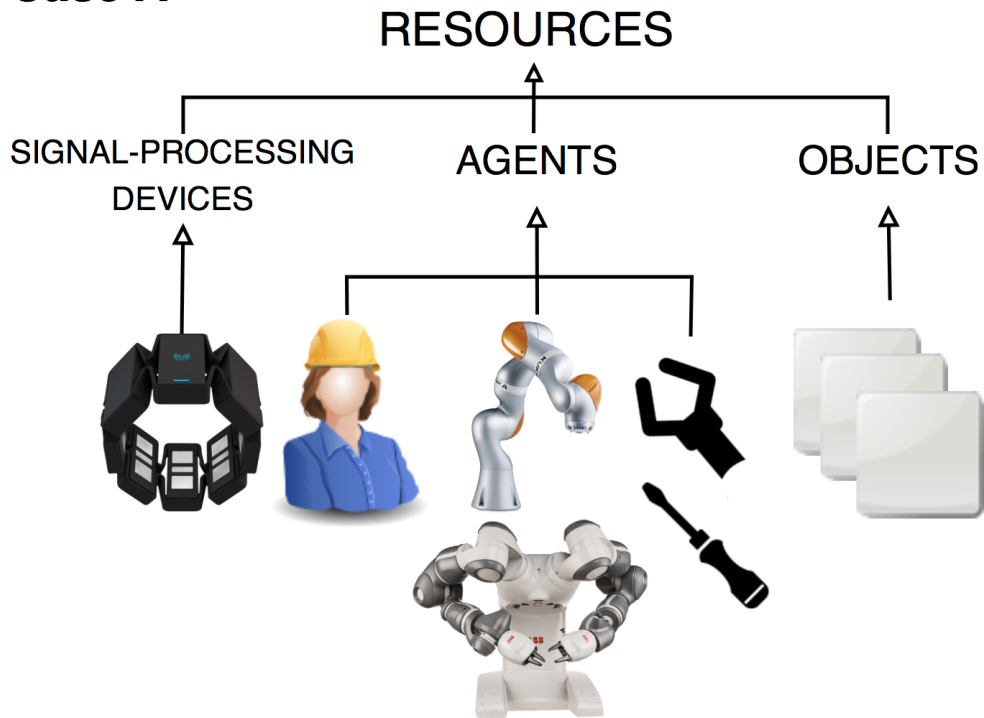


Figure 4.12: Set of resources for Case A.

Case B

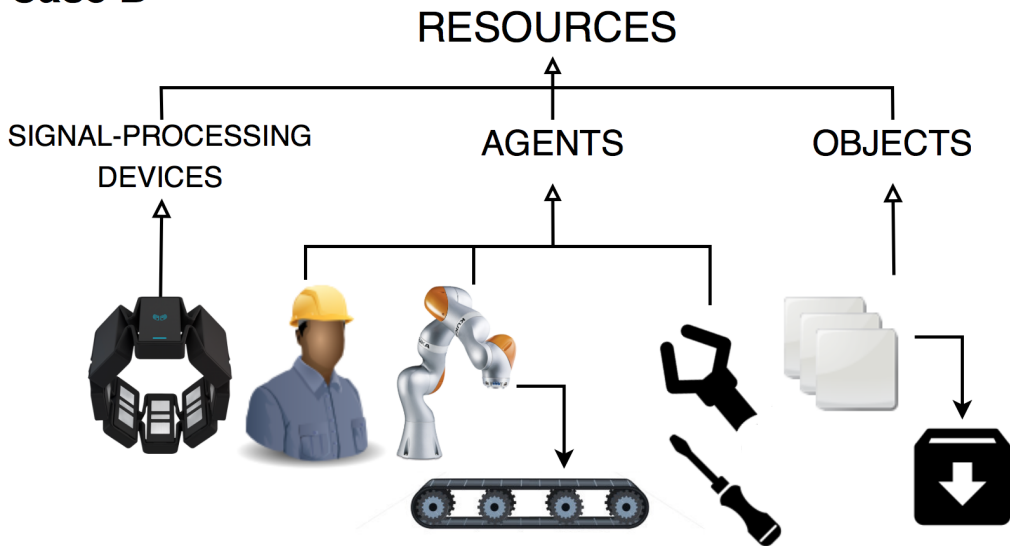


Figure 4.13: Set of resources for Case B.

Case B.

The main steps for the two cases, expressed in generic terms, are now summarized, firstly for Case A:

1. *fetching* workpieces from a *load station*;
2. *producing* the necessary screwdriver tool;
3. *holding* the workpieces in place;
4. *tightening* the fixture bolt;

and secondly for Case B:

1. *producing* the necessary screwdriver tool;
2. *holding* the workpieces in place;
3. *un-tightening* the fixture bolt;
4. *placing* the workpieces on the *unload station*.

The skills are listed without any explicit reference to the specific agent that will perform them, since different alternatives are conceivable and will be examined in later sections. In any case, even prior to such allocation process, it is possible to define the sets of *resources* and *equipment* which are required to bring these two tasks to completion. For Case A the required furnishings include, of course, the *pallet*, the workpiece *load station* (also referred to as *bin*), a *tool-changing station* and a generic *storage* accessible to the operator. For Case B the set is similar but with a symmetrical *unload station* and no storage is actually needed. Note that this kind of equipment does not have to be explicitly modeled by the notation, unless the user desires to do so for the sake of completeness. The presence of these items is more efficiently reflected by the *layout* model in terms of *obstruction* level, as will be further explained in Section 4.5.3. On the other hand, resources do require precise characterization: the respective sets, which will now be described in detail, are pictured in Figures 4.12 and 4.13. In both cases the employment of a *human* operator is indispensable. In order to broaden the available set of skills, a gesture-detecting armband (officially branded as *Myo*¹) is also included in the picture. As for the robotic systems, two different configurations are used in order to test the flexibility of the notation. For Case A a 7 D.o.F. *Kuka LBR iiwa*² with a fixed base has been selected, endowed with both a *gripper* and a *screwdriver* end-effector, exchangeable

¹<https://www.myo.com>

²<https://www.kuka.com/en-gb/products/robotics-systems/industrial-robots/lbr-iiwa>

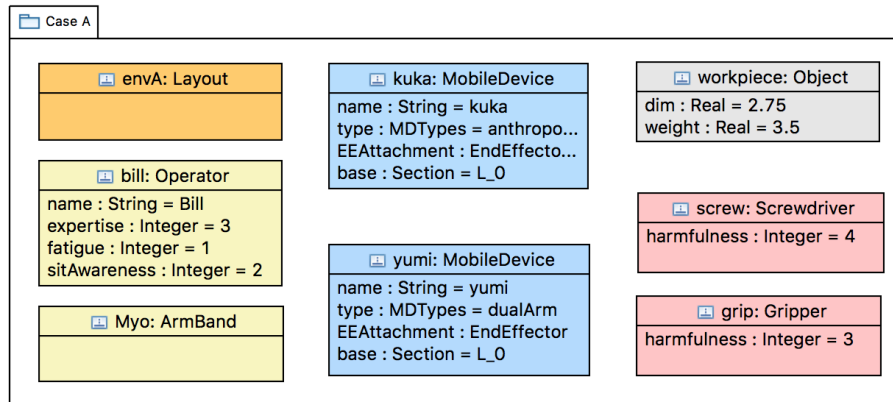


Figure 4.14: Set of instance specifications for Case A.

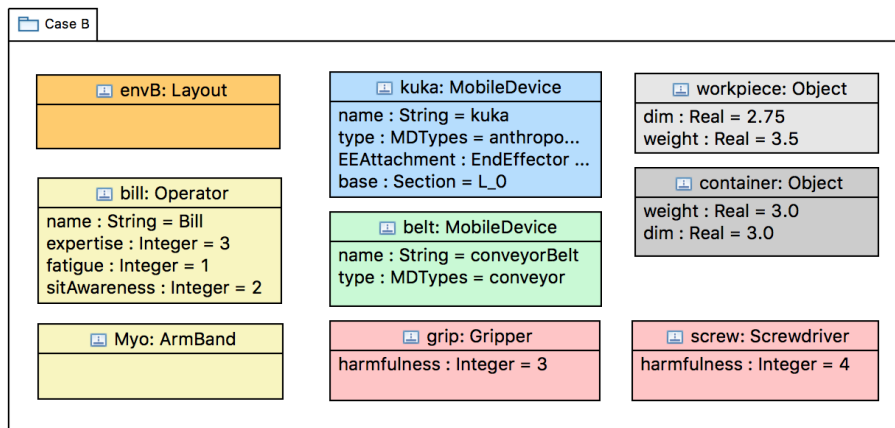


Figure 4.15: Set of instance specifications for Case B.

through the aforementioned tool-changing station. Alternatively, it is possible to replace the robotic arm with the dual-arm Yumi³ manufactured by ABB depending on the chosen alternative (see Section 4.5.4). For Case B the robotic system features the aforementioned Kuka arm equipped with the same tools but mounted on a *conveyor belt* which is instrumental in enlarging the operational *workspace*. Finally, in both cases *workpieces* are featured as manipulable objects, whereas in Case B an intermediary *container* has also been included to store workpieces once they are placed in the buffer.

4.5.2 Class Diagram Instances

The same sets must be captured by the Class Diagram in accordance with the illustrated modalities. Therefore, an instance has been created for each resource, with corresponding proper attributes, all collected in Figures 4.14 and 4.15. Note that it is possible to have both sets of instances in the same Class Diagram without any conflict, eventually customizing their appearance in order to have a clear visual indication of their usage. As planned, the Case A package includes instances of the Operator and ArmBand classes (*Bill* and *Myo* in Figure 4.14) with arbitrary values for the mental state-related attributes, shared with Case B as can be seen in Figure 4.15. The robotic arms (*Kuka* and *Yumi* for Case A, Figure 4.14, only *Kuka* for Case B, Figure 4.15) and the conveyor belt (Figure 4.15) are featured as MobileDevice instance specifications. Attributes are specified accordingly: for instance, as shown in Figure 4.14 *Kuka*'s *type* is set to *anthropomorphic* whereas *Yumi*'s is *dual-arm* (both values are chosen out of the enumeration shown in Figure 4.4). End-effectors are defined as instances of the related EndEffector specification class, i.e., *Gripper* and *Screwdriver* featured in Figure 4.5. Finally, the *workpiece* and, only for Case B, the *container* are also featured as instances of the *Object* class (see Figure 4.1). As a final remark, Figure 4.14 and Figure 4.15 also include instances for the operational environments (*envA* and *envB*) which are, more specifically, realizations of the class Layout in Figure 4.6: these will be presented in further detail in Section 4.5.3.

4.5.3 Component Diagrams

The sets of resources introduced in Figures 4.12 and 4.13 require the creation of dedicated Component Diagrams that provide a more detailed picture of their inner architecture. First, the *enhanced* operator's component, visible in Figure 4.7, contains both references to *Bill's* and *Myo armband* instances, connected by a *port-connector* pair. By doing so, when a gesture recognition action is allocated to *Bill*, it will not raise any inconsistency warning, since the functionality augmentation is stated by the *Component Diagram*. Note that this specific diagram can be used for both case studies without any adjustment.

On the other hand it is necessary to create multiple diagrams for the two implemented robotic systems since they incorporate different resources. The results are shown in Figures 4.8 and 4.9. As already explained, these diagrams work as a collective view of the system's configurations along the whole task timeline, hence the two End Effectors are simultaneously included for both case studies. The two firstly differ due to the fact that Case B features a *conveyor belt* component (see

³<http://new.abb.com/products/robotics/industrial-robots/yumi>

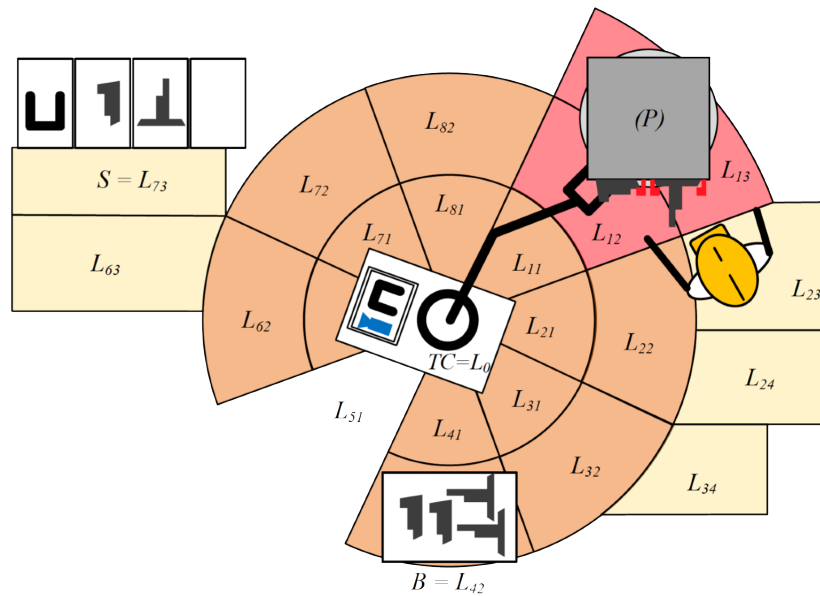


Figure 4.16: Informal representation of the layout for Case A. Colors mirror the section obstruction level: occluded (red), clear (orange), free (yellow). The grid features the robot’s base L_0 , the pallet in L_{13} , bin in L_{42} and storage in L_{73} . Note also the blind clove in L_{51} compliant with the actual robot’s envelope.

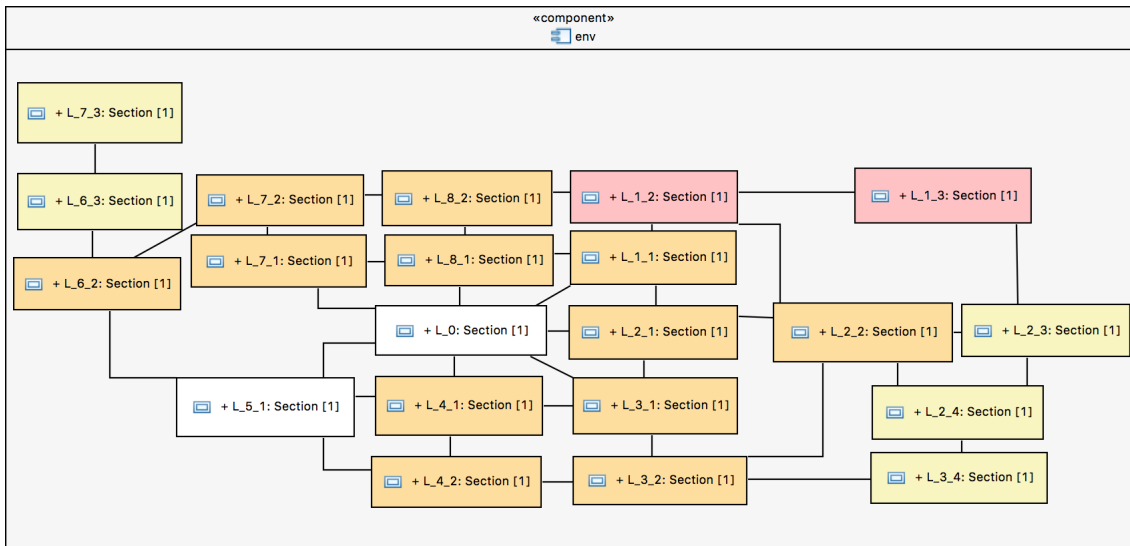


Figure 4.17: Corresponding *Component Diagram* for Case A layout. The appearance has been adjusted to better match the informal representation, but it does not hold any functional significance and is thus completely optional.

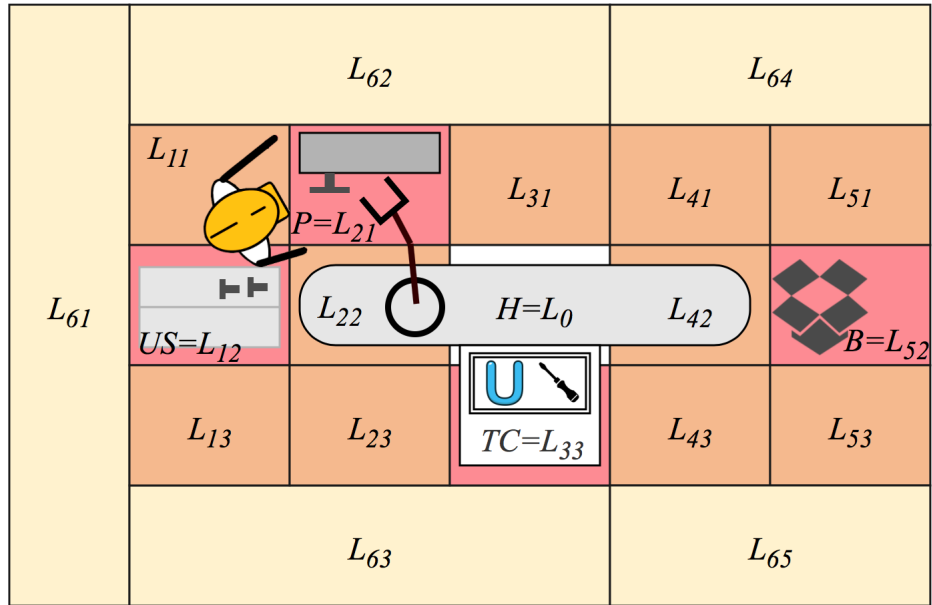


Figure 4.18: Informal representation for Case B cell layout. In this case the conveyor spans over sections L_{22} , L_0 and L_{42} . Other criticalities concern the pallet P in L_{21} , the tool changer TC in L_{33} , unload station (or buffer) US in L_{12} and bin B in L_{52} .

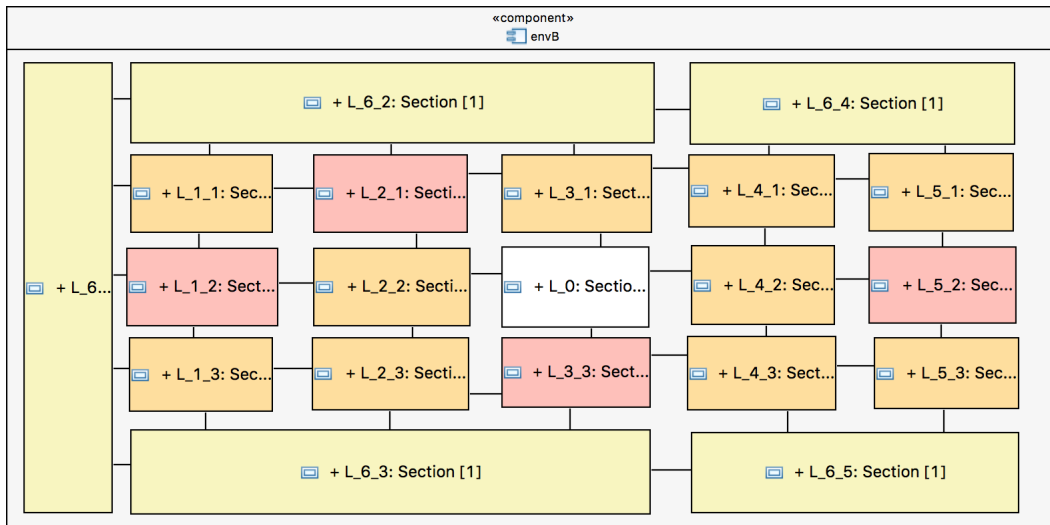


Figure 4.19: Corresponding Component Diagram for Case B layout.

conveyor belt in Figure 4.9), connected to the robotic arm to state the kinematic constraint. Furthermore the Component Diagram for Case A requires both robotic arms, the anthropomorphic and the dual-arm (*Kuka* and *Yumi* in Figure 4.8): these are not connected to each other since they work alternatively rather than jointly.

At this point of the procedure, it is necessary to define in detail the layout grid structure. Concerning Case A, since the planned robotic system has an anthropomorphic configuration and a fixed base, it is reasonable to devise a layout with a *radial* setting, as in Figure 4.16. The critical sections are the ones occupied by the pallet, the robot's base and tool-changer and the bin (or load station). On the other hand the storage, which is only required by the operator, has been placed in a section out of the robot's reach to avoid unnecessary collisions. As for Case B, due to the different architecture the *linear* subdivision in Figure 4.18 is comparably appropriate. In this case the conveyor fully occupies three of the available sections, and further limitations are related to the presence of the buffer, the bin, and -of course, pallet and tool-changer. A surrounding un-occluded path has been planned to allow the operator to navigate the cell without risking an impact at any time instant. The manually sketched blueprints in Figures 4.16 and 4.18 are followed by their equivalent Component Diagrams, respectively Figures 4.17 and 4.19. Note that this type of diagram does not capture anything about section sizes or position in space, hence the level refinement of their appearance is up to the user and mainly affects the diagram's readability. The main purpose and requirement is to model *adjacency* connectors, as shown in the resulting diagrams in Figures 4.17 and 4.19. The implemented concept is similar to the one for *simplified topological maps* used to represent metro lines, which forsake geographical accuracy to focus exclusively on stations' sequences.

4.5.4 Activity Diagrams

The operational steps for the two case studies have been illustrated in general terms in Section 4.5.2. In order to precisely model the related workflow, though, it is necessary to define a far more detailed set of actions and settle the skill allocation problem. As for Case A, a great deal of care has been given to its workflow development, to the point that different alternatives have been envisaged. The variedly allocated *sub-tasks* are:

- ST.0 Go to tool-changer (TC); change the tool (T);
- ST.1 Go to the bin (B), grasp a workpiece (wp), move wp to the pallet (P), hold wp in position, . . . , release the wp;
- ST.2 Move T to the pallet position. Note that ST.2 requires that a suitable T is available, otherwise move to TC and fetch a T before proceeding;

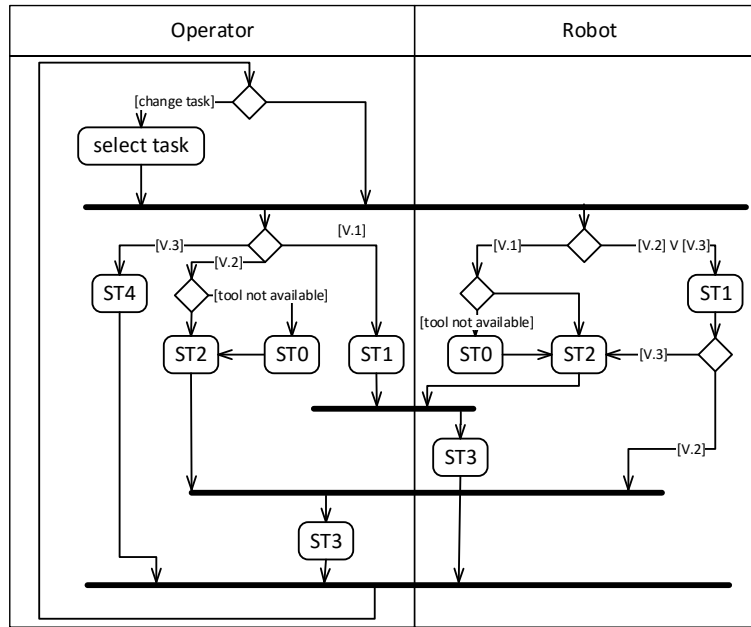


Figure 4.20: Informal representation of the task workflow for Case A. All three versions are depicted implementing decision points with properly labeled outgoing arrows. Note that sub-tasks blocks are used rather than elementary actions in order to keep the scheme as readable as possible.

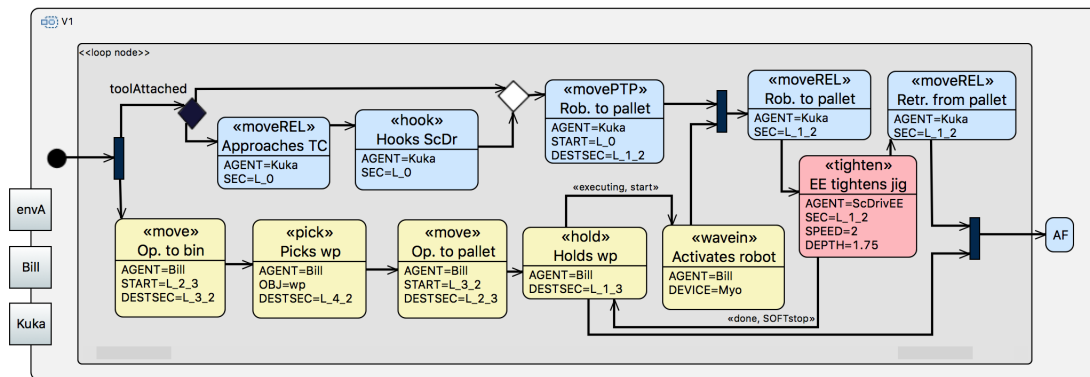


Figure 4.21: HRC-TEAM compliant Activity Diagram for Case A, V1. Note the application of stereotypes to all OpaqueActions and the implementation of arrow stereotypes to delineate complex state-based conditions.

- ST.3 Iteratively (on the number of jigs) use T to fix (e.g., tighten a bolt with T=screwdriver) the wp on the pallet, while wp is held in position;
ST.4 Go to storage (S) for inspection or any other purpose.

The different variants of the task, depending on how the operator decides (on-the-fly) to assign the execution of sub-tasks between himself and the robot, are:

- [V.1] ST.1 is done by the operator and ST.0, ST.2 and ST.3 by the robot;
[V.2] ST.1 is done by the robot and ST.0, ST.2 and ST.3 by the operator;
[V.3] ST.1, ST.2 and ST.3 are done by the robot, ST.0 and ST.4 by the operator.

Figure 4.20 presents an informal representation of the illustrated workflow, featuring all three alternatives properly labeled. The definition of multiple possibilities meets very well the flexibility and reconfigurability requirements. As a matter of fact, it may be necessary to adapt to environmental or operational variations, and having the possibility to do so without suspending all productive activities constitutes a very valuable asset. Consequently the same approach has been kept for the HRC-TEAM compliant Activity Diagram, whose full version can be found in Appendix I, whereas Figure 4.21 shows only the portion related to V1 for the sake of conciseness. The *Activity* is endowed with *Activity Parameter Nodes*, i.e., the boxes on the border in Figure 4.21 which link this perspective to the resource representation one. Therefore, in this case, the required agents are *Robotic System A* and *Bill*, which have already been examined in Sections 4.5.2 and 4.5.3, and the selected layout representation is *envA*, which is depicted in Figure 4.16 and 4.17. The Activity clearly contains a *LoopNode*, also visible in Figure 4.21, since its operations are supposed to be iterated for as many pallet fixture slots as requested. The gateway to the loop node is represented by a *DecisionNode* which captures the flow alternative selection. Similarly V1 and V2 branches also feature an internal *DecisionNode* addressing the presence of the required tool, whether it concerns the operator for V2 or the robot for V1 (corresponding to *toolAttached* in Figure 4.21). As for this specific case, the decision making process should not be arbitrary but based on data coming from sensors, but, at the current stage, this is still not a supported feature. Further matters demanding investigation regard the rendition of segments in which collaboration is preponderant: i.e., when agents are directly operating on the pallet. In these cases, arrow stereotypes are mostly exploited and prove their effectiveness. For instance, as Figure 4.21 shows, when the operator starts holding the workpiece in place, he/she is also allowed to send an activation signal to the robot («*executing, start*» arrow between *hold* and *wavein*). This enables the robot to approach the pallet, if it is already in the required section (*JoinNode* between the pair *movePTP-wavein* and *moveREL*) and tighten the fixture bolt, only after this is complete the operator is allowed to remove his/her hand from the pallet («*done, SOFTstop*» arrow between *tighten* and *hold*). Similar constructs have been envisaged for the other versions and

are visible in the Appendix A.

The same approach can be applied to the illustration of Case B workflow. Figure 4.22 summarizes the delineated sub-tasks, whose meaning will now be clarified:

- ST.0 Go to tool-changer (TC); change the tool (T);
- ST.1 Approach the pallet (P), hold wp in position, place top wp in buffer (US), . . . , place bottom wp in buffer (US);
- ST.2 Move T to the pallet position. Note that ST.2 assumes the robot starts with a suitable T attached;
- ST.3 Iteratively (on the number of jigs) use T to dismantle (e.g., untighten a bolt with T=screwdriver) the wp on the pallet, while wp is held in position;
- ST.4 Approach the bin (B) to empty it or for inspection. Note that the area the bin content is destined to is left unspecified since it lacks relevance with respect to this analysis;
- ST.5 Iteratively (on the number of containers) approach the buffer (US), pick a container, move to the bin (B), release it;

Also in this case different skill allocation strategies could be explored, but a single version will be taken into consideration to keep the analysis complexity level within a reasonable threshold. More specifically, as shown in Figure 4.23, initially the robot is in charge of untightening the bolt (ST.3) while the operator holds one workpiece in place and subsequently unloads both of them in the dedicated buffer (ST.1). After the first cycle, the robot changes its tool (ST.0) to be able to perform the iterative pick-and-place sequence (ST.5) from the buffer (US) to the bin (B), whereas the operator can choose whether to empty the bin (ST.4) or supervise until the end of the task. Collaboration is mostly predominant in the initial sequence, whereas towards the end contacts can take place only if the operator chooses to execute ST.4 or mistakenly gets close to the robot while it is performing ST.5. The example still holds significance since it tests the performance of the notation against the presence of multiple loop nodes.

A portion of the corresponding HRC-TEAM *Activity Diagram*, focusing on the initial untightening sequence, can be seen in Figure 4.23, whereas the full diagram can be found in Appendix A. Also in this case the *Activity* is endowed with Parameter Nodes invoking the required agents, hence *Operator* and *RoboticSystemB*, and the employed layout *envB*, in accordance with the outline in Section 4.5.2 and Figure 4.13. As for the actual workflow, similarly to Case A, two *LoopNodes* have been included: the first one shown in Figure 4.23, whereas both are visible in the Appendix A. Also in this case, arrow stereotypes have been widely exploited to precisely identify collaboration mechanisms.

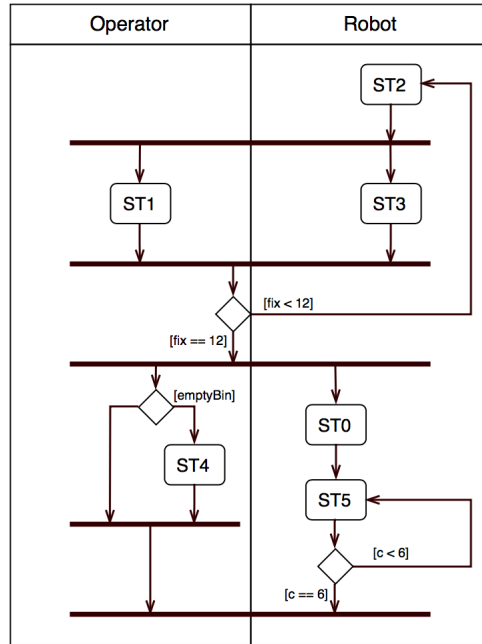


Figure 4.22: Informal representation of the task workflow for Case B. Both cycles are featured, using arbitrary variables *fix* and *c* to respectively indicate the number of untightened *fixtures* and stored *containers*.

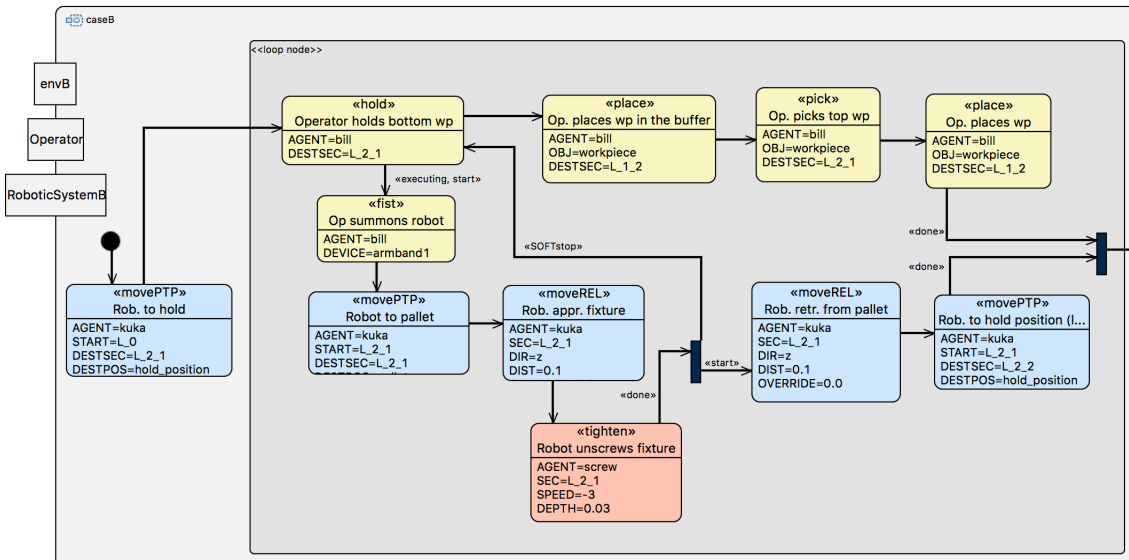


Figure 4.23: Portion of the HRC-TEAM compliant *Activity Diagram* for Case B, capturing the first loop sequence. Note the application of stereotypes to all *OpaqueActions* and the implementation of arrow stereotypes to delineate complex state-based conditions.

Chapter 5

Formal Verification-oriented Model Transformation

In order to perform formal task verification, it is necessary to translate the diagrams realized with the HRC-TEAM profile into sets of logical formulae. In particular, SAFER-HRC models must be produced from the information written in the project with respect to the UML standard. This Chapter focuses on the translation procedure, from the retrieval of data to the production of formulae, introduced in Section 5.1. The preparation of single models is more thoroughly discussed in Sections 5.1.1 and 5.1.2.

5.1 Generation of SAFER-HRC Models

In order to generate SAFER-HRC models and subsequently subject the modeled task to formal verification, the ConverTEAM tool scans and processes the produced HRC-TEAM diagrams. The procedure is centered around the selected *Activity Diagram* and the items it is associated with through the *ActivityParameterNodes*, i.e., *Component Diagrams* and instances in the *Class Diagram*. Once the necessary UML models have been retrieved, the main goal of the tool is to manipulate the collected data and produce formal models. This operation requires textual templates containing models portions, which need to be replicated without any alteration, and *keywords* which conversely need to be replaced with custom generated sets of formulae. The customization degree varies from model to model. The ones regarding standard operations such as risk estimation are plainly replicated since, at current state, no modification is envisaged. The same stands for models concerning the activation of a risk reduction measure based on current hazards and the description of

how these measures affect the system behavior. The module describing the possible hazards and the conditions required to determine their existence is modified only in relation to the maximum index that their parameter concerning the affected section can have. This is derived from the section list size, namely how many *Properties* the *Layout Component Diagram* contains.

5.1.1 Operator, Robot and Layout Models

Models representing resources require a broad set of adjustments. The operator model \mathcal{O} comprises the subdivision into eleven *body parts* whose way of distributing into space must be at least approximately constrained so that the resulting execution trace does not feature any unnatural position. The chosen strategy is to allow the head to be in any of the layout sections, as in Formula (5.1) and force all other parts to be in the same position, see Formula (5.2). The only exceptions are $bp_{11_{reg}}$ and $bp_{7_{reg}}$, i.e., lower arm and hand, which can also be in an adjacent section, see Formula (5.3). As a matter of fact, it is reasonable to assume that the operator is standing in one section with his arm stretched out to perform an operation, such as picking or holding something. Moreover, this also allows the operator to take a safer position. This means that, if he/she is performing an operation in cooperation with the robot, uninvolved parts, like the head, cannot get in the way. Therefore, in this case, the tool looks for the number of sections present in the specified *Layout Component Diagram*, in order to retrieve the value of L_{max} in Formula (5.1). Furthermore, as explained in Section 4.2.2, there may be additional spacial constraints due to excessive obstruction level, e.g., when a section is fully occupied by a piece of equipment. With regards to the operator, this is handled as an additional constraint on the head's possible position, see Formula (5.4), whereas it is considered reasonable for the hand or the lower arm to hover over these sections.

$$bp_{1_{reg}} \geq L_0 \wedge bp_{1_{reg}} \leq L_{max} \quad (5.1)$$

$$\bigwedge_{bp_i \in \mathcal{O}, i \neq 7, 11} bp_i_{reg} = bp_{1_{reg}} \wedge bp_{7_{reg}} = bp_{11_{reg}} \quad (5.2)$$

$$bp_{11_{reg}} = bp_{1_{reg}} \vee bp_{11_{reg}} = \text{adj}(bp_{1_{reg}}) \quad (5.3)$$

$$L_{i,obst} = \text{occluded} \Rightarrow bp_{1_{reg}} \neq L_i \quad (5.4)$$

At current state, SAFER-HRC only supports 2 D.o.F. robotic arms. Therefore,

the \mathcal{R} module, representing the Robot, features the definition of two links, *Link1* and *Link2*, and of the *End-Effector*. Constraints regarding their positions are linked to where the robot base is placed and how the workspace is shaped. The procedure retrieves the ID for L_{home} in Formula (5.5) from the respective *Class Diagram* instance, more specifically *base* attribute value of the *MobileDevice* class (see Figure 4.3). For rotational joints it is assumed that the first link can either be in the same section as the base or in an adjacent one, and the same constraint is propagated to the End Effector, as in Formulae (5.5), (5.6) and (5.7). Note that this heavily depends on how sections are scaled with respect to link sizes: adding adaptability to different sizes could constitute a useful future refinement. Furthermore, robot envelopes usually deviate from a full circle because of a blind circular clove, which must be taken into account to maintain feasibility by means of constraints as the one in Formula (5.8). Finally, also for the robot, there may be occluded sections which cannot be reached: such constraints are formalized by Formula (5.9).

$$R1_{reg} = L_{home} \vee R1_{reg} = \text{adj}(L_{home}) \quad (5.5)$$

$$R2_{reg} = R1_{reg} \vee R2_{reg} = \text{adj}(R1_{reg}) \quad (5.6)$$

$$EE_{reg} = R2_{reg} \vee EE_{reg} = \text{adj}(R2_{reg}) \quad (5.7)$$

$$R1_{reg} \neq L_{blindClove} \wedge R2_{reg} \neq L_{blindClove} \wedge EE_{reg} \neq L_{blindClove} \quad (5.8)$$

$$L_{i,obst} = \text{occluded} \Rightarrow R1_{reg} \neq L_i \wedge R2_{reg} \neq L_i \wedge EE_{reg} \neq L_i \quad (5.9)$$

To conclude the resource models analysis, also the Layout model, corresponding to the \mathcal{L} module, is generated. The Layout *Component Diagram* is firstly inspected to store sections IDs, i.e., the *Properties* names. Secondly *adjacency* constraints are generated by looking for all the outgoing connectors of a *Property* (see for instance Figures 4.17 and 4.19) and storing their destination element ID.

5.1.2 Task Model

The file requiring the most adjustments is the one modeling aspects of the workflow and the task in general, mainly *actions* and *decision points*. As for the latter, *DecisionNodes* featured by the selected *Activity Diagram*, e.g., *toolAttached* in Figure 4.21, are transposed into logic terms, by defining a variable for each of them and a set of constraints as in Formula (5.10). These constraints refer to the feasible range

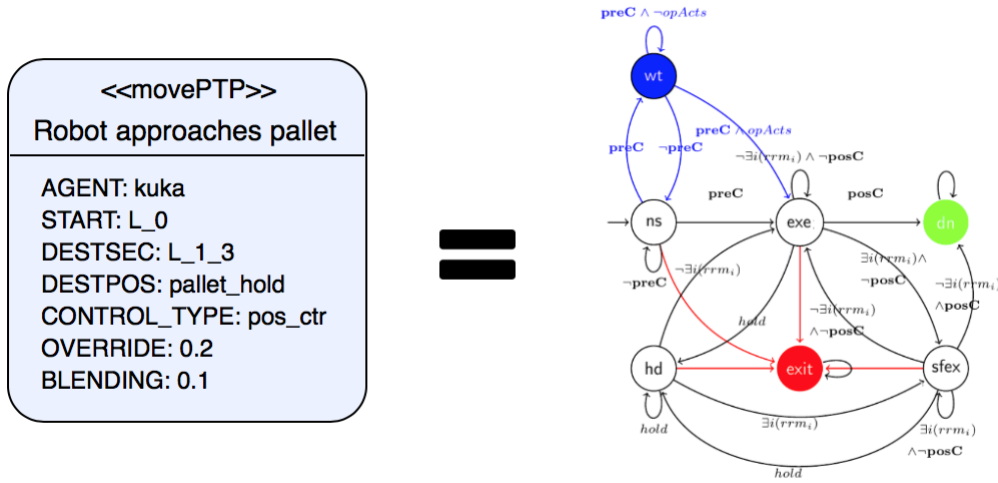


Figure 5.1: Example of how an *OpaqueAction* is translated into logic terms. Its behavior is fixed by the previously introduced Finite-State-Machine on the right, whose corresponding logic formulae are fixed for all actions. The action’s stereotype and properties affect the conditions needed by preC and posC to become true and trigger transitions.

of values, which depends on the number of outgoing arrows: e.g., two in the example reported by Formula (5.10). Moreover, once the model-checker assigns a variable value at t_0 , it must remain fixed during the whole simulation, which is also captured by Formula (5.10). This is due to the fact that a different instance of the variable is generated for each envisaged iteration, if it is contained in a loop. Therefore, allowing its value to change when the iteration is already being executed may lead to unreasonable results featuring different flow alternatives explored at the same time. Furthermore, it is possible to bypass the non-deterministic nature of these decisions by manually forcing the value, which could be useful for testing purposes when the user needs to examine a specific version.

$$\begin{aligned}
 & (\text{decNodeName} = 1 \vee \text{decNodeName} = 2) \quad \wedge \quad (5.10) \\
 & \text{AlwF} (\text{decNodeName} = \text{Past} (\text{decNodeName}, 1))
 \end{aligned}$$

Moving on to action characterization, firstly logic formulae defining the internal finite-state machine, displayed in Figure 3.2, are listed. This portion does not require any changing since it is a standardized feature, as explained in Sections 3.1.3 and 4.4.1 and shown in Figure 5.1. The tool’s main goal in this regard is to generate the set of formulae governing actions state evolution based on the *OpaqueAction* properties. This implies the retrieval of every *OpaqueAction* owned by the *Activity*.

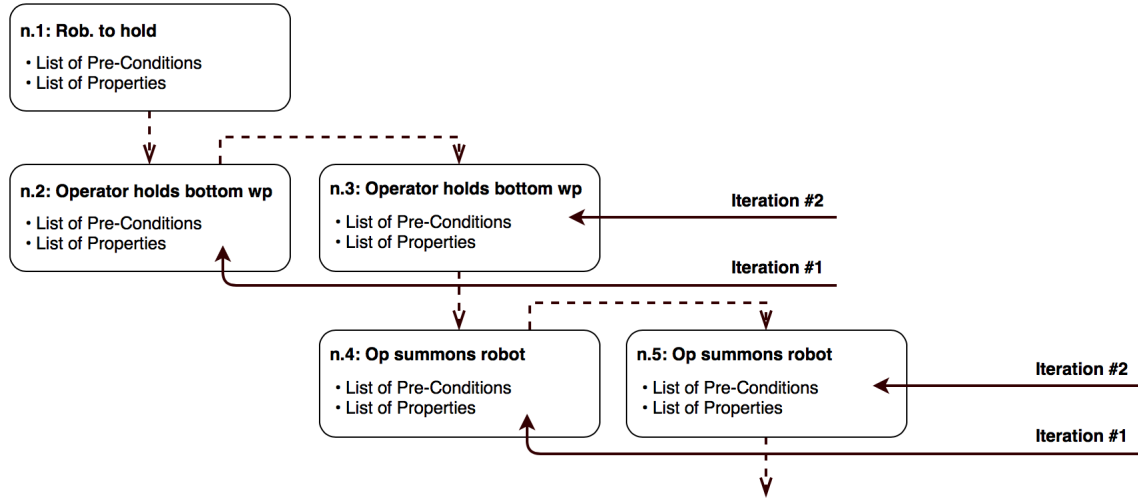


Figure 5.2: Scheme representing the action list generation starting from the *Activity Diagram*. Each *OpaqueAction* contained in a *LoopNode* (n.2 and n.4 in this case) is replicated as many times as the specified number of iterations (e.g., 2 in this case).

Moreover, the ones inside a *LoopNode* are duplicated as many times as the number of loop iterations, as shown in Figure 5.2. Firstly, the tool looks for the stereotype properties concerning the *agent ID* and collects their value to determine each action's *subject*. More specifically, *OpaqueActions* representing operations from the *opAction* set are allocated to an instance of the \mathcal{O} module; similarly for *robAction* and the \mathcal{R} instance. At current state the model only supports single instances of both the operator and the robot. Plus, as for the latter, it does not require the distinction between arm and end-effector, which are both merged into the *robot* subject, but it does not fully support prismatic joints (e.g., cartesian robots or conveyor belts), hence approximations are to be expected. Similarly actions falling into the *devAction* category are assigned to the agent associated with the device in question.

At this stage, TRIO formulations of pre-, post- and inter-conditions are generated. How the system state changes with respect to the first two has already been explained in section 3.1.3. The latter contain expressions that must hold true as long as an action is *executing* (with or without a risk reduction measure applied), either for safety or functional purposes. For instance, if an operator's *hold* action is being executed, his hand must remain in the specified section until the operation is completed. This option must be clearly disabled if one wants to encompass in his analysis also possible human *mistakes* and *misuses*, which is one of the most advanced features of HRC tasks verification.

All of the three types of guard conditions contain constraints related to the *stereotype*, i.e., the *intrinsic* nature of the action, and to the task workflow, hence *extrinsic* factors describing connections to other elements of the *Activity*. The former are deter-

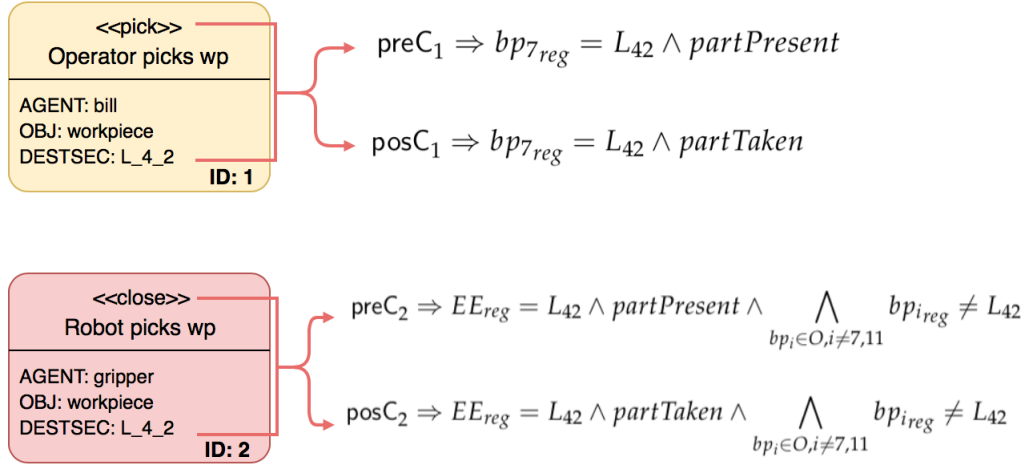


Figure 5.3: Scheme representing how stereotype-related guard-conditions are generated starting from *OpaqueActions*. In this case two *<<pick>>* actions from different agents are subject to the procedure. In both cases the DESTSEC property value is necessary for the spacial constraints concerning either the operator’s hand or the end-effector. In both cases there is also a standard predicate about the state of the object before and after the action’s execution (*partPresent* and *partTaken*). On the other hand, note the additional safety constraint concerning operator’s body parts when the action is performed by the robot.

mined by the stereotype’s category (*robAction*, *opAction*, *EEAction* and *devAction*), and the condition type (*pre*, *post* or *inter*). These factors determine the right set of conditions, which is accessed and adapted, if necessary, based on the properties values. Examples of constraints covered by this case are reported in Formulae (5.11) and (5.12) and Figure 5.3. A robot’s *hold* action, for instance, can *start* if it has previously *grabbed* an object and the End Effector is in the required *section*, as in (5.11). Similarly an operator’s *move* can be considered *complete* when all his/her-body parts have reached the *destination*, as expressed in (5.12).

$$\text{preC}_i \Leftrightarrow \text{partTaken} \wedge \text{EndEffector}_{reg} = L_{14} \quad (5.11)$$

$$\text{posC}_i \Leftrightarrow \bigwedge_{bp \in O} bp_{reg} = L_{13} \quad (5.12)$$

As for the workflow-related conditions, the set of incoming arrows is examined for each *OpaqueAction*. Each connection corresponds to a *guard condition* type: this is determined either by getting the applied *Command stereotype* or considering its absence as an indication for the default value *start*. The edge’s source defines a *Predecessor*. A Predecessor collects informal name, ID and state of the action that

triggers the event brought by the condition. However, if the source is not an *OpaqueAction* but a *Logic connector*, the function enters a recursive procedure to navigate the flow backwards, as visible in Figure 5.4. The procedure stops only when every navigation path encounters an *OpaqueAction*. Therefore, the use of connectors as intermediate *Predecessors* allows complex structures to propagate signals among actions. Apart from this distinction among *OpaqueActions* and connectors, three cases are considered depending on the position of the action with respect to a loop. The basic case regards the most common situation in which both the original element and its *Predecessor* are inside the same loop or outside any of them: identical iterations (arbitrarily equal to 1 in absence of loops) are used for both the two items. Then, if the element is immediately after a *LoopNode* (and so its predecessor is inside the loop), it is necessary to consider the maximum iteration of that loop to attach the right node/s to the tree of *Predecessors*. The last case concerns the item at the beginning of a loop, with its direct *Predecessor* outside, and presents two sub-cases:

1. the action that is the root of the tree represents the first iteration of an *OpaqueAction*, so it has to be connected to the element/s outside of the loop;
2. the action stands for the n ($n > 1$) iteration of that *OpaqueAction* and so the *Predecessor* must be retrieved from the end of the loop considering the $n - 1$ iteration;

The recursive diagram examination leads to the production of two different items depending on the nature of the predecessor: if it is a *logic operator* it is converted in the corresponding TRIO operator, as shown in Table 5.1, if it is an *OpaqueAction* it produces a logic formula in the form described by (5.13) (with the value of *state* depending on the arrow stereotype). This step is recursively reiterated until all

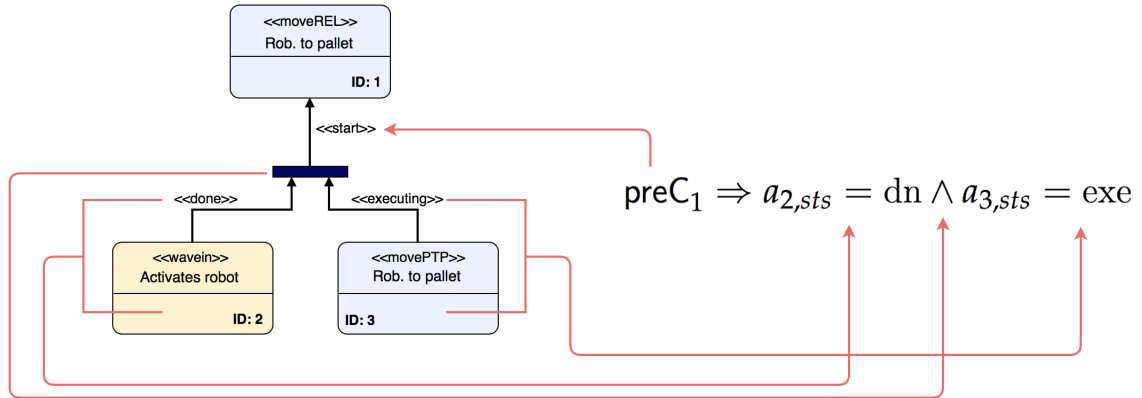
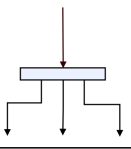
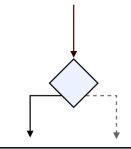
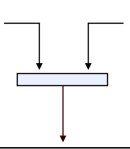
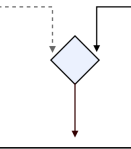


Figure 5.4: Scheme representing Pre-Condition generation starting from a portion of the *Activity Diagram* in Figure 4.23. The procedure for action 1 starts from the immediate *Predecessor*, a *JoinNode*, it stores the related logic condition, and proceeds recursively to the following level containing two branches, i.e., actions n.2 and n.3 with related stereotypes concerning the state.

upstream *OpaqueActions* have been reached and returns the complete condition, that is a set of logic expressions as the ones on the right side of Figure 5.4.

Table 5.1: Connectors logic operator counterparts

Structure				
Element Name	Fork Node	Decision Node	Join Node	Merge Node
TRIO Operator	bypassed	$decNodeName = i$	\wedge	\vee

For instance, if $action_i$ is preceded by a *JoinNode* collecting two arrows, one blank and the other stereotyped as $\langle\langle done, start \rangle\rangle$, the generated line of code reflects the condition in 5.14.

$$a_{ID,sts} = state \quad (5.13)$$

$$preC_i \Rightarrow a_{j,sts} = dn \wedge a_{k,sts} = exe \quad (5.14)$$

Formulae (5.15) and (5.16) reflect the difference between *hard* and *soft* commands illustrated in Section 4.4.2. They are respectively generated as a result of arrows stereotyped as $\langle\langle done, stop \rangle\rangle$ and $\langle\langle done, SOFTstop \rangle\rangle$. In natural language, they can be expressed as " $action_j$ must stop if $action_i$ has been completed" and " $action_j$ is allowed to stop only after $action_i$ has been completed".

$$Past(a_{i,sts} = exe \vee a_{i,sts} = sfex, 1) \wedge a_{i,sts} = dn \Rightarrow Futr(a_{j,sts} = dn, 1) \quad (5.15)$$

$$Past(a_{i,sts} = exe \vee a_{i,sts} = sfex, 1) \wedge a_{i,sts} = dn \Rightarrow SomF(a_{j,sts} = dn) \quad (5.16)$$

The final concept requiring specification is the task completion condition. Indeed, a fictitious *ActivityFinal OpaqueAction* is added to the diagram in order to store its pre-condition, which is obtained with the same aforementioned recursive procedure. Alternatively it is possible to select a single action as termination the portion of task that is going to be formally verified. The two alternatives are respectively represented by Formulae (5.17) and (5.18).

$$SomF((a_{i,sts} = dn \wedge a_{j,sts} = dn) \vee a_{k,sts} = dn) \quad (5.17)$$

$$SomF(a_{i,sts} = dn) \quad (5.18)$$

The so-produced models are ready to be checked by Zot.

Chapter 6

Deployment-oriented Model Transformation

A second translation is available for the HRC-TEAM profile to obtain a task deployable either on a real system or on a simulator. The process, introduced in Section 6.1, involves the generation of configuration files, discussed in Section 6.1.1, based on the UML diagrams which are then fed to a FB network in charge of creating the actual IEC 61499-compliant application, as explained by Section 6.1.2.

6.1 Translation Procedure

This translation procedure aims at processing diagrams developed with the HRC-TEAM notation in order to produce a version actually deployable to real resources. Under these premises, the IEC 61499 - ROS hybrid architecture introduced in Section 3.2.3 is resorted to as target infrastructure. Due to higher level of complexity of the matter and wider range of tools required by the deployment procedure, a less direct translation procedure has been envisaged. Data contained by the diagrams are reformulated using a custom notation into configuration files. The latter are subsequently fed to a Function Block network in charge of actually creating the application equivalent to the original *Activity Diagram* modeling the task. The process is summed up by Figure 6.1.

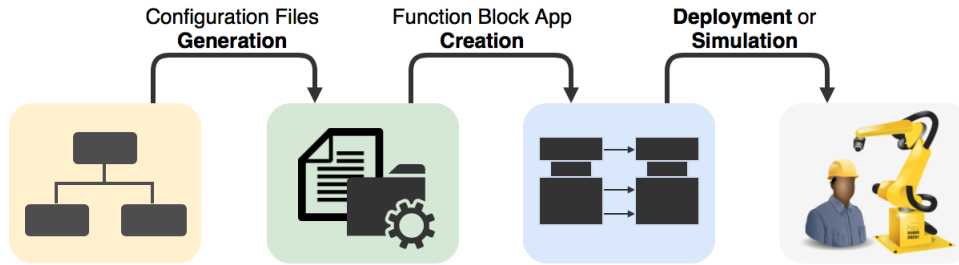


Figure 6.1: Scheme representing the procedure to translate HRC-TEAM diagrams into deployable FB applications. Firstly, custom configuration files are generated based on the diagrams content, and later fed to a FB network in charge of generating the actual application. At this point, both deployment on real resources and simulation are conceivable.

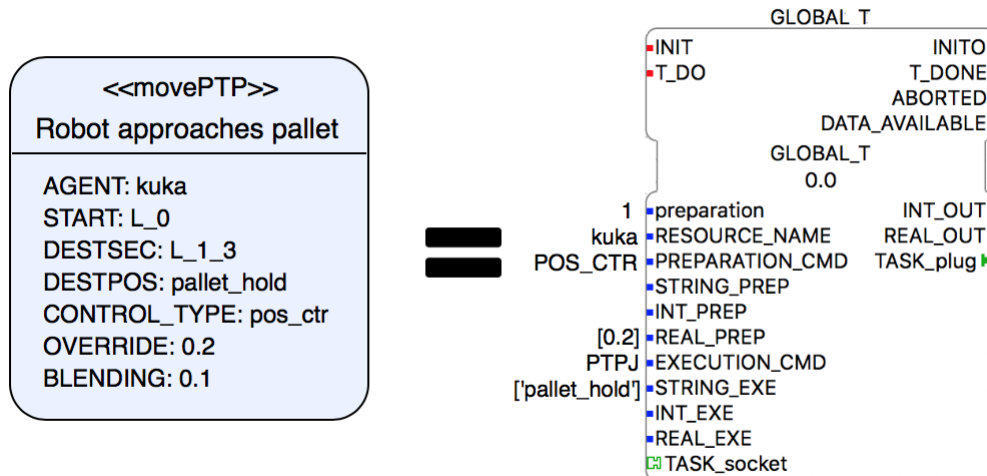


Figure 6.2: Example of how an *OpaqueAction* is transformed into a FB. The target FB class is `GLOBAL_T` (see Section 3.2.3), with a name arbitrarily set equal to the Action's informal one. The FB input parameters values are obtained from the block's *stereotype* for the `EXECUTION_CMD` (in this case `movePTP` corresponds to `PTPJ_TRAJ`) and from its list of properties. For instance, `CONTROL_TYPE` corresponds to `PREPARATION_CMD`, and the move parameters (destination and speed, i.e., `OVERRIDE`) provide values to `STRING_EXE` and `REAL_PREP`.

6.1.1 Configuration Files Generation

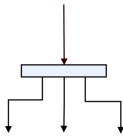
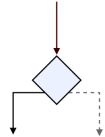
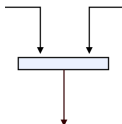
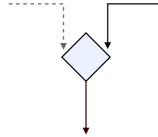
Similarly to the procedure described in Chapter 5, the information gathered by means of the interface is used to search the project for the desired *Class*, *Component* and *Activity Diagrams*. The objective is to create three files that will then be examined to generate the *FB Application*. This set of files uses a custom text-based language to describe HRC-TEAM diagrams elements which are instrumental to the generation of the Function Block network.

The first file collects data regarding the set of Actions and the task in general terms. Firstly, it mentions how many resources are involved, the names of all the agents and the name of the *Activity* (see the *HowManyRes* portion in Listing 6.3). This information is necessary to correctly generate the *Header* FB and set its parameters concerning resources and application name. As for Actions, each *OpaqueAction* will be matched by an equivalent Task-FB, as shown by Figure 6.2, carrying the same ID, agent, and parameters derived from the applied stereotype and its properties. A detail that must be considered is the management of actions inside loops: for each iteration of the same *OpaqueAction* there is a different *Action* but obviously they have the same ID. This is not admissible since the arrow must start from the precise action of the correct iteration, that is why a modification was added to the fulfillment of the address: an integer, representing the iteration at which the *Action* is associated, is affixed after the ID.

Besides FBs representing actions, also logic connectors featured by the *Activity Diagram* must be converted so that logic conditions governing actions state evolutions are properly preserved. To this aim, a set of existing blocks working as logic operators is exploited as conversion target, as illustrated by Table 6.1. The second file, a segment of which is presented in Listing 6.2, enlists nodes IDs, properly modified as previously explained in case they belong to a *LoopNode*.

Once all blocks have been properly handled, it is necessary to store information about the connections that involve them. In this case, every *Edge* contained by the *Activity Diagram* will correspond to an arrow in the FB network. Consequently, the last file contains the list of *Edges* with correlated properties, as can be seen in Listing 6.1: the IDs, the source and target types, plus –only in case they are *OpaqueActions*– the state related to the source and the command for the target.

Table 6.1: Connectors Function Block counterparts

	ForkNode	DecisionNode	JoinNode	MergeNode
Logic Connector				
Function Block	<p>ParallelFB</p> <pre> ForkNode INIT INITO T_DO_IN T_DO1_OUT T_DO2_OUT ParallelFB 0.0 TASK_s TASK_p1 TASK_p2 </pre>	<p>ForkFB</p> <pre> DecisionNode INIT INITO T_DO1_IN T_DO1_OUT T_DO2_IN T_DO2_OUT ForkFB 0.0 TASK_s TASK_p1 TASK_p2 </pre>	<p>JoinFB</p> <pre> JoinNode INIT1 INITO INIT2 T_DO T_DONE1 T_DONE2 JoinFB 0.0 TASK_s1 TASK_p TASK_s2 </pre>	<p>JoinForkFB</p> <pre> MergeNode INIT1 INITO INIT2 T_DO T_DONE1 T_DONE2 JoinForkFB 0.0 TASK_s1 TASK_p TASK_s2 </pre>

Listing 6.1: ArrowFile

```

1 Arrows
2
3 SourceID:
4 _JgyoYBXkEeentfMxR4DUhw1
5 SourceType:
6 OpaqueAction
7 TargetID:
8 _Jgv1EBXkEeentfMxR4DUhw1
9 TargetType:
10 OpaqueAction
11 SourceState:
12 done
13 TargetCommand:
14 SOFTstop
15 END
16
17
18
19
20

```

Listing 6.2: LogicFile

```

1 Connectors
2
3 JoinNodes
4
5 ID:
6 _G68yEBX1EeentfMxR4DUhw1
7 END
8
9
10
11
12
13
14
15
16

```

Listing 6.3: OpActFile

```

1 HowManyRes
2 4
3 kuka
4 gripper
5 screwdriver
6 yum
7
8 TaskName
9 caseA
10
11
12
13
14
15
16
17
18
19
20

```

6.1.2 Application Generation Mechanism

The strategy that has been adopted to bridge between the UML model and IEC 61499 application mostly revolves around *reconfiguration* features [86]. In order to limit the amount of manual input needed from the user to the sole diagrams, the Function Block application modeling the task is automatically generated based on the configuration files content. This is achieved by feeding the files paths to a fixed function block *network* that holds full responsibility for the generation process. By doing so, the user is only required to set the aforementioned parameters and trigger the sequence activation.

The network's main components are a custom-built Task-Generating FB and a set of standard blocks from the reconfiguration library, provided by the IDE, as shown in Figure 6.3. In general terms, the TaskGen-FB is in charge of producing events in a suitable order, whereas the web acts a dispatcher so that each emission produces the appropriate result.

Blocks belonging to the aforementioned library are Service Interface Function Blocks, each in charge of a distinctive operation pertaining reconfiguration. A fundamental remark about these SIFB is that their employment does not have any graphical repercussion, meaning that they can produce elements which are fully functional from FORTE's point of view but not their graphical counterparts. It follows that deployment of an application which is partially *invisible* may be a dis-

advantage for the user who cannot directly intervene to make modifications or have a clear visual feedback of the system's evolution. Nevertheless, in this regard it is necessary to remark that this purpose is already served by UML diagrams, which the FB application is supposed to replicate exactly. The available functions are:

- **creation of a new Function Block:** the required parameters concern the class of the FB to be instantiated and the informal name that will be assigned to it;
- **activation of a new Function Block:** each newly created FB must be *started* in order to work;
- **setting a parameter:** in this case, it is necessary to input the involved FB's name, the parameter's name and the desired value. Particular attention must be paid to the parameter's type so that the adopted syntax complies with the standard notation, otherwise the operation will not have a successful outcome;
- **creation of a new connection:** when constructing a FB network, creating proper connections is usually an essential step. Therefore, this block requires as input the names of the two involved FBs and of the output and input events (or parameters) which will be connected;
- **event trigger:** having this option is indispensable since performing it manually may not be feasible.

Further options are available but are not implemented in the current version of the tool. These are basically the mirrored versions of the functions that have been listed but in charge of *cancelling/deactivating* instead of creating, e.g., *deleting* a connection or *killing* a Function Block.

It is equally important to analyze in detail the pivotal element of the application, i.e., the TaskGen-FB SIFB, displayed in Figure 6.4. The required input parameters concern the conventional *activation* boolean variable, $QI()$, and three strings corresponding to the configuration files paths, as already mentioned. The output parameters are a union of the data required by the previously described SIFBs to perform their operations, and are thus accordingly connected. It possesses four input events:

- **INIT:** block activation event, manually triggered;
- **START:** initiates the file-reading phase;
- **STOP:** interrupts the file-reading;
- **REQ:** elaborates the currently required command and issues the appropriate output event.

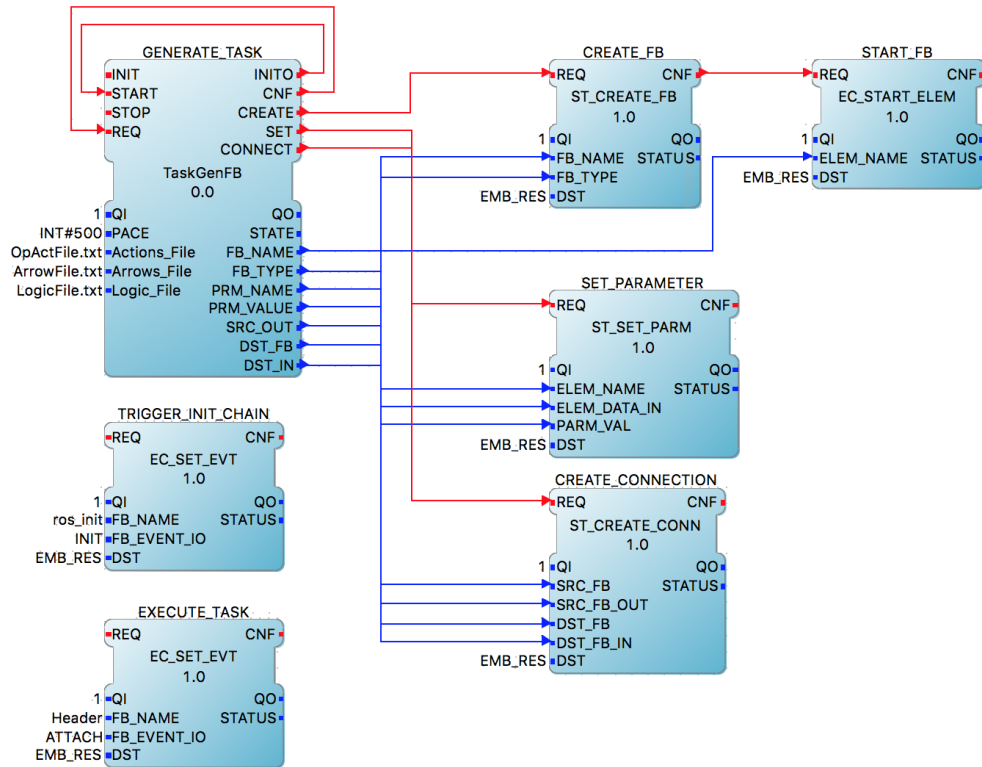


Figure 6.3: Screenshot of the FB-App Generating Network. TaskGen-FB is in charge of issuing commands (CREATE, SET and CONNECT) and updating parameters at proper times. Such events are then fed to the network and appropriately dispatched to the related branches. In particular FB-creating and -starting Reconfiguration Blocks (ST_CREATE_FB and EC_START_FB) are triggered by the CREATE event. ST_SET_PRM, which sets a parameter value, is activated by SET, whereas the block in charge of creating connections (ST_CREATE_CONN) is activated by CONNECT. The two EC_SET_EVT blocks are necessary to initiate all blocks and trigger the actual task execution (Header’s ATTACH event).

and five output events:

- **INITO**: signals the correct activation of the TaskGen-FB block, optionally connected to the input **START** if the user does not wish to separate activation from reading phase;
- **CNF**: issued once a command is ready to be processed, hence connected in retroaction to **REQ**;
- **CREATE**: the first of the three feasible commands, related to the creation of a new FB. The subsequent initialization is automatically handled by the reconfiguration network;
- **SET**: commissions the setting of a parameter;

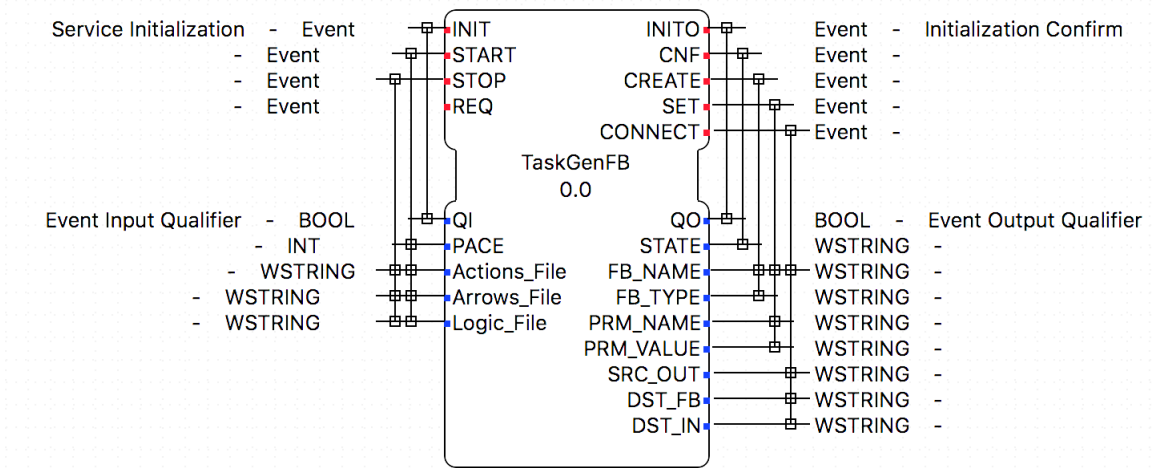


Figure 6.4: TaskGen-FB block structure. It receives as input events INIT (the only one which needs to be triggered by the user), START and STOP (referred to the *reading* phase), and REQ, which processes an internal command request. The output events are INITO, which is connected in retroaction to START so that a second manual trigger is not needed, CNF, also connected to REQ, and CREATE, SET and CONNECT which are the three generation commands. The event-parameters connections are also shown: for instance, the CREATE command requires a FB_NAME and a FB_TYPE (its class). The required input parameters are the standard activation boolean $QI()$, the procedure $PACE()$, and the three configuration files paths.

- **CONNECT:** similarly leads to the creation of a connection.

The block is endowed with an additional output parameter carrying information about the state of the block, useful to monitor whether the process is correctly unraveling or an error has occurred.

In order to work out a methodology as efficient as possible, the possibility to implement *multithreading* has been exploited. According to [87], a thread is an independent *flow of control* in charge of a sequence of execution commands. Interaction among different threads is allowed and encouraged since it may reduce the complexity of the program and improve performances. Even on non-multiprocessor machines, *parallel execution* may boost response time and throughput thanks to execution overlaps and communication. Nevertheless, multithreading poses some challenges in terms of expertise with creation and termination of threads, synchronization and deadlock detection. The order in which operations are executed is a key factor, as well as the management of access to shared data, hence several mechanisms have been devised to tackle these issues. Inter-thread communication may also give rise to problems [88] since interruption may unpredictably occur while a shared variable is being updated, hence causing other threads requiring the access to the same data set to work with out-of-date values and to thus be prone to erroneous behavior. In any case, multithreading in C++ must be managed through dedicated libraries: for

this project, *Boost.Thread*¹ has been selected among the available ones.

The way multithreading is implemented in this translation is by breaking reading and issuing commands into two different control flows. If the two operations were performed by the same thread, after each output command it would have been necessary to store the last examined line, resume reading the file and restart issuing commands only after having gone through the portion of the file that has been previously processed. This approach, although functional, is clearly not the most efficient. Having a parallel thread allows the two processes to be decoupled, so that the emission of a command does not force the file processing to undergo drastic breaks. The parallel thread is initiated once TaskGen-FB receives the START event.

The internal thread commissions the issuing of an output event –among CREATE, SET and CONNECT– at proper times whereas the main thread is in charge of actually putting it into practice. By choosing this path, the main thread processes the command as soon as it is requested without causing any error or misbehavior. Alternatively assigning this task to the internal thread causes the creation of an event queue which is disposed of only when the *internal thread* is terminated. On the other hand, parameters updates have an immediate effect under any circumstances, hence the command emission takes place only taking account the set of values present at thread termination, producing an unacceptable behavior.

With the chosen approach the internal thread is not subject to termination but it does require a *micro* interruption, applying a *monitoring* mechanism similar to a simplified version of a *semaphore* [88]. As a matter of fact, as already explained, parameter updates are instantaneously effective, whereas the command emission process does take some time in the order of microseconds. The parameter *PACE()* (see Figure 6.4) sets the pause duration in μs , allowing a correct functioning without altering the perception of the process or causing a perceivable slowdown. This can be modified by the user in case he/she is interested in slowing the procedure down (hence increasing the parameter value) for testing purposes.

In order to analyze the generation process more in detail, the logical mandatory steps to follow are:

- GS.1 **generation and activation** of action FBs
- GS.2 **parameter setting**
- GS.3 **generation and activation** of conditional FBs
- GS.4 **creation** of connections

¹Documentation available at http://www.boost.org/doc/libs/1_64_0/doc/html/thread.html

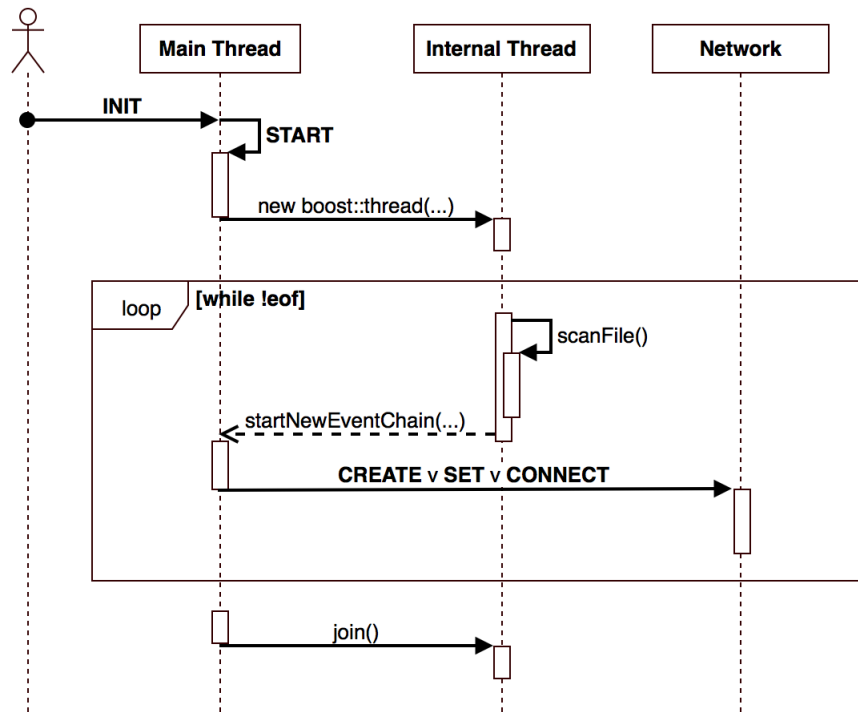


Figure 6.5: Pseudo-Sequence Diagram representing how the multithreading framework functions. The user (left-most lifeline) is only required to trigger TaskGen-FB INIT event. This initiates the main FB thread, hence self-triggering the START event if the feedback connection is present. At this point the parent thread creates a parallel one (*Internal Thread*) with the *boost :: bind* method which starts scanning the configuration files (*scanFile()* method). Once a scanning sequence is complete, it instructs the main thread (*startNewEventChain()* method) to issue the proper command (CREATE, SET or CONNECT) which is handled by the Generating Network. This sequence is iterated until the configuration files have been fully scanned, after that the main thread terminates the parallel one (*join()*) and itself, finalizing the generation process.

Changing this order may cause inconsistencies, in particular post-poning GS.1 and GS.3 with respect to GS.2 and GS.4: the application clearly does not allow customization or connection of blocks which do not exist yet.

The actual processing strategy consists of acquiring the content of the configuration files with pre-determined *interpretation keys*. For instance, when the key-phrase *Stereotype name* (see Listing 6.3) is acquired, the content of the following line is stored as value of the PRM_VALUE parameter, whereas PRM_NAME is set to EXECUTION_CMD by default, all in order to gather the type of the action FB that is under definition. In particular, GS.1 and GS.2 are performed based on the content of Actions_file, GS.3 with Logic_file and GS.4 with Arrows_file.

Chapter 7

Tools and Experimental Validations

This Chapter presents the implementation of the previously introduced procedures dedicated to design, verification and deployment of a collaborative robotic task. Section 7.1 presents the Papyrus tool, by which the user manages the UML diagrams, and the interface of ConverTEAM that allows him/her to automatically obtain inputs for Zot and the FB application. Finally, the translation mechanism is applied to the case study to prove its efficiency: in Section 7.2, the procedure to create the TRIO logic expressions and to retrieve the formal verification results is presented, whereas the generation of the configuration files and their usage for the deployment of the task is validated in Section 7.3.

7.1 Software Tools

7.1.1 Modeling Tool

Papyrus¹ is an Open Source Model-based tool, developed on the Eclipse platform and licensed under the EPL (Eclipse Public License). It can either be used as a standalone tool or as an Eclipse plugin and provides support for UML profiles, Domain Specific Languages and SysML, being a graphical editor for UML2 as defined by the Object Management Group (OMG). Since every part of Papyrus is designed to be customizable, it has been possible to adapt the environment and introduce elements that came in use for the scope of the project: e.g., the button to launch ConverTEAM, the custom palette to easily choose the stereotyped *OpaqueActions* for the *Activity Diagram*, as shown in Figure 7.1, or the definition of how to show

¹Complete documentation can be found at <https://wiki.eclipse.org/Papyrus>.

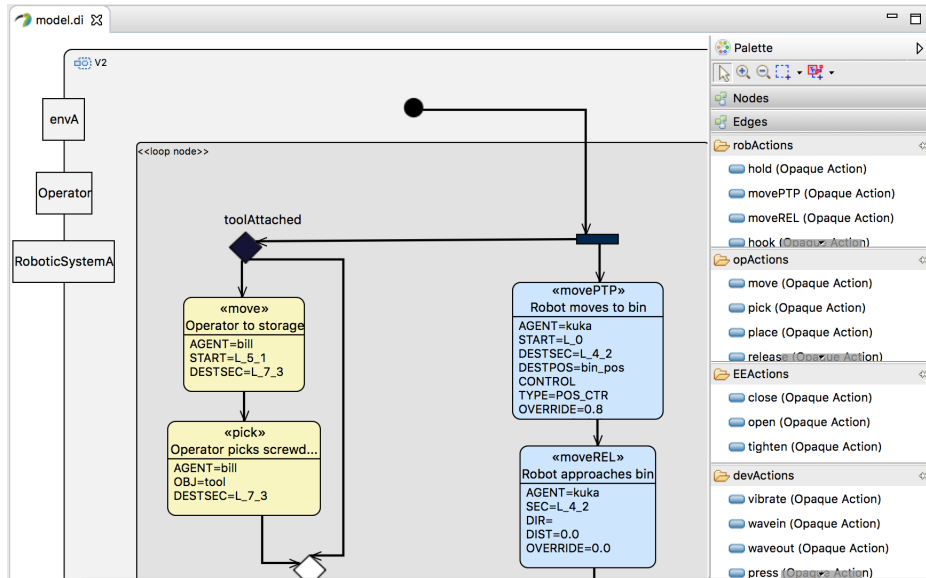


Figure 7.1: Papyrus interface screenshot capturing the customized palette. The top two compartments, *Nodes* and *Edges*, are present by default, whereas the ones corresponding to notation-specific sets have been operatively plugged in.

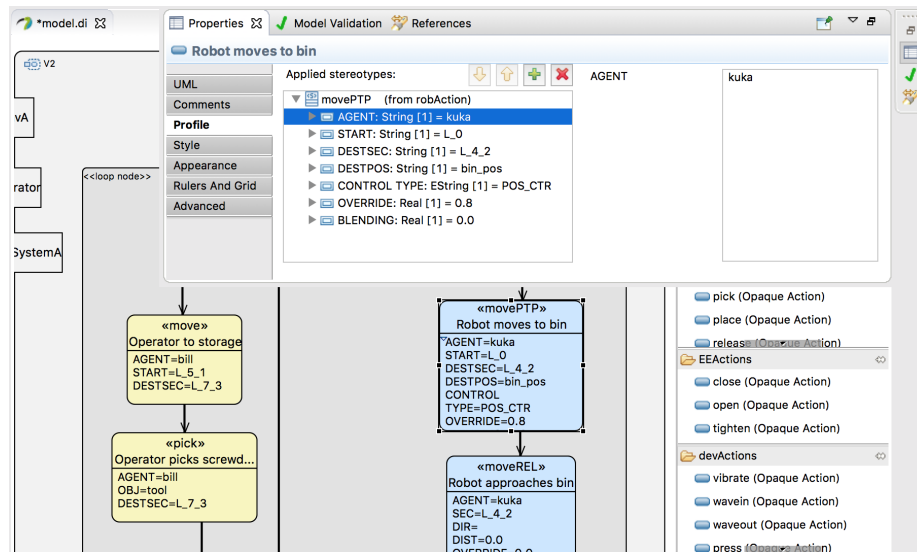


Figure 7.2: Papyrus interface screenshot capturing the profile refinement options. The box in the middle displays the applied stereotype with correlated properties list, and the one on the right allows the user to type in the value for the selected parameter. Stereotypes can be added or removed using the above buttons featuring plus and minus signs.

stereotypes properties and items in the different diagrams (see both Figures 7.1 and 7.2). Furthermore, after having added a new action in the *Activity Diagram*, its attributes can be filled out by means of the *Property* window visible in Figure 7.2.

7.1.2 Transformation Tool

The beginning of a ConverTEAM procedure involves the request of some details to the user and the most fitting tool to deal with this kind of communication is a visual interface. ConverTEAM can be directly launched from the Papyrus interface or, equivalently, as an external executable file. In any case, it is always indispensable to determine which of the four possible toolchains is interested by the current execution. Therefore, the first phase of every toolchain is always represented by a main window with four buttons, visible on the left-hand side of Figure 7.3. Each button leads to a secondary window in which it is possible to fill out all the missing information.

The first button, *ZOT Files Generation*, leads to the *Model generation* window, which is extensively displayed in Figure 7.3. It collects all the specific aspects related to SAFER-HRC models generation which cannot be retrieved from the HRC-TEAM diagrams. In particular, the user has to select the paths to recover the UML model and to set the location of the printed files; then he/she chooses, from a list of available *Activity* names, the one that has to be verified. It is also possible to specify the time bound of the simulation and the completion condition that will be used to evaluate the correct execution of the experiment. Finally, he/she can open the *Layout* subordinate window dedicated to the establishment of the forbidden sections for each agent. Once every request is satisfied, the generation of the files can be launched and, if no error occurs, a pop-up message will communicate the correct termination of the procedure. The generated files are now ready to be fed to Zot.

Then, when the results of the formal verification are returned to the user, the second and the third buttons of the main window can be pressed by the user to obtain a better readability for the Zot output. Basically, the two buttons provide the user with a way to re-parse the information contained by Zot outcome. *Textual Parsing* opens a framework in which, again, the UML project location and the desired *Activity* name are specified, but it is additionally requires to locate the Output file and to indicate a location for the new file, an example of which is visible in Listing B.3. A more sophisticated output is linked to the *Graphical Parsing* button, whose window does not require any details about the UML diagrams, but only the location of the Python script and the position for the folder that will contain the generated images.

Finally, as far as the generation of the configuration files is concerned, the con-

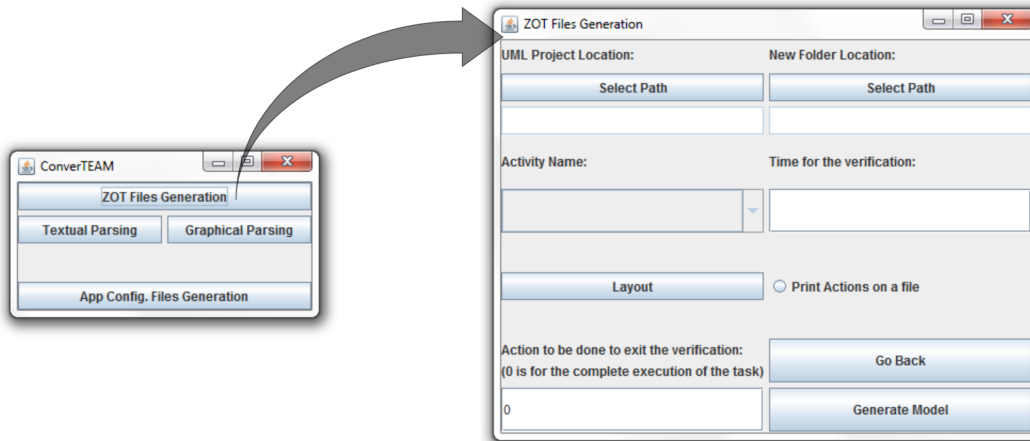


Figure 7.3: The main window, visible as soon as ConverTEAM is launched, is visible on the left-hand side. The first button is related to Formal Model generation. On the right the secondary interface can be seen, featuring the information required by user for the automatic procedure. Moreover, the *Layout* button can be pressed to specify the forbidden sections for each agent and the *Print Actions on a file* can be ticked in case a further debugging file is needed.

tribution required to the user is fulfilled by means of a different secondary window, linked to the *App Config. Files Generation* button of the GUI, shown in Figure 7.4. In particular, he/she is asked to indicate the path of the UML project, the path to the folder in which the Configuration Files will be created and the name of the *Activity* whose workflow and parameter must be utilized in the generation. Once the prerequisites are all filled in, they are retrieved and managed to perform the automatic procedure.

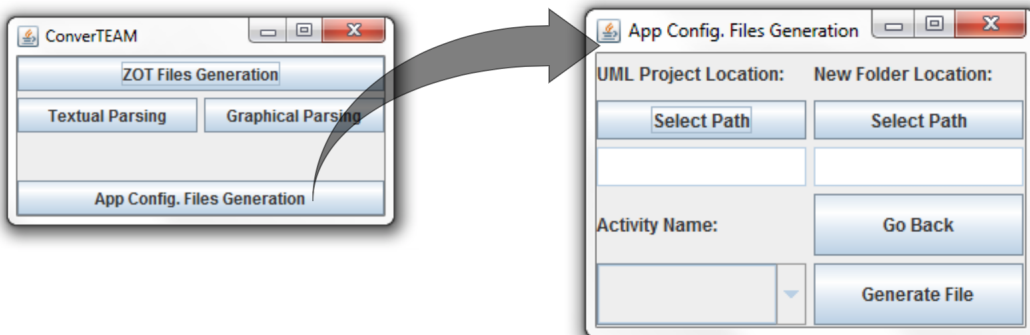


Figure 7.4: The main ConverTEAM window is visible on the left-hand side. The button on the bottom leads to Configuration Files generation. The secondary interface on the right contains information the user has to fill out before proceeding with the automatic procedure: the location of the UML project on his device, the name of the interested *Activity* and the folder location for the new Configuration files.

7.2 Validation of the Formal Verification Process

The transformation procedure has been applied to the case studies introduced in Section 4.5 to test its proper functioning. Subsequently, a series of experiments has been run using the model-checker ZOT to highlight eventual critical situations raised by the modeled tasks. Firstly, the user inputs the .XMI files paths to the GUI and selects the activity names he/she wishes to verify: in this case, *caseA* and *caseB*. Afterwards, it is possible to set the specific experimental parameters: the values picked for the experiments that will be relayed in the following paragraphs will now be listed for the sake of repeatability. Case A has been verified with a *time bound* equal to 60, and an exit condition requiring the execution of the whole task, as in Formula (7.1). As for spacial constraints, the operator's body is banned from entering sections L_0, L_{12}, L_{13} and L_{42} (see Figure 4.16 for reference). Similarly, the robot cannot reach section L_{51} due to workspace limitations. In order to keep the experiment duration under a manageable time threshold, only two loop iterations have been simulated. The setting was also chosen to allow for a comparison of the formal model generated with the new Papyrus-based tool with the manually-generated one. Moreover, *DecisionNodes*' values have indeed been manually forced in order to simulate the *worst* cases, i.e., the ones more likely to give rise to hazardous situations. For this reason, the two selected versions are V1 and V2 (*selectTask1* = 1 and *selectTask2* = 2), whereas V3 is left unverified since it does not directly involve any collaboration, hence hazards could only emerge due to operator's reckless decisions.

$$\text{SomF}((a_{18,sts} = \text{dn} \wedge a_{6,sts} = \text{dn}) \vee (a_{60,sts} = \text{dn} \wedge a_{50,sts} = \text{dn}) \vee a_{38,sts} = \text{dn}) \quad (7.1)$$

The experiment involving Case B has been run with a time bound of 90 units, and –again– the complete exit condition in Formula (7.2). In this case, the robot cannot enter the pathway created to let the operator roam freely, i.e., sections $L_{61}, L_{62}, L_{63}, L_{64}$ and L_{65} . The operator's body cannot occupy *occluded* sections, that is to say $US = L_{12}, P = L_{21}, TC = L_{33}$ and $B = L_{52}$. The loop iterations have also been reduced to 2 each, despite the unreasonableness of these values, to avoid overly time-consuming and redundant simulations. Also in this case the decision point's variable has been manually forced to its "true" value, in order to simulate the worst case in which the operator decides to empty the bin (*emptyBin0* = 1).

$$\text{SomF}(a_{51,sts} = \text{dn} \wedge a_{55,sts} = \text{dn}) \quad (7.2)$$

As for the generated models, some segments can be found in Section B.2. For instance, Listings B.7 and B.9 provide examples of decision variable generation (the

first iteration of *selectTask* in Case A) and action block definition (action n.8 in Case B). The latter shows examples of workflow-related (the action has a single *Opaque-Action* predecessor) and stereotype-related guard conditions, which adhere to the rules illustrated in Section 5.1.

Both experiments were run on a Linux desktop machine with a 3.4 GHz Intel® Core™ i7-4770 CPU and 16 GB RAM. The experiments had an approximate time duration of 15min for Case A and 50min for Case B, for a total of 68 and 58 actions respectively. Both models were found *satisfiable* hence a simulation trace was returned.

ZOT Output Parsers

Before analyzing the actual simulation results, it is worth mentioning that ZOT produces a textual output organized as to collect all predicates and all variable values for each time instant. The result though has very poor readability which utterly compromises the user’s capability to interpret the content of the simulation. For this purpose, and also in support of testing phases, two parsing tools have been developed to allow a comprehensive but much easier understanding of the experimental results.

The first one has been developed in Eclipse with a dedicated custom Java class *Hist_Reader*. Its execution produces a textual file containing time units re-arranged as shown, for example, in Figure 7.5. Firstly, it lists actions in a relevant state, hence it skips the ones which have not started yet. Secondly it indicates the sections occupied by the operator and by the single robot components (Link1, Link2, EE). Finally, it lists highlighted hazards and the involved actions (see *warning* messages in Figure 7.5), and the active risk reduction measures, if any.

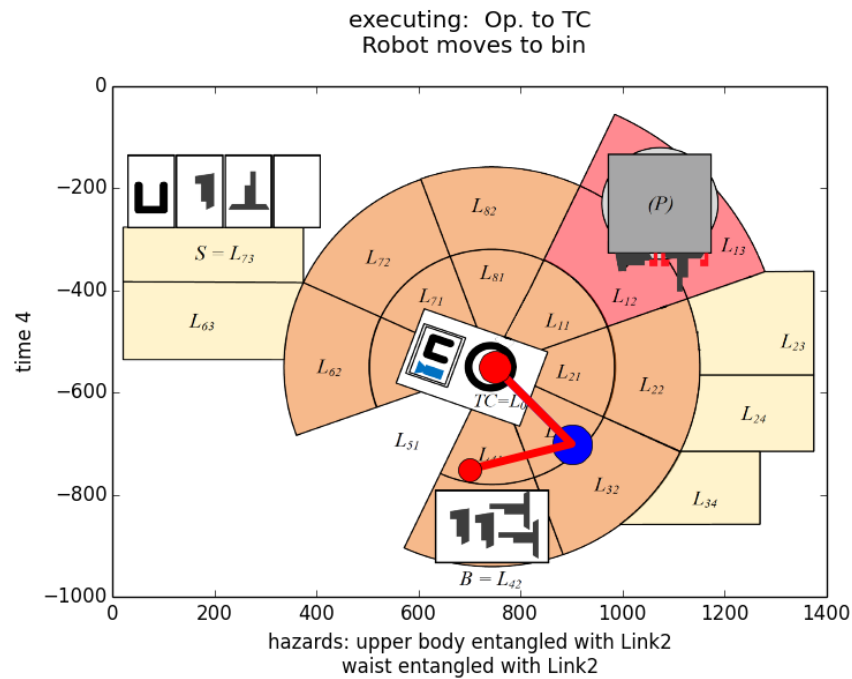
The second option has been developed as a Python² script, and aims at reinterpreting data in a graphical way. As a matter of fact, each time unit content is plotted and overlaid on the schematic layout representations in Figures 4.16 and 4.18. More specifically, it approximates the operator’s head as a blue dot (see, for example, Figure 7.8) and draws a line to the hand’s location if it is in an adjacent section. As for the robot, the base is represented by a large red circle and the End Effector by a smaller one, whereas links are drawn as straight lines connecting the occupied sections (again, see the red lines in Figure 7.8). Moreover, further information is provided in terms of plot labels. More specifically the highlighted hazards are listed as label on the x-axis, the time instant on the y-axis and the executing actions as title on top of the graph. The result is clearly a stylized representation, but it still

²The implemented version is 3.5.2, whose documentation can be found at <https://www.python.org/doc/>.

allows the user to notice unrealistic or imprecise configurations which are not easily detectable with the textual output.

Safety Assessment Outcome

The simulation has highlighted a series of hazardous situations for both cases: the most relevant examples will now be described. As one might expect, for Case A the most critical contingency is represented by the moment in which both agents are working in close proximity to each other while tightening the pallet fixtures. One of



(a)

```

----- time 4 -----
Action 23 executing      -Op. to TC 1-
Action 39 executing      -Robot moves to bin 1-

Operator in L_0 L_3_1
Robot in L_4_1 (EE) - L_3_1 (Link1) - L_3_1 (Link2) -

RRM *force decrease* active

----->WARNING: upper body ent with L2 in L_3_1
----->WARNING: waist area ent with L2 in L_3_1
----->WARNING: lower body ent with L2 in L_0
    
```

(b)

Figure 7.5: Detected Hazard n.1 (Case A): the operator is hit by the robot while reaching the storage $S = L_{73}$. Note that the model-checker has indeed activated a risk-reduction measure (force limitation).

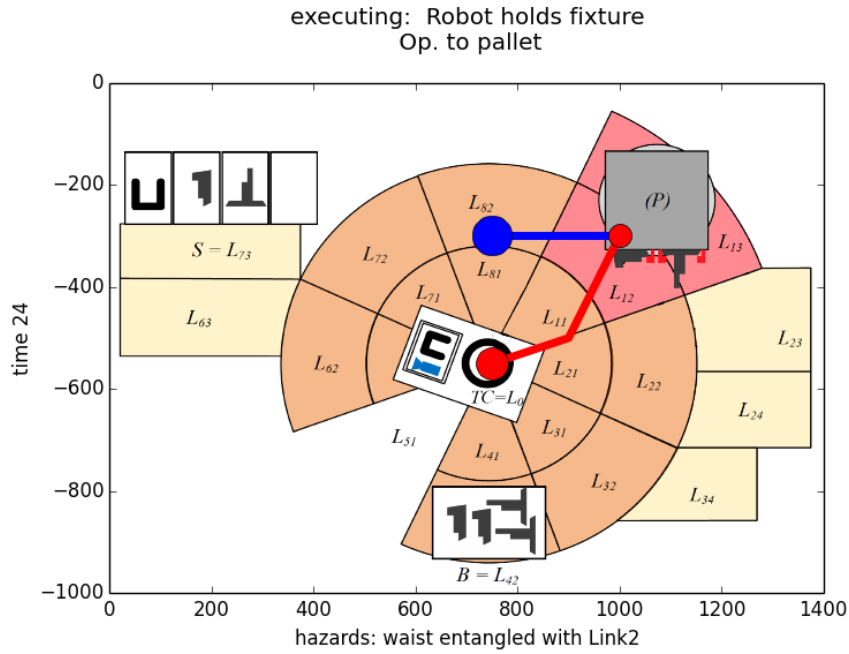


Figure 7.6: Detected Hazard n.2 (Case A): both agents are working on the pallet in close proximity to each other, hence a collision is highly probable.

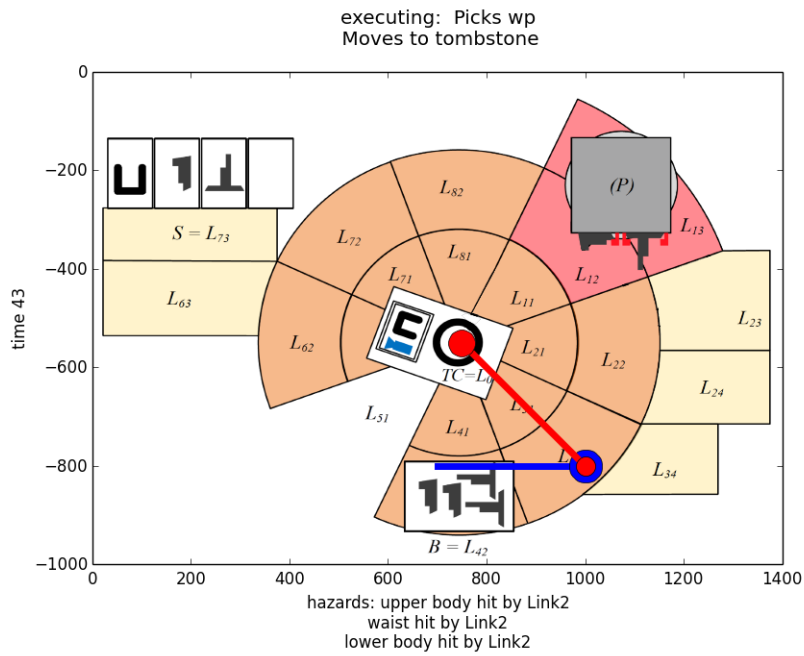


Figure 7.7: Detected Hazard n.3 (Case A): the operator is hit by the robot while reaching for a workpiece in the bin ($B = L_{42}$).

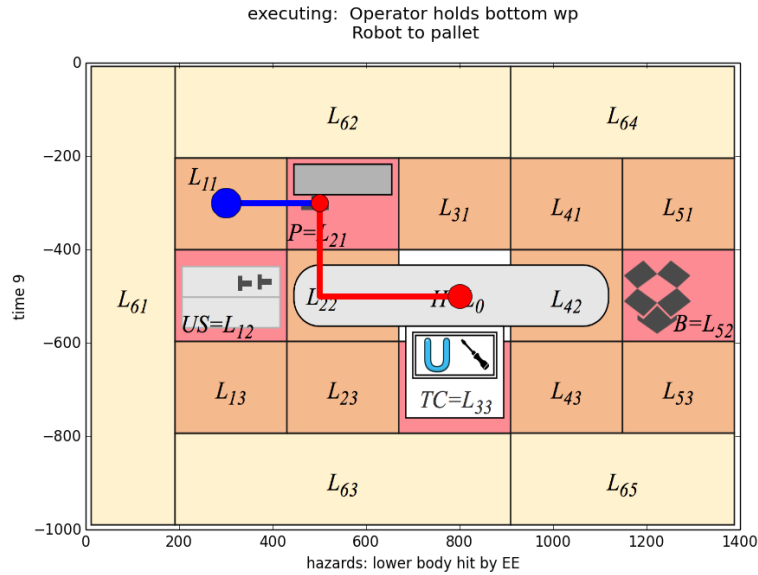


Figure 7.8: Detected Hazard n.1 (Case B): operator and robot are both working on the pallet close to each other, hence collisions are very likely to take place. Note that the seemingly unnatural configuration of the robot is due to the formal model’s partial support of prismatic joints (the conveyor belt in this case). Nevertheless, this does not impact the safety assessment result (the hazard would have still been detected with the base properly placed in L_{22}).

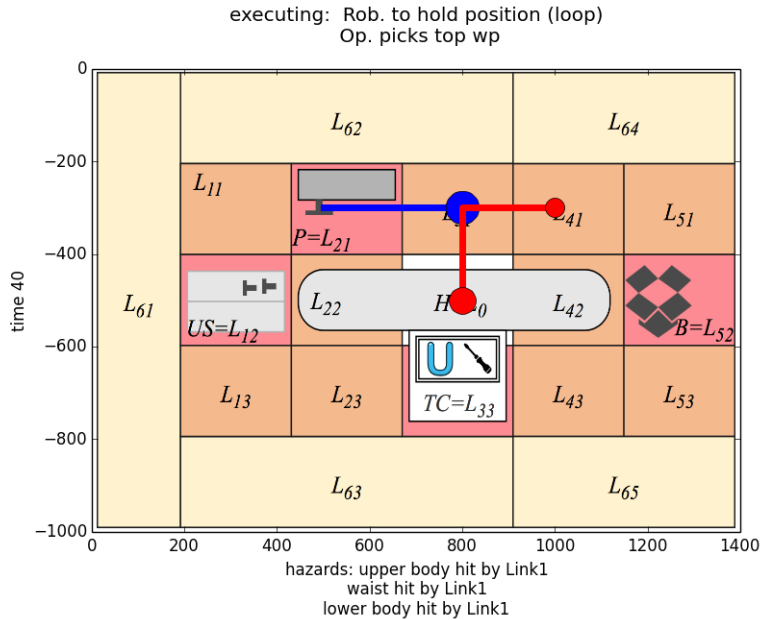


Figure 7.9: Detected Hazard n.2 (Case B): the operator is picking a workpiece from the opposite side and gets hit by the robot which is withdrawing to hold position after the untightening sequence.

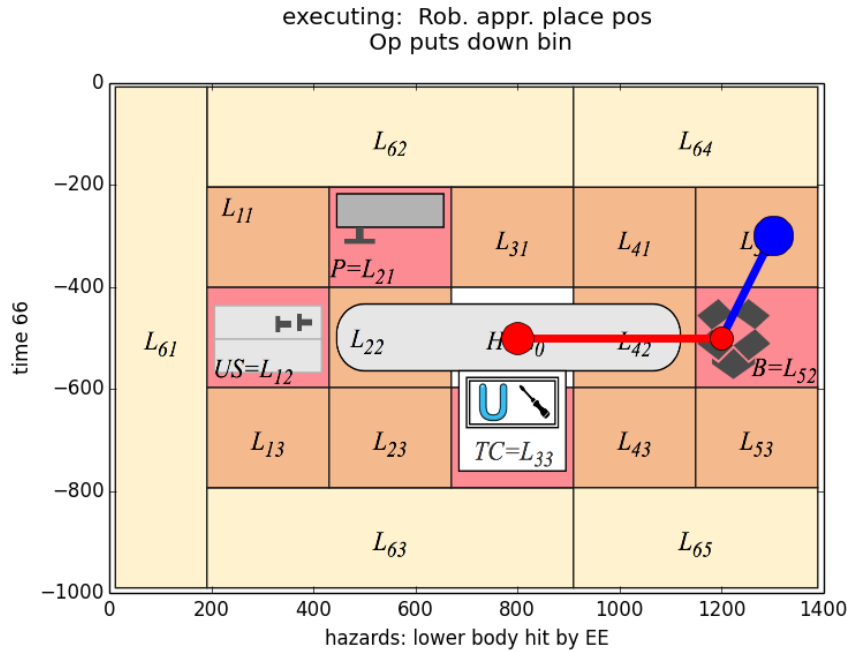


Figure 7.10: Detected Hazard n.3 (Case B): the operator is placing the bin after emptying it while the robot is approaching the same area to deposit a container, causing a collision.

the detected hazards in this context is pictured in Figure 7.6: the robot is holding the fixture in place, and the operator is approaching the pallet hence causing a collision with Link2. In this case a bulletproof risk reduction measure is hardly conceivable, since close proximity is indispensable to reach the task goal. It is reasonable to conclude that, since this hazard will always be possible, the only approach to minimize its severity is to limit the robot's speed and force and instruct the operator to act with as much alertness as possible.

Other significant detected situations are pictured in Figures 7.5 and 7.7. The first one concerns a time unit in which the robot is reaching the bin to grasp a workpiece and the operator is moving to fetch the screwdriver from the storage, and their trajectories intersect in between Sections L_0 and L_{31} . In this case, as Figure 7.5 shows, the model-checker has already activated a risk reduction measure to limit the robot's force, but an additional degree of safety might be added by instructing the operator to take the other way around to reach the storage (i.e., passing through Section L_{81} or L_{82}) since the bottom sector is needed by the robot to execute its operations. The last given example captures the opposite situation, i.e., the operator is reaching for a workpiece in the bin whereas the robot is approaching the pallet to tighten the fixture. Therefore, as Figure 7.7 shows, the operator is hit by Link2 and the End-Effector. Similarly to the previous case, instructing the operator to get close

the bin from the other side (i.e., Section L_{51}) might solve the issue. Alternatively it may also be possible to re-plan the robot's trajectories so that they do not cross such critical sections, but the pros and cons of this option must be weighted on a case by case basis.

As already mentioned, it is possible to compare these results to the ones obtained from experiments run with manually-generated model. More precisely, such conclusions can be stated only after several iterations of the formal verification procedure are performed on both models, since the simulation of the task is stochastic and may provide different results even re-using the exact same files. Under these assumptions, the comparison shows that the new tool keeps the same level of accuracy in the detection of hazards, while making the design process easier for safety experts and less time-consuming. Among the produced execution traces of the two models, the detected hazardous situations were the same and every reported warning is meaningful and related to impacts and entanglements between operator and robot.

On the contrary, as far as Case B is concerned, the UML diagrams are created *ex-novo* so that it is possible to test HRC-TEAM and, consequently, the translation toolchain from a different point of view. As a matter of fact, since a manually-created formal model is not available for this case study, the point, in this case, is to understand whether the Abstract Model and ConverTEAM are able to lead to the detection of new meaningful and consistent hazards. Results show that the interval in which operator and robot are both working on the pallet in Section L_{21} raises criticalities. As pictured in Figure 7.8, while the operator is keeping the top workpiece still his lower arm is hit by the robot which is approaching the pallet. The same conclusions as for Case A about this contingency can be drawn also for this second case. In addition, Figure 7.9 captures a further critical situation due to the robot's moving to return to holding position and the operator choosing to pick the top workpiece standing in Section L_{31} . In this case multiple solutions are conceivable: firstly instructing the operator to avoid crossing such section unless strictly necessary. In case this excessively impacts on efficiency (i.e., if they are working on a fixture slot which is closer to the pallet's right side, this might result overly uncomfortable), the robot's trajectory could be redesigned. Ultimately, an additional interrupt might be added to the model to increase the operator's control on the robot's movements: for instance, a second gesture recognition routine could be programmed so that the operator can trigger the robot's withdrawal only after he/she is done with removing workpieces. Finally, the third captured hazard, depicted in Figure 7.10, is caused by the operator choosing to empty the bin and bring it back to position while the robot is approaching the area to deposit one of the containers. This clearly causes a collision between the End-Effector and the operator's lower arm area. In this case, a further possible solution is the implementation of a vision control system to detect whether the operator is hindering the path to the bin and issue a command to

prevent the robot from moving and causing any harm. In any case, the advantages and eventual disadvantages brought by these fixes need to be evaluated by the user as he/she revisits the model.

7.3 Validation of the Deployment Process

The effectiveness and soundness of the application generation process has been tested against the Case A, V2 task model (the related *Activity Diagram* can be seen in Figure A.7). In the absence of material resources, a simulation environment has been envisaged in order to prove the procedure's correctness. Prior to the introduction of the simulator, it is necessary to discuss how the preceding steps, as pictured in Figure 6.1, apply to the specific case study.

In order to avoid unnecessary redundancies, the number of loop iterations has been reduced to 2 through the modeling tool. ConverTEAM has been thus launched, as explained in Section 7.1.2, to generate the related configuration files. Due to the presence of multiple iterations, all elements required by the files are duplicated, with the IDs properly adjusted as explained in Section 6.1.1. Since only one version of Case A is taken into account, the *selectTask DecisionNode* is not involved in the translation, whereas *toolHeld* is still required. Therefore, all four logic connectors are featured by the example. Furthermore, the example allows state-based command stereotype management to be tested also in the deployment-oriented environment, since two stereotyped arrows (one with `<< executing, start >>` and the other with `<< done, SOFTstop >>`) are featured by the chosen *Activity Diagram*.

The so-generated files are fed to the fixed application-generating FB network, pictured in Figure 6.3, and the generation process is initiated by manually triggering TaskGen-FB INIT event. This operation is performed through the 4DIAC-IDE 1.81

```
[ INFO] [1498568137.806988480]: executing: (Robot) hold
[ INFO] [1498568137.809497625]: executing: (Operator) tighten
[ INFO] [1498568137.936903809]: bill: done
[ INFO] [1498568137.937037630]: executing: (Operator) gesture recognition
[ INFO] [1498568137.956979645]: bill: done
[ INFO] [1498568138.087103466]: kuka: done
[ INFO] [1498568138.108564136]: executing: (Robot) moveREL
[ INFO] [1498568138.236881384]: kuka: done
```

Figure 7.11: Segment of ROS nodes feedback message during task simulation. The resource (*bill* and *kuka* in this close-up) sends a notification message when an action is *done* and when it starts *executing*. In the latter case, the node also communicates the action's stereotype (e.g., *hold* and *tighten*).

interface, with FORTE 1.8 M1 running on a Linux virtual machine. With parameter `PACE()` set to `500μs`, the generation process is perceived as almost instantaneous by the user. Through feedback messages printed in FORTE's shell, it is possible to verify that no error occurred during creation and connection phases. Therefore, the application is ready to be either deployed or, as in this case, simulated.

In a first step, virtual ROS nodes, representing resources required by this example (i.e., named *kuka*, *gripper*, and *bill*) are created and launched. Such nodes subscribe to the correlated topics where TaskFBs belonging to the generated application publish their control messages. Feedback notes are consequently generated in their respective shell which allow the user to check whether actions are being executed as requested by the UML model. In order to trigger task execution, firstly ROS nodes must be manually initialized, and then the Header's ATTACH event must be manually triggered as well.

In the specified experimental setting, nodes' responses, a segment of which is shown in Figure 7.11, whereas a complete iteration is shown by Listing C.5, show that the operator approaches the pallet while the robot approaches the bin, picks a workpiece and moves towards the pallet. While the robot's holding action is still executing, the operator tightens the fixture bolt and releases the robot only after completion. The sequence is repeated for the second iteration. Therefore, despite the clear limitations of this simulation technique (for instance, the necessary ROS nodes have to be specified by the user, which may not possess the necessary skills, and the information received as feedback is minimal), it still allows us to conclude that the generated application contains the correct actions and properly replicates the logic connections among them featured by the original *Activity Diagram*.

Chapter 8

Conclusion and Future Developments

8.1 Achievements

In this work we have introduced a tool-supported model-driven approach dedicated to modeling HRC applications. The HRC-TEAM profile handles UML diagrams enriched with suitable stereotypes that link all of them to each other: a *Class Diagram* and different *Component Diagrams* to describe the domain of the desired task, *Activity Diagrams* to define its workflow. Moreover, the ConverTEAM tool grants two parallel transformation toolchains, summed up by Figure 8.1, in order to generate a formal logic model based on the SAFER-HRC methodology and a set of files to configure a function block application compliant with the IEC 61499 standard. The first translation is used in the context of safety analysis to formally verify the safety conditions of the designed task, whereas the second translation grants direct deployment of the application on specific resources in a real environment.

Due to the graphical nature of the approach, the user is allowed to precisely design HRC applications in an easier, faster and more automated way. The UML notation lets users quickly modify existing designs and easily build new task models as variations of previous ones. Besides, even if they do not have an extensive background in formal modeling approaches, it is possible for them to instantaneously generate formal models to perform a formal verification of the task: the results can be used, together with internal consistency checks, to iteratively improve the design until a proper level of correctness and estimated risk is reached. Similarly, the user is not required to replay the modeling phase in order to create a deployable application, but the shift to the real environment can be automatically performed with minimum manual input and significant time savings.

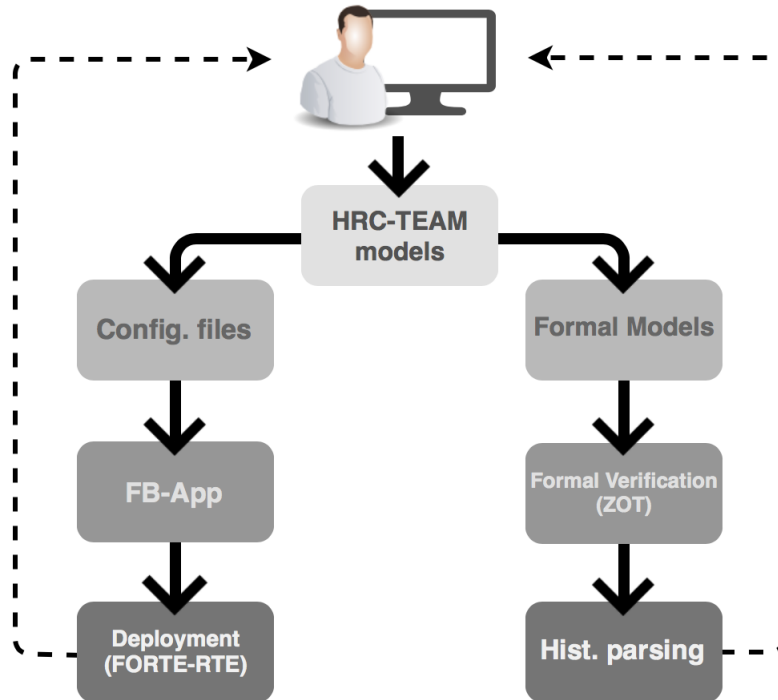


Figure 8.1: Final toolchain scheme. Each fragment represents a step of the procedure, starting from the HRC-TEAM models that are necessary to represent the collaborative task. Then, they are either transformed into the configuration files or the formal logic models, respectively re-parsed to generate the FB application or verified through the model-checker ZOT. Finally, the user can retrieve and parse the results of the verification to modify the Abstract Model, iterating the procedure until deemed necessary, and deploy the FB application on the real system or on the simulator.

8.2 Future Work

Besides the illustrated achievements and advantages brought by this work, there still is room for future improvements to enhance the aforementioned characteristics.

First of all, communication through the GUI should be developed in order to refine the management of preferred settings, for example by saving the selected sections forbidden to the agents or the different paths chosen to apply the procedure. Moreover, interaction with the user could be extended in order to further customize simulation settings, for example a-priori selecting the outcome of Decision Points.

Some features could also be added to the profile itself. For instance, reconfiguration and risk reduction measures are an important concept both for formal verification and IEC 61499 applications. Therefore, the profile could be extended in order to capture these alternative flowlines and the criterion behind respective

activations and dismissals. This could, for example, be achieved by deepening *DecisionNodes* characterization in order to endow them with significant conditions rather than their current non-deterministic connotation. However, the main improvement will regard the overall portability of the tool: HRC-TEAM should be turned into a rigorous UML profile. As a matter of fact, the user should be able to dynamically apply and un-apply the desired perspective to each of his/her models, rather than consistently having to replicate the Class Diagram in accordance with the template. Furthermore, the notation could also benefit from stronger inter-diagram connections. In fact, at current state, while it is necessary to have a dedicated stereotype for each of the Class Diagram Agents' operations if one is willing to employ them in the Activity Diagram, its actual application to the specific method does not have any practical effect. This implies that efforts could be put in the future into solving this issue.

As for the formal verification-oriented translation, improvements can be made about more complex robot architectures support. In the current version, the conversion tool is not able to handle systems composed by an arbitrary number of links or not classified as *antropomorphic* arm. As a matter of fact, SAFER-HRC only supports models containing two rotational links and one end-effector. Another enhancement could be the extension to more than two agent instances, since at this moment the model is able to manage only one Operator and one Robotic arm. Furthermore, although models obtained manually and through the automated ConverTEAM tool are currently comparable, they are not identical. To be more precise, an absolute 100% match will hardly ever be conceivable, but some work can be done towards closing this gap. A major step forward could be made by implementing *contextual variations* in relation to the generation of stereotype-related guard-conditions. In fact, right now the tool performs this operation by adamantly following fixed patterns (each stereotype corresponds to a fixed set of conditions). This feature could be improved by having it dynamically take into consideration contextual factors (e.g., which actions could be simultaneously executing and which agents could happen to be nearby), similarly to what a human programmer is naturally driven to do while manually drafting the models. Finally, also indicators about physical and mental state of the operator, i.e., fatigue and awareness, which are already featured by HRC-TEAM models, could be transposed into logical terms and exploited by the formal verification. This improvement would add a further layer of realism and accuracy to the safety assessment results.

As for the evaluation of the approach in its entirety, bringing it to the attention of actual practitioners could provide a meaningful unbiased assessment of its overall utility. First of all, the efficiency of the model-to-deployment branch should be tested on material resources, in order to determine whether the shift to a real environment entails an excessive model complexity increase hurting performances. Furthermore,

the formal verification process should be subject to evaluation by a safety expert. As a matter of fact, the time and effort required by an external auditor to acquire expertise on the notation and develop diagrams from scratch could be compared to the effort required by directly drafting the formal models or the FB application. As a further step forward, the very decision to adopt a formal verification method based on model-checking versus a standard manual risk assessment technique could be called into question to objectively appraise advantages and drawbacks.

Appendix A HRC-TEAM Models

A.1 Class Diagram

Fig. A.1 displays the complete developed Class Diagram. *Classes* are displayed in light blue, whereas *InstanceSpecifications* are portrayed in different colors based on their nature which are then inherited by following diagram for a better visual understanding of the elements meaning. The two main superclasses *Resource* and *Layout* are connected by an *association*, whereas the other Classes are featured as *specifications* of either of the two. A complete description of all the featured elements, their attributes and their methods can be found in Section 4.2.1.

A.2 Component Diagrams

Figures A.2, A.3, A.4, A.5 and A.6 carry complete examples of Component Diagrams developed for Case A and Case B. As for the Operator (Figure A.2), this is exploited to state an *association* with the ArmBand. For the Robotic Systems (Figure A.3 and Figure A.4), the inter-component connectors are implemented to declare a *kinematic constraint* among single items (e.g., between the conveyor and the robotic arm). Finally, in the layouts cases (Figure A.5 and Figure A.6), single sections are instantiated and connected to each other to create *adjacency* constraints.

A.3 Activity Diagrams

Figures A.7 and A.8 feature the complete Activity Diagrams for the two case studies, A and B. Note that all *OpaqueActions* are endowed with a stereotype, and so are some of the arrows carrying state-based commands different from the default one (see Section 4.4 for the complete analysis). Also note that for Case A (Figure A.7) the three different versions are handled by the *DecisionNode selectTask*, whereas Figure A.8 features both *LoopNodes* necessary for Case B and the *DecisionNode emptyBin* referring to possible operator's choices (explained in detail in Section 4.5.4).

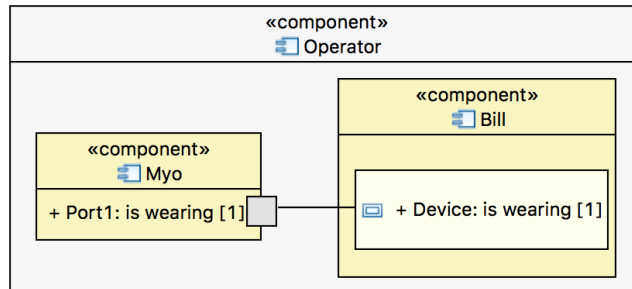


Figure A.2: Operator Component Diagram for Case A and Case B.

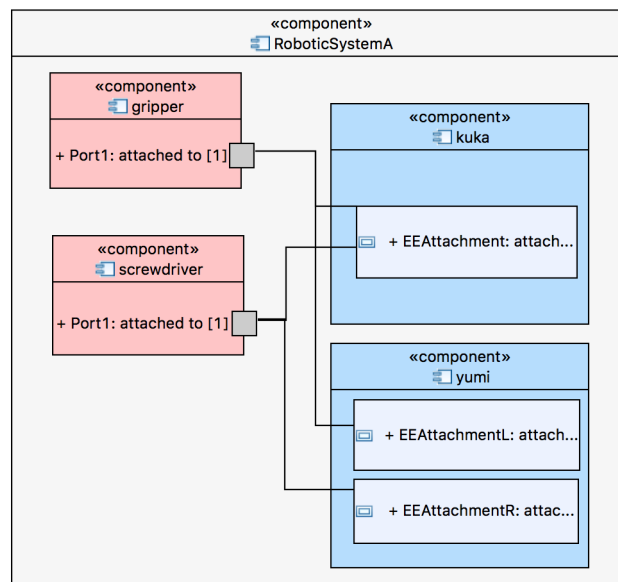


Figure A.3: Robotic System Component Diagram for Case A.

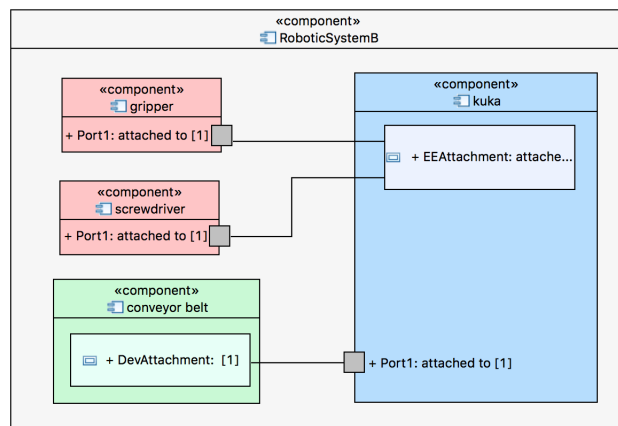


Figure A.4: Robotic System Component Diagram for Case B.

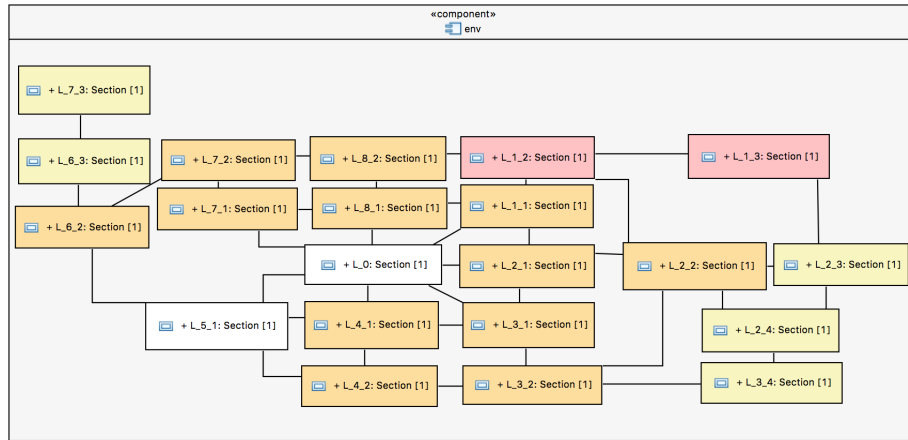


Figure A.5: Layout Component Diagram for Case A.

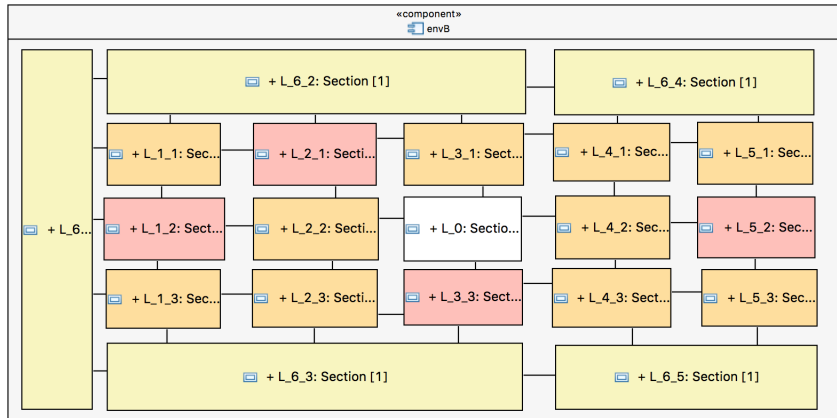


Figure A.6: Layout Component Diagram for Case B.

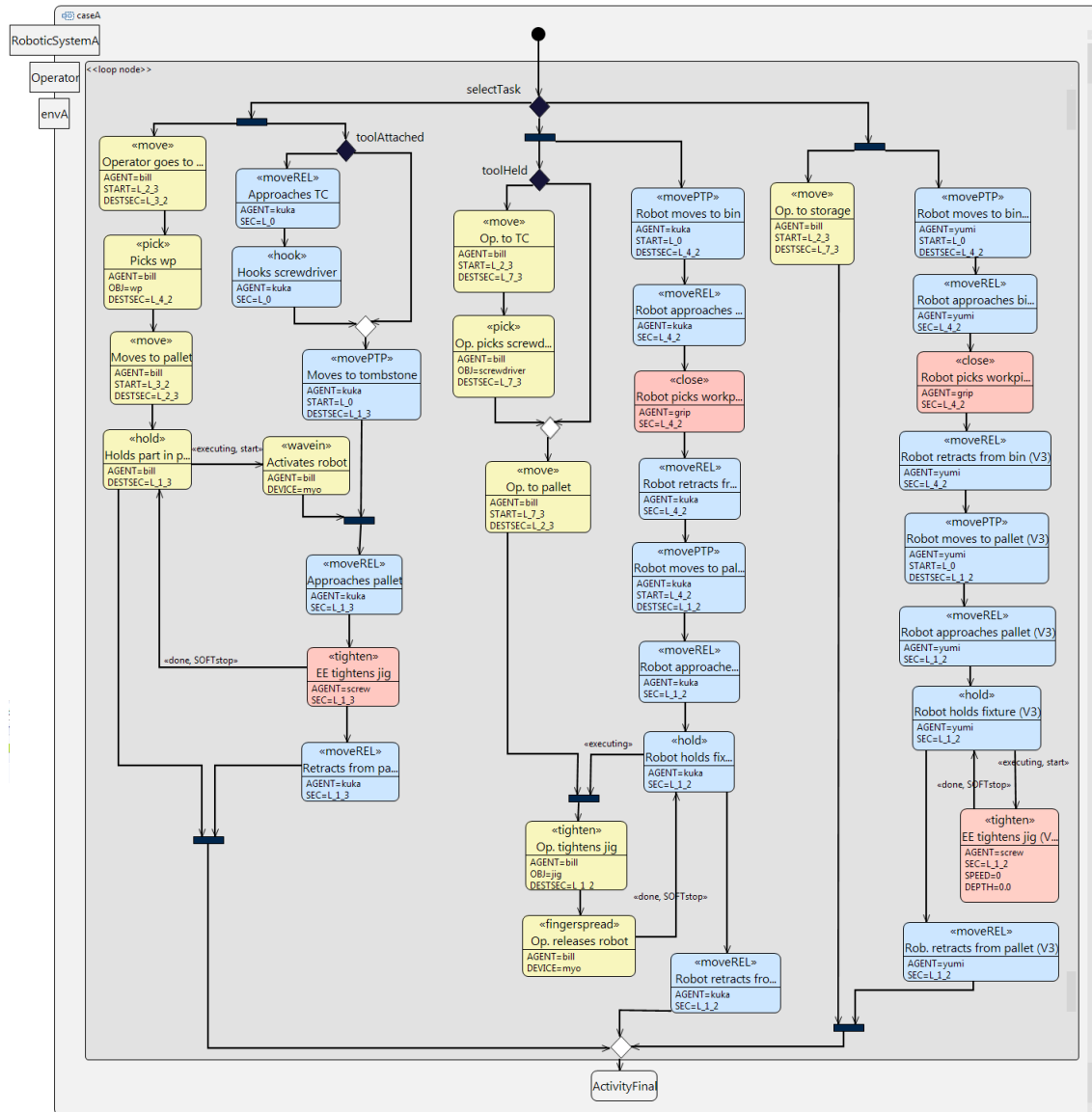


Figure A.7: Complete Activity Diagram for Case A.

Appendix B SAFER-HRC Models

B.1 Java Code

Listing B.1: Portion of Total Action List (TAL) creating function

```
1 public void getActionsTAL(Activity act){
2     List<Action> TAL = new ArrayList<Action>();
3     int id = 1;
4     for(ActivityNode n: act.getNodes()){
5         // Action is out of any loop
6         if (n instanceof OpaqueAction){
7             OpaqueAction op = (OpaqueAction) n;
8             Action ac = new Action(op);
9             ac.oldID = id;
10            ac.iter = 0;
11            ac.ID = id++;
12            if(n.getAppliedStereotypes().size() == 0){ // only the ActivityFinal
13                OpaqueAction has no applied stereotype!
14                ac.ID = -1;
15            }
16            TAL.add(ac);
17        }
18        // Action is inside a loop
19        if (n instanceof StructuredActivityNode){
20            StructuredActivityNode san = (StructuredActivityNode) n;
21            int loopIter = Integer.parseInt(san.getName());
22            for (ActivityNode n2: san.getContainedNodes()){
23                if(n2 instanceof OpaqueAction){
24                    int idfix = id;
25                    for(int i=0;i<loopIter;i++){
26                        OpaqueAction op2 = (OpaqueAction) n2;
27                        Action ac = new Action(op2);
28                        ac.oldID = idfix;
29                        ac.iter = i+1;
30                        ac.ID = id++;
31                        TAL.add(ac);
32                    }
33                }
34            }
35        }
36        this.aList = TAL;
37        this.getPreCondTAL(TAL);
38        this.getDecNodes(act);
39    }
```

Listing B.2: Portion of Pre-Conditions list creating function

```

1 public void getPreCondTAL(List<Action> TAL, List<ActivityEdge> ee){
2     for(Action a: TAL){
3         int iter = a.iter;
4         for(ActivityEdge e: a.op.getIncomings()){
5             ActivityNode source = e.getSource();
6             PreCondition pc = new PreCondition();
7             Predecessor p = new Predecessor();
8
9             // Set pre-condition command
10            for(Stereotype st: e.getAppliedStereotypes()){
11                if (st.getProfile().getName().equals("tempOperatorCommand")){
12                    pc.command = st.getName();
13                }
14            }
15
16            // Set pre-condition p
17            if(a.op.getInStructuredNode() != source.getInStructuredNode()){
18                // First element of a loop, with iteration > 1
19                :
20                if(source instanceof OpaqueAction){ // OpaqueAction case
21                    for(Action a2: TAL){
22                        if(a2.op == source && a2.iter == iter-1){
23                            p.name = a2.op.getName();
24                            for(Stereotype st: e.getAppliedStereotypes()){
25                                if (st.getProfile().getName().equals("
tempOperatorState")){
26                                    p.what = st.getName();
27                                }
28                            }
29                            p.ID = a2.ID;
30                        }
31                    }
32                }
33                else{ // Connector case, recursive call
34                    if(source instanceof InitialNode){
35                        p.what = "InitialNode"; }
36                    else{
37                        p = this.getPreCondTALconn(e, p, TAL, iter-1, ee); }
38                }
39            }
40            // First element of a loop, with iteration = 1
41            else if(a.op.getInStructuredNode() != null){
42                if(source instanceof OpaqueAction){ // OpaqueAction case
43                    :
44                    else if(a2.op.getInStructuredNode() != null){ // but the precondition
is inside another loop
45                    :
46                }
47            }
48            else{ // Connector case, recursive call
49                :
50            }
51        }
52        // Action a is outside any loop, whereas Precond is inside a loop
53        else{
54            int totalIter = Integer.parseInt(source.getInStructuredNode().
getName());
55            if(source instanceof OpaqueAction){ // OpaqueAction case

```

```

56         :
57     }
58     else{ // Connector case, recursive call
59         :
60     }
61 }
62 }
63 // Action a and Precond are inside the same loop or both outside
64 else{
65     if(source instanceof OpaqueAction){ // OpaqueAction case
66         :
67     }
68     else{ // Connector case, recursive call
69         :
70     }
71     pc.p = p;
72     a.precond.add(pc);
73 }
74 }
75 }

```

B.2 ZOT Input

Listing B.3: Generated operator body constraints for Case A/O.lisp

```

1 (defconstant *Operator_Body*
2   (alwf
3     (&&
4       (<->
5         (-P- OperatorStill)
6         (-A- i body_indexes ([=](-V- Body_Part_pos i) (yesterday (-V-
7           Body_Part_pos i))))
8       )
9       ([>=](-V- Body_Part_pos 1) L_0)
10      ([<=](-V- Body_Part_pos 1) L_4_2)
11
12      (!!([=] (-V- Body_Part_pos 1) L_0))
13      (!!([=] (-V- Body_Part_pos 1) L_1_2))
14      (!!([=] (-V- Body_Part_pos 1) L_1_3))
15      (!!([=] (-V- Body_Part_pos 1) L_4_2))
16
17      (||
18        ([=] (-V- Body_Part_pos 1) (yesterday(-V- Body_Part_pos 1)))
19        (Adj (-V- Body_Part_pos 1) (yesterday(-V- Body_Part_pos 1)))
20      )
21
22      ([=] (-V- Body_Part_pos 1) (-V- Body_Part_pos 2))
23
24      :
25      ([=] (-V- Body_Part_pos 11) (-V- Body_Part_pos 7))
26
27      :
28      (|| ([=] (-V- Body_Part_pos 1) (-V- Body_Part_pos 11))
29        (Adj (-V- Body_Part_pos 1) (-V- Body_Part_pos 11)))

```

```
28 )
29 ) ) )
```

Listing B.4: Generated robot structure constraints for Case B/R.lisp

```
1 (defconstant *Robot_Structure*
2   (alwf
3     (&&
4       ([=] (-V- End_Eff_F_Position) (-V- End_Eff_B_Position))
5
6       (||
7         (Adj (-V- LINK1_Position) L_0)
8         ([=] (-V- LINK1_Position) L_0)
9       )
10
11      (||
12        (Adj (-V- LINK1_Position) (-V- LINK2_Position))
13        ([=] (-V- LINK1_Position) (-V- LINK2_Position))
14      )
15
16      (||
17        (Adj (-V- End_Eff_B_Position) (-V- LINK2_Position))
18        ([=] (-V- End_Eff_B_Position) (-V- LINK2_Position))
19      )
20
21      (!!([=] (-V- LINK1_Position) L_6_2))
22      (!!([=] (-V- LINK2_Position) L_6_2))
23      (!!([=] (-V- End_Eff_B_Position) L_6_2))
24
25      :
26
27      (->
28        (-P- Robot_Idle)
29        (&&
30          (!! (-P- LINK1_Moving)) (!! (-P- LINK2_Moving)) (!! (-P-
31            End_Eff_Moving))
32        )
33
34      (<->
35        (-P- Robot_Homing)
36        (&&
37          ([=] (-V- End_Eff_B_Position) L_0) ([=] (-V- LINK2_Position) L_0)
38          ([=] (-V- LINK1_Position) L_0)
39          (!! (-P- LINK1_Moving)) (!! (-P- LINK2_Moving)) (!! (-P-
40            End_Eff_Moving))
41        )
42      )
43    )
44  )
45 ) ) )
```

Listing B.5: Generated layout structure for Case A/L.lisp

```
1 ; Layout
2
3 (defvar L_0 0)
4 (defvar L_7_1 1)
5 (defvar L_4_1 2)
6 (defvar L_3_2 3)
7 (defvar L_2_2 4)
8
9 :
10 (defvar L_8_2 17)
11 (defvar L_7_2 18)
```

```

11 (defvar L_6_2 19)
12 (defvar L_4_2 20)

```

Listing B.6: Generated layout adjacency constraints for Case B/L.lisp

```

1 (defun Adj (i j)
2   (||
3     (&&([=] i L_1_2)(|| ([=] j L_2_2)([=] j L_1_1)([=] j L_1_3)([=] j L_6_1)))
4     (&&([=] i L_2_1)(|| ([=] j L_1_1)([=] j L_2_2)([=] j L_3_1)([=] j L_6_2)))
5     (&&([=] i L_2_2)(|| ([=] j L_1_2)([=] j L_2_1)([=] j L_2_3)([=] j L_0)))
6     (&&([=] i L_2_3)(|| ([=] j L_1_3)([=] j L_2_2)([=] j L_3_3)([=] j L_6_3)))
7
8     ;
9     (&&([=] i L_6_3)(|| ([=] j L_6_1)([=] j L_6_5)([=] j L_1_3)([=] j L_2_3)([=]
10      j L_3_3)))
11    (&&([=] i L_6_1)(|| ([=] j L_6_2)([=] j L_1_1)([=] j L_1_2)([=] j L_1_3)([=]
12      j L_6_3)))
13    (&&([=] i L_6_4)(|| ([=] j L_6_2)([=] j L_4_1)([=] j L_5_1)))
14    (&&([=] i L_6_5)(|| ([=] j L_6_3)([=] j L_4_3)([=] j L_5_3)))
15  )

```

Listing B.7: Generated decision variable constraints for Case A/T.lisp

```

1 (define-tvar selectTask1 *int*)
2 (defconstant *selectTask1config*
3   (alw
4     (&&([>=] (-V- selectTask1) 1)
5     ([<=] (-V- selectTask1) 3)
6     ([=] (-V- selectTask1) (yesterday(-V- selectTask1)))
7   )
8 )
9 )

```

Listing B.8: Examples of action block configuration constants in Case A/T1.lisp

```

1 ; Moves to tombstone
2
3 (defconstant *Action22T1*
4   (alwf (&&
5     ; Pre-Condition
6     (->
7       ([=] (-V- Action_Pre 22 T1) 1)
8       (&&
9         (||
10          (&& ([=] (-V- toolAttached2) 2)
11          (&& ([=] (-V- selectTask2) 1)
12          (||
13            (&&
14              ([=] (-V- actions 17 1 T1) done)
15              ([=] (-V- actions 5 1 T1) done)
16            )
17            (&&
18              ([=] (-V- actions 59 1 T1) done)
19              ([=] (-V- actions 49 1 T1) done)
20            )
21            ([=] (-V- actions 37 1 T1) done)
22          )
23          )
24          )
25          ([=] (-V- actions 10 1 T1) done)
26        )
27        ([=] (-V- End_Eff_B_Position) L_0)

```

```
28     ) )
29     ; Post-Condition
30     (->
31         ([=] (-V- Action_Post 22 T1) 1)
32         (&&
33             ([=] (-V- End_Eff_B_Position) L_1_3)
34 ) ) ) )
```

Listing B.9: Examples of action block configuration constants in Case B/T1.lisp

```
1 ; Op. places wp in the buffer
2
3 (defconstant *Action8T1*
4     (alwf (&&
5         ; Pre-Condition
6         (->
7             ([=] (-V- Action_Pre 8 T1) 1)
8             (&&
9                 ; Workflow-related
10                ([=] (-V- actions 2 1 T1) done)
11                ; Stereotype-related
12                (-P- partTaken)
13                ([=] (-V- Body_Part_pos hand) L_1_2)
14            )
15        )
16        ; Post-Condition
17        (->
18            ([=] (-V- Action_Post 8 T1) 1)
19            (&&
20                (!! (-P- partTaken))
21                (-P- partPresent)
22                ([=] (-V- Body_Part_pos hand) L_1_2)
23            )
24        )
25        ; Inter-Condition
26        (->
27            (||
28                ([=] (-V- actions 8 1 T1) executing)
29                ([=] (-V- actions 8 1 T1) exrm)
30            )
31            (&&
32                ([=] (-V- Body_Part_pos hand) L_1_2)
33            )
34 ) ) ) )
```

B.3 Parsed ZOT Output

Listing B.10: Sample of parsed ZOT output for Case A experiment

```
1 ----- time 0 -----
2
3 Operator in L_2_2 L_2_3
4 Robot in L_4_2 (EE) - L_3_1 (Link1) - L_3_2 (Link2) -
5
6 RRM *force decrease* active
7
8
```

```

9 ----- time 1 -----
10 Action 23 waiting          -Op. to TC 1-
11
12 Operator in L_2_3
13 Robot in L_4_2 (EE) - L_3_1 (Link1) - L_3_2 (Link2) -
14
15
16 ----- time 2 -----
17 Action 23 executing        -Op. to TC 1-
18
19 Operator in L_2_1 L_2_2
20 Robot in L_4_2 (EE) - L_3_1 (Link1) - L_3_2 (Link2) -
21
22 RRM *force decrease* active
23
24
25 ----- time 3 -----
26 Action 23 executing        -Op. to TC 1-
27 Action 39 executing        -Robot moves to bin 1-
28
29 Operator in L_3_1 L_3_2
30 Robot in L_0 (EE) - L_3_1 (Link1) - L_3_1 (Link2) -
31
32
33 ----- time 4 -----
34 Action 23 executing        -Op. to TC 1-
35 Action 39 executing        -Robot moves to bin 1-
36
37 Operator in L_0 L_3_1
38 Robot in L_4_1 (EE) - L_3_1 (Link1) - L_3_1 (Link2) -
39
40 RRM *force decrease* active
41
42 ----->WARNING: upper body ent with L2 in L_3_1
43 ----->WARNING: waist area ent with L2 in L_3_1
44 ----->WARNING: lower body ent with L2 in L_0
45
46 ----- time 5 -----
47 Action 23 executing        -Op. to TC 1-
48 Action 39 executing        -Robot moves to bin 1-
49
50 Operator in L_3_1 L_4_1
51 Robot in L_4_1 (EE) - L_7_1 (Link1) - L_0 (Link2) -
52
53 RRM *vel decrease* active
54
55
56 ----- time 6 -----
57 Action 23 executing        -Op. to TC 1-
58 Action 39 done             -Robot moves to bin 1-
59 Action 41 executing        -Robot approaches bin 1-
60
61 Operator in L_4_2 L_5_1
62 Robot in L_4_2 (EE) - L_3_1 (Link1) - L_3_2 (Link2) -
63
64
65 ----- time 7 -----
66 Action 23 executing        -Op. to TC 1-
67 Action 39 done             -Robot moves to bin 1-
68 Action 41 done             -Robot approaches bin 1-
69 Action 45 executing        -Robot picks workpiece 1-
70
71 Operator in L_6_2
72 Robot in L_4_2 (EE) - L_3_1 (Link1) - L_4_1 (Link2) -

```


Appendix C IEC 61499 Application

C.1 Java Code

Listing C.1: Portion of Total Edge List (TEL) creating function

```
1 public void getEdgesTEL(Activity act){
2     for(ActivityEdge e: act.getEdges()){
3         int mIter = 1;
4         ActivityNode tempSource = e.getSource();
5         ActivityNode tempTarget = e.getTarget();
6         Edge temp = new Edge();
7
8         // Assigning the correct value to mIter
9         if(tempSource.getInStructuredNode() != null
10            && tempSource.getInStructuredNode() == tempTarget.
11               getInStructuredNode()){
12             mIter = Integer.parseInt(e.getSource().getInStructuredNode().getName());
13         }
14
15         // Get stereotypes related to Source state and command for Target
16         for(Stereotype st: e.getAppliedStereotypes()){
17             if(st.getProfile().getName().equals("tempOperatorState")){
18                 temp.sourceState = st.getName();
19             }
20             else if(st.getProfile().getName().equals("tempOperatorCommand")){
21                 temp.targetCommand = st.getName();
22             }
23         }
24
25         // Target of the Arrow
26         temp.edgeTarget = tempTarget;
27         if(tempTarget instanceof JoinNode){
28             :
29         }
30         else if(tempTarget instanceof MergeNode){
31             :
32         }
33
34         // Source of the Arrow
35         temp.edgeSource = tempSource;
36         if(tempSource instanceof ForkNode){
37             :
38         }
```

```

38     else if(tempSource instanceof DecisionNode){
39         :
40     }
41
42     // Copying the Arrow in case of loops
43     for(int i=0;i<mIter;i++){
44         Edge temp2 = new Edge(temp);
45         temp2.iter = i+1;
46         TEL.add(temp2);
47     }
48 }
49
50 // Creating arrows for SpecialReturn (from the end of the loop to a new
51 // iteration of it)
52 for(ActivityEdge e: act.getEdges()){
53     if(e.getSource().getInStructuredNode() != e.getTarget().getInStructuredNode
54        ()
55        && e.getTarget().getInStructuredNode() != null){
56         for(ActivityEdge e2: act.getEdges()){
57             if(e != e2 && e2.getSource().getInStructuredNode() != e2.getTarget()
58                .getInStructuredNode()){
59                 if(e.getTarget().getInStructuredNode() == e2.getSource().
60                    getInStructuredNode()){
61
62                     // Target of the Arrow
63                     temp.edgeTarget = e.getTarget();
64                     if(tempTarget instanceof JoinNode){
65
66                         :
67                     }
68                     else if(tempTarget instanceof MergeNode){
69
70                         :
71                     }
72
73                     // Source of the Arrow
74                     temp.edgeSource = e2.getSource();
75                     if(tempSource instanceof ForkNode){
76
77                         :
78                     }
79                     else if(tempSource instanceof DecisionNode){
80
81                         :
82                     }
83
84                     // Copying the Arrow in case of loops
85                     int mIter = Integer.parseInt(e.getTarget().
86                        getInStructuredNode().getName());
87                     for(int i=0; i<mIter-1; i++){
88                         Edge temp2 = new Edge(temp);
89                         temp2.iter = -(i+1);
90                         TEL.add(temp2);
91                     }
92                 }
93             }
94         }
95     }
96     this.eList = TEL;
97 }

```

C.2 TaskGen-FB Root C++ Code

Listing C.2: Function run by the internal thread to scan configuration files

```

1 void FORTE_TaskGenFB::threadFcn()
2 {
3     int intervalDur = PACE();
4     CTimerHandler::sm_poFORTETimer->registerTimedFB(&m_stTimeListEntry, DT);
5     usleep(intervalDur);
6     while (!exit){
7         STATE() = "reading";
8         switch(currFile){
9             case 1:
10                std::getline(actionFile, line);
11                if (line=="HowManyRes"){
12                    howManyRes = std::atoi(getPRM(currFile).getValue());
13                    FB_NAME() = "Header";
14                    std::stringstream ss; ss<<howManyRes;
15                    std::string s; s = "H"; s.append(ss.str()); //s.append("RES");
16                    FB_TYPE().fromString(s.c_str());
17                    processComm(1);
18                    // Infrastructure Creation
19                    :
20                }
21                // Creates GLOBAL_T blocks and sets their parameters
22                if (line=="ID:"){
23                    FB_NAME() = getPRM(currFile); FB_TYPE() = "GLOBAL_T";
24                    processComm(1); }
25                if (line=="Agent:"){
26                    PRM_NAME() = "RESOURCE_NAME"; PRM_VALUE() = getPRM(currFile);
27                    processComm(2); }
28                if (line=="Stereotype name:"){
29                    PRM_NAME() = "EXECUTION_CMD"; PRM_VALUE() = getEX_COMM();
30                    processComm(2); }
31                if (line=="DESTPOS:" || line=="CONTROL TYPE:" || line=="OVERRIDE:"){
32                    PRM_NAME() = getPRM_NAME(); PRM_VALUE() = getPRM(currFile);
33                    processComm(2); }
34                if (actionFile.eof()){
35                    actionFile.close(); currFile++; }
36                break;
37            case 2:
38                std::getline(logicFile, line);
39                if (line=="JoinNodes"){
40                    FB_TYPE() = "JoinFB"; }
41                :
42                break;
43            case 3:
44                std::getline(arrowFile, line);
45                if (line=="SourceID"){
46                    FB_NAME() = getPRM(currFile); handle_conn(); }
47                :
48                break;
49        }
50    }
51    STATE() = "idle";
52 }

```

Listing C.3: Function run by the internal thread to process commands

```

1 void FORTE_TaskGenFB::processComm(int comm)
2 {
3     switch(comm){
4         case 1:
5             sendCreate = true;
6             std::cout << "-----> Creating block " << FB_NAME().
              getValue() << " (" << FB_TYPE().getValue() << ")" << std::endl;
7             break;
8         case 2:
9             sendSet = true;
10            // Debug message
11            break;
12         case 3:
13            sendConnect = true;
14            // Debug message
15            break;
16    }
17    // Triggers the execution of cg_nExternalEventID case
18    CTimerHandler::getDeviceExecution()->startNewEventChain(m_stTimeListEntry.
              m_poTimedFB);
19    usleep(PACE());
20 }

```

Listing C.4: Main TaskGen-FB switch case managing response to input events

```

1 void FORTE_TaskGenFB::executeEvent(int pa_nEIID){
2     switch(pa_nEIID){
3         case scm_nEventINITID:
4             // Send Output Event INITO
5             break;
6         case scm_nEventSTARTID:
7             :
8             actionFile.open(Actions_File().getValue());
9             arrowFile.open(Arrows_File().getValue());
10            logicFile.open(Logic_File().getValue());
11
12            currFile = 1;
13            // Creates parallel thread
14            internal_thread = new boost::thread(boost::bind(&FORTE_TaskGenFB::threadFcn ,
              this));
15            break;
16         case scm_nEventSTOPID:
17            exit = true;
18            break;
19         case scm_nEventREQID:
20            // Sends output command based on the currently true boolean
21            if(sendCreate){
22                sendOutputEvent(scm_nEventCREATEID); sendCreate = false;
23            }
24            if(sendSet){
25                sendOutputEvent(scm_nEventSETID); sendSet = false;
26            }
27            if(sendConnect){
28                sendOutputEvent(scm_nEventCONNECTID); sendConnect = false;
29            }
30            break;
31         case cg_nExternalEventID:
32            sendOutputEvent(scm_nEventCNFID);
33            break;
34     }
35 }

```

C.3 Simulation Log

Listing C.5: ROS resource nodes feedback during task simulation

```
1 [ INFO] [1498568136.917132863]: executing: (Robot) movePTP to: bin_pos
2 [ INFO] [1498568136.917462592]: executing: (Operator) move
3 [ INFO] [1498568137.046908940]: bill: done
4 [ INFO] [1498568137.047201555]: kuka: done
5 [ INFO] [1498568137.067645132]: executing: (Robot) moveREL
6 [ INFO] [1498568137.196951915]: kuka: done
7 [ INFO] [1498568137.218149398]: executing: (EndEff) close
8 [ INFO] [1498568137.347011208]: gripper: done
9 [ INFO] [1498568137.357044553]: executing: (Robot) moveREL
10 [ INFO] [1498568137.486967870]: kuka: done
11 [ INFO] [1498568137.507099052]: executing: (Robot) movePTP to: pre_pallet
12 [ INFO] [1498568137.636930878]: kuka: done
13 [ INFO] [1498568137.658462346]: executing: (Robot) moveREL
14 [ INFO] [1498568137.786860931]: kuka: done
15 [ INFO] [1498568137.806988480]: executing: (Robot) hold
16 [ INFO] [1498568137.809497625]: executing: (Operator) tighten
17 [ INFO] [1498568137.936903809]: bill: done
18 [ INFO] [1498568137.937037630]: executing: (Operator) gesture recognition
19 [ INFO] [1498568137.956979645]: bill: done
20 [ INFO] [1498568138.087103466]: kuka: done
21 [ INFO] [1498568138.108564136]: executing: (Robot) moveREL
22 [ INFO] [1498568138.236881384]: kuka: done
```


Bibliography

- [1] H. Lasi, P. Fettke, H.-G. Kemper, T. Feld, and M. Hoffmann, “Industry 4.0,” *Business & Information Systems Engineering*, vol. 6, no. 4, p. 239, 2014.
- [2] S. Pfeiffer, “Robots, Industry 4.0 and Humans, or Why Assembly Work Is More than Routine Work,” *Societies*, vol. 6, no. 2, p. 16, 2016.
- [3] D. Gorecky, M. Schmitt, M. Loskyll, and D. Zühlke, “Human-Machine-Interaction in the Industry 4.0 era,” in *Industrial Informatics (INDIN), 2014 12th IEEE International Conference on*, pp. 289–294, IEEE, 2014.
- [4] G. Hoffman and C. Breazeal, “Collaboration in human-robot teams,” in *AIAA 1st Intelligent Systems Technical Conference*, p. 6434, 2004.
- [5] K. Kosuge, H. Yoshida, D. Taguchi, T. Fukuda, K. Hariki, K. Kanitani, and M. Sakai, “Robot-human collaboration for new robotic applications,” in *Industrial Electronics, Control and Instrumentation, 1994. IECON'94., 20th International Conference on*, vol. 2, pp. 713–718, IEEE, 1994.
- [6] C. W. Lewis, “The facts about Co-Bot Robot sales,” *RobotEconomics, (A)n Intelligent Future*, 2015.
- [7] R. Parasuraman, T. B. Sheridan, and C. D. Wickens, “A model for types and levels of human interaction with automation,” *IEEE Transactions on systems, man, and cybernetics-Part A: Systems and Humans*, vol. 30, no. 3, pp. 286–297, 2000.
- [8] M. Askarpour, “Risk assessment in collaborative robotics,” in *Formal Methods Doctoral Symposium, FM-DS*, vol. 1744 of *CEUR-WS*, 2016.
- [9] M. Askarpour, D. Mandrioli, M. Rossi, and F. Vicentini, “SAFER-HRC: Safety Analysis Through Formal vERification in Human-Robot Collaboration,” in *International Conference on Computer Safety, Reliability, and Security*, pp. 283–295, Springer, 2016.
- [10] A. Zoitl, G. Ebenhofer, and M. Hofmann, “Developing a monitoring infrastructure for IEC 61499 devices,” in *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*, 2013.

- [11] International Electro-technical Commission, *International Standard IEC61499*, 2005.
- [12] R. Lewis, *Modelling control systems using IEC 61499: Applying function blocks to distributed systems*. No. 59, Iet, 2001.
- [13] S. Panjaitan and G. Frey, “Functional design for iec 61499 distributed control systems using UML Activity Diagrams,” in *Proceeding International Conference Instrumentation, Communication and Information Technology (ICICI) 2005*, pp. 64–70, 2005.
- [14] N. Iannacci, M. Giussani, F. Vicentini, and L. M. Tosatti, “Robotic cell workflow management through an IEC 61499-ROS architecture,” in *Emerging Technologies and Factory Automation (ETFA), 2016 IEEE 21st International Conference on*, pp. 1–7, IEEE, 2016.
- [15] D. Georgakopoulos, M. Hornick, and A. Sheth, “An overview of workflow management: From process modeling to workflow automation infrastructure,” *Distributed and Parallel Databases*, vol. 3, no. 2, pp. 119–153, 1995.
- [16] Z. Hemel, R. Verhaaf, and E. Visser, “WebWorkFlow: an object-oriented workflow modeling language for web applications,” *Model Driven Engineering Languages and Systems*, pp. 113–127, 2008.
- [17] K. Salimifard and M. Wright, “Petri net-based modelling of workflow systems: An overview,” *European journal of operational research*, vol. 134, no. 3, pp. 664–676, 2001.
- [18] W. M. van der Aalst, “Three good reasons for using a petri-net-based workflow management system,” in *Proceedings of the International Working Conference on Information and Process Integration in Enterprises (IPIC’96)*, pp. 179–201, Citeseer, 1996.
- [19] W. M. Van Der Aalst and A. H. Ter Hofstede, “Yawl: yet another workflow language,” *Information systems*, vol. 30, no. 4, pp. 245–275, 2005.
- [20] “Workflow patterns home page.” <http://www.workflowpatterns.com>.
- [21] W. M. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros, “Workflow patterns,” *Distributed and Parallel Databases*, vol. 14, no. 1, pp. 5–51, 2003.
- [22] F. Paternò, C. Mancini, and S. Meniconi, “Concurtasktrees: A diagrammatic notation for specifying task models,” in *Human-Computer Interaction INTERACT’97*, pp. 362–369, Springer, 1997.

- [23] C. Martinie, P. Palanque, M. Ragosta, and R. Fahssi, “Extending procedural task models by systematic explicit integration of objects, knowledge and information,” in *ACM International Conference Proceeding Series*, p. European Association for Cognitive Ergonomics (EAC, 2013.
- [24] M. L. Bolton, R. I. Siminiceanu, and E. J. Bass, “A systematic approach to model checking human–automation interaction using task analytic models,” *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, vol. 41, no. 5, pp. 961–976, 2011.
- [25] B. Selic, “A systematic approach to domain-specific language design using UML,” in *Object and Component-Oriented Real-Time Distributed Computing, 2007. ISORC’07. 10th IEEE International Symposium on*, pp. 2–9, IEEE, 2007.
- [26] A. Nordmann, N. Hochgeschwender, and S. Wrede, “A Survey on Domain-Specific Languages in Robotics,” *Journal of Software Engineering for Robotics (JOSER)*, vol. 1, no. July, pp. 195–206, 2016.
- [27] S. Dhouib, S. Kchir, S. Stinckwich, T. Ziadi, and M. Ziane, “Robotml, a domain-specific language to design, simulate and deploy robotic applications,” in *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, pp. 149–160, Springer, 2012.
- [28] D. S. Lortal, Gaele and S. Gerard, “Integrating Ontological Domain Knowledge into a Robotic DSL,” *Models in Software Engineering*, pp. 401–414, 2011.
- [29] A. Van Deursen, P. Klint, J. Visser, *et al.*, “Domain-specific languages: An annotated bibliography,” *Sigplan Notices*, vol. 35, no. 6, pp. 26–36, 2000.
- [30] G. Biggs and B. Macdonald, “A Survey of Robot Programming Systems,” in *Proceedings of the Australasian conference on robotics and automation*, pp. 1–3, 2003.
- [31] A. Steck and C. Schlegel, “SmartTCL: An execution language for conditional reactive task execution in a three layer architecture for service robots,” in *Int. Workshop on DYNAMIC languages for RObotic and Sensors systems (DYROS/SIMPAR)*, pp. 274–277, 2010.
- [32] C. Schlegel, A. Lotz, M. Lutz, D. Stampfer, J. F. Inglés-Romero, and C. Vicente-Chicote, “Model-driven software systems engineering in robotics: covering the complete life-cycle of a robot,” *it-Information Technology*, vol. 57, no. 2, pp. 85–98, 2015.
- [33] X. Blanc, J. Delatour, and T. Ziadi, “Benefits of the MDE approach for the development of embedded and robotic systems Application to Aibo,” in *Proceedings of the 2nd National Workshop on Control Architectures of Robots: from*

- Models to Execution on Distributed Control Architectures, CAR. 2007*, no. 2, 2007.
- [34] R. Bischoff, T. Guhl, E. Prassler, W. Nowak, G. Kraetzschmar, H. Bruyninckx, P. Soetens, M. Haegele, A. Pott, P. Breedveld, *et al.*, “Brics-best practice in robotics,” in *Robotics (ISR), 2010 41st International Symposium on and 2010 6th German Conference on Robotics (ROBOTIK)*, pp. 1–8, VDE, 2010.
- [35] M. A. A. Rahman, A. Yasuda, K. Mayama, M. Mizukawa, and T. Takasu, *Model-Driven Development of Intelligent Mobile Robot Using Systems Modeling Language (SysML)*. INTECH Open Access Publisher, 2011.
- [36] D. Alonso, C. Vicente-chicote, F. Ortiz, J. Pastor, and B. Alvarez, “V3CMM: a 3-View Component Meta-Model for Model-Driven Robotic Software Development,” *Journal of Software Engineering for Robotics (JOSER)*, vol. 1, no. January, pp. 3–17, 2010.
- [37] M. Klotzbücher, P. Soetens, and H. Bruyninckx, “Bcm: A minimal robotic component model for multitarget system and component generation,” tech. rep., Technical report, Best Practice in Robotics, EU FP7 project, 2010.
- [38] M. Dumas and A. H. Ter Hofstede, “UML activity diagrams as a workflow specification language,” in *International Conference on the Unified Modeling Language*, pp. 76–90, Springer, 2001.
- [39] R. Bastos, D. Dubugras, and A. Ruiz, “Extending UML Activity Diagram for Workflow Modelling in Production Systems,” in *Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS’02)-Volume 9*, no. c, p. 291, 2002.
- [40] J. Brüning and M. Gogolla, “UML metamodel-based workflow modeling and execution,” in *Proceedings - IEEE International Enterprise Distributed Object Computing Workshop, EDOC*, pp. 97–106, 2011.
- [41] M. Bruccoleri, S. N. La Diega, and G. Perrone, “An object-oriented approach for flexible manufacturing control systems analysis and design using the unified modeling language,” *International Journal of Flexible Manufacturing Systems*, vol. 15, no. 3, pp. 195–216, 2003.
- [42] V. Vyatkin, “Software engineering in industrial automation: State-of-the-art review,” *IEEE Transactions on Industrial Informatics*, vol. 9, no. 3, pp. 1234–1249, 2013.
- [43] K. C. Thramboulidis, “Using UML in control and automation: a model driven approach,” in *Industrial Informatics, 2004. INDIN’04. 2004 2nd IEEE International Conference on*, pp. 587–593, IEEE, 2004.

- [44] T. Ritala and S. Kuikka, “UML automation profile: Enhancing the efficiency of software development in the automation industry,” in *IEEE International Conference on Industrial Informatics (INDIN)*, vol. 2, pp. 885–890, 2007.
- [45] International Standards Organization, *ISO10218-2:2011: Robots and robotic devices – Safety requirements for industrial robots – Part 2: Robot systems and integration*, 2011.
- [46] International Standards Organization, *ISO12100, Safety of machinery – General principles for design – Risk assessment and risk reduction*, 2010.
- [47] G. Joshi and H. Joshi, “Fmea and alternatives v/s enhanced risk assessment mechanism,” *International Journal of Computer Applications*, vol. 93, no. 14, 2014.
- [48] Q. A. Do Hoang, J. Guiochet, D. Powell, and M. Kaâniche, “Human-robot interactions : model-based risk analysis and safety case construction,” *6th European Congress on Embedded Real-Time Software and Systems*, p. 6, 2012.
- [49] M. Felder and A. Morzenti, “Validating real-time systems by history-checking TRIO specifications,” in *Proceedings - International Conference on Software Engineering*, no. 14, pp. 199–211, 1992.
- [50] C. Ghezzi, D. Mandrioli, and A. Morzenti, “TRIO: A Logic Language for Executable Specifications of Real-time Systems,” *Journal of Systems and software*, vol. 12, pp. 107–123, May 1990.
- [51] A. Coen-Porisini, M. Pradella, and P. Pietro, “A finite-domain semantics for testing temporal logic specifications,” in *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pp. 41–54, Springer, 1998.
- [52] A. Morzenti, E. Ratto, M. Roncato, and L. Zoccolante, “Trio, a logic formalism for the specification of real-time systems,” in *Real Time, 1989. Proceedings., Euromicro Workshop on*, pp. 26–30, IEEE, 1989.
- [53] C. A. Furia, D. Mandrioli, A. Morzenti, and M. Rossi, *Modeling time in computing*. Springer Science & Business Media, 2012.
- [54] M. Pradella, A. Morzenti, and P. S. Pietro, “Refining real-time system specifications through bounded model- and satisfiability-checking,” in *ASE 2008 - 23rd IEEE/ACM International Conference on Automated Software Engineering, Proceedings*, pp. 119–127, 2008.
- [55] E. A. Emerson, “Handbook of theoretical computer science (vol. b),” ch. Temporal and Modal Logic, pp. 995–1072, Cambridge, MA, USA: MIT Press, 1990.

- [56] EN-ISO10218-1, *Robots for industrial environments - Safety requirements - Part 1*. 2011.
- [57] International Standards Organization, *ISO/TR14121-2, Safety of machinery – Risk assessment – Part 2: Practical guidance and examples of methods*, 2012.
- [58] T. Strasser, A. Zoitl, F. Auinger, and C. Sünder, “Towards engineering methods for reconfiguration of distributed real-time control systems based on the reference model of iec 61499,” in *International Conference on Industrial Applications of Holonic and Multi-Agent Systems*, pp. 165–175, Springer, 2005.
- [59] A. Zoitl, T. Strasser, and G. Ebenhofer, “Developing modular reusable IEC 61499 control applications with 4DIAC,” in *IEEE International Conference on Industrial Informatics (INDIN)*, pp. 358–363, 2013.
- [60] The Eclipse Foundation, “Framework for Distributed Industrial Automation and Control (4DIAC),” in *The IEEE International Conference on Industrial Informatics*, 2010.
- [61] V. Vyatkin, “IEC 61499 as enabler of distributed and intelligent automation: State-of-the-art review,” *IEEE Transactions on Industrial Informatics*, vol. 7, no. 4, pp. 768–781, 2011.
- [62] J. H. Christensen, T. Strasser, A. Valentini, V. Vyatkin, and A. Zoitl, “The IEC 61499 Function Block Standard : Overview of the Second Edition,” *ISA Automation Week*, 2012.
- [63] T. Strasser, M. Rooker, G. Ebenhofer, and A. Zoitl, “Standardized Dynamic Reconfiguration of Control Applications in Industrial Systems,” *International Journal of Applied Industrial Engineering (IJAIE)*, vol. 2, no. 1, pp. 57–73, 2014.
- [64] V. Vyatkin and H.-M. Hanich, “Modeling of IEC 61499 function blocks a clue to their verification,” in *XI Workshop on Supervising and Diagnostics of Machining Systems*, vol. 76, pp. 59–68, 2000.
- [65] L. H. Yoong, P. S. Roop, Z. E. Bhatti, M. M. Kuo, *et al.*, *Model-Driven Design Using IEC 61499*. Springer, 2015.
- [66] V. Vyatkin, “The IEC 61499 standard and its semantics - Bridging the Gap Between PLC Programming Languages and Distributed Systems,” *IEEE Industrial Electronics Magazine*, vol. 3, no. 4, pp. 40–48, 2009.
- [67] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, “ROS: an open-source Robot Operating System,” in *ICRA workshop on open source software*, vol. 3, p. 5, Kobe, 2009.

- [68] J. H. Christensen, “Hms/fb architecture and its implementation,” in *Agent-based manufacturing*, pp. 53–87, Springer, 2003.
- [69] J. H. Christensen, T. Strasser, V. Vyatkin, and A. Zoitl, “The IEC 61499 Function Block Standard : Software Tools and Runtime Platforms,” *ISA Automation Week*, vol. 2012, 2012.
- [70] M. Staron, *Improving modeling with UML by Stereotype-based language customization*. Doctoral dissertation, Blekinge Institute of Technology, 2005.
- [71] OMG, *OMG Unified Modeling LanguageTM(OMG UML)*, 2015.
- [72] R. K. Pandey, “Object constraint language (OCL),” *ACM SIGSOFT Software Engineering Notes*, vol. 36, no. 1, p. 1, 2011.
- [73] H. C. Purchase, L. Colpoys, D. Carrington, and M. McGill, “UML class diagrams: an empirical study of comprehension,” in *Software Visualization*, pp. 149–178, Springer, 2003.
- [74] M. Neerincx, J. van Diggelen, and L. van Breda, “Interaction Design Patterns for Adaptive Human-Agent-Robot Teamwork in High-Risk Domains,” *Engineering Psychology and Cognitive Ergonomics*, vol. 5639, pp. 211–220, 2009.
- [75] L.-P. Yuan, G.-F. Ma, and R.-S. Sun, “An analysis of fatigue and its characteristics: A survey on chinese air traffic controller,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9736, pp. 38–47, 2016.
- [76] J. Y. C. Chen, M. J. Barnes, and M. Harper-Sciarini, “Supervisory control of multiple robots: Human-performance issues and user-interface design,” *IEEE Transactions on Systems, Man and Cybernetics Part C: Applications and Reviews*, vol. 41, no. 4, pp. 435–454, 2011.
- [77] A. Steinfeld, T. Fong, D. Kaber, M. Lewis, J. Scholtz, A. Schultz, and M. Goodrich, “Common metrics for human-robot interaction,” in *Proceeding of the 1st ACM SIGCHI/SIGART conference on Human-robot interaction - HRI '06*, p. 33, 2006.
- [78] M. R. Endsley, “Toward a Theory of Situation Awareness in Dynamic Systems,” *Human Factors: The Journal of the Human Factors and Ergonomics Society*, vol. 37, no. 1, pp. 32–64, 1995.
- [79] B. Siciliano and O. Khatib, *Handbook of Robotics*. Springer.
- [80] B. Choudhury, a. Mishra, and B. Biswal, “A Model for Performance Measure of Industrial Robots,” in *2010 Second International Conference on Computer and Network Technology*, pp. 457–461, 2010.

- [81] E.-D. Zhang, L.-L. Qi, and S. Murphy, "Method and system for optimizing the layout of a robot work cell," Oct. 29 2013. US Patent 8,571,706.
- [82] W. Bu, Z. Liu, and J. Tan, "Industrial robot layout based on operation sequence optimisation," *International Journal of Production Research*, vol. 47, no. 15, pp. 4125–4145, 2009.
- [83] M. Tay and B. Ngoi, "Optimising Robot Workcell Layout," *Advanced Manufacturing Technology*, pp. 377–385, 1996.
- [84] F. Sadeghpour and M. Andayesh, "The constructs of site layout modeling: an overview," *Canadian Journal of Civil Engineering*, vol. 42, no. August 2014, pp. 199–212, 2015.
- [85] T. Hegazy and E. Elbeltagi, "Evosite: Evolution-based model for site layout planning," *Journal of Computing in Civil Engineering*, vol. 13, no. 3, pp. 198–206, 1999.
- [86] A. Zoitl, C. Sunder, and I. Terzic, "Dynamic reconfiguration of distributed control applications with reconfiguration services based on iec 61499," in *Distributed Intelligent Systems: Collective Intelligence and Its Applications, 2006. DIS 2006. IEEE Workshop on*, pp. 109–114, IEEE, 2006.
- [87] J. Chen and W. Watson III, "A C++ thread package for concurrent and parallel programming," tech. rep., Thomas Jefferson National Accelerator Facility, Newport News, VA (US), 1999.
- [88] J. English, "Multithreading in C++," *Acm sigplan notices*, vol. 30, no. 4, pp. 21–28, 1995.