Journal of Intelligent & Robotic Systems (2021) 102:59
https://doi.org/10.1007/s10846-021-01386-2

REGULAR PAPER

Formally-based Model-Driven Development of Collaborative Robotic Applications

Mehrnoosh Askarpour[1] · Livia Lestingi[2] · Samuele Longoni[2] · Niccolò Iannacci[3] · Matteo Rossi[3] · Federico Vicentini[4]

Received: 25 November 2019 / Accepted: 31 March 2021

Abstract
The development of Human Robot Collaborative (HRC) systems faces many challenges. First, HRC systems should be adaptable and re-configurable to support fast production changes. However, in the development of HRC applications safety considerations are of paramount importance, as much as classical activities such as task programming and deployment. Hence, the reconfiguration and reprogramming of executing tasks might be necessary also to fulfill the desired safety requirements. Model-based software engineering is a suitable means for agile task programming and reconfiguration. We propose a model-based design-to-deployment toolchain that simplifies the routine of updating or modifying tasks. This toolchain relies on (i) UML profiles for quick model design, (ii) formal verification for exhaustive search for unsafe situations (caused by intended or unintended human behavior) within the model, and (iii) trans-coding tools for automating the development process. The toolchain has been evaluated on a few realistic case studies. In this paper, we show a couple of them to illustrate the applicability of the approach.

Keywords Formal verification · Model-based software engineering · Safety analysis · Task deployment · Task modeling · UML profiling

1 Introduction

Collaborative robotic applications often need to be adaptable and reconfigurable to change behavior with respect to programmed tasks, to update devices or layout, etc. Modularity, composability and reusability in component-based engineering are key enablers for replacing and reloading the tasks of a robot system in human-centered manufacturing [33]. Such abilities are essential in flexible production where operational requests are unknown or unavailable at design time, and humans have a substantial decision power. Operators in Flexible Manufacturing Systems (FMS) are no longer bound to a single station with fixed action schedules;instead, they can devote more added-value time to supervising tasks, as in the collaborative robotic application used to validate the approach presented in this work (see Fig. 1), which upgrades a formerly manual load/unload station. Examples of physically- or mentally-heavy workload tasks that benefit from collaborative robots include, but are not limited to: setting fixtures, mounting/dismounting workpieces into fixtures before/after machining, inspecting the pre-machining setups, inspecting the quality of machined parts. In these cases, robots can provide a number of assistive tasks such as kitting parts and tools, handling parts, supporting manual assembly, moving sensors for inspections, etc., according to *nominal* task plans or inline *alternatives* requested by operators.

Normally, collaborative tasks can be freely organized by operators—e.g., suspend and resume, swap assignments between human and robot, temporarily switch to totally autonomous robot execution and later resume some manual tasks. Correspondingly, layouts (e.g., robot approach directions to the pallet, types of tools, geometries of workpieces and fixtures) can also change. The actual creation of collaborative tasks can be done by different actors in several ways: through automatic composition of programs by factory planners/schedulers [25, 53], if robot-level control is part of their architecture [30] and whenever human tasks can be modeled [9]; through off-line programming

These authors contributed equally to the work presented in this article: Mehrnoosh Askarpour, Livia Lestingi, Samuele Longoni

✉ Livia Lestingi
livia.lestingi@polimi.it

Extended author information available on the last page of the article.

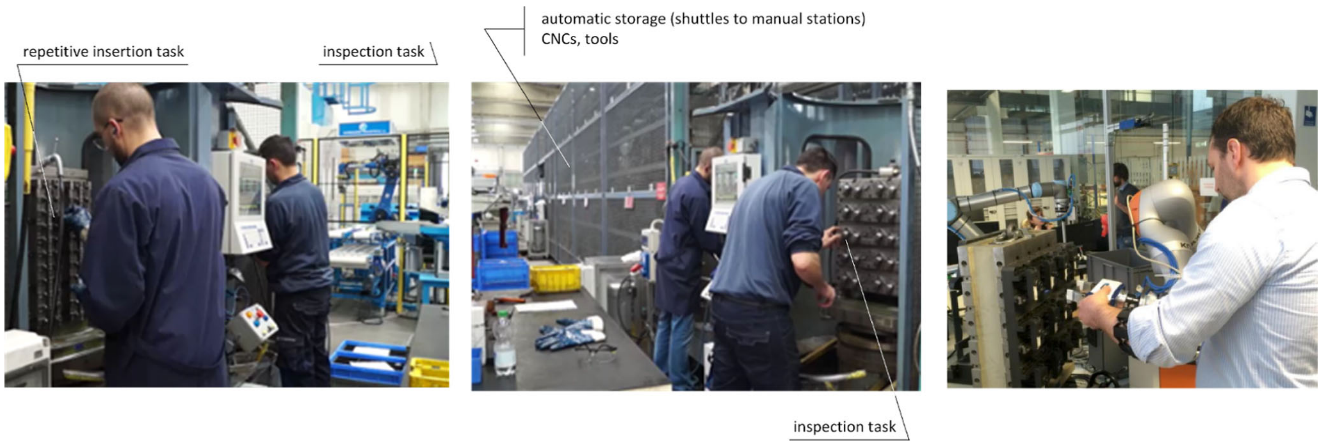

repetitive insertion task
inspection task
automatic storage (shuttles to manual stations)
CNCs, tools
inspection task

Fig. 1 The first two photos from the left show a production scenario with heavy-duty manual tasks, whereas the last one shows a collaborative version of pallet preparation

by technicians; inline by users, through interactive lead-through or GUI-based task composition. No matter the approach used for task design and composition, software development needs to efficiently support the rapid creation of new/ updated target applications for robots.

This variability increases the difficulty of ensuring that the designed applications satisfy safety requirements before their deployment. In fact, the close proximity and frequent interactions between robots and humans make risk assessment more challenging than usual, even when merely considering the *intended uses* (i.e., behaviors that do not deviate from the designed plan). Further, human operators are prone to errors or out-of-training behaviors, and may follow different non-codified workflows (e.g., circumventing instructions or hazardous troubleshooting), depending on the situation. In addition, user-driven customization of applications usually improves their usability and synchronization with human activities (e.g., setting a preferred comfortable robot trajectory). However, while changing the timing of robots could result in a greater attention by operators on a single task, it could also jeopardize their situation awareness when multiple tasks are supervised. As a result, misbehavior during transient conditions might become very common (e.g., wrong actions for catching up with a robot operation). Finally, with the advent of mobile manipulation, layout organization can change over time, increasing the number of possible combinations of physical interaction between robots, operators and objects inside the environment as a result of task scheduling, as in Nielsen et al. [31].

Hence, it is crucial that task and workspace design provide suitable protection for operators in changing conditions, and that the actual application deployment conforms to such a safe design.

To tackle the issues highlighted above, in this paper we propose a toolchain that facilitates the development of HRC applications by covering the main steps of design, safety verification, and deployment of the application in the shopfloor. The toolchain relies on the following main ingredients: (i) **a domain-specific UML profile** for the design of tasks composing HRC applications; (ii) **a formal model of HRC applications**, expressed in terms of metric temporal logic, that precisely captures the concepts defined in the UML profile and that is amenable to automated formal verification; (iii) **a mechanism to translate each UML model** conforming to the aforementioned profile **into a metric temporal logic one**, on which the safety of the designed application is thoroughly and precisely assessed through a process based on automated formal verification techniques; (iv) **a mechanism to automatically deploy the designed task on the chosen infrastructure**, in a way that preserves the safety measures introduced at design time. Developments (i), (iii) and (iv) are presented in this paper for the first time. To make the paper self-contained, we also briefly summarize the formal model, which has previously been presented in Vicentini et al. [49].

As depicted in Fig. 2, our toolchain is made of three major modules, organized in a development lifecycle (outer loop) and safety verification iteration(s) (inner loop).

The DESIGN module, described in Section 4, implements the dedicated UML profile and provides the model-based abstraction of target tasks, with the definition, instantiation and organization of available resources (robots, operators, devices), layouts, actions, goals, etc. This module is the starting point of applications' development lifecycle, where any change in the system or goals entails a new iteration in the development process. The main user of this module is the solution developer, or any other task-composing actor (e.g., automatic planners and reasoners).

The SAFETY VERIFICATION module, presented in Section 5, implements the translation of UML models into corresponding metric temporal logic ones and performs automated verification on the latter to assess the safety of

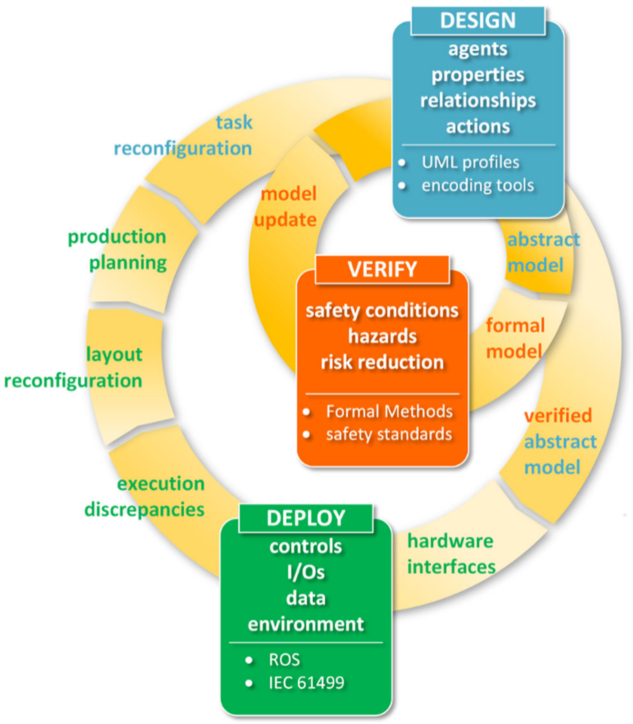


Fig. 2 Overview of the main modules of the toolchain. The inner loop is dedicated to the verification and update of the model generated by the design tool. The outcome of the verification step is then passed to the outer loop, where the deployment and interaction with the actual real environment take place. Any change in the executable re-enters the outer/inner loops

the designed application. All safety-critical aspects of the target scenario introduced by the abstract model—hazards, their associated risks and all the means for mitigating such risks—need to be formally verified in order to ensure the consistency and validity of assumptions in the candidate system. The inner verification loop is executed whenever the task model is modified, to analyze the effects of changes on the safety of the application. Verification results are then looped back to the abstract model to (i) drive necessary modifications when the design does not meet its goals, or (ii) greenlight the deployment into the physical system when goals are met. The main users of this module are the safety stakeholders participating in the application risk assessment.

The DEPLOYMENT module, described in Section 6, generates, compiles and launches controls and routines on target hardware components, starting from verified abstract models. The deployment framework provides the architecture for combining hardware interfaces, protocols, commands and data, as well as the runtime middleware for field operations. In collaborative robotics, typical recipient devices of deployed functions are robot controllers, intuitive user interfaces, end-effectors and external axes. Additionally, since the collaborative robotic application is part of a larger automation and production system, the framework supports both horizontal (e.g., other workcells, logistics) and vertical integration (e.g., shopfloor management, factory planning and scheduling). Observations from the field (e.g., sensors, tracking services, user interactions) are implemented to monitor the correspondence between runtime executions and abstract models, to enable both system updates due to new requests and re-planning activities when actual executions do not match their models. This module is primarily used by developers, for hardware and framework configuration, application maintenance and interfacing with devices (e.g., sensor placement to monitor shopfloor activity).

1.1 Structure of the Paper

The rest of this paper first discusses relevant related literature (Section 2), then it provides some details about the features provided by the modules of the toolchain. In particular, Section 3 describes the running example, which is used to illustrate the features of the toolchain and to validate its effectiveness. Section 4 presents the UML profile that we developed, which is the starting point of the whole approach. Section 5 first explains how the toolchain generates instances of the formal model from UML ones; then, it describes how the toolchain allows designers to (i) use generated formal models to assess the safety of the application and (ii) iteratively update the abstract model by introducing new risk mitigating factors. Section 6 describes how the toolchain allows users to deploy the designed tasks in the shopfloor. Section 7 reports on experiments we carried out on two realistic applications to validate the effectiveness of the toolchain. Section 8 discusses how our model-verification and model-deployment loops can support reconfiguration of HRC applications. Finally, Section 9 concludes.

2 Related Work

As mentioned in Section 1, the work presented in this paper covers a variety of aspects concerning the development of HRC applications: high-level task modeling, their safety assessment through an exhaustive analysis of human-robot interactions, their deployment in shopfloors. To the best of our knowledge, there is no available toolchain that covers all aforementioned aspects, even though the various issues tackled by this work have been separately researched in the past.

In this section we briefly explore the literature on modeling and deployment issues. We invite interested readers to find more details in Askarpour [5]; Lestingi and Longoni [27].

2.1 Modeling

Various works model task workflows through Petri nets (e.g., van der Aalst [2], Salimifard and Wright [40]), which have a rigorous mathematical foundation and employ a rather intuitive state-based approach, but are weaker when it comes to capturing more advanced patterns [3]. Petri nets also are at the basis of YAWL [48], which aims to efficiently cover as many advanced patterns as possible.

Safety in HRC environments involves not only workflows, but also other types of features and relations among elements, such as geometry (e.g., the layout of the robotic cell), proximity, or attachments. Hence, a notation focusing exclusively on describing workflows, however effective, is not enough for our goals. Therefore, we broadened the investigation to more comprehensive Domain-Specific Modeling Languages, which are usually part of a wider toolchain involving one or multiple conversions. Blanc et al. [12] propose a model-driven approach for Aibo, one of the robotic pets developed and manufactured by Sony, which is able to react to audio and visual inputs. The proposal consists of three meta-models, *Robot*-, *Validation*- and *Behavior*-specific, which can also be translated into code thanks to predefined sets of rules, one for each meta-class. Another example is V^3CMM by Alonso et al. [4], a 3-View Component Meta-Model directly specified using OMG's Meta-Object Facility (MOF) and including three different views (*structural*, *coordination* and *algorithmic*) of the system under analysis, which can be mapped to platform-specific primitives to generate code. Other DSLs, such as those introduced by Bischoff et al. [11] and Klotzbucher et al. [26], share a similar structure and allow users to model different perspectives of the same system, mostly revolving around *architectural*, *behavioral* and *environmental* aspects critical for deployment- or formal verification-oriented transformations. On the other hand, HRC task-modeling requires to capture similar concepts, but also to maintain a higher-level point of view, since delving in such detail into technical issues (e.g., the instantiation of specific control system components) is less significant than representing the way agents interact with one another.

UML, on the other hand, has enough expressive power and predisposition to automated translation to meet these requirements. Nevertheless, its standard semantics is typically too generic for Domain-Specific applications. A typical approach to overcome this limitation is through stereotypes that provide a custom semantics to standard UML elements, thus casting them in a collaborative industrial task-modeling perspective [43]. These stereotypes are typically collected in a custom UML profile. The strength of this approach has been proved in various works, such as for example Ritala and Kuikka [38], which aims at covering all requirements of a control application, and RobotML [17]. The latter is an Eclipse-based toolchain for the development and deployment of robotic missions. Despite leading to thorough results, it mainly emphasizes the definition of the robotic system's technical components, hence it covers only partially the safety-related concerns that are at the core of our work.

2.2 Deployment

Process development for robot control and abstraction of components have been major objectives for robot software engineering for decades. Among others, Sousa and Ramos [41] present a holonic architecture where the scheduling function, resources and tasks are represented by interacting holons. Scheduling and resource allocation is thus obtained after an initial negotiation phase and holons exchange messages with a proper negotiation protocol. Castelli et al. [16] propose a scheduling system based on a multi-step negotiation protocol by using a detailed simulation model of an existing factory. Strasser et al. [44] present methods for obtaining an adaptive control system which allow dynamic reconfiguration at runtime. Brugali and Scandurra [13] focus on software component encapsulation, describing design principles and implementation guidelines that enable the development of reusable and maintainable software building blocks (task engineering). Brugali and Shakhimardanov [14], instead, discuss the role of software components as architectural units of large, possibly distributed, robotic systems. They address issues related to hardware integration (standard interfaces), with special attention to hardware heterogeneity, computational and communication resources and to design techniques to assemble components into systems. Valente and Carpanzano [47] demonstrate a dynamic scheduling algorithm for automation tasks via task modeling, which offers a control system with more responsiveness, reduced impact of uncertainties and unexpected changes, and higher resource utilization. Task modeling is also the focus of Wang et al. [52], where modular composite function blocks based on sequences of operations are built and activated during runtime. Regarding skill-based task modeling, Backhaus and Reinhart [8] propose a planning module for composing robot applications, ready for deployment through the encoding of planned robot tasks modeled in AutomationML.

Among the possible alternatives of component-based and modular architectures for robot system control, we choose to deploy models to the physical layer using the specifications proposed in the IEC 61499 standard [28, 51, 54]. The standard is based on the notion of Function Blocks (FB), which are modular and reusable entities that constitute one of the best-known approaches to component-based engineering. FBs can be a natural extension of abstract models, encompassing concepts like self-consistent *tasks* and hardware *resources*. Some previous works developed methods for automatic or semi-automatic building of executable targets from higher-abstraction models, which are translated into control components. Panjaitan and Frey [34], which provide an early example of IEC 61499-compliant stand-alone automation application, use a UML activity diagram as starting model for the generation of the IEC 61499 application. The work by M. Plasch [29] and Plasch et al. [35]) introduces an Eclipse plug-in that supports the modeling of workflows and nested operations through the syntax of FBs, and that can automatically translate a model of activity into a IEC 61499 application. Both approaches usually result in rigid control sequences and complex connections, which need a further (manual) optimization phase by the programmer.

3 Running Example

Before explaining the proposed toolchain and its modules, we describe the scenario shown in Fig. 1, used as a running example throughout the paper. This scenario has been used as a showcase demonstration at the EuRoC Project [50], and it is the basis for the empirical evaluation of Section 7.1. Please notice that, although the framework is application-agnostic, in the following sections we describe modeling components tailored to this scenario.

In this scenario, one human operator and one 7-DoF KUKA LBR manipulator arm collaborate to perform a machine tool pallet assembly task. The robot is not mobile, it is mounted on a still platform, and it has two interchangeable end-effectors (a gripper and a screwdriver). The operator has a gesture-detecting armband[1] that sends moving or stopping signals to the robot.

The layout is an industrial manufacturing setting with an assembly pallet, a close-by bin (reachable by the robot arm), and an area for the operator to move to or supervise. The final goal is for the robot and human to pick several workpieces from the bin, take them to the assembly pallet, screwdrive, and tighten them on the pallet. The workpieces should be held on the pallet steady up until the tightening is completed. There are therefore two main activities to be done for each workpiece: bin-picking actions (including picking a workpiece from the bin, taking it to the pallet, holding it while the tightening is executing), and tightening actions (moving to the front of the pallet, positioning the screwdriver on the right spot, screwdrive until the workpiece is tightened, retract from the pallet and move away). These actions are repeated for each workpiece, until the desired number of workpieces are taken care of, creating an iterative loop. In each iteration, bin-picking actions are done by one of the agents (human or robot), and tightening actions are done by the other. The agent roles do not have to be the same for all of the iterations, meaning agents could swap roles at any iteration, which generates multiple variant workflows. Additionally, swapping roles adds changing end-effector actions (detecting the need for changing the end-effector, sending stop signals to the robot, loosening the current end-effector, removing it, placing the right end-effector) to the task. For example, if the operator chooses to do bin picking, then the robot should have a screwdriver end-effector; otherwise, the human should change it. In the opposite case, the end-effector must be a gripper.

One possible variation is shown in Fig. 13, where the human operator (Bill) moves to the bin location, picks a workpiece and brings it to the pallet; in the meantime, he double checks if the robot has a suitable end-effector. The operator then places the workpiece on the pallet and sends an activation signal to the robot. Consequently, the robot arm approaches the pallet, tightens the piece with its screwdriver end-effector, and then retracts from the pallet.

Another variation is for the operator to choose to do the tightening, in which case the robot moves to the bin, goes down the bin, and picks a workpiece with its gripper. It then gets out of the bin while holding the workpiece, takes it to the pallet, approaches the pallet, and places the workpiece on it; then, the human starts tightening the piece, while it is being held by the robot. After finishing, the human sends a signal to the robot, which retracts from the pallet and moves away.

At any point in the middle of the first or second variation, the swapping of roles could happen, thus creating additional workflows.

4 Task Design

This section describes the first module of the toolchain, DESIGN. In particular, it presents the novel high-level modeling notation, based on the UML standard [32], that is at the core of the DESIGN module, and that allows users to describe the different aspects of an HRC application: the task to be collaboratively performed by human operators and robots and the workspace in which they operate. We organized the key elements of human-robot shared tasks, with their architectural and functional relationships, in a UML profile, which is briefly presented in the rest of this section (interested readers can find more details in Lestingi and Longoni [27]). The use of a standard modeling language in the toolchain facilitates the creation of models which can be systematically explored during the two main trans-coding steps (i.e., DESIGN→VERIFICATION and DESIGN→DEPLOYMENT). The profile presented in this paper has been built using standard principles for the creation of UML profiles described, for example, in Gogolla and Henderson-sellers [20], UML [46], Sprinkle [42] and applied in Bruning and Gogolla [15].

[1]Myo by Thalmic Labs, Inc. www.myo.com

The UML profile models the main elements of HRC applications (operators, robots, layout) via *Class Diagrams*. Different sections of the layout and their connectivity are modeled via *Component Diagrams*. Tasks to be executed and their elementary actions are represented by a customized version of *Activity Diagrams*. The rest of this section briefly presents the different elements of the UML profile, using the application presented in Section 3 to exemplify them. A snippet of a model representing our running example is shown in Fig. 4. Note that the approach described in this section is generic and can be used for any other industrial collaborative scenario. The profile presented in Fig. 3 is general, and can be used to describe instances of specific scenarios such as the one shown in Fig. 4, but also others. The rest of this section describes each of the above-mentioned diagrams in some detail.

4.1 Class Diagrams

The two main concepts in the modeling of HRC applications are those of *Resource* and *Layout*, which are captured by the stereotype classes of the same name shown in Fig. 3. Class *Resource* is specialized into sub-classes, which differ for their type of participation to the task goal. In particular, class *Agent* refers to entities which play an *active* role in the task, hence human operators and robotic systems. These classes include attributes that are relevant for the safety assessment of the application. With regards to the operator (see Fig. 4), these refer to her mental and physical state, in terms of *expertise*, *fatigue* and *situation awareness*, which can affect the hazards that arise in the application [6]. The relevant features of robotic devices (whose corresponding classes are not shown here for the sake of brevity)

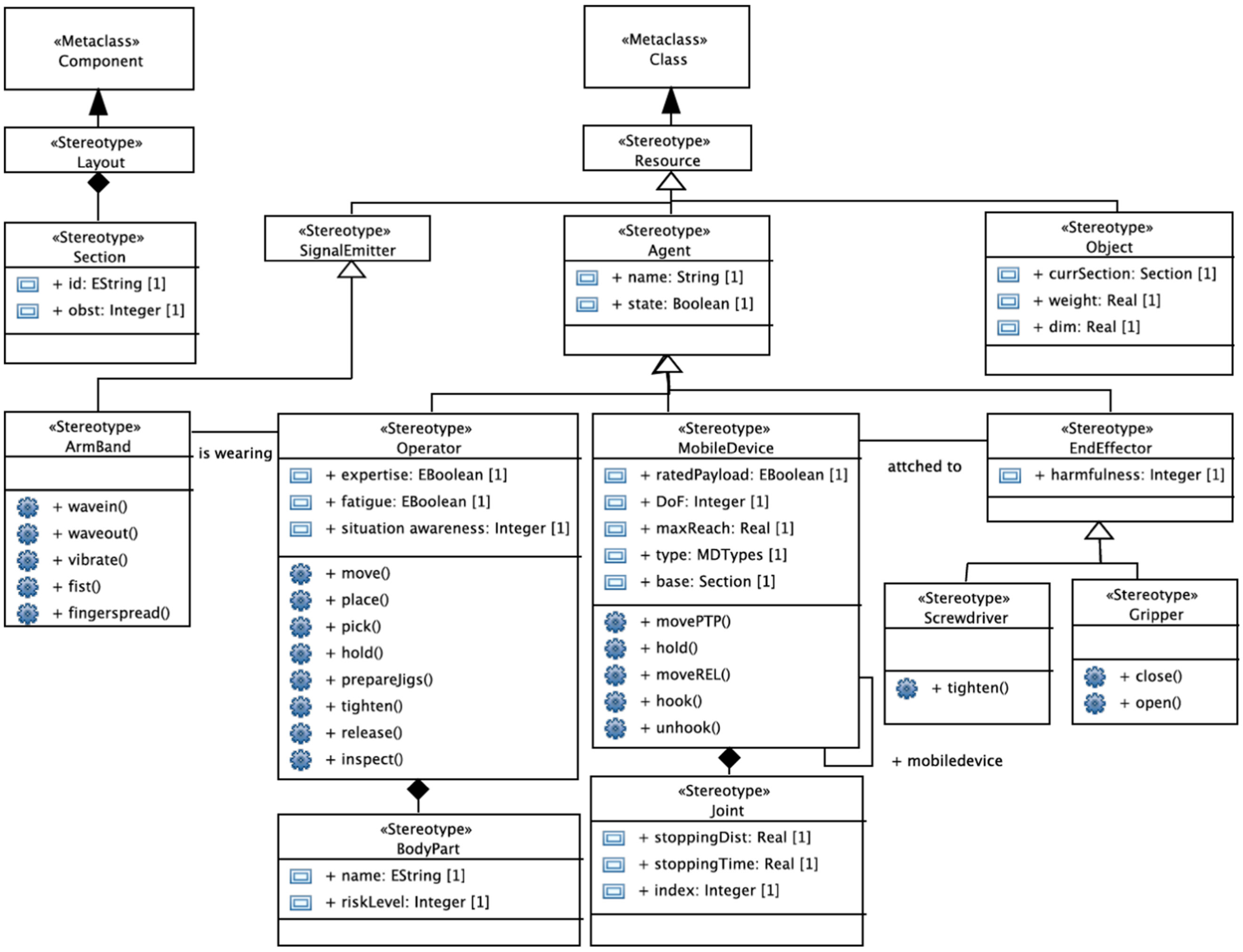


Fig. 3 Meta model of HRC applications, containing their main common concepts

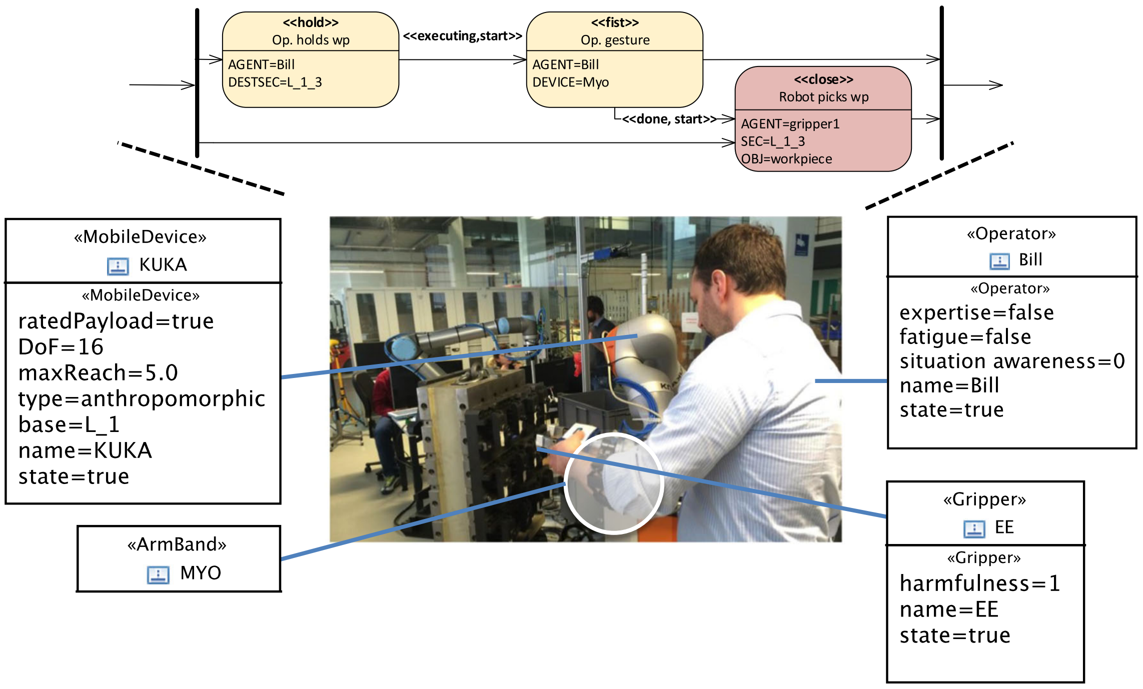


Fig. 4 An example of modeling a HRC application using our proposed UML profile. The operator, robot system and other devices are defined via stereotypes presented in Fig. 3. Additionally, the executing task is modeled by stereotypes on customized activity diagrams which are introduced in Section 4.3

primarily concern their architecture, and in particular their *structure*, *operational workspace*, degree of *base mobility* and *rated payload*. Each agent-type class introduces a set of operations, which represent the elementary actions that the agent is able to perform during a task. As shown in Section 4.3, the steps to be executed by an agent in a task are defined by applying suitable stereotypes to the activities of Activity Diagrams. These stereotypes include, e.g., the ability to *move*, *grasp* and *release* an object, *hold* something in place, *use a tool* (only available to operators), and *hook* and *unhook* an end-effector (only available to robotic arms for tool-changing sequences). Classes *Mobile device*, *Joint*, *EndEffector* capture the various types of elements that can appear in a robotic system. The relations defined between them, instead, capture the kinematic chain constraints—e.g., the fact that at any time the end-effector and the robotic arm cannot reside at opposite ends of the layout—that exist in real life because of physical attachments, and which are then translated into suitable logic formulae (see Section 5.2). Unlike agents, resources that are of type *SignalEmitter* play a *semi-active* role, since they cannot operate autonomously, but provide an extension to an agent's capabilities. Examples of devices are buttons and other interfaces, as well as wearable devices like gesture-detecting armbands. They are particularly useful for communication between operators and robots, for example to issue commands or to receive feedback about the state of the system. Class *Object* of Fig. 3, instead, captures *passive* items manipulated by agents during the task. These include *fixtures* that can be assembled and *workpieces* which will be subjected to manufacturing alterations. Class *Object* does not have any methods, but it includes attributes concerning the geometry of the object—which are parameters required by some operations. To describe an HRC application through the meta model showed in Fig. 3, a safety engineer must define the instances of the elements that are part of the application. For example, the diagrams in Fig. 4 introduce an instance of Class *Operator* (named "Bill"), and an instance of Class *Armband*, i.e., a subset of *SignalEmitter* for gesture recognition.

4.2 Component Diagrams

To capture the *Layout* of an HRC application, our approach employs a grid-like abstraction similar to the one presented by Sadeghpour and Andayesh [39]. In our approach, a *Layout* is discretized and divided in a set of *Sections*. Class *Section* includes attributes describing the *ID* and *obstruction level* of each section.

The actual layout of the application is described through a Component Diagram, wherein each component corresponds to a section of the layout, and connectors represent adjacency constraints between sections. For example, the layout of the case study described in Section 7.2 (the workcell depicted in Fig. 14(top)) is captured by the Component Diagram of Fig. 14(middle).

4.3 Customized Activity Diagrams

The description of the workflow of a desired task, which provides a *dynamic* perspective of the application, is given in terms of Activity Diagrams. Indeed, Activity Diagrams [32] are typically used to represent the procedural flow of actions, focusing on their order of execution and on the conditions that trigger or guard them. We customize standard UML Activity Diagrams through stereotypes that allow us to link each *Activity* to the concepts introduced as class operations in Fig. 3. Figure 4 (top) shows a fragment of an Activity Diagram including custom stereotypes. A task is made of actions, which are modeled as suitably stereotyped UML *OpaqueActions*. More precisely, each *OpaqueAction* must be tagged with a stereotype corresponding to one of the methods appearing in the Class Diagram of Fig. 3. The instances of classes presented in the diagram of Fig. 4 (bottom) illustrate the application of the stereotypes introduced in Fig. 3 to model the instances for the running example explained in Section 3. Indeed, for each operation executable by an agent there is a precise stereotype with a corresponding list of properties. For example, the properties of action movePTP—which represents point-to-point movements made by robots—include the *agent* that performs it, and the layout sections that are the *starting point* and the *destination* of the action.

Other elements of our Customized Activity Diagrams (*ControlFlows*, *ControlNodes*) provide the ordering and synchronization (if any) among actions. *ControlFlow* edges depict precedences between actions. Each *OpaqueAction* can have multiple incoming and outgoing edges, and each edge can have an associated stereotype. Specifically, each edge between two activities a_1, a_2 has both a state-describing stereotype (*done*, *executing*, *hold* or *not started*) and a command-describing stereotype (*start*, *stop*, *pause*, *resume*), where the former identifies the state which must be entered by a_1 for a_2 to execute the transition corresponding to the latter stereotype (see Fig. 4(top)). If the synchronization between a_1 and a_2 is not strict—i.e., a_2 can make the transition sometime after a_1 enters the desired state, but not necessarily immediately—stereotypes SOFTstart, SOFTstop, SOFTpause, SOFTresume can be used. If no stereotype is indicated, by default the *done* state and the *start* command are assumed.

The incoming and outgoing edges correspond to pre- and post-conditions of actions. We use UML *ControlNodes* to represent logical combinations of the various conditions (see also Table 2): *JoinNode* and *MergeNode* represent, respectively, AND and (exclusive) OR combinations of the conditions corresponding to the incoming arcs of the *ControlNode*; dually, *ForkNode* and *DecisionNode* indicate AND and OR combinations of the conditions related to the *ControlNode*'s outgoing edges. This allows for the creation of complex combinations of conditions by cascading UML *ControlNodes*. For example, in the fragment of Customized Activity Diagram shown in Fig. 4 (top), stereotype ≪executing,start≫ on the *ControlFlow* between *Actions* ≪hold≫ and ≪fist≫ states that the latter can start only when the former is executing. Stereotype ≪done,start≫ between *Actions* ≪fist≫ and ≪close≫, instead, means that the former can start only when the latter is completely done. The rightmost *JoinNode* defines that the rest of the task can continue only after both the ≪fist≫ and ≪close≫ actions have been completed. Finally, UML *LoopNodes* are used to concisely depict iterative executions of actions. The *name* field of a *LoopNode* states the number of iterations of the loop. Only two edges can cross the border of a *LoopNode*, one entering the first action of the iteration and one exiting the last action (see an example in Fig. 5).

Models of collaborative applications, and in particular task flowcharts, are created through a prototype tool which has been implemented by customizing the open-source Papyrus UML modeler.[2] The tool allows users to select stereotyped *Actions* through custom palettes (see Fig. 5), to edit the properties of stereotypes and of actions, and to navigate the diagrams. The tool is the front-end and entry point to the design process; it provides the interfaces to launch the automated procedures for trans-coding the models for verification and deployment purposes (see Fig. 2) and to iterate the development procedures.

5 Task Verification

This section describes the SAFETY VERIFICATION module of the toolchain (see Fig. 2). The safety verification of target tasks is based on a formal model that is generated from the UML profile, which only accounts for operational aspects of collaborative tasks. A formal verification tool is used to comprehensively explore and check all possible interactions between robots and human operators, with respect to *risks*, along the designed task. The verification procedure is in charge of exploring also the *unintended* interactions, where some potential hazards may occur.

In the rest of this section we first introduce (Section 5.1) the necessary preliminaries to understand the formal model and its automated verification; then, we present the general structure of the formal model (Section 5.2), the automated translation of UML models into formal ones (Section 5.3), and an additional feature to the toolchain to more realistically reflect the impact of human or hardware errors in an HRC scenario (Section 5.4); finally, we explain how the toolchain can be used to perform the formal verification, analyse the outputs (Section 5.5), and improve

[2]eclipse.org/papyrus

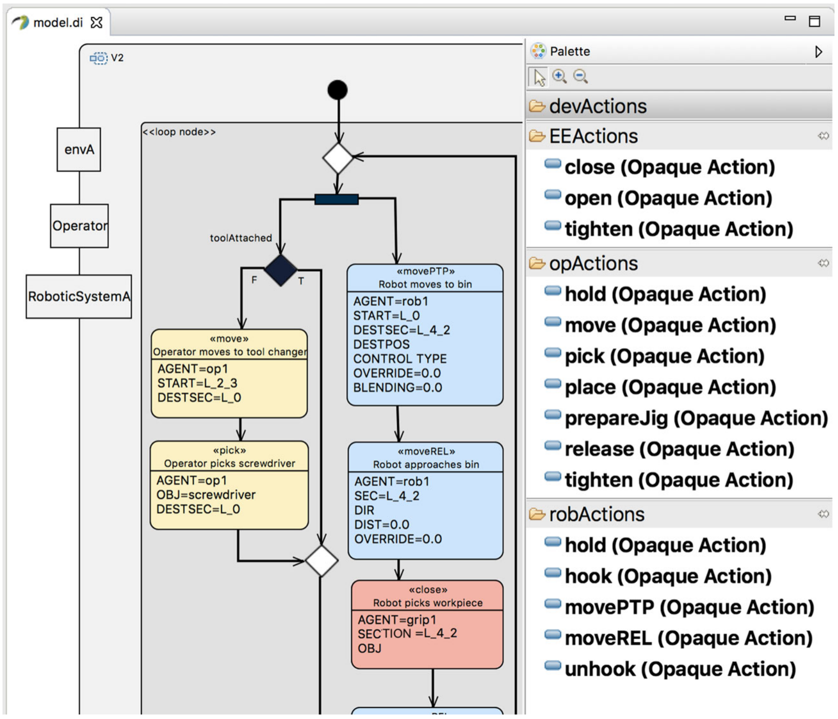


Fig. 5 Screenshot of the Papyrus interface with customized palettes for adding actions of end-effector, operator or robot

the model according to the observed verification results (Section 5.6).

5.1 Preliminaries on Formal Modeling and Verification

The formal model is formulated using a temporal logic language, called TRIO ([19]), suitable for expressing the evolution over time of events and specific situations (e.g., "the operator approaches the pallet *before* the robot turns on the tool"). The TRIO language features a quantitative notion of time, which is useful for describing tasks such as "the robot *always* reaches the working pose 3 time units *after* leaving its homing position". In a typical logic-based approach, all relevant elements of a task being analyzed—e.g., the intended workflow—and the causal relationships among resources are captured through temporal logic predicates and constraints.

TRIO formulae contain the usual first-order connectives, operators, and quantifiers, as well as a single basic modal operator, called Dist, that relates the *current time* to another time instant (see list of operators in Table 1). In particular, given a time-dependent formula ϕ (i.e., a term representing a mapping from the time domain to truth values) and a (arithmetic) term t indicating a time distance (either positive or negative), formula $\mathsf{Dist}\phi, t$ specifies that ϕ holds at a time instant at a distance of exactly t time units from the current one. While TRIO can exploit both discrete and dense time domains, here we assume the standard model of the nonnegative integers $\mathbb{N}$ as discrete time domain.

TRIO formulae are well suited for expressing the temporal characteristics of a HRC system that could affect safety. For example, a different intensity occurs in a collision between human and robot depending on which actor arrives at the collision point first (e.g., they arrive together, or the robot arrives first, hence it is still

Table 1 List of derived TRIO operators; ϕ, ψ denote propositions, and v is a variable and d is a constant value

TRIO Operator	Definition	Meaning
Past (ϕ, d)	$d > 0 \wedge \mathsf{Dist}\phi, -d$	ϕ occurred d time units in the past
Futr (ϕ, d)	$d > 0 \wedge \mathsf{Dist}\phi, d$	ϕ occurs d time units in the future
Alw (ϕ)	$\forall t\, (\mathsf{Dist}\phi, t)$	ϕ always holds
Som (ϕ)	$\exists t\, (\mathsf{Dist}\phi, t)$	ϕ occurs sometimes
Until (ϕ, ψ)	$\exists t\, (\mathsf{Futr}\,(\psi, t) \wedge (\forall t' (0 < t' < t) \Rightarrow \mathsf{Dist}\phi, t^{prime}))$	ψ will eventually occur and ϕ will hold till then
Until$_w$ (ϕ, ψ)	$\mathsf{Until}\,(\phi, \psi) \vee \mathsf{Alw}\,(\phi)$	weak until: ψ may never occur in the future
WithinF (ϕ, d)	$\exists t\, (0 < t < d \wedge \mathsf{Dist}\phi, t)$	ϕ will occur within d time units

when the collision happens). These timing differences can be conveniently modeled in TRIO through its *derived* temporal operators, which are defined from the basic Dist through propositional composition and first-order logic quantification. Table 1 defines some of the most significant ones, including those used in this work.

The satisfiability of TRIO formulae is in general undecidable. However, in this paper we consider a decidable subset of the language, that can be handled by automated tools, to build the system model and to express its properties.

[1] is a bounded satisfiability checker for TRIO formulae ([37]). We use Zot in this work to check the model of the system against desired safety properties. In case the property is not satisfied, Zot provides a counterexample witnessing a system execution that violates the property.

We adopted TRIO and its supporting tool Zot because of the generality of the language, which can be applied in natural way to different types of applications,[3] and for the efficiency of the verification in comparison with the state of the art [36]. For a comprehensive survey and comparison among various logic languages tailored towards modeling time-dependent phenomena see [18].

5.2 Formal Model

This section summarizes the formal model defined in Vicentini et al. [49]. Here we explain the essence of the model to make the current paper self-contained. As mentioned above, the approach presented in this section is general, and it relies on the scenario introduced in Section 3 (and visualized in Figs. 4 and 7) only as an illustrative example.

The human body is discretized into a set $\mathcal{O}$ of eleven body regions $\{O_{head}, \ldots, O_{leg}\}$ [24], where each region $O_i = \langle p_i, v_i, m_i, k_i \rangle$ is characterized by a set of predicates, such as for example p_i, representing the location of body region i, v_i, representing its velocity, etc. Similarly, the model $\mathcal{R}$ of a robot includes a set of predicates $R_j = \langle p_j, v_j, f_j, m_j, shape_j \rangle$ for each of its n links.

As mentioned above, the model of the layout $\mathcal{L}$ includes a tuple of predicates for each of the sections in which it is divided, $L_k = \langle shape_k, material_k, obst_k \rangle$. Some predicates, like $shape$, $material$ or clearance/ occupancy $obst_k$ carry safety-related information.

Every task is broken down into several atomic actions, where each action $a_i = \langle a_i^{exeT}, a_i^{agent}, preC_i, posC_i, a_i^{sts} \rangle$ is characterized by a set of features: a constant (a_i^{exeT}) representing the required time for termination, a predicate (a_i^{agent}) defining the action's expected executor agent (operator or robot), a set $preC_i$ of formulae (pre-conditions) that enable the execution of the action, a set $posC_i$ of formulae (post-conditions) that capture conditions that are enforced by the action at the end of its execution, a predicate (a_i^{sts}) capturing its current state (not started `ns`, waiting `wt`, executing `exe`, safe-executing `sfex`, hold `hd`, done `dn`, exit `exit`).

The behavior of each operator action—i.e., an action in which the agent is the operator—is governed by the finite state automaton depicted in Fig. 6 (a similar automaton, not shown here for brevity, captures the behavior of robot actions), which is formalized in temporal logic.

The transition between states is governed by the features of the action. For example, the transition between states `ns` and `exe` occurs when the pre-condition $preC$ of the action holds; similarly, the transition between states `wt` and `exe` occurs when the operator starts the action within Δ time units from its enabling (i.e., from when the pre-condition of the action holds) which is captured by formula $preC \wedge \text{opStarts} \wedge c \leq \Delta$. For brevity, in this paper we do not show the full formalization of all conditions appearing on the transitions of the automaton of Fig. 6, but only provide, for most of them, an informal name (as in the case of label "opStarts within Δ" in the transition from `wt` to `exe`); interested readers can find further details in Vicentini et al. [49].

Using the features of $\mathcal{O}$, $\mathcal{R}$, $\mathcal{L}$ introduced above, the formal model defines formulae that capture various safety-related aspects of HRC applications. For example, the following formula expresses a kinematic constraint for the

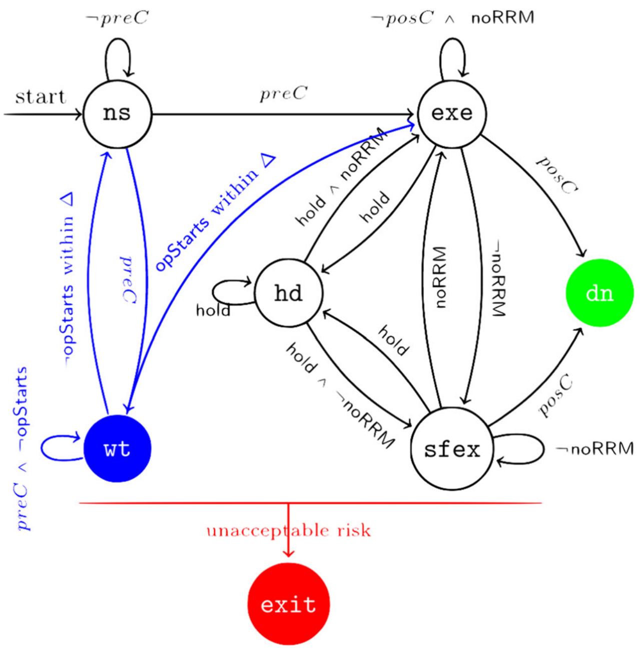


Fig. 6 Finite State Automaton governing the behavior of actions (from Vicentini et al. [49])

[3] Together with MTL, TRIO is one the first temporal logic languages dealing with time in a metric way.

robot through the features of $\mathcal{R}$ and $\mathcal{L}$ introduced above. More precisely, it states that, always, the positions $p_1^{\mathtt{ro}}$ and $p_2^{\mathtt{ro}}$ of the first two attached links of the robot (R_1 and R_2) correspond to the same layout section, or to adjacent ones:

$$\mathsf{Alw}\left(p_1^{\mathtt{ro}} = p_2^{\mathtt{ro}} \vee \mathsf{Adj}(p_1^{\mathtt{ro}}, p_2^{\mathtt{ro}})\right) \quad (1)$$

where $\mathsf{Alw}(F)$ means that formula F holds in all time instants, and Adj is a predicate of $\mathcal{L}$ that defines when sections of the layout are next to one another.

A core part of the formal model concerns the definition of hazardous situations—e.g., contacts in physical human-robot interactions—and of Risk Reduction Measures (RRMs) introduced to lower the risk associated with such hazards. For example, two general types of hazards are introduced by [24]: *quasi-static (Qs)*—sustained contacts of body parts against a constraining object with continuous energy flow from the robot—and *transient (Tr)*—fast contacts where body parts are hit and then recoil because of the kinetic energy transferred to the body. These situations are formulated in terms of spatial and temporal relationships among resources, so that informal conditions like "being in between", "standing close-by", "approaching the same place at the same time", are captured through suitable logic formulae. The following formula is an example of the content of the model and formalizes Qs hazardous contact $\mathsf{hzd}_{ijk}^{\mathsf{Qs}}$ occurring in section L_k of the layout between the i-th robot link (e.g., the end-effector) and the j-th body part (e.g., the human hand):

$$\begin{aligned}\mathsf{hzd}_{ijk}^{\mathsf{Qs}} \Leftrightarrow\ & \mathsf{InSameL}_{ijk} \wedge \mathsf{moving}_{R_i} \wedge \mathsf{Past}\left(\mathsf{Sep}_{ij} > \mathtt{close}, 1\right) \\ & \wedge \mathsf{Past}\left(p_i^{\mathtt{ro}} = L_k \wedge p_j^{\mathtt{op}} \neq L_k, 1\right) \wedge \\ & \left(obst_k = \mathtt{occluded} \vee \exists R_{m \neq i} \in \mathcal{R}\left(\mathsf{InSameL}_{imk}\right)\right)\end{aligned} \quad (2)$$

where $\mathsf{Past}(F, 1)$ means that F held in the previous instant from the current one, $\mathsf{InSameL}_{ijk}$ and moving_{R_i} are abbreviations for formulae describing, respectively, that R_i and O_j are in the same layout section, and that R_i is moving, and Sep_{ij} is a predicate capturing the current separation between R_i and O_j. The last two lines of the formula formalize, respectively, the situation of R_i and O_j getting closer to one another (with the robot arriving in L_k *before* the operator), and the presence of obstacles (i.e., the occlusion of section L_k) in that condition, or the internal crunching of arms inside a manipulator.

Any hazardous situation is scored in terms of the associated risk whenever it is encountered (recall that the actual state is generated at verification time, and not assigned *a priori*). When necessary, TRIO predicates are used to capture the severity and occurrence of hazards according to normative safety methodologies [22]. In particular, the hybrid method for risk estimation in ISO/TR 14121-2 [23] is used for setting severity with discrete scores in $\{1, 2, 3, 4\}$ and occurrence factors (frequency of situations, probability of errors and failures, probability of the avoidability of the situation) with discrete values cumulated in an aggregate score in the $\{1, \ldots, 15\}$ range. Modeled hazards hzd_{ijk} are then assigned a combined score $\mathsf{risk}_{ijk} \in \{0, 1, 2\}$, according to the referenced method. A discussion on the determinism of score assignment and update of values is provided in Vicentini et al. [49].

The formal model includes also the safety strategy to be implemented, if necessary in presence of unacceptable risk level. The following formula, instead, captures an RRM in the family of collision avoidance strategies for preventing contact hazard hzd_{ijk} by activating the safety mode Speed and Separation Monitoring (SSM, [24]):

$$\begin{aligned}\mathsf{RRM}_{ijk}^{SSM} \Rightarrow\ & \left(\mathsf{Sep}_{ij} < \mathsf{Sep}_{min} \Rightarrow \mathsf{Futr}\left(v_{ij} = \mathtt{none}, 1\right)\right) \\ & \wedge \left(\mathsf{Sep}_{min} \leq \mathsf{Sep}_{ij} \leq \mathsf{Sep}_{mid} \Rightarrow \mathsf{Futr}\left(v_{ij} \leq \mathtt{mid}, 1\right)\right)\end{aligned} \quad (3)$$

where $\mathsf{Futr}(F, 1)$ means that F holds in the next instant from the current one, and Sep_{min} and Sep_{mid} are two thresholds on the distance between O_j and R_i, moving at relative speed v_{ij}. Sep_{mid} is used as a threshold for slowing down R_i, so to maintain a Sep_{min} threshold.

5.3 From UML to Formal Models

Models created through the UML profile described in Section 4 are automatically translated into a set of TRIO formulae, using an approach similar to the one presented in Baresi et al. [10] to carry out automated formal verification on UML models. This translation process effectively provides a formal semantics—one that is suitable for automated formal verification—to our custom UML profile, since each one of its elements is associated with a set of formal axioms.

In this section we provide an overview of the translation process, which parses the XMI file created by Papyrus, extracts the relevant information, and generates different parts of the formal model; a more detailed explanation is presented in Lestingi and Longoni [27].

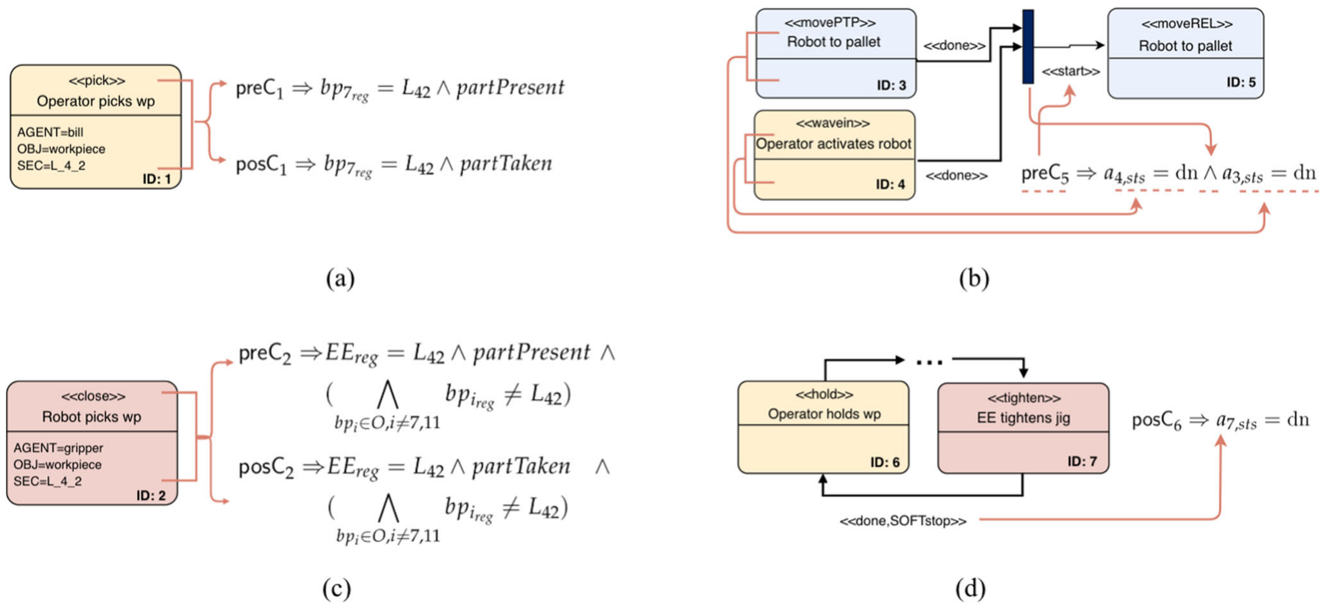
Class and Component Diagrams are used to derive the properties of the actual robots and operators involved in the designed task, and of the layout in which they move. For example, from a component diagram representing the layout of a workcell such as the one of Fig. 14(middle), the tool determines that predicate $\mathsf{Adj}(L_9, L_{23})$ holds, and produces the corresponding formula stating this fact. In addition, the tool automatically produces formulae such as Formula Eq. 1, which define kinematic constraints on robot and human parts, and which rely on the definition of predicate Adj previously introduced.

The formalization of the behavior of the application along time originates from the stereotypes associated with *Actions* and from the control nodes in the Customized Activity Diagram. As described above, each action is formalized through logic formulae that correspond to the automation of Fig. 6. The formalization depends on the action *pre-* and *post-conditions*, whose corresponding logic formulae are produced through an automatic examination of the Customized Activity Diagram capturing the task, depending on the stereotypes associated with edges, the intrinsic nature of *Actions*, their assigned resource, and the type of *Control Nodes* that link actions. This step is recursively applied to each *Action* in the Customized Activity Diagram until all upstream *Actions* are analyzed.

More precisely, for each *Action* i in the Customized Activity Diagram, a set of TRIO formulae formalizing the behavior of the automaton of Fig. 6 is produced. These formulae include *Action*-specific predicates $preC_i$, $posC_i$ capturing the pre and post-conditions that govern the entry and exit of the states of the automaton for that action. The formulae defining predicates $preC_i$, $posC_i$ are generated partly from the stereotypes associated with each action, and partly by recursively examining the structure of the Customized Activity Diagram. Figure 7 shows examples of the two cases.

Consider, for instance, *Action* 2 of Fig. 7b; it is tagged with stereotype ≪close≫, its attribute AGENT indicates that it is executed by the gripper end-effector, and its attribute SEC defines that the operation is carried out in section L_{42} of the layout. This information is used to produce the corresponding definitions for predicates $preC_2$ and $posC_2$. In particular, $preC_2$ implies that: the end-effector must be in the desired section L_{42}; the part to be grasped is present (a condition which is captured by predicate $partPresent$); and the human body parts "lower arm" and "hand"—whose positions are captured by predicates $bp7_{reg}$ and $bp11_{reg}$ in the formal model—are not in section L_{42}. Predicate $posC_2$ is similarly defined. Notice that, together, the pre- and post-conditions of an *Action* tagged as ≪close≫ state that the position of the agent is the same before and after the execution of the action. *Action* 1 shown in Fig. 7a is similarly translated taking into account its ≪pick≫ tag and its attributes.

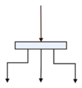
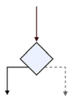
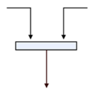
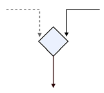
To illustrate how the Customized Activity Diagram structure—i.e., the connections between atomic actions (*OpaqueActions* in UML parlance) and control nodes (*JoinNodes*, *MergeNodes*, etc.)—helps determine actions' pre- and post-conditions, let us consider actions on the right side of Fig. 7. In general, for each *OpaqueAction* that appears in the Customized Activity Diagram the translation mechanism determines its *predecessor* actions by navigating backwards the incoming edges, through the control nodes, until another *OpaqueAction* is reached. In the fragment of Customized Activity Diagram of Fig. 7b,d, the predecessors of *Action* 5 "Robot to pallet" tagged as ≪moveREL≫ are, through the *JoinNode*, *Action* 3 and *Action* 4—tagged, respectively, with the ≪movePTP≫ and ≪wavein≫ stereotypes. The pre- and post-conditions for each action are created from the list of predecessor actions, taking into account the stereotypes (if any) associated with each edge, and the type of control nodes the edges go through. For example, the formula in Fig. 7(b) is produced by considering that the node before *Action* 5 is a join—that is, the "and"—of two flows, originating from *Actions* 3 and 4 and tagged as ≪done,start≫ and ≪executing,start≫, respectively; this corresponds to the condition "*Action* 5 can start only if *Action* 3 and 4 are both done", which is formalized by the formula in the figure. Table 2 summarizes



<<pick>>
Operator picks wp
AGENT=bill
OBJ=workpiece
SEC=L_4_2
ID: 1
$preC_1 \Rightarrow bp7_{reg} = L_{42} \wedge partPresent$
$posC_1 \Rightarrow bp7_{reg} = L_{42} \wedge partTaken$
(a)
<<movePTP>>
Robot to pallet
ID: 3
<<done>>
<<start>>
<<moveREL>>
Robot to pallet
ID: 5
<<wavein>>
Operator activates robot
ID: 4
<<done>>
$preC_5 \Rightarrow a_{4,sts} = \mathrm{dn} \wedge a_{3,sts} = \mathrm{dn}$
(b)
<<close>>
Robot picks wp
AGENT=gripper
OBJ=workpiece
SEC=L_4_2
ID: 2
$preC_2 \Rightarrow EE_{reg} = L_{42} \wedge partPresent \wedge (\bigwedge_{bp_i \in O, i \neq 7,11} bp_{i_{reg}} \neq L_{42})$
$posC_2 \Rightarrow EE_{reg} = L_{42} \wedge partTaken \wedge (\bigwedge_{bp_i \in O, i \neq 7,11} bp_{i_{reg}} \neq L_{42})$
(c)
<<hold>>
Operator holds wp
ID: 6
<<tighten>>
EE tightens jig
ID: 7
$posC_6 \Rightarrow a_{7,sts} = \mathrm{dn}$
<<done,SOFTstop>>
(d)

Fig. 7 Example of pre-/post-condition generation from portions of a customized activity diagram

Table 2 Converting customized activity diagram control nodes to logical connectors for producing pre-/post-conditions

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

how each type of control node is interpreted in the formal model.

As mentioned in Section 4.3, the synchronization between interdependent actions can be either *strict* or *soft*, which have different formalizations. For example, Formulae Eqs. 4 and 5 from Vicentini et al. [49] formalize the difference between stereotypes ≪done,stop≫ and ≪done,SOFTstop≫ applied to a flow between two *Actions* i and j. More precisely, the former means "j stops immediately after i has been completed", whereas the latter means "j stops sometimes after i has been completed".

$$\mathsf{Past}\left(a_i^{sts} = \mathtt{exe} \vee a_i^{sts} = \mathtt{sfex}, 1\right) \wedge a_i^{sts} = \mathtt{dn} \Rightarrow \mathsf{Futr}\left(a_j^{sts} = \mathtt{dn}, 1\right) \tag{4}$$

$$\mathsf{Past}\left(a_i^{sts} = \mathtt{exe} \vee a_i^{sts} = \mathtt{sfex}, 1\right) \wedge a_i^{sts} = \mathtt{dn} \Rightarrow \mathsf{SomF}\left(a_j^{sts} = \mathtt{dn}\right) \tag{5}$$

For example, the formula in Fig. 7d shows that the requirement of *Action* 6 to end is for *Action* 7 to terminate in advance or simultaneously.

5.4 Exploring Human Errors and Hardware Failures

The formal model automatically generated from UML diagrams captures the nominal behavior of the system through pre-/post-conditions. However, deviations from the nominal behavior could strongly affect safety. Hence, we have enriched the model to capture also the operator's *unintended* uses. In [6] we proposed a formal model of human error phenotypes, which captures the possible misuses that violate the *temporal*, *spatial* or *functional* constraints of a task execution. For example, repeating—or attempting to redo—an already terminated action is a temporal error that is formalized by the following formula, which states that a "repetition" error for action x occurs when the latter is started again by the operator even if its status is $\mathtt{dn}$, or when the operator does not terminate it (i.e., $\neg\mathsf{opStops}_x$ holds) even after its post-conditions hold:

$$\mathsf{Repetition}_x \Leftrightarrow a_x^{agent} = \mathtt{op} \wedge \begin{pmatrix} a_x^{sts} = \mathtt{dn} \wedge \mathsf{opStarts}_x \\ \vee \\ \mathsf{Past}\left(a_x^{sts} \neq \mathtt{dn}, 1\right) \wedge posC_x \wedge \neg\mathsf{opStops}_x \end{pmatrix} \tag{6}$$

The full definition of the phenotypes and examples can be found in Askarpour et al. [6].

5.5 Model Verification and Analysis of Outputs

The automatically-encoded formal model can be verified through mechanisms that generate the model dynamics (i.e., the states), analyze the sequences of intended actions, systematically generate also unintended sequences from errors or misuses (see above), and detect unacceptable conditions whenever the level of risk is estimated as non-negligible. Such mechanisms are enabled by the Zot tool,[4] which implements several techniques—employing a so-called bounded satisfiability checking approach based on off-the-shelf Satisfiability Modulo Theories (SMT) solvers—for checking the satisfiability of TRIO formulae in an automated manner (see also Pourhashem Kallehbasti et al. [36]).

The Zot formal verification tool analyzes the global state of the model in every instant of its evolution. The formal definition of a hazard can be matched inside each state, in which case the corresponding risk is estimated—as mentioned in Section 5.2, the estimation of risks associated with hazards (i.e., the function $\mathsf{hzd}_{ijk} \rightarrow \mathsf{risk}_{ijk}$) is part of the model itself. In particular, the Zot tool is used to check all possible states where a hazard is present and whose level of risk is higher than a certain threshold τ. Risk threshold τ is decided at design time; it is usually set to $\tau = 1$ (negligible/low risk) for scores $\mathsf{risk}_{ijk} \in \{0, 1, 2\}$ assigned using the ISO/TR 14121-2 hybrid method (see Section 5.2). A low or negligible risk value (the desired target!) can be obtained if a protective function is already enforced or can be enabled. Such capabilities are captured through predicates RRM_{ijk}^{y}, where y is the type of RRM that mitigates risk_{ijk}. The example RRM_{ijk}^{SSM} provided in Section 5.2 is the model of the form of collision avoidance known as Speed and Separation Monitoring (SSM)—normatively defined in ISO/TS 15066 [24]—which could be activated, for instance, to reduce the risks of the unintended behavior generated by errors in occupying the programmed position as in Section 5.4.

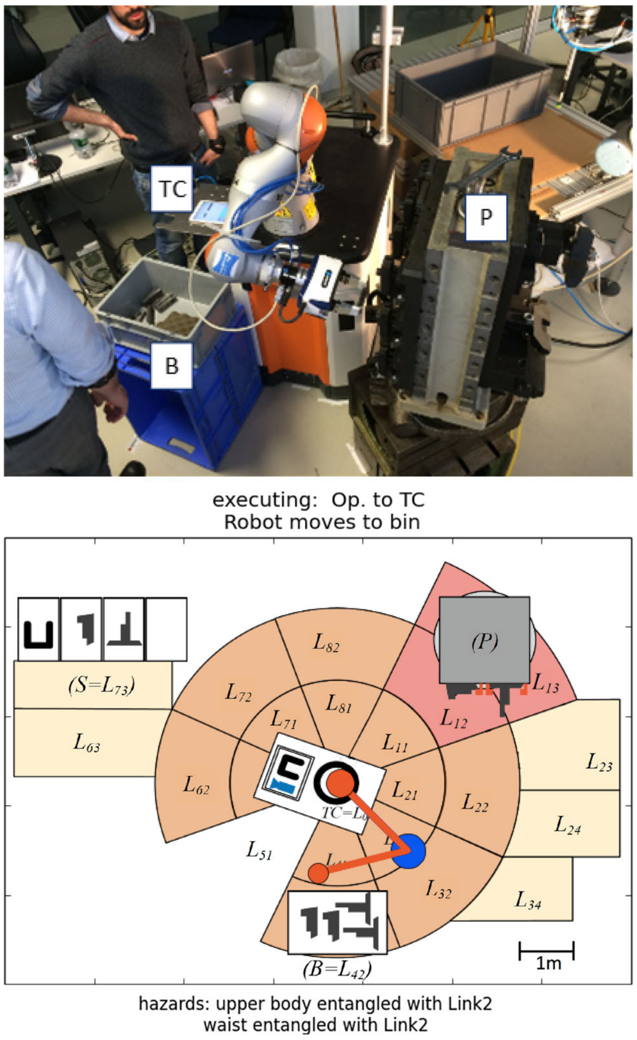
[4]Available from github.com/fm-polimi/zot.

In general, the effect of a RRM^y_{ijk} is to change the severity and/or the occurrence factors of a given risk_{ijk}. This standard identification-mitigation procedure informally describing the process of ensuring safety is formalized by the following property, which states that either the value of risk should always be less than a certain τ, or that there must be a RRM that will mitigate the risk at the next time instant (i.e., $\mathsf{Futr}\left(\mathsf{risk}_{ijk} \leq \tau, 1\right)$ holds) if the risk value exceeds τ. The property needs to be verified for each combination ijk identified as potentially critical.

$$\mathsf{Alw}\left(\begin{array}{l} \mathsf{risk}_{ijk} \leq \tau \vee \\ \mathsf{risk}_{ijk} > \tau \wedge \exists y \left(\begin{array}{l} \mathsf{RRM}^y_{ijk} \wedge \\ \mathsf{Futr}\left(\mathsf{risk}_{ijk} \leq \tau, 1\right) \end{array}\right) \end{array}\right) \tag{7}$$

When the Zot tool is fed the property above to check whether it holds for the model or not, it can either report that no hazards with unacceptable risk levels are present, or it returns a sequence of states which highlights, for each of them, the salient features of the system in that state, and in particular what hazards are present (if any), and what RRMs (if any) are active. The toolchain includes a pair of tools that aid the designer in interpreting the outcome of the formal verification step. The first tool graphically overlays the most critical parts of the operator and the robot on top of a schematic representation of the layout. An example of output from this tool is shown in the middle of Fig. 8, in which the operator's head (blue dot) and the robot (red dots with links, the larger dot being the robot base) are depicted, together with the label of the active hazards at the current step (the top of Fig. 8 shows the real-world application to which the example refers). This rendering can be navigated for each instant, highlighting possible hazardous conditions. The second tool, instead, produces a more detailed textual output which highlights additional information, such as active RRMs. Figure 8 (bottom) shows an example of table output by the second tool, which details, in addition to the active actions, the layout sections occupied by the parts of the operator and by those of the robot, the hazards present, and what RRMs are activated (force limitation in this case).

It should be noted that the safety requirement could be modified or replaced, which would consequently require to update the structure of the model accordingly. For example, different values for τ ($\tau = 1$ or $\tau = 2$) in Formula Eq. 7 could result in defining two very different system behaviors. One allows for the risk value to reach up to two, while the other triggers RRMs before that even happens. This would consequently change the set of introduced RRMs as well. As another example, consider replacing constraint $\mathsf{Futr}\left(\mathsf{risk}_{ijk} \leq \tau, 1\right)$ with formula $\mathsf{Futr}\left(\mathsf{risk}_{ijk} \leq \tau, 5\right)$. It would result in longer allowed intervals for RRMs to mitigate the risk, which means RRMs could be defined very differently to reduce the frequency with which sudden



```
---------------------------------- step 4 ---------------------
Action 23 executing              -Op. to TC 1-
Action 39 done               -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
```

Fig. 8 (top) Aerial view of the workcell of the experiment. (middle) Graphical rendering of a detected (mitigated) hazard. (bottom) Detailed textual description of a state reached by the system (step 4, in the snapshot)

emergency stops are triggered. This affects the trade-off between safety and performance as well. Thus, the desired safety requirement is essential to the definition of the system behavior, and so are any changes made to it.

5.6 Introducing Risk Mitigation Factors

The introduction of a RRM in the model/target is a decisive design action, which is ultimately under the responsibility of an overall safety supervisor. Different types of measures y (i.e., different RRM^{y}_{ijk}) can be chosen for mitigating risk_{ijk}, depending on the capabilities of the robot system (i.e., controls, sensors, etc.), and also on the possibility to modify the task itself, if necessary. As a consequence of the compliance with safety requirements, production routines may be variously affected by the RRM strategy. For instance, slower safe modes in robot controls, re-arranged layouts, or forcing constrained action sequences by controlling access to locations, all have a substantial impact on the organization and/or efficiency of tasks.

The criteria for the selection of RRMs are a matter of performance-based design; also, multiple RRMs are often available as suitable solutions for attaining the required risk reduction. Providing guidance in the selection of RRMs is currently out the of scope of the presented toolchain; still, the toolchain indirectly supports this step, by allowing users to analyze the effects of potential solutions on risks [7]. Indeed, although it is possible to let a computational unit automatically choose the RRMs to be introduced, the final decisions must be taken by the person who is ultimately liable for the deployment of the task under analysis. The verification tool can provide support and valuable insight to make these choices, but in industrial scenarios an explicit acknowledgment of executed procedures is necessary.

Introducing an RRM, or altering the task, whatever is the agent in charge of completing such procedure, corresponds to updating the model, which needs to be then verified again before any deployment. Re-verification and deployment are actually part of the main workflow of our toolchain.

6 Deployment

This section describes the third module of the toolchain, DEPLOYMENT (see Fig. 2).

Vertical and horizontal integration in factories benefit from robust middlewares and frameworks, which are able to interface robots with different automation protocols and data-representation standards.

In our toolchain, we selected the IEC 61499 standard to support the access to the physical control layer and to other automation protocols. This standard provides specifications for compiling and interfacing data and events of modular Function Blocks (FBs) for building event-driven architectures. Its design objective is to support the integration of distributed and scalable automation systems [54].

Basic Function Blocks (BFB) defined by the standard have a simple behavior for handling events, with a standardized automaton-like Execution Control Chart (ECC). Since BFBs can be nested, we designed a custom FB, called Action Function Block (AFB) representing a template for robot (or human) actions. From typical execution profiles in robotics, actions are split into a preparation phase (e.g., setting control strategies, configuring parameters) and the actual running phase. The AFB is then able to synchronize events (requests for configuration) and continuous streams of data. As such, AFBs represent pure logic components of program execution, handling dataframes and signals at runtime. There is a one-to-one functional correspondence between AFBs and abstract models of actions, so that AFBs are used to build deployable UML *Actions* (see Section 6.3). Another type of standardized FBs—i.e., Service Interface Function Blocks—is instead used for coding the interfaces for actual resources. *Resource* FBs are dedicated to individual physical units (robots, sensors, drivers, etc.). Examples of implemented protocols include EtherCAT and POWERLINK fieldbuses and ROS publish/subscribe mechanisms (see Fig. 9, and Iannacci et al. [21] for full implementation details). Hardware resources can be reused and allocated by several applications—i.e., several chains of *action* FBs, according to the target workflows. Multiple blocks of type AFB are connected with data and event streams to individual *Resource* FBs, which in turn have event/data access policies for synchronizing and scheduling the actuation of individual actions from the hardware resource.

The rest of this section first provides some details on the decision to use modular FBs (Section 6.1), then introduces some preliminary knowledge required to develop FBs (Section 6.2), and finally describes the automated translation of UML models into FBs (Section 6.3).

6.1 Motivation and Advantages of Using Modular FBs

The design principle facilitated by FBs is the breakdown of applications in logical sub-tasks (actions) entirely separated from the resources executing them. This approach is compatible with the abstractions used in the DESIGN and VERIFICATION modules.

With the assigned granularity of FBs to individual actions, FB-based applications can be effectively distributed across different target nodes through any middleware implementing the specification of the IEC 61499 standard. Standardized interfaces provided by FBs allow for the recursive composition of groups of connected AFBs. From the scalability standpoint, entire clusters of devices and associated tasks can be nested and relocated in (i.e., controlled from) different parts of an automation network. For robotic applications, it is straightforward to wrap factory-relevant information (e.g., from/to ERPs), including data and events, in standard interfaces to generate compatible bridges/gateways.

On top of standard interfaces that offer maintainability, FBs and connection rules encompass a low variability

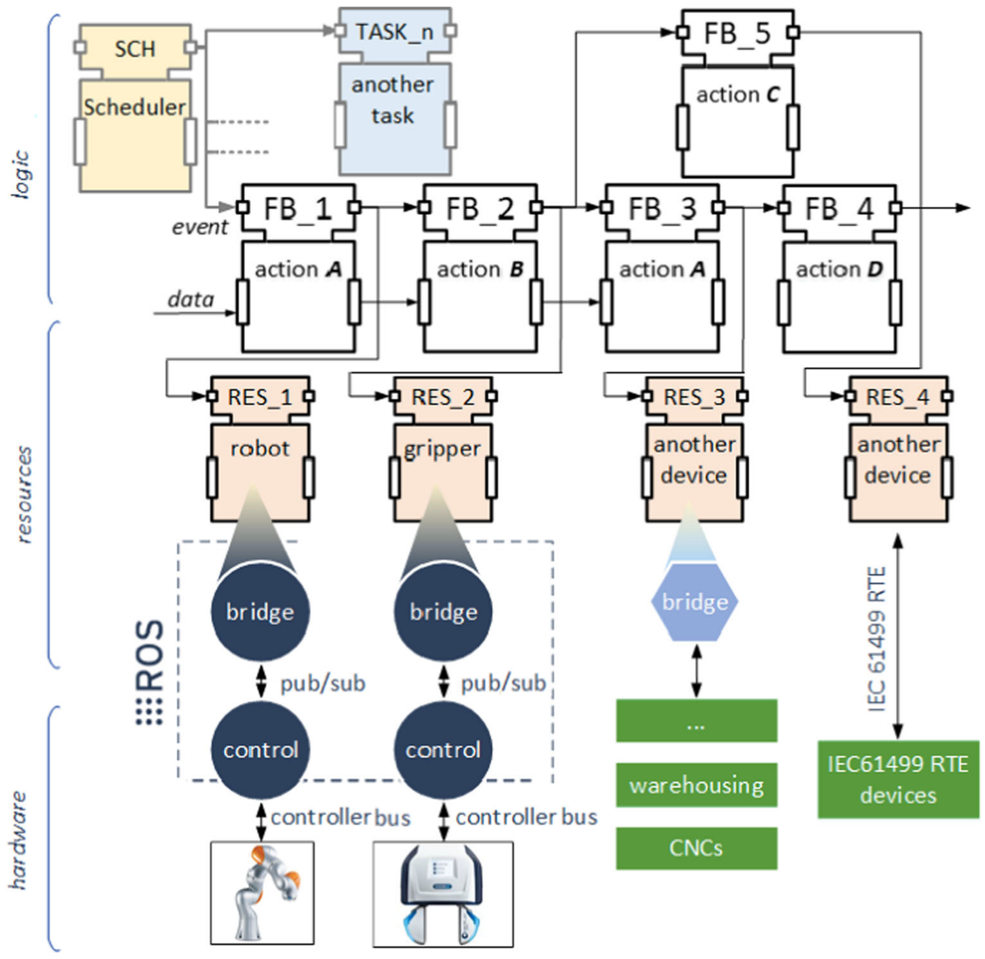


Fig. 9 Architecture of the deployment framework, based on IEC 61499. Tasks correspond to flowcharts of logic-FBs FB_n. The example sequence of actions is A-B-||(A-D,C). Multiple applications can be allocated to available resources. Other tasks (Task_n) and a scheduling node (SCH) depict the potential concurrent executing tasks. In *resources*, each FB (RES_n) is dedicated to a single device interface, and could be shared by multiple logic-FBs, from multiple applications. In *hardware*, controllers or protocols are interfaced through bridges implemented in the RES_n blocks, to parse protocol-specific dataframes

language (LVL) which is effective for verification, self-composing and self-diagnostics. In particular, the properties of FB interfaces are exploited for developing a self-generating application able to assemble AFBs and resources.

6.2 Modes and Tools for FB Development

The IEC 61499 standard is implemented in tool suites that include an Integrated Development Environment (IDE) and a runtime component. These suites enable the definition, composition and building of FBs into runtime components deployed at corresponding hardware targets (e.g., I/O devices, actuator drives, PCs). We use in particular the open-source 4DIAC-FORTE project[5] for supporting our deployment module.

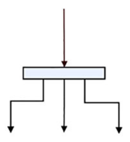
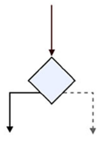
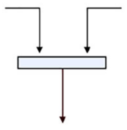
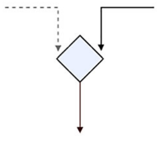
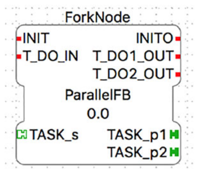
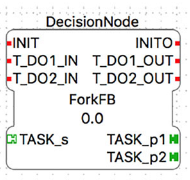
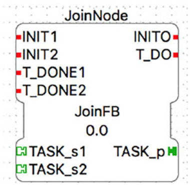
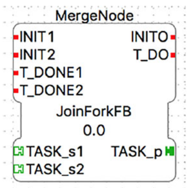
When designing and building applications according to the IEC 61499 standard using stand-alone IDEs, developers can take advantage of features such as visual editing of FBs to create application workflows, the availability of FB libraries for components or functions, and an extensive compatibility with PLC development through IEC 61131 languages. Graphical programming supports a clear identification of workflows and their deployed (expected) behavior in executable applications.

This desirable property is however shadowed by the integration of the IEC 61499 modelling standard and language inside our toolchain, where the design of applications is supported by an even cleaner layer of abstraction (see the UML models in Section 4). FB executables are built automatically from higher-abstraction specifications, without manual intervention.

Inside our toolchain, 4DIAC IDE is used only at configuration or maintenance time for the building of resource-FBs and basic logic-FBs—i.e., for generating or maintaining a library of general-purpose components. 4DIAC IDE is of course used for developing the stand-alone utility application that *automatically generates* all subsequent actual applications from the verified abstract model.

[5]4DIAC-FORTE (www.eclipse.org/4diac) is the foremost IEC 61499-compliant IDE and runtime environment suite.

Table 3 Connectors function block counterparts

	ForkNode	DecisionNode	JoinNode	MergeNode
Logic connector				
Function block	ParallelFB  ForkNode INIT INITO T_DO_IN T_DO1_OUT T_DO2_OUT ParallelFB 0.0 TASK_s TASK_p1 TASK_p2	ForkFB  DecisionNode INIT INITO T_DO1_IN T_DO1_OUT T_DO2_IN T_DO2_OUT ForkFB 0.0 TASK_s TASK_p1 TASK_p2	JoinFB  JoinNode INIT1 INITO INIT2 T_DO T_DONE1 T_DONE2 JoinFB 0.0 TASK_s1 TASK_p TASK_s2	JoinForkFB  MergeNode INIT1 INITO INIT2 T_DO T_DONE1 T_DONE2 JoinForkFB 0.0 TASK_s1 TASK_p TASK_s2

6.3 From Verified UML Models to the Shopfloor

The transformation of verified abstract models into target builds is based on the ability of FBs to self-generate executable applications, thanks to the so-called *reconfigurability* property of the IEC 61499 standard [54].

The procedure is based on two elements. First, abstract UML models built using the profile presented in Section 4 are explored and parsed into a set of configuration files that encode the number of resources, names of agents, names of activities, etc. For example, logic connectors introduced in the Customized Activity Diagram are translated into existing blocks shown in Table 3. Edges and control flows of diagrams also are converted to configuration files. Second, `taskGen`, which is a stand-alone IEC 61499 utility application depicted in Fig. 10, generates and connects ordered networks of FBs according to such configuration files. A managing GENERATE_TASK FB carries the whole application label TaskGenFB, initiates the reading of the configuration files, and issues the following events: CREATE, which triggers the CREATE_FB and START_FB blocks for creating and starting other FBs according to their FB_NAME listed in the configuration files; SET, which triggers the SET_PARAMETER FB needed to set the value of a custom block's input parameter; CONNECT, which triggers the CREATE_CONNECTION FB that is in

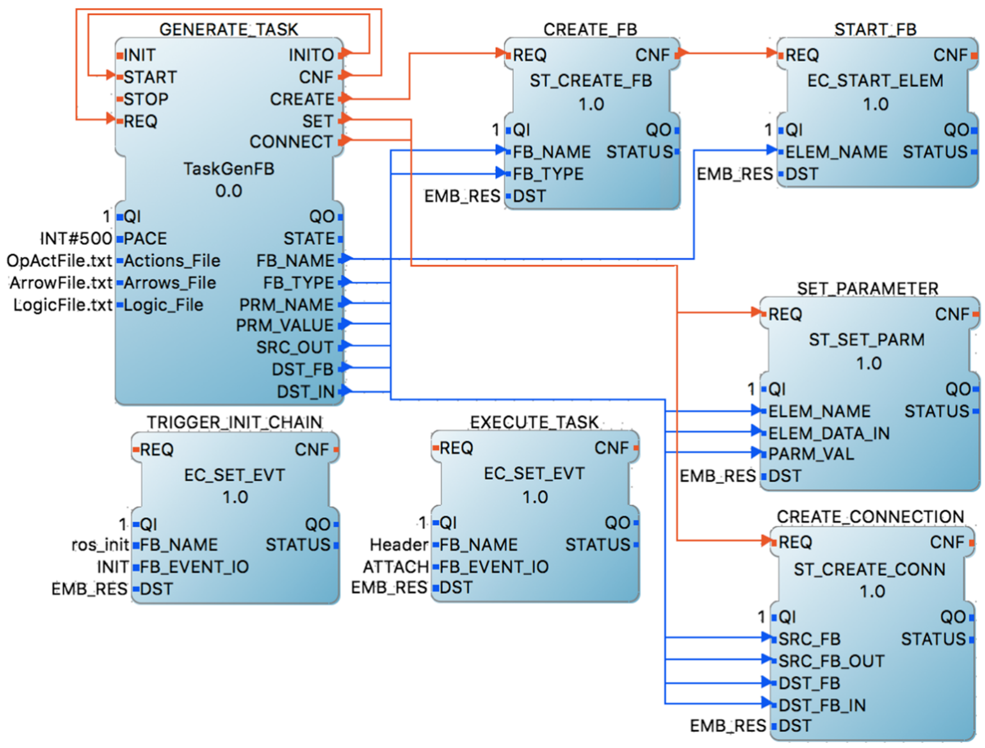

GENERATE_TASK
INIT INITO START CNF STOP CREATE REQ SET CONNECT
TaskGenFB 0.0
1 QI QO
INT#500 PACE STATE
OpActFile.txt Actions_File FB_NAME
ArrowFile.txt Arrows_File FB_TYPE
LogicFile.txt Logic_File PRM_NAME
PRM_VALUE SRC_OUT DST_FB DST_IN
CREATE_FB
REQ CNF ST_CREATE_FB 1.0
1 QI QO FB_NAME STATUS FB_TYPE EMB_RES DST
START_FB
REQ CNF EC_START_ELEM 1.0
1 QI QO ELEM_NAME STATUS EMB_RES DST
SET_PARAMETER
REQ CNF ST_SET_PARM 1.0
1 QI QO ELEM_NAME STATUS ELEM_DATA_IN PARM_VAL EMB_RES DST
TRIGGER_INIT_CHAIN
REQ CNF EC_SET_EVT 1.0
1 QI QO ros_init FB_NAME STATUS INIT FB_EVENT_IO EMB_RES DST
EXECUTE_TASK
REQ CNF EC_SET_EVT 1.0
1 QI QO Header FB_NAME STATUS ATTACH FB_EVENT_IO EMB_RES DST
CREATE_CONNECTION
REQ CNF ST_CREATE_CONN 1.0
1 QI QO SRC_FB STATUS SRC_FB_OUT DST_FB DST_FB_IN EMB_RES DST

Fig. 10 Screenshot of the FB-App generating network

charge of actually building the topology; CNF, an output event that keeps the process alive, issuing a command to be processed in response to a REQ input event. REQ elaborates the currently required command and issues one of the events listed above. TRIGGER_INIT_CHAIN and EXECUTE_TASK are utility blocks for initializing and enabling the execution of the application. The parsing tool is available as an extension of the Papyrus modeler and includes a procedure for the launching of the `taskGen` FB application (refer to Lestingi and Longoni [27] for full details). The pre-requisite for running `taskGen` is the availability in a local 4DIAC library (hence also in the runtime environment) of individual *action* FBs templates that are compiled off-line and capture basic abilities of robot systems (e.g., move, grasp, set). The main advantage of this approach is the possibility of maintaining and updating the library templates independently from any single code generation process and across different implementations.

At build time, `taskGen` maps UML *Action* elements featuring the same ID, agent and parameters into corresponding *action* FBs, as represented in Fig. 11.

Additionally, `taskGen` maps every connection between activities—i.e., every *Control Flow* in the Customized Activity Diagram—into FB connectors (event and data wires). Then, the stereotypes of *Actions* are mapped into groups of AFBs, depending on the rules described in the configuration files (e.g., the state of the source action and the command for the target action). Stereotypes, in fact, codify the way in which AFBs must be ordered and triggered. Indeed, setting the topology and the policy for triggering events is the most critical step in creating consistent FB applications. Notably, connections are usually the major source of bugs and deadlocks in the manual development of FB applications.

7 Testing and Evaluation of the Toolchain

In this section, we provide a step-by-step walkthrough of our toolchain, and then apply it to two test-case scenarios—described in Section 7.1 and Section 7.2, respectively—to evaluate its effectiveness. The second test case, in particular, is larger and more complex than the first one in terms of layout and combinations of tasks, to evaluate the scalability of our toolchain. Finally, Section 7.3 provides a comparison between the results (in terms of hazards identified) obtained through the application of our toolchain to the test case of Section 7.2 with those obtained through a manual safety assessment carried out by experts in the field.

Figure 12 shows the process through which users apply the toolchain in practice. (Steps 1-2) The users (safety assessors and application designers) use the GUI to create a model of the application through the approach described in Section 4, and to launch the translation into the formal model. (Step 3) The formal representation of the UML model is automatically created. (Step 4) The produced formal model is checked with Zot. (Step 5) The results of verification are reported via screenshots/textual reports. (Step 6) The user can choose to modify the UML model and iteratively repeat steps 2-5, until the desired model is achieved. (Step 7) Once the formal model is verified, the user can choose the FB translator via the GUI. (Step 8) The configuration files are automatically created. (Step 9) The user can use and external simulator (e.g., ROS nodes) to validate the correctness of the model. If the simulation highlights a problem in the application, the process can be repeated from step 1 by modifying the UML model. The next section explains how the process works in practice by applying it to a prototype assembly task.

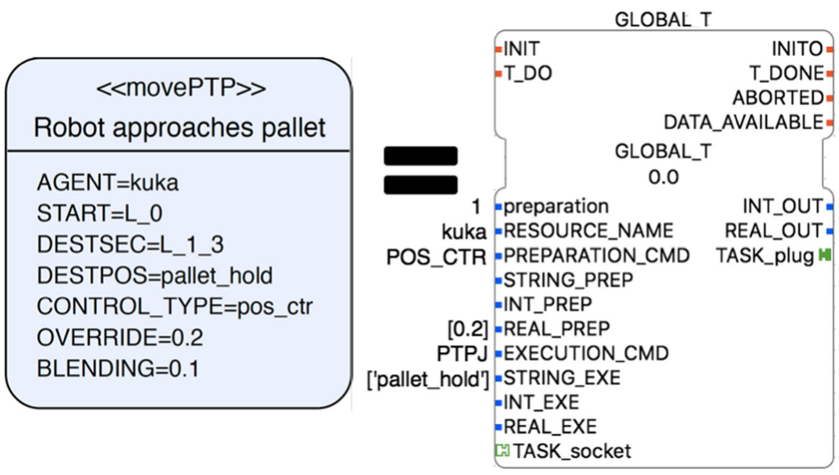


Fig. 11 Example of transformation of a UML *Action* (left, from Papyrus) into an *action* FB (right, from 4DIAC IDE). Class GLOBAL_T mentioned on top of the FB defines a general-purpose action in our toolchain

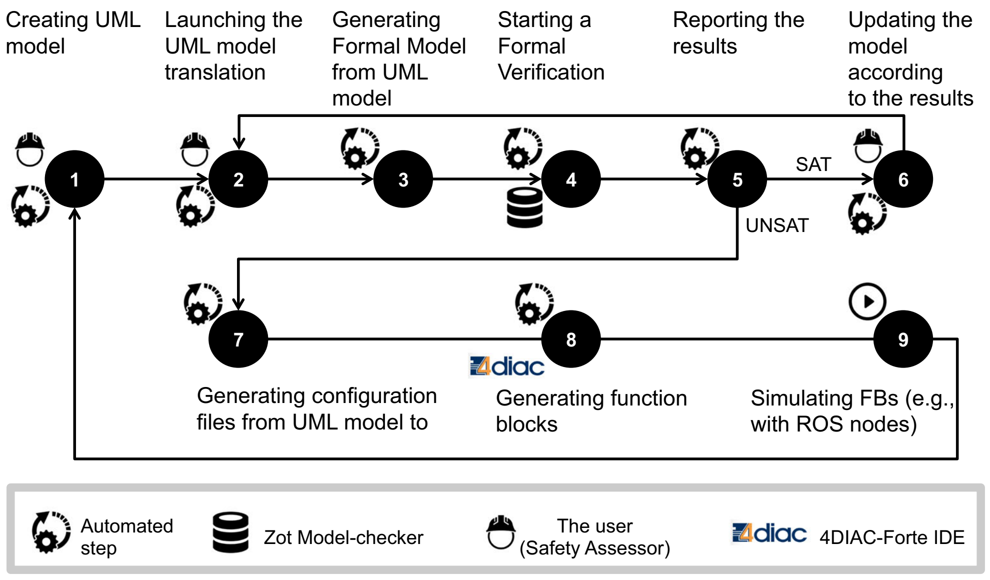

Creating UML model
Launching the UML model translation
Generating Formal Model from UML model
Starting a Formal Verification
Reporting the results
Updating the model according to the results
1
2
3
4
5
6
SAT
UNSAT
7
8
9
4diac
Generating configuration files from UML model to
Generating function blocks
Simulating FBs (e.g., with ROS nodes)
Automated step
Zot Model-checker
The user (Safety Assessor)
4diac 4DIAC-Forte IDE

Fig. 12 Flowchart showing a walkthrough usage of the toolchain. Fully automated steps alternate with steps that require interaction of the user with the tool (depicted with dual symbols). The last deployment step is in a ready-to-play state, potentially re-entering the flowchart for model upgrade/modification

7.1 Experiments on a Prototype Assembly Task

The scenario of Section 3 has been tested during a demonstration run. The main sequence of the intended use is modeled as a Customized Activity Diagram with three main variants, of which one is depicted in Fig. 13. The safety verification procedure in this experiment had three main steps (verify, improve, update).

Formal Verification First, we prepared the initial model of the task and formally verified it through the Zot tool against the property captured by Formula Eq. 7 of Section 5.5. By analyzing the output as shown, e.g., in Fig. 8, we found several false positive hazards due to the unrealistic positioning of the operator/robot allowed by the model. During this first iteration, it was possible to analyze the degree of approximation introduced by the discretization of the layout and model dynamics (physical interactions, kinematics, etc.). The issue of granularity of the model representation is fully discussed in Vicentini et al. [49].

Second, we updated the model by constraining some pre-/post-conditions to avoid certain situations, and we introduced new risk mitigation factors (RRMs) in the model, or combined existing ones. In the deployment stage, such changes corresponded to either updates in robot programs, or signal-event conditions, or safety configurations (e.g.,

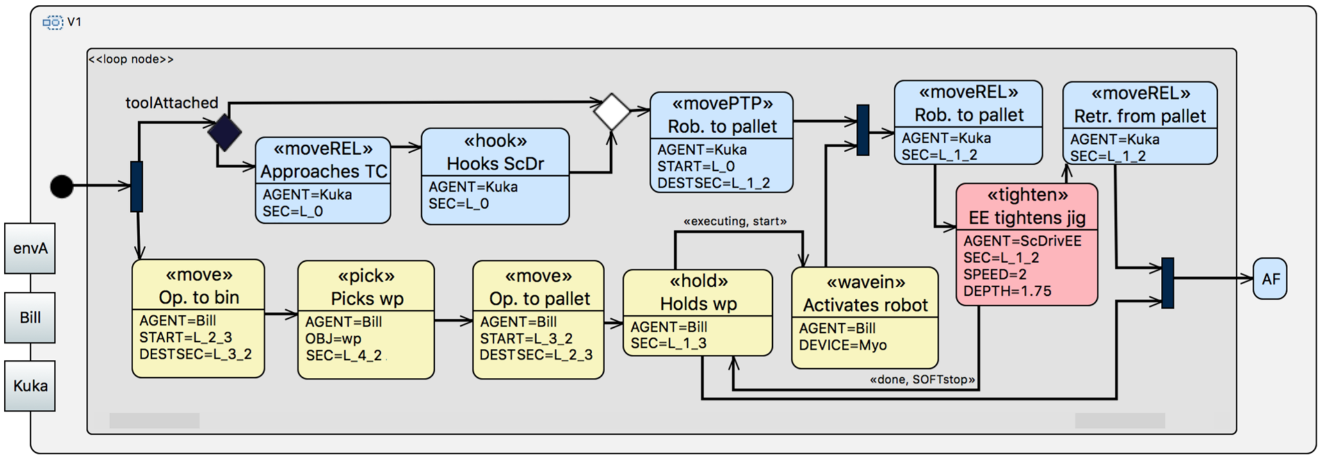


Fig. 13 The customized activity diagram of one possible executional variation of the first experimental case

speed or force limits). Once the verification did not highlight unmitigated hazards, the system entered the execution stage (see Section 6).

Finally, during the regular execution of the task (deployment of the nominal workflow), a test operator decided to insert an inline inspection of the parts, amidst the assembly sequence. Expecting a major effect on layout occupancy and physical interaction by the operator, the model was updated and verified before actually enabling the deployment of the newly requested feature. Again, after a run of Zot verification had highlighted potential hazards, some robot force limits and position zoning were added to the robot controller, corresponding to constraints in the formal model about robot properties and accessible layout sections, respectively. These modeling actions corresponded to the introduction/selection of RRMs. After a new run of verification, the routines were added to the deployment flow and executed.

Different runs (model versions) had different execution times and different exploration depths. Each run explored at least 40 steps in the task (enough to analyze the task to its completion) and it did not take more than 480 seconds[6] of real time to perform a complete exploration of every possible execution trace within the search depth.

Deployment Here we discuss how the Steps 7-9 shown in Fig. 12 apply to this case study. We have launched the tool to generate the related configuration files. Then, the so-generated files are fed to the fixed application-generating FB network pictured in Fig. 10, and the generation process is initiated by manually triggering `TaskGen-FB` INIT event. This operation is performed through the 4DIAC-IDE 1.81 interface, with FORTE 1.8 M1 running on a Linux virtual machine. The generation process is perceived as almost instantaneous with timing parameter (`PACE()`) set to 500ms. When FORTE does not report any error during the creation and connection phases, the application is ready to be either deployed or simulated. We have used ROS nodes with looped messages from/to the hardware layer, to avoid the actual reading of command topics by the hardware interface nodes. Despite its limitations, the rendered simulation still allowed us to conclude that the generated application contains the correct actions, and it properly replicates the logic connections among them described in the original Customized Activity Diagram.

7.2 Experiments on a Larger Task

Scalability has always been a critical issue in formal verification. As the model grows, the verification time naturally increases; oftentimes the increase is considerable, showing the so-called state-space explosion problem. To verify the scalability of our toolchain, we have analyzed a larger case study, in which the original manipulator is set on top of a mobile unit able to autonomously relocate in a layout shown in Fig. 14. The robot unit can travel and access the whole workspace (the blue area) and moves between load/unload areas for raw materials and finished parts, and three assembly stations—1, 2 and 3—and a sensor-based inspection station 4, as shown in Fig. 14a. Two human operators (OP_1 and OP_2) are employed in the application. OP_1 is mostly present in stations 1 and 2, while OP_2 works mainly in 3, or executes auxiliary manual tasks on the workbench in 4. Both operators can freely hold and resume their tasks, swap posts, or join one another in some area. The main robot-assisted tasks are: pallet assembly at stations 1 and 2, including bin-picking from a local storage carried by the mobile unit; pallet disassembly (reversal of assembly) at 1 and 2, including bin-dumping; pallet inspection at station 3; lead-through programming of assembly, disassembly, and inspection tasks (trajectories, parameters, etc.) at stations 1, 2 and 3; material handling on load/unload areas. Other manual tasks by OP_1 and OP_2 include manual loading of parts/boxes; visual inspection of pallet at stations 1, 2 and 3; manual assembly/disassembly of pallet at stations 1 and 2; manual measurements of parts at station 4; cleaning pallets at stations 1 and 2; kitting of tools and parts at stations 1, 2 and 3; general supervision (programming other tasks at HMI during operations, consulting production data at HMI, etc.) at stations 1, 2 and 3. Note that *all* combinations of robot/manual task assignments are admitted (e.g., robot holds and OP_1 screw-drives jigs and vice versa, switching tasks on the fly, quitting a manual task and assigning the robot to proceed autonomously). Frequently, robot base and operators move side-to-side across the central aisle, or other operators may transit along the aisle because the target area is part of a larger plant and access to it is not restricted (Fig. 14(middle)).

Figure 14(middle) shows that the workspace is discretized in 23 sections with different characteristics. Each section is captured by a component of the diagram that—in the formal model—is given a two-level layered representation: more precisely, we have upper and lower layers, with boundary set at human hip height, rendering a realistic partitioning of a 2.5D volume where torso/arms and legs are correctly located. The mobile base of the robot is always allowed in the lower layer, while the manipulator arm can move in both layers. In this case study, not all areas have static aspects. The robot is constantly moving around, and some areas—mostly those situated on the aisle—can have different characteristics from time to time, depending on the presence of the robot unit.

First, after describing the layout (Fig. 14) and the workflow (Fig. 15) through UML diagrams, we used the

[6]On commodity hardware, a Linux desktop machine with a 3.4 GHz Intel® Core™ i7-4770 CPU and 16 GB RAM.

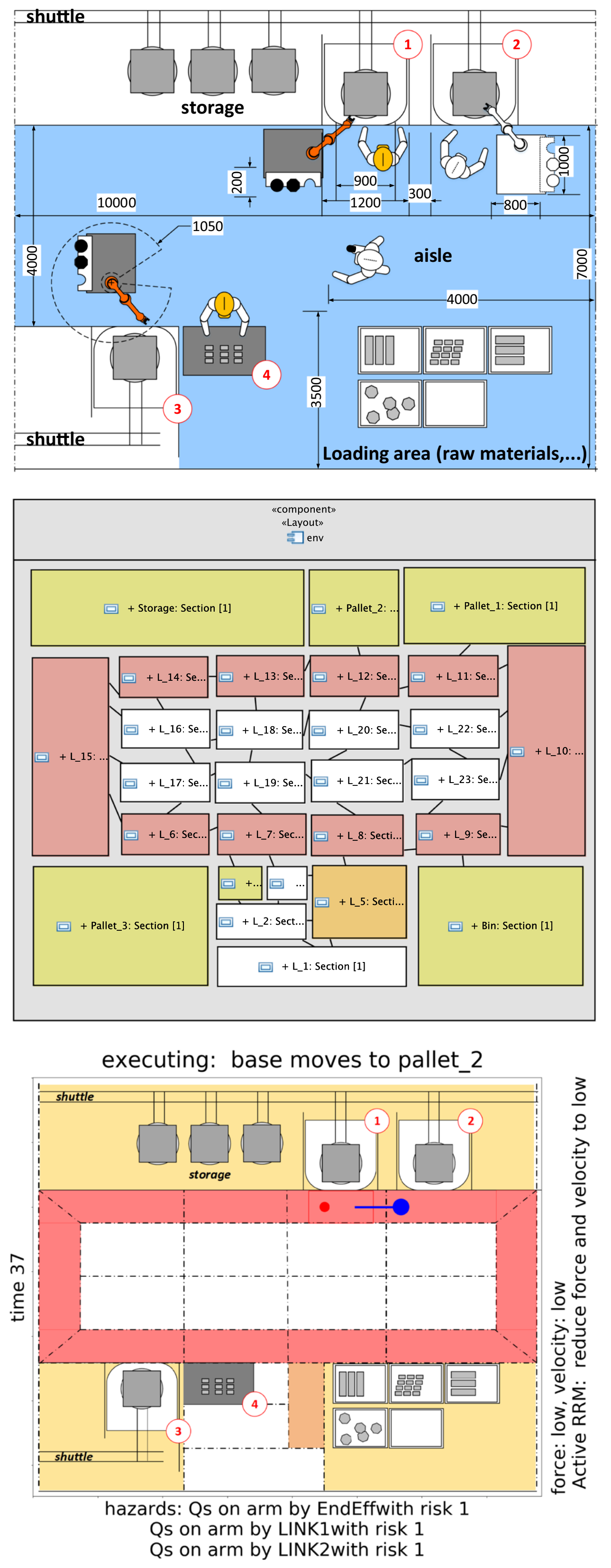


Fig. 14 Second case study: (top) precise workcell depiction; (middle) component diagram describing the layout in the UML model; (bottom) tool exemplary output

tool to generate the formal model. In every iteration, we learned new issues about the correct definition of pre-/post-conditions of actions in order to make the flow of actions realistic. The tool provides warnings and messages to acknowledge any error or impractical definition in the model.

Second, we analyzed the outputs of the tool produced in the previous step (see Fig. 14(bottom) as an example), introduced RRMs accordingly in the model, and iteratively updated and analyzed the model until the verification highlighted no unmitigated hazards.

Each run explored at least 45 steps (finishing at least the job for one pallet) in the task and did not take more than 1200 seconds[7] of real time to perform a complete exploration of every possible trace within the search depth. More details on this test case with and without human erroneous phenotypes is provided, respectively, in Askarpour et al. [6] and Vicentini et al. [49].

7.3 Validation of the Approach Compared to Manual Assessment

We argue that our approach produces stronger results than manual or simulation techniques while requiring less time and effort. We have asked a different team, expert in machinery safety, to apply their regular safety analysis (which is done manually) on the example of Section 7.2. Table 4 summarizes the results of a comparison between our results and those of the manual analysis and shows that the formal verification approach finds various instances of hazards more comprehensively ($60 >> 31$). Note that the 20 man-hours reported as the required time for our approach includes multiple runs of the experiment plus the amount of the time that Zot took to produce an output, while 40 hours are for manually performing the safety analysis only once. However, we do not claim that every safety requirement has been tackled. Moreover, it seems that our approach finds a different balance of Tr and Qs hazards ($73\% - 23\%$ as opposed to $55\% - 45\%$), which could be an indicator to the fact that abstraction and discretization of the system model could have caused loss of precision in detecting contacts. Hence, we suggest our technique to be complemented with 3D simulators as well to increase the level of confidence of the final results.

8 About Task Reconfiguration

After deployment, many practical situations contribute to deviate from the designed nominal task. Examples include,

[7]On commodity hardware, with a 2,6 GHz Intel® Core™ i5 CPU and 8 GB RAM.

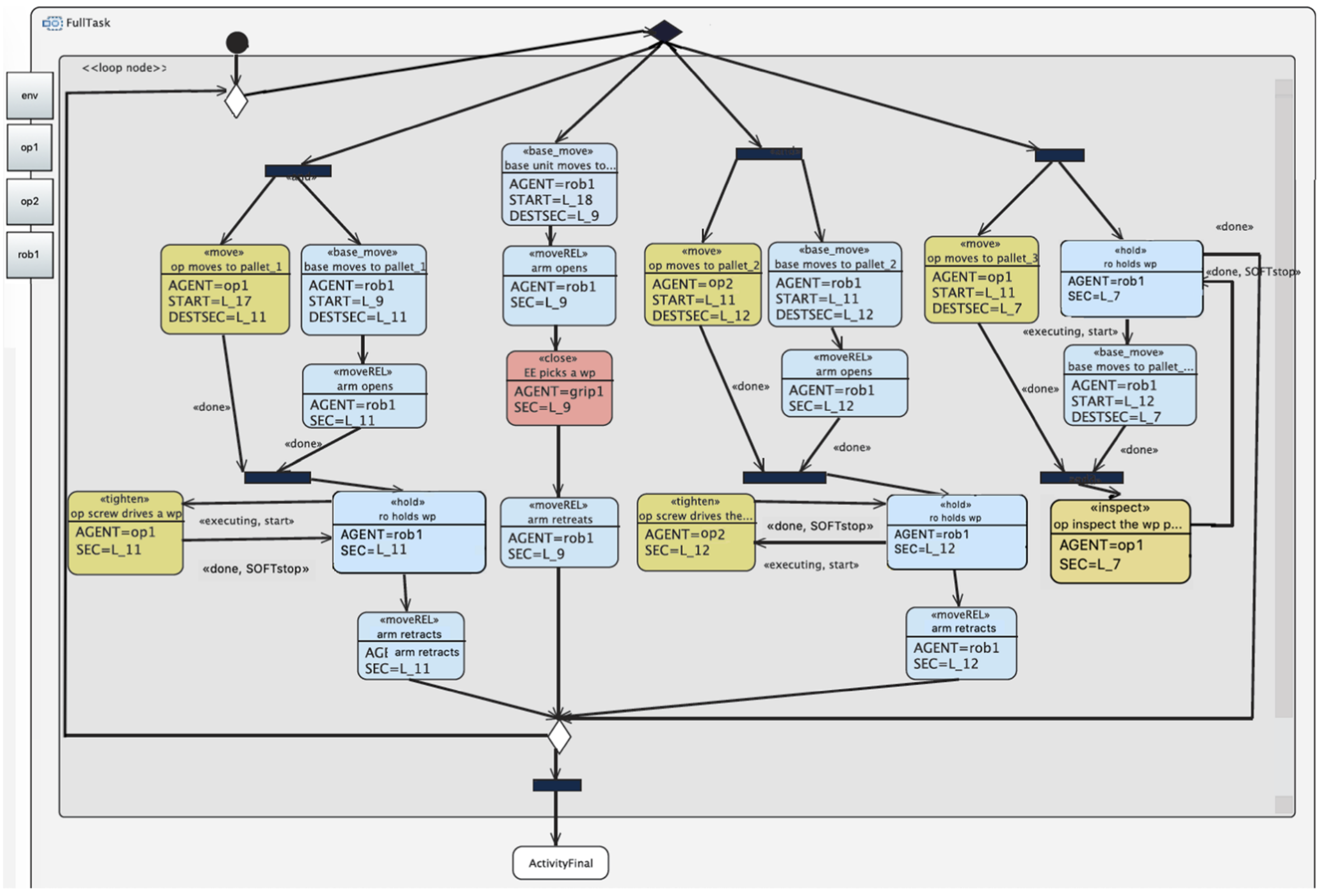


Fig. 15 Snippet of Customized Activity Diagram of the workflow for the second case study, showing one of the possible sequences of actions. This snippet corresponds to a sequence of "bin-picking", "inspection at pallet 3 by operator 1 (op1)" and "unloading at pallet 2 by op2" activities executed iteratively (as many times as defined by the value assigned to the loop node). The complete Customized Activity Diagram allows more alternations between the two operators

but are not limited to: modification of robot poses, replacement of robot tools, replacement of fixtures, re-arrangement of portions of layouts, (purposeful) modification of manual sequences. Some changes are major, and trigger the necessity of performing a new risk assessment. Some changes are minor—e.g., the *limited* variation of the geometry of a robot tool—and likely have a negligible effect on production (task execution) and safety conditions. Nonetheless, it is usually very hard to predict the long-term effects of discrepancies in execution, when these accumulate. For instance, cascading effects on altered procedures may increase the rate of error or unintended use of machinery. Also, in model-based planning of tasks at factory level, excessive drifts of the actual execution with respect to the plan can have disruptive effects on global optimization.

On the other hand, task alteration is a core feature of lean and adaptable solutions: reconfiguration of layouts and re-sequencing of schedules are examples of purposeful operations regularly present in hybrid human-robot environments. Such alterations need of course to be tracked and propagated in the abstract model of tasks to ensure the consistency of available information and the validity of possible updates [45]. For instance, in the experiment of the pallet assembly task (Figs. 1 and 8) the addition of a single inspection action in the middle of the regular flow had disruptive effects on the whole sequence because of the resulting major

Table 4 Validation of our approach by comparing it to the results of manual safety assessment

	#Hazards	#Tr Hazards	#Qs Hazards	Required Time
Manual Assessment	31	17(55%)	14(45%)	40 man-hours
Formal Verification	60	44(73%)	16(23%)	20 man-hours

timing alterations, new transient situations that arise when resuming the regular execution, and much closer direct physical interaction between robot and operator.

Task reconfiguration has significant effects on development and—especially—on safety verification. Indeed, reconfiguration cannot be underestimated since it has a non-negligible rate of occurrence in practice. The main advantage of a development toolchain, like the one proposed in this work, is the ability to support the automatic (re)generation of verification models and deployable components from any model update. The design module, in fact, has the ability to automatically translate the high-level UML model into both the formal verification model (see the actions-to-formal-predicates transformation of Fig. 7) and the controller to be reloaded (see the actions-to-IEC-61499-AFBs transformation in Fig. 11). Fully- or highly-automated procedures for reconfiguring and maintaining developed tasks offer greater chances of obtaining high *availability*, which in terms of production contributes to limit the system downtime. Efficient toolchains, in fact, can be seen as major contributors to the overall *dependability* of robotic applications, which in case of collaborative solutions is of utmost relevance. Finally, semi-automated safety verification procedures decrease the costs of safety assessments, which can be substantial with respect to the overall development costs.

9 Conclusion

In this paper we have introduced a tool-supported model-driven approach to (i) create (formal) models of HRC applications, (ii) automatically perform formal verification on them, and (iii) deploy corresponding function blocks. The effectiveness of the presented toolchain has been validated by carrying out some experiments on a few realistic case studies, two of which have been described in this paper.

The prototype tool generates temporal logic models from UML diagrams annotated with suitable stereotypes to facilitate the creation of formal models. The UML-based approach facilitates the development of HRC applications by allowing users to quickly modify existing designs, to easily build new task models as variations of previous ones, and to continuously and automatically verify the safety of the design with each new iteration. The tool also allows users to deploy corresponding function blocks via the 4DIAC IDE framework.

Future work will focus on improving the tool by including, for example, functions to automatically suggest RRMs and modifications to the design depending on the detected hazards.

Author Contributions **Mehrnoosh Askarpour:** Methodology, Conceptualization, Software, Writing - original draft. **Livia Lestingi:** Software, Investigation, Writing - original draft. **Samuele Longoni:** Software, Investigation, Writing - original draft. **Niccolò Iannacci**: Conceptualization, Resources. **Matteo Rossi:** Conceptualization, Supervision, Writing - review & editing, Funding acquisition. **Federico Vicentini:** Conceptualization, Resources, Writing - review & editing.

Funding Open access funding provided by Politecnico di Milano within the CRUI-CARE Agreement. The Italian Ministry of Education, University and Research is acknowledged for the support provided through the Project “Department of Excellence LIS4.0 - Lightweight and Smart Structures for Industry 4.0”.

Availability of data and materials The datasets generated and/or analysed during the current study are available from the corresponding author on reasonable request.

Declarations

Ethics approval Not applicable (this article does not contain any studies with human participants or animals performed by any of the authors).

Consent to participate Not applicable (this article does not contain any studies with human participants or animals performed by any of the authors).

Consent for Publication All authors have approved the manuscript and agree with its publication on Journal of Intelligent & Robotic Systems.

Competing interests The authors have no financial or proprietary interests in any material discussed in this article.

References

1. Zot: a bounded satisfiability checker. available from http://github.com/fm-polimi/zot (2012)
2. van der Aalst, W.M.: Three good reasons for using a petri-net-based workflow management system. In: Proc. of the Int. Working conf. on Info. and Process Integration in Enterprises, pp. 179–201. Citeseer (1996)
3. van der Aalst, W.M., ter Hofstede, A., Kiepuszewski, B., Barros, A.: Workflow patterns. Distrib Parallel Databases **14**(1), 5–51 (2003)

4. Alonso, D., Vicente-chicote, C., Ortiz, F., Pastor, J., Alvarez, B.: V3CMM: a 3-view component meta-model for model-driven robotic software development. JOSER **1**(January), 3–17 (2010)
5. Askarpour, M.: Safer-HRC: a methodology for safety assessment through formal verification in HRC. Doctoral dissertation, Politecnico di Milano (2018)
6. Askarpour, M., Mandrioli, D., Rossi, M., Vicentini, F.: Formal model of human erroneous behavior for safety analysis in collaborative robotics. Robot Comput. Integr. Manuf. **57**, 465–476 (2019)
7. Askarpour, M., Lestingi, L., Buran, F., Rossi, M., Vicentini, F.: Model-driven risk analysis for the design of safe collaborative robotic applications. In: 2020 IEEE International Conference on Human-Machine Systems (ICHMS), pp. 1–6 https://doi.org/10.1109/ICHMS49158.2020.9209450 (2020)
8. Backhaus, J., Reinhart, G.: Digital description of products, processes and resources for task-oriented programming of assembly systems. J Intell Manuf **28**(8), 1787–1800 (2017)
9. Banziger, T., Kunz, A., Wegener, K.: Optimizing human–robot task allocation using a simulation tool based on standardized work descriptions. J Intell Manuf (2018)
10. Baresi, L., Morzenti, A., Motta, A., Pourhashem Kallehbasti, M.M., Rossi, M.: A logic-based approach for the verification of uml timed models. ACM Trans. Softw. Eng. Methodol. **26**(2), 7:1–7:47 (2017)
11. Bischoff, R., Guhl, T., Prassler, E., Nowak, W., Kraetzschmar, G., Bruyninckx, H., Soetens, P., Haegele, M., Pott, A., Breedveld, P., Broenink, J., Brugali, D., Tomatis, N.: BRICS - best practice in robotics. In: Proc. of ISR and ROBOTIK, pp. 968–975 (2010)
12. Blanc, X., Delatour J, Ziadi T: Benefits of the MDE approach for the development of embedded and robotic systems application to aibo. Proc of the Workshop on Control Architecture of Robots (2007)
13. Brugali, D., Scandurra, P.: Component-based robotic engineering (part i)[tutorial]. IEEE Robot. Autom. Mag. **16**(4), 84–96 (2009)
14. Brugali, D., Shakhimardanov, A.: Component-based robotic engineering (part ii). IEEE Robot. Autom. Mag. **17**(1), 100–112 (2010)
15. Bruning, J., Gogolla, M.: Uml metamodel-based workflow modeling and execution. In: IEEE 15th Int. Enterprise Distr. Object Computing conf., pp. 97–106 (2011)
16. Castelli, L., Nicola, A., Pesenti, R., Ukovich, W.: Autonomous agent system using dispatching rules in the negotiation protocol. In: Kulianic, E. (ed.) AMST'02 Advanced Manufacturing Systems and Technology, pp. 577–584. Springer, Vienna (2002)
17. Dhouib, S., Kchir, S., Stinckwich, S., Ziadi, T., Ziane, M.: 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)
18. Furia, C.A., Mandrioli, D., Morzenti, A., Rossi, M.: Modeling time in computing: A taxonomy and a comparative survey. ACM Comput Surv **42**(2), 6:1–6:59 (2010)
19. Furia, C.A., Mandrioli, D., Morzenti, A., Rossi, M.: Modeling time in computing. Monographs in Theoretical Comp. Sci. An EATCS Series, Springer (2012)
20. Gogolla, M., Henderson-sellers, B.: Analysis of uml stereotypes within the uml metamodel. In: Proceedings of UML, pp. 84–99. Springer (2002)
21. Iannacci, N., Giussani, M., Vicentini, F., Molinari Tosatti, L.: Robotic cell work-flow management through an IEC 61499-ROS architecture. In: ETFA, pp. 1–7. IEEE (2016)
22. ISO 12100: Safety of machinery – General principles for design – Risk assessment and risk reduction. Int. Organ. for Standardization (2010)
23. ISO/TR 14121-2: Safety of machinery – Risk assessment – Part 2: Practical guidance and examples of methods. Int. Organ. for Standardization (2012)
24. ISO/TS 15066: Robots and robotic devices – Collaborative robots. Int. Organ. for Standardization (2016)
25. Jiménez, P.: Survey on assembly sequencing: a combinatorial and geometrical perspective. J Intell Manuf **24**(2), 235–250 (2013)
26. Klotzbucher, M., Soetens, P., Bruyninckx, H.: Bcm: A minimal robotic component model for multitarget system and component generation. Tech. rep., Technical report, Best Practice in Robotics, EU FP7 (2010)
27. Lestingi, L., Longoni, S.: HRC-TEAM: A model-driven approach to formal verification and deployment of collaborative robotic applications. Master's thesis, Politecnico di Milano (2017)
28. Lewis, R.: Modelling distributed control systems using iec 61499: Applying function blocks to distributed systems. 59, Iet (2001)
29. Plasch, M., Rooker, M., Pichler, A.: Simplified programming of modular robotic systems based on workflow modeling. In: Austrian Robotics Workshop (2012)
30. Maoudj, A., Bouzouia, B., Hentout, A., Kouider, A., Toumi, R.: Distributed multi-agent scheduling and control system for robotic flexible assembly cells. J Intell Manuf (2017)
31. Nielsen, I., Dang, Q.V., Bocewicz, G., Banaszak, Z.: A methodology for implementation of mobile robot in adaptive manufacturing environments. J Intell Manuf **28**(5), 1171–1188 (2017)
32. OMG: OMG unified modeling language™(OMG UML). Tech. Rep. March, Object Management Group. http://www.omg.org/spec/UML/2.5 (2015)
33. Pacaux-Lemoine, M.P., Trentesaux, D., Rey, G.Z., Millot, P.: Designing intelligent manufacturing systems through human-machine cooperation principles: A human-centered approach. Comput Ind Eng **111**, 581–595 (2017)
34. Panjaitan, S., Frey, G.: Functional design for IEC 61499 distributed control systems using uml activity diagrams. In: Int. Conf, Instrumentation, Communication and Information Technology, pp. 64–70 (2005)
35. Plasch, M., Pichler, A., Bauer, H., Rooker, M., Ebenhofer, G.: A plug & produce approach to design robot assistants in a sustainable manufacturing environment. In: 22nd Int. Conf. on Flexible Automation and Intelligent Manufacturing (2012)
36. Pourhashem Kallehbasti, M.M., Rossi, M., Baresi, L.: On how bit-vector logic can help verify ltl-based specifications. IEEE Trans. Softw. Eng: 1–15 (2020)
37. Pradella, M., Morzenti, A., San Pietro, P.: Bounded satisfiability checking of metric temporal logic specifications. ACM TOSEM **22**(3), 20:1–20:54 (2013)
38. Ritala, T., Kuikka, S.: UML automation profile: Enhancing the efficiency of sw development in the automation industry. In: Proceedings of INDIN, pp. 885–890 (2007)
39. Sadeghpour, F., Andayesh, M.: The constructs of site layout modeling: an overview. Canadian J. Civil Eng. **42**(August 2014), 199–212 (2015)
40. Salimifard, K., Wright, M.: Petri net-based modelling of workflow systems: An overview. European J. Oper. Res. **134**, 664–676 (2001)
41. Sousa, P., Ramos, C.: A distributed architecture and negotiation protocol for scheduling in manufacturing systems. Computers in Industry **38**(2), 103–113 (1999)
42. Sprinkle, J., Rumpe, B., Vangheluwe, H., Karsai, G.: 3 metamodelling. In: Dagstuhl Workshop on Model-Based Engineering of Embedded Real-Time Systems, pp. 57–76. Springer (2007)
43. Staron, M.: Improving modeling with UML by stereotype-based language customization. Doctoral dissertation, Blekinge Institute of Technology (2005)

44. Strasser, T., Zoitl, A., Auinger, F., Sunder, C.: Towards engineering methods for reconfiguration of distributed real-time control systems based on the reference model of IEC 61499. Springer (2005)
45. Strasser, T., Zoitl, A., Auinger, F., Sunder, C.: Towards engineering methods for reconfiguration of distributed real-time control systems based on the reference model of IEC 61499. Springer (2005)
46. UML O: 2.4. 1 superstructure specification. Tech. rep., document formal/2011-08-06. OMG (2011)
47. Valente, A., Carpanzano, E.: Development of multi-level adaptive control and scheduling solutions for shop-floor automation in reconfigurable manufacturing systems. CIRP Annals-Manuf Technol **60**(1), 449–452 (2011)
48. Van Der Aalst, W.M., Ter Hofstede, A.H.: Yawl: yet another workflow language. Info Sys **30**(4), 245–275 (2005)
49. Vicentini, F., Askarpour, M., Rossi, M.G., Mandrioli, D.: Safety assessment of collaborative robotics through automated formal verification. IEEE TRO **36**(1), 42–61 (2020)
50. Vicentini, F., Pedrocchi, N., Beschi, M., Giussani, M., Iannacci, N., Magnoni, P., Pellegrinelli, S., Roveda, L., Villagrossi, E., Askarpour, M., Maurtua, I., Tellaeche, A., Becchi, F., Stellin, G., Fogliazza, G.: PIROS: Cooperative, safe and reconfigurable robotic companion for CNC pallets load/unload stations, pp. 57–96. Springer International Publishing, Cham (2020)
51. Vyatkin, V.: IEC 61499 function blocks for embedded and distributed control systems design. ISA-Instrumentation, Systems, and Automation Society (2007)
52. Wang, L., Keshavarzmanesh, S., Feng, H.Y.: A function block based approach for increasing adaptability of assembly planning and control. Int J Prod **49**(16), 4903–4924 (2011)
53. Zhang, J., Ding, G., Zou, Y., Qin, S., Fu, J.: Review of job shop scheduling research and its new perspectives under industry 4.0. J Intell Manuf (2017)
54. Zoitl, A., Strasser, T.: Distributed Control Applications: Guidelines, Design Patterns, and Application Examples with the IEC 61499. Industrial Information Technology. CRC Press, Boca Raton (2016)

Mehrnoosh Askarpour is an adjunct assistant professor at McMaster University. Her current research interests include verification of safety-critical system properties and application of formal methods for safe robotics and autonomous vehicles.

Livia Lestingi earned her M.Sc. degree in Automation Engineering from Politecnico di Milano in 2017. She is currently a Ph.D. student in Information Technology at Politecnico di Milano. Her research interests include the analysis of human-robot interaction through formal methods and formal modeling techniques of human behavior.

Samuele Longoni has completed his University career at Politecnico di Milano with a Master Thesis regarding formal validation and deployment of human-robot applications. Now, he is focusing on functional-safety requirements for mechatronic systems in the automotive sector, working in the R&D department of an Italian company dedicated to braking technology.

Niccolò Iannacci received the M.S. degree in mechanical engineering from Politecnico di Milano, Milan, Italy, in 2014. His research interests include optimal control theory and artificial intelligence applied to humanrobot cooperation.

Matteo Rossi is an associate professor at Politecnico di Milano. His research interests are in formal methods for safety-critical and real-time systems, architectures for real-time distributed systems, and transportation systems both from the point of view of their design, and of their application in urban mobility scenarios.

Federico Vicentini received the M.Sc. degree in mechanical engineering and the Ph.D. degree in mechanical system engineering from the Politecnico di Milano, Milano, Italy, in 2003 and 2007, respectively. He was a Researcher with the National Research Council (CNR), Italy until 2019. His research interests include industrial robot safety, human-robot interaction, and validation procedures. Mr. Vicentini serves as a member of national and international standardization committees for robot and machine safety. He is now with Boston Dynamics, Inc. USA.

Affiliations

Mehrnoosh Askarpour[1] · **Livia Lestingi**[2] · **Samuele Longoni**[2] · **Niccolò Iannacci**[3] · **Matteo Rossi**[3] · **Federico Vicentini**[4]

Mehrnoosh Askarpour
askarpom@mcmaster.ca

Samuele Longoni
samuele.longoni@mail.polimi.it

Niccolò Iannacci
niccolo.iannacci@polimi.it

Matteo Rossi
Matteo.Rossi@polimi.it

Federico Vicentini
vicentinifederico@gmail.com

1 Computing and Software Department, McMaster University, Hamilton, Canada

2 DEIB, Politecnico di Milano, Milan, Italy

3 Dipartimento di Meccanica, Politecnico di Milano, Milan, Italy

4 National Research Council (CNR), Rome, Italy