CHALMERS
UNIVERSITY OF TECHNOLOGY

Control components for Collaborative and Intelligent Automation Systems

Control components for Collaborative and Intelligent Automation Systems

Martin Dahl
Department of Electrical Engineering
Chalmers University of Technology
Gothenburg, Sweden
martin.dahl@chalmers.se

Endre Erős
Department of Electrical Engineering
Chalmers University of Technology
Gothenburg, Sweden
endree@chalmers.se

Atieh Hanna
Research & Technology Development
Volvo Group Trucks Operation
Gothenburg, Sweden
atieh.hanna@volvo.com

Kristofer Bengtsson
Department of Electrical Engineering
Chalmers University of Technology
Gothenburg, Sweden
kristofer.bengtsson@chalmers.se

Martin Fabian
Department of Electrical Engineering
Chalmers University of Technology
Gothenburg, Sweden
fabian@chalmers.se

Petter Falkman
Department of Electrical Engineering
Chalmers University of Technology
Gothenburg, Sweden
petter.falkman@chalmers.se

***Abstract*—Collaborative and intelligent automation systems need intelligent control systems. Some of this intelligence exist on a per-component basis in the form of vision, sensing, motion, and path planning algorithms. To fully take advantage of this intelligence, also the coordination of subsystems need to exhibit intelligence. While there exist middleware solutions that eases communication, development, and reuse of such subsystems, for example the Robot Operating System (ROS), good coordination also requires knowledge about how control is supposed to be performed, as well as expected behavior of the subsystems. This paper introduces lightweight components that wraps ROS2 nodes into composable control components from which an intelligent control system can be built. The ideas are implemented on a use case involving collaborative robots with on-line path planning, intelligent tools, and human operators.**

***Index Terms*—Control Architectures and Programming; Factory Automation; Planning, Scheduling and Coordination**

I. Introduction

Production systems are quickly getting increasingly complex, with manual off-line programming of robot tasks rapidly being replaced by online algorithms that dynamically performs tasks based on the state of the environment [1], [2]. This complexity will be pushed even further when collaborative robots [3] together with other intelligent and autonomous machines and human operators, replaces more traditional automation solutions. To fully benefit from these collaborative and intelligent automation systems, also the control system needs to be more intelligent – it needs to *react to* and *anticipate* what the environment and each subsystem will do. Combined with the traditional challenges of automation software development, such as safety, reliability, and efficiency, this increased intelligence adds additional complexity that needs to be handled by the control system.

Intelligent and collaborative automation systems often comprise of several robots, machines, smart tools, human-machine interfaces, cameras, safety sensors, etc. Distributed large-scale automation systems require a communication architecture that enable reliable messaging, well-defined communication, good monitoring, and robust task planning and discrete control. In order to ease integration and development of different types of online algorithms for sensing, planning, and control of the hardware, various platforms have emerged as middle-ware solutions, one of which stands out is the Robot Operating System (ROS) [4].

Fig. 1. Collaborative robot assembly station controlled by a network of ROS2 nodes.

At the same time, traditional automation software has started a move away from field bus protocols towards Ethernet based communication platforms [5]. One such platform is the Data Distribution Service (DDS) [6], which, based on real world usage and performance [7], [8], enables implementation of large scale of industrial automation use cases.

The next version of ROS, ROS2 [9] is currently being developed on top of DDS. With ROS2 on DDS, the benefits of ROS to development and integration can be taken advantage

*This work has been supported by UNIFICATION, Vinnova, Produktion 2030 and UNICORN, Vinnova, Effektiva och uppkopplade transportsystem.

of for implementation of large scale industrial automation use cases [10].

This paper deals with a use case from a truck engine manufacturing facility. The challenge involves a collaborative robot and a human operator performing assembly operations on a diesel engine in a collaborative or coactive fashion. The assembly system can be seen in Fig. 1. In order to achieve this, a wide variety of hardware as well an extensive library of software including intelligent algorithms has to be used. The physical setup consists of a six degree of freedom collaborative robot (a UR10), an autonomous mobile platform, two different specialized end-effectors with connected docking stations, a smart nutrunner tool, a lifting system and docking station for the nutrunner, a camera and RFID reader system, as well as eight computers dedicated to different tasks. The system is run solely on ROS2, with a number of nodes having their own dedicated ROS1 master behind a bridge.

Intelligent and collaborative automation systems combine the challenges of high level intelligent task and motion planning for the robots with I/O based control on the level of sensors and actuators seen in more traditional automation systems. The automation system needs to keep track of the state of all resources and products, as well as the environment. The control system also needs means of *restarting* production should something go wrong. In [10], a control architecture with simultaneous support for both the low-level (sensors and actuators) and the high level (task and motion planning) control and decision taking was introduced. This paper aims to extend that work by providing the concept of lightweight *control components* that can be used to quickly compose a control system.

In Section II, the control architecture in which these control components operate is described. To give further context, the implementation of the architecture is discussed in III. Section IV shows how the described control components have been used for control of the automation system for the use case described above. Finally, Section V contains some concluding remarks.

II. Discrete control architecture

ROS2 systems are composed of a set of nodes communicating by sending typed messages over named topics using a publish/subscribe mechanism. This enables a quick and semi-standardized way to introduce new drivers and algorithms to a system. When composing a system of heterogeneous ROS2 nodes, care needs to be taken to understand the behavior of each node. While the interfaces are clearly separated into message types on named topics, knowledge about the workings of each node is not readily available. This is especially true when nodes have internal state that is not visible to the outside world. In order to be able to coordinate different ROS2 nodes, the control system needs to know both *how* to control the node, as well as how the nodes *behave*. In this work the *how* is referred to as *control actions* and the *behavior* is referred to as *effects*. The aim is to create reusable companion components that describe the interface and behavior of the ROS2 nodes to allow quick composition for coordinated control. But first let's take a look at the context of the overall control architecture. An overview of the architecture, first introduced in [10], can be seen in Fig. 2.

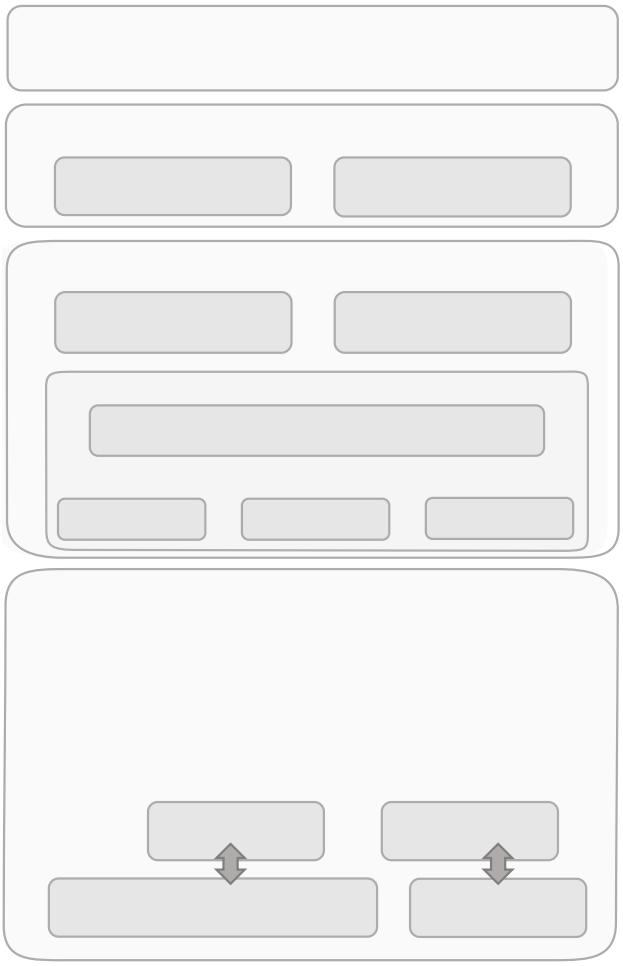

Fig. 2. The layers of the hierarchical control architecture used in this work.

In this figure the automation system is divided into four layers. Layer 0, *ROS2 nodes and pipelines*, concerns individual device drivers and ROS2 nodes in the system. *Transformation pipelines* are defined in this layer to map ROS2 messages coming from and going out to the nodes in the system to variables within the control system that is based on state rather than messages. State is divided into *measured state* - coming from the ROS2 network, *estimated state* - inferred from previous actions of the control system, and *command state* - to be sent out on the ROS2 network. Layer 1, *System state and behavior*, concerns modeling the different resources and their *ability operations* that define the low-level tasks the resources can perform. Depending on the system state, ability operations can perform *actions* which trigger changes to the *command state* variables that is eventually transformed by the pipelines in layer 0 to ROS2 messages. Resources and their abilities are modeled in two steps: first individually, then the system specific interactions are modeled as global specifications.

Specifications in layer 1 are generally safety oriented: ensuring that nothing "bad" can happen. Layer 2, *Control intent*, define the desired behavior of the automation system by defining *planning operations*. Planning operations are generally defined on a higher abstraction level (e.g. "assemble part A and part B") to define *intent* rather than "how". Planning operations are dynamically matched to sequences of suitable abilities (the how) during run-time using on-line planning. Finally, layer 3, *High level planning and optimization* represents a high level planning, optimization, or scheduling system that decides in which order to execute the planning operations of layer 2.

The remainder of this section will describe the control architecture in more detail, starting with definitions of resources and operations from which a state transition system can be constructed both for planning and formal verification.

A. Resources

Devices in the system are modeled as *resources*, which groups the device's local state and discrete descriptions of the tasks the device can perform. The system state is encoded into variables of three kinds: *measured state*, *command state*, and *estimated state*. Messages on the ROS2 network are mapped into corresponding variables using transformation pipelines, see [10].

Definition 1. A resource i is defined as $r_i = \langle V_i^M, V_i^C, V_i^E, O_i, C_i \rangle, r_i \in R$ where V_i^M is a set of *measured* state variables, V_i^C is a set of *command* state variables, V_i^E is a set of *estimated* state variables, and O_i is a set of *generalized operations* defining the resource's abilities. Variables are of finite domain. The set $V_i = V_i^M \cup V_i^C \cup V_i^E$ define all state variables of a resource. The *control state* of the resource i is a set of tuples C_i, containing a valuation of each variable within that variable's domain. R is the set of all resources in the system.

One of the tasks for the system depicted in Fig. 1 is to bolt down a cover plate onto the engine using a smart nutrunner tool. The smart nutrunner tool publishes its current state on a "state" topic and receives instructions on a "command" topic. Based on the messages received it will start or stop driving the tool. The strictly typed messages of ROS2 make it possible to automatically generate a mapping that transforms this into control state of the correct type. To ease notation in the coming sections, a shorter variable name is introduced in the comments of Listing 1, where measured state variables are denoted with a subscript "?" and command state variables are denoted with a subscript "!".

The resource nr defining the smart nutrunner can then be defined as $r_{nr} = \langle \{ti_?, tr_?, ttr_?\}, \{ti_!, tr_!\}, \emptyset, O_{nr} \rangle$. Notice that $V_{nr}^e = \emptyset$. This is the ideal case, because it means all local state of this resource can be measured.

B. Generalized operations

Control of an automation system can be abstracted into performing *operations*. By constructing a transition system modeling how operations modify the state of the resources in a system, formal techniques for verification and planning can be applied. To do this in a manner suitable to express both low-level ability operations and later on planning operations, we define the *generalized operation*.

```
# ROS2 topic: /smart_nutrunner/state
bool tool_is_idle                 # => ti?
bool tool_is_running_forward      # => tr?
bool programmed_torque_reached    # => ttr?

# ROS2 topic: /smart_nutrunner/command
bool set_tool_idle                # => ti!
bool run_tool_forward             # => tr!
```

Listing 1: ROS2 message definitions for messages on the topics from and to the smart tool.

Definition 2. A generalized operation j operating on the state of a resource i is defined as $o_j = \langle P_j, G_j, T_j^d, T_j^a, T_j^E \rangle, o_j \in O_i$. P_j is a set of named predicates over the state of the resource variables V_i. G_j is a set of un-named guard predicates over the state of V_i. The sets T^d and T^a define *control* transitions that update V_i, where T^d define transitions that require (external from the ability) deliberation and T^a define transitions that are automatically applied whenever possible. T_j^E is a set of *effect* transitions describing the possible consequences to V_i^M of being in certain states. A transition $t_i \in \{T^d \cup T^a \cup T^E\}$ has a guard predicate which can contain elements from P_j and G_j and a set of actions that update the current state if and only if the corresponding guard predicate evaluates to true.

T_j^d, T_j^a, and T_j^E have the same formal semantics, but are separated due to their different uses. The effect transitions T_j^E define how the *measured state* is updated, and as such they are not used during actual low-level control like the control transitions T_j^d and T_j^a . They are important to keep track of however, as they are needed for on-line planning and formal verification algorithms, as well as for simulation based validation.

It is natural to define when to take certain actions in terms of what state the resource is currently in. To ease both modeling, planning algorithms and later on online monitoring, the guard predicates of the generalized operations are separated into one set of named (P_j) and one set of un-named (G_j) predicates. The named predicates can be used to define in what state the operation currently is in, in terms of the set of local resource states defined by this predicate. The un-named predicates are used later in Section II-E where the name of the state does not matter.

C. Ability operations

The behavior of the resources in the system is modeled by *ability operations* (abilities for short). While it is possible to define transitions using only the un-named guard predicates in G (from Definition 2), it is useful to define a number of "standard" predicate names for an ability to ease modeling, reuse, and support for online monitoring. In this work common meaning is introduced for the following predicates:

enabled (ability can start), *starting* (ability is starting, e.g. a handshaking state), *executing* (ability is executing, e.g. waiting to be finished), *finished* (ability has finished), and *resetting* (transitioning from finished back to enabled). In the general case, the transition between *enabled* and *starting* has an action in T^d, while the transition from *finished* has an action in T^a. In other types of systems other "standard" names could be used (e.g. *request* and *receive*).

Let's exemplify again with the smart nutrunner. Table I shows the transitions of a typical ability, where each line makes up one transition of the ability. The ability models the task of running a nut, by starting to run the motors forward until a pre-programmed torque ($ttr_?$) has been reached. Notice that for this ability, $G = \emptyset$.

TABLE I

pred. name	predicate	control actions	effects
enabled	$ti_? \wedge \neg tr_? \wedge ti_! \wedge \neg tr_!$	$ti_! = F$, $tr_! = T^\bigstar$	-
starting	$\neg ti_! \wedge tr_! \wedge \neg tr_?$	-	$ti_? = F$, $tr_? = T$
executing	$\neg ti_! \wedge tr_! \wedge tr_?$	-	$ttr_? = T$
finished	$\neg ti_! \wedge tr_! \wedge ttr_?$	$ti_! = T$, $tr_! = F$	-
resetting	$ti_! \wedge \neg tr_! \wedge tr_?$	-	$ti_? = T$, $tr_? = F$

Table I: The named predicates and action functions making up the transitions of the "Run nut" ability.

The resource and ability defined, combined with its ROS2 node and the pipelines, makes for a well isolated and reusable ***control component***, spanning layer 0 and the resource half of layer 1 in Fig. 2. However, at this point it cannot do anything useful other than being used for manual or open loop control. One also need to be able to model *interaction* between resources, for example the robot and the nutrunner, or the human operator and the nutrunner.

D. Modeling resource interaction

Layer 1 of Fig. 2 illustrates two types of interaction between resources: safety specification and adding additional effects which springs from the interaction between resources. As it might not be possible to outright *measure* these effects, many of them will be modeled as control actions updating estimated state variables.

The generalized operations defined in Definition 2 can be instantiated into a *global* resource r_G ($r_G \notin R$) with its state defined as the current valuation of $V_G = \bigcup V_i, \forall i$ s.t. $r_i \in R$. An instantiation with an additional guard $ur.pos = bp_1$ conjuncted with the ability's *finished* predicate, and an additional action on the transition possible when the *finished* predicate is satisfied $\hat{b_1}$ = tightened, where $ur.pos$ (a variable in V_G) references a named pose (bp_1, in the domain of $ur.pos$) of the robot and $\hat{b_1} \in (empty, placed, tightened)$ is an *estimated* state variable in V_G introduced to keep track of the bolt pairs. The changes compared to Table I are highlighted in Table II. Instantiations such as this can be generated for all bolt pairs in the system.

Formal specifications are used to ensure that the resources can never do something "bad". In order to be able to work with individual devices, as well as isolating the complexities

TABLE II

pred. name	predicate	control actions	effects
finished	$\neg ti_! \wedge tr_! \wedge ttr_? \wedge ur.pos = bp_1$	$ti_! = T$, $tr_! = F, \hat{b_1} = t$	-

Table II: The instantiated ability "Run nut bolt pair one".

that arise from their different interactions, modeling should rely on using global specifications. The abilities defined so far, together with a set of global specifications can be used to formulate a supervisor synthesis problem from the variables and transitions in r_G. Using the method described in [11], the solution to this synthesis problem can be obtained as additional guard predicates on the deliberate transitions in T^d. Examples of this modeling technique can be found in [12], [13].

E. Planning operations

While ability operations define the low-level tasks different devices can perform, planning operations model how to make the system do something *useful* (in this case, produce engines). The planning operation k is defined as the generalized operation O_k defined in r_G and the estimated state variable $O_k^E \in V_G$ with the domain $\{i, e, f\}$. For the operation k, $P_k = \{<\text{init}, O_k^E = i>, <\text{executing}, O_k^E = e, <\text{finished}, O_k^E = f>\}$, $T_k^d = \text{init} \wedge g_j / O_k^E := e$, $T_k^a = \text{executing} \wedge g_k / O_k^E := f$, $T_k^E = \emptyset$ (where / is used to separate the guard predicate and action function making up the transition). $g_j \in G_k$ is a predicate defining the *precondition* of O_k and $g_k \in G_k$ is a predicate defining the *postcondition* of O_k.

Consider the tightening of a bolt in our example. A natural way to model this as an operation (let's call it `TightenBolt`), is to start in the pre-state defined by the precondition $p \in G_k$ defined by $\hat{b}$ = placed and end in the post-state defined by the postcondition $p_2 \in G_k$ defined by $\hat{b}$ = tightened.

Modeling operations in this way does two things. First, it makes it possible to add and remove resources from the system more easily - as long as the desired goals can be reached, the upper layers of the control system does not need to be changed. Second, it makes it easier to model on a high level, eliminating the need to care about specific sequences of abilities. As will be shown in Section IV, the operation `TightenBolt` involves sequencing several abilities controlling the robot and the tool to achieve the goal state of the operation.

III. Control implementation

The components defined thus far are used for running the system by executing the operations to reach some kind of goal. At the same time the control must also react to external inputs like state changes or events from machines, operators, sensors or cameras. The control therefore continuously deliberate the best execution sequence of abilities to reach the current goal defined by the planning operations.

A. The runner

The control system keeps track of the current state of all variables, the planning operations, the ability operations, and

two deliberation plans. When the state is updated by any of the pipelines that continuously transforms ROS2 messages into state changes, the runner is triggered. The runner consists of two stages, the first evaluates and triggers planning operation transitions and the second evaluates and triggers ability operation transitions. When a transition is evaluated to true in the current state and is enabled, the transition is triggered and the state is updated. After the two steps have been executed, the outgoing pipelines are triggered and transforms the updated state into ROS2 messages.

Each stage includes both deliberation transitions and automatic transitions. The automatic transitions will always trigger when their guard predicate is true in the current state, but the deliberation transitions must also be enabled by the deliberation plan. The deliberation plan is a sequence of transitions defining the execution order of the deliberation transitions. The plan for the planning operation stage is defined either by a manual sequence or by a planner or optimizer in level 3. For the ability stage, the deliberation plan is continuously refined by an automated planner that finds the best sequence of abilities to reach a goal defined by the planning operations.

B. Automated planning

The approach to planning taken in this work in based on finding counter examples using bounded model checking (BMC) [14]. Modern SAT-based model checkers are very efficient in finding counter examples, even for systems with hundreds of variables. BMC also has the useful property that counter examples have minimal length due to how the problem is unfolded into SAT problems iteratively. As such, a counter-example (or deliberation plan) is always minimal in the number of transitions taken. Additionally, well-known and powerful specification languages like *Linear Temporal Logic* (LTL) [15] can be used. For the implementation done in Section IV, the SAT based bounded model checking capabilities of the nuXmv symbolic model checker [16] was used.

C. From planning operations to a goal

The planning operations define the current high-level purpose of the system as well as the general execution sequence. These operations normally define the nominal behavior of the system, for example defining the sequence of assembling the parts on the engine in the use case, but can also be manual operations defining other types of goals.

The postcondition g_{ik} of each planning operation with their *exeucting* predicate O_i^E satisfied are conjuncted to form a LTL specification $s = \bigwedge_{O_i \in O} (O_i^E \implies \Diamond g_{ik})$, where $\Diamond$ is the LTL operator specifying that the predicate eventually becomes true. The automated planner then looks for counter-examples that disprove the negation of s. Such a counter-example (if found) will contain the fewest possible number of transitions that take the system to a state where all g_{ik} have been reached.

During error handling it is possible to create temporary planning operations, for instance creating operations which has other operations precondition as their postcondition, effectively allowing the control system attempt to go back to a previous state to perform some planning operation again. An example of this is given in Section IV-D.

IV. Use case - control of a collaborative intelligent automation system

In this section we describe how modeling and control of a collaborative intelligent automation system is performed by composing the control components described. For clarity, this section focuses on a small part of the system: moving an engine cover plate from a kitting wagon onto the engine and bolting it down. This involves several components; the UR10 robot, the connector, an end-effector for lifting the plate, the smart nutrunner already described, as well as the human operator. The aim is to show how the different components are modeled and used for control. Note that the components described are simplified to illustrate the concepts while fitting the paper format.

1) Robot end-effector connector component: A node similar to the one for the smart tool exist for the connector attached to the robot end-effector, the message types of which is described below.

```
# ROS2 topic: /connector/state
bool connected                  # => c?
bool not_connected              # => nc?
bool connection_failure         # => cf?

# ROS2 topic:  /connector/command
bool lock_rsp                   # => l!
bool unlock_rsp                 # => u!
```

This component define two abilities, `lock` and `unlock`. The `lock` ability is defined by Table III.

TABLE III

pred. name	predicate	control actions	effects
enabled	$nc_? \wedge \neg c_? \wedge u_! \wedge \neg l_!$	$u_! = F, l_! = T^\bigstar$	-
executing	$\neg u_! \wedge l_! \wedge \neg c_?$	-	$c_? = T$
finished	$\neg u_! \wedge l_! \wedge c_?$	-	-

Table III: The named predicates and action functions making up the transitions of the "lock" ability.

2) UR10 component: The UR10 component defines a resource which has state relating ROS2 nodes for both path planning using *MoveIt!* [17] and a lower level UR driver for script execution. Part of its state is defined below. Discretization of the robot poses is done in the pipelines [10]; the domain of $p_?$ and $r_!$ is an enumeration type based on this discretization.

```
# ROS2 topic: /ur10/state
bool moving                     # => m?
string act_pos                  # => p?

# ROS2 topic:  /ur10/command
string ref_pos                  # => r!
```

An ability is defined for the robot: `goto` which takes a (discrete, i.e. named) goal position ($r_!$) as an input.

Notice that the ability defined in Table IV does not define any control action. This is added when the ability is instantiated, effectively creating one ability per target pose (e.g. `gotoHome`).

TABLE IV

pred. name	predicate	control actions	effects
enabled	$\neg m_? \wedge p_? \neq u \wedge r_! = p_?$	-	-
starting	$\neg m_? \wedge r_! \neq p_?$	-	$m_? = T$, $p_? = u$
executing	$m_? \wedge r_! \neq p_?$	-	$m_? = F$, $p_? = r_!$
finished	$\neg m_? \wedge p_? = r_!$	-	-

Table IV: The named predicates and action functions making up the transitions of the `goto` ability.

3) Human operator: The human operator is also a control component. Like the previous components, this component wraps a ROS2 node, pipelines for transformations to and from its resource state and a number of abilities. The difference is that the ROS2 node only instructs the human what to do next via an interface on a smart watch, as well as registers when the task is done. The detailed resource state and abilities are omitted for brevity.

A. Resource interaction

An estimated state variable $\hat{rt} \in \{none, nr, lf\}$ is introduced defining what tool the robot is holding. Similarly to the instantiation of the run nut ability detailed in Table II, two instantiations of the connector ability `lock` are created with an additional action that updates this state depending on the robot position (the pose names of the tool docking positions). The UR10 `goto` ability is also instantiated with additional guards referencing variables about which tool (or no tool) needs to be connected at different poses. See Table V for an example where the robot is required to be connected to a certain tool in order to be allowed to move to the pose bp_1. Recall that $\hat{b_1} \in (empty, placed, tightened)$ was introduced

TABLE V

pred. name	predicate	control actions	effects
enabled	$\ldots \wedge \hat{rt} = nr$	$r_! = bp_1$	-

Table V: The instantiated ability "Goto bolt pair one".

in Section II-D. The human instruction ability is instantiated to create a number of `placeBolt` abilities, which updates $\hat{b_n}, n \in \{1, .., 6\}$ from empty to placed.

B. Planning operations

Table VI shows one possible way to define the planning operations required in order to perform the tasks described in Section IV. In essence these describe in a declarative way what the desired goal states are. Nothing has to be defined about which resource(s) are eventually utilized to reach these states, allowing the system great flexibility to operate in a reactive manner. Note that the "size" of the chosen operations need to be considered, making sure the planning system has goals that can be expected to be solved for in a reasonably short time period. See Section IV-C for a notion of how long this time period can be. Consider the `TightenBolt` operations in Table VI; having $\hat{rt} = nr$ in the precondition is not strictly necessary – if omitted the planner would still make sure to get the correct tool before moving into position. However, the

TABLE VI

operation	precondition	postcondition
`GetLFTool`	$\hat{rt} \neq lf$	$\hat{rt} = lf$
`GetNRTool`	$\hat{rt} \neq nr$	$\hat{rt} = nr$
`LFToEngine`	$\hat{rt} = lf \wedge \hat{lf} = onMir$	$\hat{lf} = onEngine$
`PlaceBolt_i`	$\hat{lf} = onEngine \wedge bp_i =$ empty	$bp_i =$ placed
`TightenBolt_i`	$\hat{lf} = onEngine \wedge bp_i =$ placed $\wedge \hat{rt} = nr$	$bp_i =$ tightened

Table VI: The planning operations in the described use case. The operations suffixed with `_i` exist in multiple versions with $i \in 1..6$.

resulting plan would potentially be much longer (if $\hat{rt} = lf$, first the other tool would have to be put back), which affects planning time negatively.

Fig. 3 shows an illustration of a possible execution of this example. To the left is shown three *planning operations*, with their respective preconditions above their name, and their respective postcondition below it. To the right is shown one possible set of sequences of *ability operations* generated by the on-line planning system. Note that these sequences will naturally change depending on the state of the system. It is also possible for multiple operations to execute simultaneously, which in the example mean that both `GetNRTool` and `PlaceBolt1` can run at the same time.

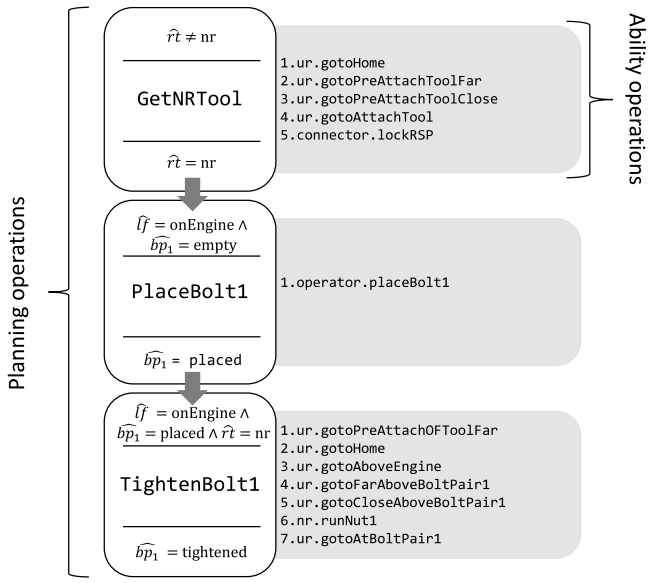

Fig. 3. Three *planning operations* each matched to a sequence of *ability operations* by the on-line planning system.

C. Planning performance

The system described in this work has $242,295$ reachable states. To give an overview of the planning performance, Table VII show the planning times for n transitions into the future. Measurements were made on a consumer grade laptop computer.

The table suggest that planning operations can comfortably search around 20 steps into the future while keeping on-line performance. This roughly correspond to executing ten ability operations in sequence.

TABLE VII

n	time (ms)
10	776
20	1652
30	3836
40	6259
50	34576

Table VII: Time consumed by the planning system for plans which need n transitions to reach the goal state.

D. Handling a restart situation

Consider the case where the nutrunning for the first bolt fails (i.e. the `TightenBolt1` planning operation). In this section we consider how the control system can handle two different scenarios in which this can occur.

1) Robot error: The robot could stop moving during execution of its `goto` ability operation for example if the operator pushes the robot out of his or her way. The discretization of the robot's positions makes the system unaware of exactly where the robot is located. By temporarily adding a *restart* ability operation to the runner that allow the robot to go back to its previous position (using motion planning) the state of the control system and the robot resource can be synchronized, after which the nominal behavior can continue.

2) Operator error: Another reason for this to occur is that the operator had not in fact put the bolts in correctly. Hence the *estimated* state of our system is out of sync with reality. The operator can now manually synchronize the estimated state in a frond-end, setting $\hat{b_1} =$ empty. If a temporary planning operation is now added, which has the precondition of the failed `TightenBolt1` planning operation as its postcondition, this will result in a plan that includes telling the operator to place the bolts again.

V. Conclusion

This paper has described a lightweight type of reusable control component for modeling and control of ROS2 based systems. The components can be composed into a global system that allows using formal methods to ensure various properties as well as enabling on-line planning during execution.

The described components have been applied in the modeling and control of an industrial assembly station. Based on our experience from this implementation, the high-level modeling of the planning operations was found to be easy to understand and work with. Turning the ROS2 nodes into control components required some work, but while this could be tedious, it was not very difficult. This work is also intended to be reusable. The more challenging engineering effort has instead been moved to correctly specifying the interactions which arises between the components. Future work should involve generating both the effects and safety-oriented specifications required in a generalized way. Due to the difficulties in anticipating what the planning system will do, virtual validation is key during development of these interactions. Work is ongoing to include virtual simulation nodes in the control components [18], to make it simple to switch between the logical *effects* described in this paper and simulation which may also include continuous dynamics.

References

[1] R. Alterovitz, S. Koenig, and M. Likhachev, "Robot planning in the real world: Research challenges and opportunities," *AI Magazine*, vol. 37, no. 2, pp. 76–84, Summer 2016.

[2] L. Perez, E. Rodriguez, N. Rodriguez, R. Usamentiaga, and D. F. Garcia, "Robot guidance using machine vision techniques in industrial environments: A comparative review," *Sensors*, vol. 16, no. 3, 2016. [Online]. Available: http://www.mdpi.com/1424-8220/16/3/335

[3] A. Bauer, D. Wollherr, and M. Buss, "Human-robot collaboration: A survey," *International Journal of Humanoid Robotics*, vol. 05, no. 01, pp. 47–66, 2008. [Online]. Available: https://doi.org/10.1142/S0219843608001303

[4] "ROS," http://www.ros.org, 2019, [Online; accessed 25-Feb-2019].

[5] J.-D. Decotignie, "Ethernet-based real-time and industrial communications," *Proceedings of the IEEE*, vol. 93, no. 6, pp. 1102–1117, 2005.

[6] G. Pardo-Castellote, "Omg data-distribution service: architectural overview," in *23rd International Conference on Distributed Computing Systems Workshops, 2003. Proceedings.*, May 2003, pp. 200–206.

[7] P. Bellavista, A. Corradi, L. Foschini, and A. Pernafini, "Data distribution service (dds): A performance comparison of opensplice and rti implementations," in *2013 IEEE Symposium on Computers and Communications (ISCC)*, July 2013, pp. 000 377–000 383.

[8] S. Profanter, A. Tekat, K. Dorofeev, M. Rickert, and A. Knoll, "Opc ua versus ros, dds, and mqtt: Performance evaluation of industry 4.0 protocols," in *IEEE International Conference on Industrial Technology (ICIT), Melbourne, Australia*, 02 2019.

[9] "ROS 2," https://index.ros.org/doc/ros2/, 2019, [Online; accessed 25-Feb-2019].

[10] M. Dahl, E. Erös, A. Hanna, K. Bengtsson, and P. Falkman, "Sequence planner - automated planning and control for ros2-based collaborative and intelligent automation systems," https://arxiv.org/abs/1903.05850, 2019.

[11] S. Miremadi, B. Lennartson, and K. Åkesson, "A BDD-based approach for modeling plant and supervisor by extended finite automata," *Control Syst. Technol. IEEE Trans.*, vol. 20, no. 6, pp. 1421–1435, 2012.

[12] P. Bergagård, P. Falkman, and M. Fabian, "Modeling and automatic calculation of restart states for an industrial windscreen mounting station," *IFAC-PapersOnLine*, vol. 48, no. 3, pp. 1030–1036, 2015.

[13] M. Dahl, K. Bengtsson, M. Fabian, and P. Falkman, "Automatic modeling and simulation of robot program behavior in integrated virtual preparation and commissioning," *Procedia Manufacturing*, vol. 11, pp. 284–291, 2017.

[14] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without bdds," in *International conference on tools and algorithms for the construction and analysis of systems*. Springer, 1999, pp. 193–207.

[15] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. IEEE, 1977, pp. 46–57.

[16] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, "The nuxmv symbolic model checker," in *CAV*, 2014, pp. 334–342.

[17] I. A. Sucan and S. Chitta, "MoveIt!" http://moveit.ros.org, 2018, [Online; accessed 26-Feb-2019].

[18] E. Endre, M. Dahl, A. Albo, H. Atieh, P. Falkman, and K. Bengtsson, "Integrated virtual commissioning of a ros2-based collaborative and intelligent automation system," in *Submitted to the 24th IEEE Conference on Emerging Technologies and Factory Automation (ETFA2019)*, May 2019.