Development of an Industry 4.0 Demonstrator Using Sequence Planner and ROS2

Development of an Industry 4.0 Demonstrator Using Sequence Planner and ROS2

Endre Erős, Martin Dahl, Atieh Hanna, Per-Lage Götvall, Petter Falkman, and Kristofer Bengtsson

Abstract In many modern automation solutions, manual off-line programming is being replaced by online algorithms that dynamically perform tasks based on the state of the environment. Complexities of such systems are pushed even further with collaboration among robots and humans, where intelligent machines and learning algorithms are replacing more traditional automation solutions. This chapter describes the development of an industrial demonstrator using a control infrastructure called Sequence Planner (SP), and presents some lessons learned during development. SP is based on ROS2 and it is designed to aid in handling the increased complexity of these new systems using formal models and online planning algorithms to coordinate the actions of robots and other devices. During development, SP can auto generate ROS nodes and message types as well as support continuous validation and testing. SP is also designed with the aim to handle traditional challenges of automation software development such as safety, reliability and efficiency. In this chapter, it is argued that ROS2 together with SP could be an enabler of intelligent automation for the next industrial revolution.

E. Erős · M. Dahl · P. Falkman · K. Bengtsson (✉)
Chalmers University of Technology, Gothenburg, Sweden
e-mail: kristofer.bengtsson@chalmers.se

E. Erős
e-mail: endree@chalmers.se

M. Dahl
e-mail: martin.dahl@chalmers.se

P. Falkman
e-mail: petter.falkman@chalmers.se

A. Hanna · P.-L. Götvall
Research & Technology Development, Volvo Group Trucks Operations, Gothenburg, Sweden
e-mail: atieh.hanna@volvo.com

P.-L. Götvall
e-mail: Per-Lage.Gotvall@volvo.com

A. Koubaa (ed.), *Robot Operating System (ROS)*, Studies in Computational Intelligence 895, https://doi.org/10.1007/978-3-030-45956-7_1

1 Introduction

As anyone with experience with real automation development knows, developing and implementing a flexible and robust automation system is not a trivial task. There are currently many automation trends in industry and academia, like Industry 4.0, cyber-physical production systems, internet of things, multi-agent systems and artificial intelligence. These trends try to describe how to implement and reason about flexible and robust automation, but are often quite vague on the specifics. When it comes down to the details, there is no silver bullet [1].

Volvo Group Trucks Operations has defined the following vision to better guide the research and development of next generation automation systems: Future Volvo factories [2], will be *re-configurable* and *self-balancing* to better handle rapid changes, production disturbances, and diversity of the product. Collaborative robots and other machines can support operators at workstations, where they use the same smart tools and are interchangeable wit operators when it comes to performing routine tasks. Operators are supported with mixed reality technology, and are provided with digital work instructions and 3D geometry while interacting intuitively with robots and systems. Workstations are equipped with smart sensors and cameras that feed the system with real-time status of products, humans, and other resources in the environment. Moreover, they are supported by advanced yet intuitive control, dynamic safety, online optimization and learning algorithms.

Many research initiatives in academia and industry have tried to introduce collaborative robots ("cobots") in the final assembly [3–5]. Despite their relative advantages, namely that they are sometimes cheaper and easier to program and teach [5] compared to conventional industrial robots, they are mostly deployed as robots "without fences" for co-active tasks [6]. Current cobot installations are in most cases not as flexible, robust or scalable as required by many tasks in manual assembly. Combined with the lack of industrial preparation processes for these types of systems, new methods and technologies must be developed to better support the imminent industrial challenges [7].

1.1 The Robot Operating System and Sequence Planner

This chapter discusses some of the challenges of developing and implementing an industrial demonstrator that includes robots, machines, smart tools, human-machine interfaces, online path and task planners, vision systems, safety sensors, etc. Coordination and integration of the aforementioned functionality requires a well-defined communication interface with good monitoring properties.

During the past decade, various platforms have emerged as middle-ware solutions trying to provide a common ground for integration and communication. One of them is the Robot Operating System (ROS), which stands out with a large and enthusiastic community. ROS enables all users to leverage an ever-increasing num-

ber of algorithms and simulation packages, as well as providing a common ground for testing and virtual commissioning. ROS2 takes this concept one step further, integrating the scalable, robust and well-proven Data Distribution Service (DDS) as its communications layer, eliminating many of the shortcomings of ROS1.

ROS2 systems are composed of a set of nodes that communicate 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 into a system. However, having reliable communication, monitoring tools, as well as a plethora of drivers and algorithms ready to be deployed is not enough to be able to perform general automation. 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 contain an internal state that is not visible to the outside world. In order to be able to coordinate different ROS2 nodes, a control system needs to know both *how to control* the nodes making up the system, as well as how these nodes *behave*.

To control the behavior of nodes, an open-source control infrastructure called Sequence Planner (SP) has been developed in the last years. SP is used for controlling and monitoring complex and intelligent automation systems by keeping track of the complete system state, automatically planning and re-planning all actions as well as handling failures or re-configurations.

This chapter presents the development of a flexible automation control system using SP and ROS2 in a transformed truck engine final assembly station inspired by the Volvo vision. The chapter contributes with practical development guidelines based on the **lessons learned** during development, as well as detailed design rationales from the final demonstrator, using SP built on top of ROS2.

The next section introduces the industrial demonstrator that will be used as an example throughout the chapter. Section 3 discusses robust discrete control, why distributed control states should be avoided and the good practice of using state-based commands. The open-source control infrastructure SP is introduced in Sect. 4 and the generation of ROS2 code for logic and tests is presented in Sect. 5.

2 An Industrial Demonstrator

The demonstrator presented in this paper is the result of a transformation of an existing manual assembly station from a truck engine final assembly line, shown in Fig. 1, into an intelligent and collaborative robot assembly station, shown in Fig. 2.

In the demonstrator, diesel engines are transported from station to station in a predetermined time slot on Automated Guided Vehicles (AGVs). Material to be mounted on a specific engine is loaded by an operator from kitting facades located adjacent to the line. An autonomous mobile platform (MiR100) carries the kitted material to be mounted on the engine, to the collaborative robot assembly station.

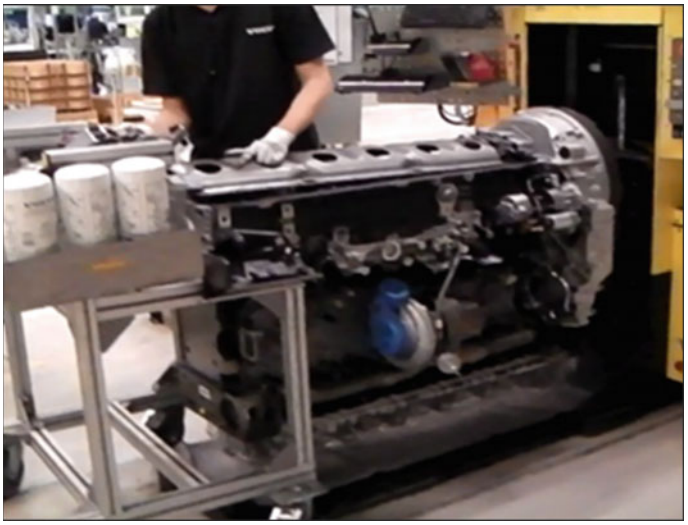


Fig. 1 The original manual assembly station

Fig. 2 Collaborative robot assembly station controlled by a network of ROS2 nodes. A video clip from the demonstrator: https://youtu.be/TK1Mb38xiQ8

In the station, a robot and an operator work together to mount parts on the engine by using different tools suspended from the ceiling. A dedicated camera system keeps track of operators, ensuring safe coexistence with machines. The camera system can also be used for gesture recognition.

Before the collaborative mode of the system starts, an authorized operator has to be verified by a RFID tag. After verification, the operator is greeted by the station

and operator instructions are shown on a screen. If no operator is verified, some operations can still be executed independently by the robot, however, violation of safety zones triggers a safeguard stop.

After the AGV and the kitting material have arrived, a Universal Robots (UR10) robot and an operator lift a heavy ladder frame on to the engine. After placing the ladder frame on the engine, the operator informs the control system with a button press on a smartwatch or with a gesture, after which the UR10 leaves the current end-effector and attaches to the nutrunner used for tightening bolts. During this tool change, the operator starts to place 24 bolts, which the UR10 will tighten with the nutrunner.

During the tightening of the bolts, the operator can mount three oil filters. If the robot finishes the tightening operation first, it leaves the nutrunner in a floating position above the engine and waits for the operator. When the operator is done, the robot attaches a third end-effector and starts performing the oil filter tightening operations. During the same time, the operator attaches two oil transport pipes on the engine, and uses the same nutrunner to tighten plates that hold the pipes to the engine. After executing these operations, the AGV with the assembled engine and the empty MiR100 leave the collaborative robot assembly station. All of this is controlled by the hierarchical control infrastructure Sequence Planner that uses ROS2.

2.1 ROS2 Structure

ROS2 has a much improved transport layer compared to ROS1, however, ROS1 is still ahead of ROS2 when it comes to the number of packages and active developers. In order to embrace the strengths of both ROS1 and ROS2, i.e. to have an extensive set of developed robotics software and a robust way to communicate, all sub-systems in the demonstrator communicate over ROS2 where a number of nodes have their own dedicated ROS1 master behind a bridge [8].

A set of related nodes located on the same computer are called a hub [8], where one or more hubs can be present on the same computer. ROS hubs are considered to have an accompanying set of bridging nodes that pass messages between ROS and ROS2.

In the demonstrator, ROS nodes are distributed on several computers shown in Table 1, including standard PCs as well as multiple Raspberry Pies and a Lattepanda Alpha (LP Alpha). Most of the nodes are only running ROS2.

All computers in the system are communicating using ROS2 and they can easily be added or removed. All computers connect to the same VPN network as well, to simplify the communication over cellular 4G.

One important **lesson learned** was that ROS1 nodes should only communicate with a ROS1 master on the same computer, else the overall system becomes hard to setup and maintain. During the demonstrator development, we tried to use ROS2 nodes whenever possible, and we only used ROS2 for communication between computers.

Table 1 Overview of the computers in the demonstrator. The ROS versions used were Kinetic for ROS1 and Dashing for ROS2. Some nodes, like number 5 and 7, need both

No.	Name	ROS v.	Computer	OS	Arch.	Explanation
1	Tool ECU	Dashing	Rasp. Pi	Ubuntu 18	ARM	Smart tool and lifting system control
2	RSP ECU	Dashing	Rasp. Pi	Ubuntu 18	ARM	Pneumatic conn. control and tool state
3	Dock ECU	Dashing	Rasp. Pi	Ubuntu 18	ARM	State of docked end-effectors
4	MiRCOM	Dashing	LP Alpha	Ubuntu 18	amd64	ROS2 to/from REST
5	MiR	Kinetic+Dashing	Intel NUC	Ubuntu 16	amd64	Out-of-the-box MiR100 ROS suite
6	RFIDCAM	Dashing	Desktop	Win 10	amd64	Published RFID and camera data
7	UR10	Kinetic+Dashing	Desktop	Ubuntu 16	amd64	UR10 ROS suite
8	DECS	Dashing	Laptop	Ubuntu 18	amd64	Sequence planner

There is a number of available implementations of DDS, and since DDS is standardized, ROS2 users can choose an implementation from a vendor that suits their needs. This is done through a ROS Middleware Interface (RMW), which also exposes Quality of Service (QoS) policies. QoS policies allow ROS2 users to adjust data transfer in order to meet desired communication requirements. Some of these QoS policies include setting message history and reliability parameters. This QoS feature is crucial, especially in distributed and heterogeneous systems, since setups are usually unique.

However, in the demonstrator described in this chapter, default settings were sufficient for all nodes since there were no real-time requirements. Even though standard QoS settings were used, how the messaging and communication was setup highly influenced the robustness of the system. In the next section, we will study this in more details. The communication between the hubs and the control system is in some cases auto-generated by Sequence Planner and will be introduced in more details in Sect. 5.

The main focus when developing this demonstrator was to perform the engine assembly using both humans and intelligent machines in a robust way. Other important aspects are non-functional requirements like safety, real-time constraints and performance. However, these challenges were not the main focus when developing the demonstrator. This will be important in future work, especially to handle the safety of operators.

3 Robust Discrete Control

The presented demonstrator consists of multiple sub-systems that need to be coordinated and controlled in a robust way. A robust discrete control system must handle unplanned situations, restart or failures of sub-systems, as well as instructing and monitoring human behavior.

During development, we also **learned** that one of the most complex challenges for a robust discrete control system, is how to handle asynchronous and non-consistent control states. In this section, we will therefore take a look at this and why it is important to avoid having a distributed control state and why state-based commands should be used instead of event-based. In many implementations, this challenge is often neglected and handled in an ad-hoc fashion. To better understand this challenge and how to handle it, let us start with some definitions.

3.1 States and State Variables

There are many different ways to define the state of a system. For example, if we would like to model the state of a door, we can say that it can be: $\{opened, closed\}$, or maybe even: $\{opening, opened, closing, closed\}$. We could also use the door's angle, or maybe just a boolean variable. Depending on what we would like to do with the model and what can we actually measure, we will end up with very different definitions. In this chapter, a model of a state of a system is defined using state variables:

Definition 1 A state variable v has a domain $V = \{x_1, \ldots, x_n\}$, where each element in V is a possible value of v. There are three types of variables: v^m: *measured* state variables, v^e: *estimated* state variables, and v^c: *command* state variables. For simplicity, we will use the following notation when defining a variable: $v = \{x_1, \ldots, x_n\}$.

State variables are used when describing the state of a system, where a state is defined as follows:

Definition 2 A state S is a set of tuples $S = \{\langle v_i, x_i\rangle, \ldots, \langle v_j, x_j\rangle\}$, where v_i is a state variable and $x_i \in V_i$ is the current value of the state variable. Each state variable can only have one current value at the time.

Let us go back to the door example to better understand states and state variables. The door can for example have the following state variables:

$$pos^e = \{opened, closed\}$$
$$locked^m = \{true, false\}$$

In this example, the position of the door is either $opened$ or $closed$ and is defined by the *estimated* state variable pos^e. It is called estimated since we can not measure the position of this example door. The pos^e variable must therefore be updated by the control system based on what actions have been executed (e.g. after we have opened the door, we can assume that it is opened).

The door also has a lock, that can either be locked or not, and is defined by the *measured* state variable $locked^m$. It is called measured since this door has a sensor that tells us if the door is locked or not.

The door can be in multiple states, for example, the state $\{\langle pos^e, closed\rangle, \langle locked^m, true\rangle\}$ defines that the door is closed and locked. To be able to change the state of the door, a control system needs to perform actions that unlocks and opens the door.

3.2 Event-Based Commands and Actions

When controlling different resources in an automation system, a common control approach is for the control system to send commands to resources, telling them what to do or react on. Also, the control system reacts on various events from the resources. When communicating in this fashion, which we can call event-based communication, all nodes will wait for and react on events. This type of communication and control often leads to a highly reactive system but is often challenging to make robust.

In ROS2, nodes communicate via topics, services or actions. It is possible to get some communication guarantees from the QoS settings in DDS, however, even if the communication is robust, we **learned** the hard way that it is not easy to create a robust discrete control system using only event-based communication. To better explain this, let us study how one of the nodes from the demonstrator could have been implemented, if we have had used event-based communication:

Example 1: The nutrunner: event-based communication

In the demonstrator, either the robot or the human can use the nutrunner. When started, it runs until a specific torque or a timeout has been reached. In the example, the nutrunner is controlled by the following commands: *start* or *reset*, and responds with the events *done* or *failed*. The communication could be implemented with ROS actions but for simplicity, let us use two publish-subscribe topics and the following ROS messages:

```
# ROS2 topic:  /nutrunner/command
string cmd     # 'start', or 'reset'

# ROS2 topic:  /nutrunner/response
string event   # 'done', or 'failed'
```

To start the nutrunner, the *start* command message is sent to the nutrunner node, which will start running. When a specific torque is reached, the node sends a *done* event back. If for some reason the torque is not reached after a timeout, or if the nutrunner fails in other ways, it will respond with a *failed* event. To be able to start the nutrunner again, the *reset* command must be sent to initialize the nutrunner.

If we can guarantee that each message is only received once and everything behaves as expected, this may work quite fine. However, if for some reason either the controller or the node fails and restarts, we will have a problem. If the controller executes a sequence of actions, it will assume that it should first send the *start* command and then wait for *done* or *failure*. However, if the nutrunner node is in the wrong state, for example if it is waiting for a *reset* command, nothing will happen. Then the control sequence is not really working, since the states of the two nodes are out of sync (i.e. the control node thinks that the nutrunner node is in a state where the control can send start).

A common problem in many industrial automation systems is that the state of the control system gets out of sync with the actual states of some resources. The reason for this is that many implementations are based on sequential control with event-based communication to execute operations, with the assumption that the sequence always starts in a well defined global state. However, this is often not the case, e.g. when a system is restarted after a break or a failure.

During development, we **learned** that in order to achieve flexibility and to handle failures and restart, a system should never be controlled using strict sequences or if-then-else programming. When the control system gets out of sync, the common industrial practice is to remove all products and material from the system, restart all resources and put them in well defined initial “home” states. Since we need a robust control that can handle resource failures, flexible and changing work orders and other unplanned events, a better control strategy is needed.

So, if the problem is that some nodes need to know the states of other nodes, maybe we can solve this by sharing internal state of the node and keep the event-based communication?

3.3 Distributed State

If each node shares its state and is responsible for updating it, we can say that the global state and the control is distributed on all nodes. This is a popular control strategy in many new automation trends, since it is simple to implement and hide some of the node details from other nodes. Let us take a look at how a distributed control strategy could have been implemented in the nutrunner example:

Example 2: The nutrunner: Sharing state

In this example, the node communicates with the hardware via 1 digital output and 3 digital inputs. These can be defined as state variables:
$run^c = \{false, true\}$
$run^m = \{false, true\}$
$tqr^m = \{false, true\}$
$fail^m = \{false, true\}$
To run the nutrunner, the command state variable run^c is set to true, which is a digital output, and then the nutrunner responds back by setting the measured state variable run^m to true when it has started. When the torque has been reached, tqr^m becomes true, or if it fails or timeouts, $fail^m$ becomes true. These inputs are reset to false by setting run^c to false again, which can also be used during execution to stop the nutrunner.

Since the nutrunner node is controlled via events (like in the example before), it does not need to expose these details, but instead implements this as an internal node control loop. In our example it communicates its state using the following message:

```
# ROS2 topic:  /nutrunner/state
string state   # 'idle', 'running', 'torque', or 'failed'
```

Now, when the nutrunner node is sharing its state, it is easier for an external control system to know the correct state and act accordingly. However, it is quite challenging to implement a robust state machine that actually works as expected.

When hiding implementation details by using a distributed control state, like in the nutrunner example, the control will initially appear to be simple to implement and may work at first. However, based on our struggling during the demonstrator development, we found out that it is still quite hard to implement fully correct logic in each node and keep the states of all nodes in sync with each other. The main reason for this is that the control system anyway needs to know the details of each node to actually figure out how to restart a system when something happens. So, the **lesson learned:** restart of most nodes almost always depends on the state of other nodes.

It is also challenging to handle safety concerns where the system must guarantee some global specifications when the detailed control is distributed. We have learned

the hard way that a local control loop and internal state in each node almost always becomes tangled with multiple cross-cutting concerns making the complete system hard to maintain and troubleshoot. Another approach is to centralize the control and to use state-based commands.

3.4 State-Based Commands

The benefit of centralized control is that the control system knows about everything and can guarantee global requirements. It can figure out the optimal operation sequence and can restart the complete system in an efficient way.

In the door control earlier, the controller sent the *open* event and then it was waiting for the *done* event. When using state-based commands, the controller instead sends a reference position, $refpos^c$, telling the door what state it wants it to be in while the door is sending back its real state. These messages are sent when something changes as well as at a specific frequency. If one of the commands is lost, another one will soon be sent.

Let us look at how a state-based command could be implemented for the nutrunner example.

Example 3: The nutrunner: State-based communication

Instead of using events to control the nutrunner, the control system and the node instead communicate using the already defined state variables with the following messages:

```
# ROS2 topic:  /nutrunner/command
bool run_c

# ROS2 topic:  /nutrunner/state
bool run_m
bool tqr_m
bool fail_m
bool run_c
```

The control system is sending a reference state while the nutrunner is returning its current state as well as mirroring back the reference. The centralized control system now knows everything about the nutrunner since in practice, the detailed control loop is moved from the nutrunner node to the control node. The mirror is monitored by an external ROS2 node which checks that all nodes are receiving their commands.

During the development of the demonstrator, we discovered that by using state-based commands and avoiding local states in each node, it is possible to implement control strategies for efficient restart, error handling, monitoring and node implementation. It is much easier to recover the system by just restarting it, since the nodes do

not contain any local state. Stateless nodes also makes it possible to add, remove or change nodes while running, without complex logic. This has probably been one of the most important **lessons learned** during the development.

The control of the demonstrator is using centralized control, a global state, and state-based communication. While state-based communication requires a higher number of messages to support control compared to event-based communication, and centralized control with a global state is harder to scale compared to decentralized approaches, we believe that these trade-offs are worth it when creating a robust and efficient control architecture for these types of automation systems. Perhaps the bigger challenge with this approach though, is that the control system needs to handle all the complexity to make it robust. To aid in handling this complexity, we have developed a new control infrastructure called Sequence Planner for robust automation of many sub-systems.

4 Sequence Planner

The use of state-based commands introduced in Sect. 3 implies that a number of devices should continuously get a reference (or goal) state. An overall control scheme is then needed in order to choose what these goal states should be at all time. At the same time, the control system should react to external inputs like state changes or events from machines, operators, sensors, or cameras. Developing such systems quickly becomes difficult due to all unforeseen situations one may end up in. Manual programming becomes too complex and executing control sequences calculated off-line become very difficult to recover from when something goes wrong.

Several frameworks in the ROS community are helping the user with composing and executing robot tasks (or algorithms). For example, the framework ROSPlan [9] that uses PDDL-based models for automated task planning and dispatching, SkiROS [10] that simplifies the planning with the use of a skill-based ontology, eTaSL/eTC [11] that defines a constraint-based task specification language for both discrete and continuous control tasks or CoSTAR [12] that uses Behavior Trees for defining the tasks. However, these frameworks are mainly focused on robotics rather than automation.

4.1 A New Control Infrastructure

Based on what we **learned**, we have employed a combination of on-line automated planning and formal verification, in order to ease both modeling and control. Formal verification can help us when creating a model of the correct behavior of device nodes while automated planning lets us avoid hard-coded sequences and “if-then-else” programming and allowing us to be resource-agnostic on a higher level. We end

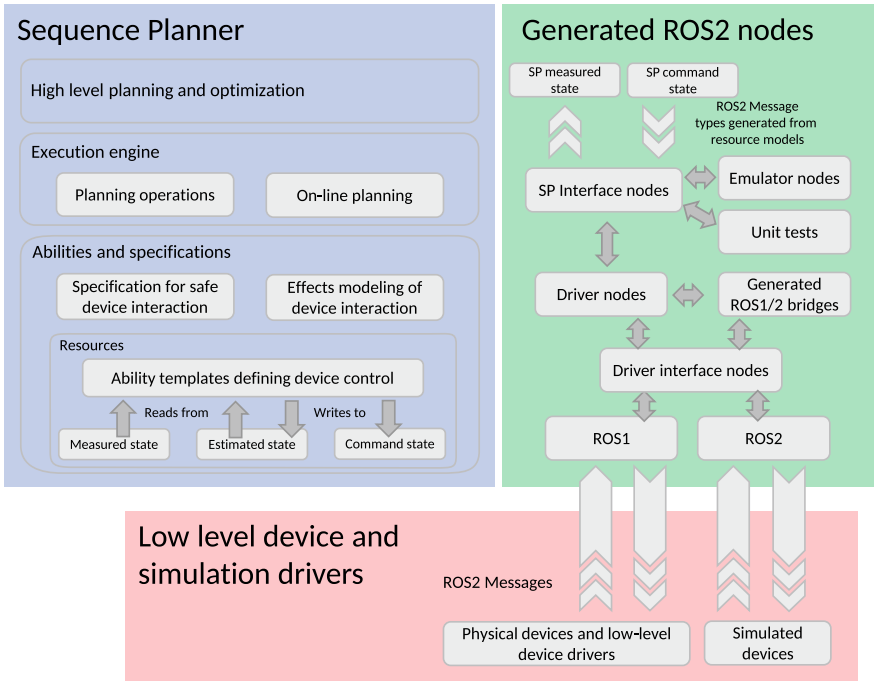


Fig. 3 Sequence Planner control infrastructure overview

up with a control scheme that continuously deliberates the best goals to distribute to the devices. This is implemented in our research software Sequence Planner (SP).

SP is a tool for modeling and analyzing automation systems. Initially [13], the focus was on supporting engineers in developing control code for programmable logical controllers (PLCs). During the first years, algorithms to handle product and automation system interaction [14], and to visualize complex operation sequences using multiple projections [15] were developed. Over the years, other aspects have been integrated, like formal verification and synthesis using *Supremica* [16], restart support [17], cycle time optimization [18], energy optimization and hybrid systems [19], online monitoring and control [20], as well as emergency department online planning support [21]. Recently, effort has been spent to use the developed algorithms and visualization techniques for control and coordination of ROS- and ROS2-based systems [22]. This section will give an overview of how SP can be used to develop ROS2-based automation systems.

The remainder of this section will describe how these SP control models look like, starting from the bottom up of the left part of Fig. 3. In Sect. 5 it is described how the model is translated into ROS2 nodes with corresponding messages (top right of Fig. 3. Because model-based control is used, it is essential that the models accurately represent the underlying systems (bottom of Fig. 3). Therefore Sect. 5.2 describes how randomized testing is used to ease the development of ROS2 nodes.

4.2 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. Recall Definition 1, which divides system state into variables of three kinds: *measured state*, *command state*, and *estimated state*.

Definition 3 A resource i is defined as $r_i = \langle V_i^m, V_i^c, V_i^e, O_i \rangle, r_i \in R$ 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. R is the set of all resources in the system.

The resources defined here are eventually used to generate ROS2 nodes and message definition files corresponding to state variables.

4.3 Generalized Operations

Control of an automation system can be abstracted into performing *operations*. By constructing a transition system modeling how operations modify states 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 high-level planning operations, we define a *generalized operation*.

Definition 4 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 resource variables V_i. G_j is a set of un-named guard predicates over the state of V_i. Sets T^d and T^a define *control* transitions that update V_i, where T^d defines 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 since 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 Sect. 4.6 where the name of the state does not matter.

4.4 Ability Operations

The behavior of resources in the system is modeled by *ability operations* (abilities for short). While it is possible to define transitions using only un-named guard predicates in G (from Definition 4), 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*).

Example 4: The nutrunner: resource and ability template

The resource nr containing state variables of the nutrunner can be defined as $r_{nr} = \langle \{run^m, tqr^m, fail^m\}, \{run^c\}, \emptyset, O_{nr} \rangle$. Notice that $V^e_{nr} = \emptyset$. This is the ideal case, because it means all local state of this resource can be measured.

The table below shows the transitions of a "run nut" ability, where each line makes up one possible 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 (tqr^m) has been reached. Notice that for this ability, $G = \emptyset$. Control actions in T^d are marked by $\bigstar$.

pred. name	predicate	control actions	effects
$enabled$	$\neg run^c \wedge \neg run^m$	$run^c = T^{\bigstar}$	$-$
$starting$	$run^c \wedge \neg run^m$	$-$	$run^m = T$
$executing$	$run^c \wedge run^m \wedge \neg tqr^m$	$-$	$tqr^m = T \vee fail^m = T$
$finished$	$run^c \wedge run^m \wedge tqr^m$	$run^c = F$	$-$
$failed$	$run^c \wedge run^m \wedge fail^m$	$run^c = F$	$-$
$resetting$	$\neg run^c \wedge run^m$	$-$	$run^m = F, tqr^m = F, fail^m = F$

To implement this logic, we have **learned** that it is as complex as implementing it in a distributed state machine, which we talked about in Sect. 3. However, since this model is directly verified and tested together with the complete system behavior, we directly find the errors and bugs. This is much harder in a distributed and asynchronous setup.

While abilities make for well isolated and reusable components from which larger systems can be assembled, they can mainly be used for manual or open loop con-

trol. The next step is therefore to model *interaction* between resources, for example between the robot and the nutrunner, or between the state of the bolts and the nutrunner.

4.5 Modeling Resource Interaction

Figure 3 illustrates two types of interaction between resources: "specification for safe device interaction" and "effects modeling of device interaction". The latter means modeling what can happen when two or more devices interact. As it might not be possible to *measure* these effects, many of them will be modeled as control actions updating estimated state variables.

Given a set of resources and generalized operations as defined by Definition 4, a complete control model is created by instantiating the needed operations into a *global* resource r_g ($r_g \notin R$). Additional estimated state added if needed in order to express the result of different resources interacting with each other. See Example 5.

Safety specifications are created to ensure that the resources can never do something "bad". The instantiated ability operations can be used together with a set of global specifications to formulate a supervisor synthesis problem from the variables and transitions in r_g. Using the method described in [23], 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 [24, 25]. By keeping specifications as a part of the model, we **learned** that there are fewer points of change which makes for faster and less error-prone development compared to changing the guard expressions manually.

4.6 Planning Operations

While ability operations define low-level tasks that different devices can perform, planning operations model how to make the system do something *useful*. As the name suggests, planning operations define intent in the form goal states.

A planning operation has a preconditions which define when it makes sense to try to reach the goal, and a postcondition specifying a target, or goal, state. Planning operations also introduce an estimated state variable o^e to keep track of its own state, $o^e = \{initial, executing, finished\}$. When in the initial state, if the precondition is satisfied, the planning operation updates its estimated state variable to *executing* which will trigger this planning operation to be considered by the on-line planner. When the goal is reached, the operation's state variable transitions to *finished*, and the goal is removed from the on-line planner.

Example 5: The nutrunner: Specifying device interaction and planning operations

We would like to create a planning operation called `TightenBolt1`. When a bolt is in position, which is a precondition of `TightenBolt1`, the operation instructs the system to reach the goal where bolt1 has been tightened. This will happen when the nutrunner has reached the correct torque and the robot is in the correct position at the bolt.

The robot has a number of state variables and in this example, we are interested in its position, where the *measured* state variable $ur.pos^m = \{pos_1, \ldots, pos_n\}$ defines what named pose the robot can be in. In our system we would also like to know the state of bolt1, which is modeled with the *estimated* state variable $bp_1^e = \{empty, placed, tightened\}$.

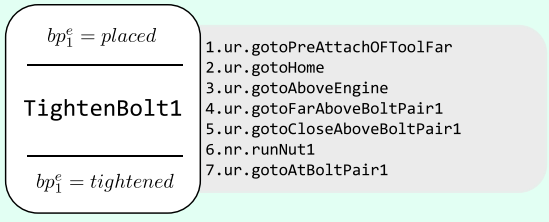
This state variable is not only used by one ability, but in multiple abilities and planning operations. If a variable is only used by planning operations, it is updated by their actions when starting or when reaching the goal, but in this case also other abilities need to know the state of the bolts.

For the planner to be able to find a sequence of abilities that can reach the goal, an ability needs to take the action $bp_1^e =$ tightened. The nut running ability or the robot can be extended with this extra action, but then special variants of abilities need to be made. If, for example, we extend the nut running ability, it needs to be duplicated for each bolt with just minor differences.

A better approach is to generate new special abilities to track these results. In this case, an ability is created only with the following transition:

pred. name	predicate	control actions	effects
$-$	$tqr^m \wedge ur.pos^m = bp_1$	$bp_1^e = tightened$	$-$

A natural way to model `TightenBolt1`, is to start in the pre-state defined by its precondition $bp_1^e = placed$ and end in the post-state defined by the postcondition $bp_1^e = tightened$. The figure below shows a planning operation, with a sequence of dynamically matched ability operations (to the right) to reach the goal state $bp_1^e = tightened$.



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 do 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 shown in Example 5, the operation `TightenBolt1` involves sequencing several abilities controlling the robot and the tool to achieve the goal state of the operation.

4.7 Execution Engine

The execution engine of the control system keeps track of the current state of all variables, planning operations, ability operations and two deliberation plans. The execution engine 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, updated resource state is sent out as 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 deliberation transitions. The plan for the planning operation stage is defined either by a manual sequence or by a planner or optimizer on a higher level. 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.

In this work, planning is done by finding counter-examples using bounded model checking (BMC) [26]. Today's SAT-based model checkers are very efficient in finding counter-examples, even for systems with hundreds of variables. By starting in the current state, the model of the system is unrolled to a SAT problem of increasing bound, essentially becoming an efficient breadth-first search. As such, a counter-example (or deliberation plan) is minimal in the number of transitions taken from the current state. Additionally, well-known and powerful specification languages like *Linear Temporal Logic* (LTL) [27] can be used to formulate constraints.

When decisions are taken by the SP execution engine instead of by the resources themselves, nodes essentially become shallow wrappers around the actual devices in the system. The fact that the core device behavior is modeled in SP allows for generation of a lot boilerplate code (and tests!) in the nodes, which is described in the next section.

5 Auto Generated ROS Nodes

The architecture of the system is based on the ROS hubs described in Sect. 2.1 and in [8], which allow us to separate the system into functional entities. Each ROS hub represents one SP resource which is exchanging messages based on the SP model of

the resource. Based on these models of resources, SP's component generator module generates a set of ROS2 nodes with accompanying messages during compile-time.

5.1 Auto Generated Code for Resource

Five nodes per resource hub are generated: *interfacer*, *emulator*, *simulator*, *driver* and *test*. The *interfacer* node serves as a standardized interface node between SP and nodes of the resource. *Emulator* nodes are generated based on abilities defined in SP and are used when testing the control system since they emulate the expected behavior of the ability. *Simulator* and *driver* nodes implement the actual device control and also an initial template to be used when implementing the connection with the real drivers in ROS. *Test* nodes implement automatic testing of *simulator* and *driver* nodes based on the SP model.

To use the abilities in formal planning algorithms and to support testing of the SP model without being connected to the real subsystem, *measured* variables must be updated by *someone*. We have chosen to place this updating as individual *emulation* nodes [28] rather than to let SP perform it internally. This allows us to always maintain the structure of the network which allows an easy transition to simulated or actual nodes at any time, while at the same time keeping the core execution engine uncluttered. The internal structure of the emulator node is quite simple and can be fully generated based on the SP model. This is because the internal state of an emulation does not have to reflect the internal state of the target which it is emulating, it only needs to mimic the observable behavior to match an existing target.

After the SP model has been tested with generated emulator nodes, the next step in the workflow is to use a variety of simulators instead of emulators connected in the ROS2 network via *simulator* nodes. These nodes are partially generated based on the model of the resource and partially manually assembled. The manually assembled part is decoupled from the main node and is imported into the *simulator* so that the node itself can be independently re-generated when the SP model changes.

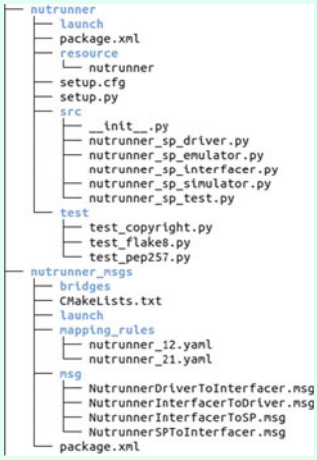
These imports contain interface definitions for specific hardware and software. In the example of the nutrunner, the *driver* node is run on a Raspberry Pi and its manually written part contains mapping between variables and physical inputs and outputs.

Per resource, the *Simulator* node talks to SP over the *interfacer* node using same message types generated for all three nodes, *emulator*, *simulator* and *driver*. This way, the *interfacer* node does not care if the equipment is real, simulated or just emulated.

The driver nodes are in most cases exactly the same as simulator nodes, however we distinguish them by trying to be general. Both the simulator and the driver nodes use same imported manually assembled interfacing components.

Example 6: The nutrunner: model-based component generation

Given the nutrunner's "run nut" ability defined in Example 4, the component generator module generates one ROS2 package containing the different node types for that resource and another package containing the necessary message types:



5.2 Model Based Testing

Unit-testing is a natural part of modern software development. In addition to the nodes generated from SP models, *tests* are also generated. Because the SP model clearly specifies the intended behavior of the nodes via *effects*, it is also possible to generate tests which can be used to determine if the model and the underlying driver implementation do in fact share the same behavior. The generated tests are run over the ROS2 network, using the same interfaces as the control system implementation. This enables *model based testing* using a seamless mixture of emulated, simulated, and real device drivers, which can be configured depending on what is tested. Based on what we **learned**, the generated unit tests can:

- ensure that the device drivers and simulated devices adhere to the behavior specified in the SP model.
- help eliminate "simple" programming errors that waste development resources, for example making sure that the correct topics and message types are used for user-written code.
- provide means to validate if the specifications in the SP model make sense. While formal methods can guarantee correctness w.r.t. some specification, writing the specification is still difficult and needs to be supported by continuous testing.

The tests generated by SP employ property-based testing using the Hypothesis library [29], which builds on ideas from the original QuickCheck [30] implementation. These tools generate random test vectors to see if they can break user-specified properties. If so, they automatically try to find a minimal length example that violates the given properties. The properties that need to hold in our case are that effects specified in the SP model always need to be fulfilled by the nodes implementing the resource drivers. Of course, when bombarded by arbitrary messages, it is not surprising to also find other errors (e.g. crashes).

Example 7: The nutrunner: automated testing during node development

Let's exemplify with the nutrunner resource again. The figure below shows the invocation of *pytest* for executing a generated unit test. The test will send arbitrary commands to the nutrunner node, via the same interface that SP uses for control, and check that the responses from the system match what is expected in the SP model.

```
$ pytest nutrunner_sp_tests.py
============================================= test session starts ==============================================
platform linux -- Python 3.6.8, pytest-4.6.3, py-1.8.0, pluggy-0.12.0
hypothesis profile 'default' -> database=DirectoryBasedExampleDatabase('/home/martin/rosbook/dashing_ws/sr
c/nutrunner/src/.hypothesis/examples')
rootdir: /home/martin/rosbook/dashing_ws/src/nutrunner
plugins: hypothesis-4.24.3, repeat-0.8.0, rerunfailures-7.0, cov-2.5.1
collected 1 item

nutrunner_sp_tests.py F                                                                              [100%]

=================================================== FAILURES ===================================================
_____________________________________________ test_nutrunner_random _____________________________________________

    @given(cmdMsg)
>   @settings(max_examples=50, deadline=None)
    def test_nutrunner_random(data):

nutrunner_sp_tests.py:26:
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

data = {'seq': 0, 'set_run': False}

    @given(cmdMsg)
    @settings(max_examples=50, deadline=None)
    def test_nutrunner_random(data):
        [ ... code removed ... ]
E           AssertionError: Node did not produce desired effect within time limit
E           assert False

nutrunner_sp_tests.py:195: AssertionError
---------------------------------------------- Hypothesis ----------------------------------------------
Falsifying example: test_nutrunner_random(data={'seq': 0, 'set_run': False})
expecting effect: ['tqr_m = False']
Last message seen: nutrunner_msgs.msg.NutrunnerInterfacerToSP(got_set_run=False, run_m=False, tqr_m=True,
fail_m=False)
========================================== 1 failed in 14.36 seconds ===========================================
$
```

The test fails, showing the generated test vector for which the effect failed to emerge, as well the message last observed on the ROS2 network. This makes it easier to spot problems which often arise due to the model and the actual device behavior differing. In this case, it turns out that the driver node does not properly reset tqr^m when run^c resets. Investigating how the driver node works yields:

```
msg.tqr_m = GPIO.input(self.gpi5) == 1
```

It can be seen from the snippet above that the driver node simply forwards what the lower-level device emits to some input pin. It turns out that for this particular nutrunner, the torque reached flag is reset only when the output used to set start running forward goes high.

Instead of introducing more state, and with that, complexity to the driver node, the proper fix for this is to go back and change the SP model to reflect the actual behavior of the node. By moving the torque reset effect from the *resetting* state to the *starting* state of the ability template definition and re-generating our package, the generated test now succeeds:

```
$ pytest nutrunner_sp_tests.py
============================================= test session starts ==============================================
platform linux -- Python 3.6.8, pytest-4.6.3, py-1.8.0, pluggy-0.12.0
hypothesis profile 'default' -> database=DirectoryBasedExampleDatabase('/home/martin/rosbook/dashing_ws/sr
c/nutrunner/src/.hypothesis/examples')
rootdir: /home/martin/rosbook/dashing_ws/src/nutrunner
plugins: hypothesis-4.24.3, repeat-0.8.0, rerunfailures-7.0, cov-2.5.1
collected 1 item

nutrunner_sp_tests.py .                                                                          [100%]

========================================== 1 passed in 35.07 seconds ===========================================
$
```

The nutrunner node should now be safe to include in the automation system.

The unit-test described in Example 7 deals only with testing one resource individually: it is not the instantiated abilities but the templates that are tested. This means that additional preconditions (e.g. handling zone booking) are not tested in this step. As such, while the real driver for the nutrunner could, and probably should be subjected to these kinds of tests, it is not safe to do with, for instance, a robot. Triggering random target states (e.g. robot movements) would be dangerous and such tests need to be run in a more controlled manner. Luckily ROS2 makes it simple to run the tests on a simulated robot by simply swapping which node runs.

It can also be interesting to generate tests from the complete SP model. Consider an ability `ur.moveTo` for moving the UR robot between predefined poses, that is instantiated for a number of different target poses, each with a different precondition describing in which configuration of the system the move is valid. If a simulated node triggers a simulated protective stop when collisions are detected (e.g. between collision objects in the MoveIt! framework), it becomes possible to test whether the SP model is correct w.r.t allowed moves (it would not reach the target if protective stop is triggered). In fact, such a test can be used to *find* all the moves that satisfy the stated property, which eases modeling.

5.3 Node Management and Integrated Testing

The first step in the engineering workflow would be testing the model with emulator nodes, iterating the model until the desired behavior is reached. Afterwards, a simulation or real hardware can be interfaced for one of the resources, while the others remain emulated. This way, targeted model properties can be tested.

To extend the usability of code generation, a node manager is generated to launch node groups for different testing and commissioning purposes, supporting testing of targeted model properties.

Testing the model using emulator nodes can be referred to as virtual commissioning (VC) [31]. The purpose of VC is to enable the control software, which controls

and coordinates different devices in a production station, to be tested and validated before physical commissioning.

The concept of integrated virtual preparation and commissioning (IVPC) [32] aims to integrate VC into the standard engineering workflow as a continuous support for automation engineers. Exactly this is gained with the described workflow of continuous model improvements, autogeneration of components and testing using ROS2 as a common platform that supports integration of different smart software and hardware components.

So, during the development, we have **learned**, that model-based code generation speeds up development and eliminates a lot of coding related errors since components are generated as a one-to-one match to the model. This enables for continuous model improvements during development based on behavior of the generated emulator node.

6 Conclusions

This chapter has presented the development of a industrial demonstrator and the control infrastructure Sequence Planner, together with some practical development guidelines and lessons learned. The demonstrator includes robots, machines, smart tools, human-machine interfaces, online path and task planners, vision systems, safety sensors, optimization algorithms, etc. Coordination and integration of all these functionality has required the well-defined communication interface provided by ROS2 as well as the robust task planning and control of Sequence Planner.

From a technical perspective, ROS2, is an excellent fit for industry 4.0 systems as shown in this chapter. However, being a well-structured and reliable communication architecture is just one part of the challenge. As we have shown, various design decisions will greatly influence the end result, for example in choosing how to communicate between nodes or in how the system behavior is modeled.

Since the presented control approach uses state based communication, the nodes are continuously sending out messages. This implies that care needs to be taken to avoid flooding the network. However, during work on the demonstrator, the bottleneck was more related to planning time rather than network capacity. Future work should include studying how many resources can be handled by SP, at what publishing frequency messages can be reliably received and how this is influenced by different QoS settings.

During the development and implementation of the industrial demonstrator, we learned that:

- To be able to restart the control system, it was important to avoid distributed control state, as the restart of a node almost always depends on the states of other nodes.
- To make the intelligent automation robust and functional, it was much easier to have a centralized control approach.

- It was easier to handle failures, troubleshoot, and maintain the system when using state-based commands.
- For the system to be flexible and robust, we had to stay away from hard-coded control sequences and if-then-else programming.
- Online automated planning and optimization is necessary for any type of intelligent automation.
- It is impossible to develop these types of systems without continuous testing and verification.

The major challenges during development were always related to implementation details and we learned the hard way that the devil is in the details. Developing any type of automation systems will never be a simple task, but it is possible to support the creative design process using algorithms. The control infrastructure Sequence Planner is an open source project that can be found here: http://github.com/sequenceplanner/sp.

Acknowledgements This work was funded by Volvo Groups Trucks Operations and the Swedish Innovation Agency Vinnova through the Production 2030 project Unification and the FFI project Unicorn.

References

1. F.P. Brooks Jr., No silver bullet: Essence and accidents of software engineering. Computer **20**, 10–19 (1987)
2. Volvo GTO Vision. https://www.engineering.com/PLMERP/ArticleID/18868/Vision-and-Practice-at-Volvo-Group-GTO-Industry-40-and-PLM-in-Global-Truck-Manufacturing.aspx. Accessed 14 Apr 2019
3. P. Tsarouchi, A.-S. Matthaiakis, S. Makris, G. Chryssolouris, On a human-robot collaboration in an assembly cell. Int. J. Comput. Integr. Manufact. **30**(6), 580–589 (2017)
4. Å. Fast-Berglund, F. Palmkvist, P. Nyqvist, S. Ekered, M. Åkerman, Evaluating cobots for final assembly, in *6th CIRP Conference on Assembly Technologies and Systems (CATS), Procedia CIRP*, vol. 44 (2016), pp. 175–180
5. V. Villani, F. Pini, F. Leali, C. Secchi, Survey on human-robot collaboration in industrial settings: safety, intuitive interfaces and applications. Mechatronics **55**, 248–266 (2018)
6. W. He, Z. Li, C.L.P. Chen, A survey of human-centered intelligent robots: issues and challenges. IEEE/CAA J. Autom. Sin. **4**(4), 602–609 (2017)
7. A. Hanna, K. Bengtsson, M. Dahl, E. Erős, P. Götvall, and M. Ekström, Industrial challenges when planning and preparing collaborative and intelligent automation systems for final assembly stations, in *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)* (2019), pp. 400–406
8. E. Erős, M. Dahl, H. Atieh, K. Bengtsson, A ROS2 based communication architecture for control in collaborative and intelligent automation systems, in *Proceedings of 29th International Conference on Flexible Automation and Intelligent Manufacturing (FAIM2019)* (2019)
9. M. Cashmore, M. Fox, D. Long, D. Magazzeni, B. Ridder, A. Carreraa, N. Palomeras, N. Hurtós, M. Carrerasa, Rosplan: planning in the robot operating system, in *Proceedings of the Twenty-Fifth International Conference on International Conference on Automated Planning and Scheduling, ICAPS'15* (AAAI Press, 2015), pp. 333–341

10. F. Rovida, M. Crosby, D. Holz, A.S. Polydoros, B. Großmann, R.P.A. Petrick, V. Krüger, *SkiROS—A Skill-Based Robot Control Platform on Top of ROS* (Springer International Publishing, Cham, 2017), pp. 121–160
11. E. Aertbeliën, J. De Schutter, ETASL/ETC: a constraint-based task specification language and robot controller using expression graphs, in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems* (2014), pp. 1540–1546
12. C. Paxton, A. Hundt, F. Jonathan, K. Guerin, G.D. Hager, Costar: Instructing collaborative robots with behavior trees and vision, in *2017 IEEE International Conference on Robotics and Automation (ICRA)* (2017), pp. 564–571
13. P. Falkman, B. Lennartson, K. Andersson, Specification of production systems using PPN and sequential operation charts, in *2007 IEEE International Conference on Automation Science and Engineering* (2007), pp. 20–25
14. K. Bengtsson, B. Lennartson, C. Yuan, The origin of operations: interactions between the product and the manufacturing automation control system, in *13th IFAC Symposium on Information Control Problems in Manufacturing, IFAC Proceedings Volumes*, vol. 42, no. 4, (2009), pp. 40–45
15. K. Bengtsson, P. Bergagard, C. Thorstensson, B. Lennartson, K. Akesson, C. Yuan, S. Miremadi, P. Falkman, Sequence planning using multiple and coordinated sequences of operations. IEEE Trans. Autom. Sci. Eng. **9**, 308–319 (2012)
16. P. Bergagård, M. Fabian, Deadlock avoidance for multi-product manufacturing systems modeled as sequences of operations," in *2012 IEEE International Conference on Automation Science and Engineering: Green Automation Toward a Sustainable Society, CASE 2012, Seoul, 20–24 August 2012* (2012), pp. 515–520
17. P. Bergagård, *On restart of automated manufacturing systems*. Ph.D. Thesis at Chalmers University of Technology (2015)
18. N. Sundström, O. Wigström, P. Falkman, B. Lennartson, Optimization of operation sequences using constraint programming. IFAC Proc. Vol. **45**(6), 1580–1585 (2012)
19. S. Riazi, K. Bengtsson, R. Bischoff, A. Aurnhammer, O. Wigström, B. Lennartson, Energy and peak-power optimization of existing time-optimal robot trajectories, in *2016 IEEE International Conference on Automation Science and Engineering (CASE)* (2016)
20. A. Theorin, K. Bengtsson, J. Provost, M. Lieder, C. Johnsson, T. Lundholm, B. Lennartson, An event-driven manufacturing information system architecture for industry 4.0. Int. J. Product. Res. 1–15 (2016)
21. K. Bengtsson, E. Blomgren, O. Henriksson, L. Johansson, E. Lindelöf, M. Pettersson, Å. Söderlund, Emergency department overview - improving the dynamic capabilities using an event-driven information architecture, in *IEEE International Conference on Emerging technologies and factory automation (ETFA)* (2016)
22. M. Dahl, E. Erős, A. Hanna, K. Bengtsson, M. Fabian, P. Falkman, Control components for collaborative and intelligent automation systems, in *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pp. 378–384 (2019)
23. S. Miremadi, B. Lennartson, K. Åkesson, A BDD-based approach for modeling plant and supervisor by extended finite automata. IEEE Trans. Control Syst. Technol. **20**(6), 1421–1435 (2012)
24. P. Bergagård, P. Falkman, M. Fabian, Modeling and automatic calculation of restart states for an industrial windscreen mounting station. IFAC-PapersOnLine **48**(3), 1030–1036 (2015)
25. M. Dahl, K. Bengtsson, M. Fabian, P. Falkman, Automatic modeling and simulation of robot program behavior in integrated virtual preparation and commissioning. Proc. Manufact. **11**, 284–291 (2017)
26. A. Biere, A. Cimatti, E. Clarke, 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
27. A. Pnueli, The temporal logic of programs, in *18th Annual Symposium on Foundations of Computer Science (SFCS 1977)* (IEEE, 1977), pp. 46–57

28. E. Erős, M. Dahl, A. Hanna, A. Albo, P. Falkman, K. Bengtsson, Integrated virtual commissioning of a ROS2-based collaborative and intelligent automation system, in *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)* (2019), pp. 407–413
29. D.R. MacIver, Hypothesis 4.24 (2018), https://github.com/HypothesisWorks/hypothesis
30. K. Claessen, J. Hughes, Quickcheck: a lightweight tool for random testing of haskell programs, in *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP '00* (ACM, New York, 2000), pp. 268–279
31. C.G. Lee, S.C. Park, Survey on the virtual commissioning of manufacturing systems. J. Comput. Des. Eng. **1**(3), 213–222 (2014)
32. M. Dahl, K. Bengtsson, P. Bergagård, M. Fabian, P. Falkman, Integrated virtual preparation and commissioning: supporting formal methods during automation systems development. IFAC-PapersOnLine **49**(12), 1939–1944 (2016). 8th IFAC Conference on Manufacturing Modelling, Management and Control MIM 2016

Endre Erős received his M.Sc. degree in Control Engineering in 2018 at the Faculty of Mechanical Engineering, University of Belgrade, Serbia. He is currently working as a Ph.D. student in the automation research group at Chalmers University of Technology, researching discrete control of complex systems. Endre is involved in developing, testing and integrating discrete control methods in two projects that aim to bring humans and robots closer together in working, as well as in living environments.

Martin Dahl received the M.Sc.Eng. degree in Computer Science and Engineering in 2015 from Chalmers University of Technology in Gothenburg, Sweden. He is currently working towards a Ph.D. degree in Electrical Engineering, also at Chalmers University of Technology. His research is currently focused on discrete control of complex systems.

Atieh Hanna received the M.Sc.Eng. degree in Electrical Engineering in 2001 from Chalmers University of Technology in Gothenburg, Sweden. From 2001 to 2011, she was working at Volvo Group Trucks Technology as a Development Engineer and from 2012 as a Research and Development Engineer at Volvo Group Trucks Operations within Digital and Flexible Manufacturing. She is currently working towards a Ph.D. degree and her research is focused on planning and preparation process of collaborative production systems.

Per-Lage Götvall received his B.Sc. from Chalmers in 1996 after an eight-year career in the Royal Swedish navy as a marine engineer. He joined the Volvo Group the same year and has worked in various roles at the company since then. In 2015 started a work aimed at understanding what was required for humans and robotic machines to work together. This work, as a manager for Volvo Group Trucks area of Transport system, robotics, was followed by a position as a Senior research engineer at Volvo Group Trucks Operations within the area of Flexible manufacturing, which is also the current position at the Volvo Group.

Petter Falkman received the Ph.D. degree in Electrical Engineering in 2005 from Chalmers University of Technology, Göteborg, Sweden. Dr. Falkman is currently Associate Professor at the Department of Signals and Systems, Chalmers University of Technology. He received his Master of Science in Automation and Mechatronics at Chalmers in 1999. Dr. Falkman is vice head of utilization in the electrical engineering department. Dr. Falkman is program director for the Automation and Mechatronics program at Chalmers. He is profile leader in the Chalmers production area of advance. He is also part of the Wingquist Laboratory, a research center for virtual product and production development at Chalmers. His main research topic concerns information integration, virtual preparation, machine learning, control, and optimization of automation systems.

Kristofer Bengtsson received the Ph.D. degree in Automation in 2012 from Chalmers University of Technology, Gothenburg, Sweden. From 2001 to 2005 he was with Advanced Flow Control AB developing automation control systems and user interfaces, and from 2005 to 2011 with Teamster AB, an automation firm in Gothenburg. From 2012 he is a researcher in the Automation research group at Chalmers and from 2011 he is also with Sekvensa AB, a research consulting firm. His current research interest includes control architectures based on ROS2 and new AI-based algorithms for complex automation systems as well as supporting healthcare providers with online prediction, planning and optimization.