2023 IEEE International Conference on Robotics and Automation (ICRA 2023)
May 29 - June 2, 2023. London, UK

RoboSC: a domain-specific language for supervisory controller synthesis of ROS applications

Bart Wesselink[1], Koen de Vos[2], Ivan Kuertev[1], Michel Reniers[2] and Elena Torta[2]

***Abstract*— The paper presents a novel domain-specific language, RoboSC, for developing supervisory controllers for robotic applications. RoboSC supports concepts of ROS/ROS2 and supervisory control theory. It enables users to focus on the modeling and the synthesis process of supervisory controllers for ROS applications only because it generates all artifacts needed to connect such controllers to ROS applications and deploy them. Validation tests with actual and simulated robots show the approach's feasibility and indicate reduced coding effort.**

I. INTRODUCTION

Robots are more and more used in safety critical domains where interaction with humans is either unavoidable or required. Examples are nursing homes [1] or search and rescue scenarios [2]. Particularly when human interaction is unavoidable, ensuring that the robotic system will guarantee requirements (e.g., safety) is of primary importance. Validation/verification of requirements is addressed at different levels of the design [3]. Algorithmic performance can be formally proved [4], discrete event controllers (e.g., state machines) can be formally verified [5] and extensive validation tests can be performed. Ensuring guarantees at design-time is the cheapest option with shorter lead time. To facilitate the design of robotic systems with requirements guarantee at design-time this paper proposes and validates a tool (i.e. RoboSC), in the form of a domain-specific language, that allows synthesis of discrete events supervisory controllers for ROS applications from discrete event models of nodes and the formalization of requirements. The tool provides code generation capabilities to effortlessly integrate the supervisors in robots supporting ROS [6] and ROS2 [7] resulting in reduced coding effort.

II. RELATED WORK

The ROS and ROS2 frameworks provide native tools to specify discrete event controllers. SMACH [8] is perhaps one of the most well-known. These tools, however, do not provide options to formally prove satisfaction of requirements directly (e.g., livelock and deadlock freeness). The generic lack of tooling to facilitate formal verification of requirements is also reported by Nordmann et al. [9] in their review paper. One of the few examples is reported in Heinzemann and Lange [10] who developed vTSL: a formally verifiable DSL for specifying robotic tasks. The language uses an external model checker (i.e. Spin [11]) to ensure that the robot adheres to specified properties. The tool supports ROS integration and code generation. Another language that supports formal verification of the controller is Salty [12]. The language uses Generalized Reactivity(1) [13] specifications to automatically generate controllers. Users specify generalized reactivity relations between the state of the environment and guarantees the robot should make under those conditions. Based on all relations that the user specifies, the language synthesizes the controller.

A language that provides code generation but no formal verification is MontiArcAutomaton [14]. It adheres to the component and connector (C & C) architecture. In this architecture, the composition of all components is separated from the individual behaviour of a single component. Communications between components are defined as ports. The behaviour of a component can be described as an automaton or using a problem-specific language.

Other approaches presented in literature are Koord: a DSL that focuses on programming and verifying applications for distributed robots [15] and RobotML [16], which is developed around the ontology of a robot.

In this paper we present a new language, RoboSC, that builds on concepts of supervisory control theory [17] and allows to formally synthesize supervisory controllers based on automata models of ROS/ROS2 nodes and requirements on their synchronized behavior. Contrary to Heinzemann and Lange [10] and languages like Koord [15] or RobotML [16], RoboSC uses a specification-based approach, which integrates the verification into the language, rather than an external tool. Furthermore, RoboSC also provides support for communication using services and actions, which vTSL does not. Similarly to Salty [12] we adopt a synthesis-based approach but RoboSC is grounded in supervisory control theory instead of Generalized Reactivity. RoboSC is modular in a similar way as MontiArcAutomaton [14]. The models described with RoboSC can support ROS and ROS2 independently. A preliminary investigation of the concepts expressed in RoboSC was presented in Kok et al. [18] and Torta et al. [19]. In this paper we present a tool that builds on those concepts enabling their easier specification and integration alongside extending them to ROS2.

III. METHOD

A. Supervisory Control Theory

In this section we briefly and informally present the relevant concepts of Supervisory Control (SC) theory (see [17] for a theoretical introduction) upon which RoboSC is based.

[1] Faculty of Computer Science, Eindhoven University of Technology, The Netherlands {b.b.a.wesselink, i.kurtev} at tue.nl
[2]Faculty of Mechanical engineering, Eindhoven University of Technology, The Netherlands {m.a.reniers, k.d.vos, e.torta} at tue.nl

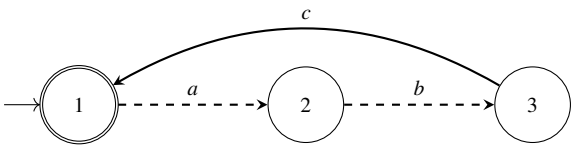

Fig. 1. Automaton representation. Controllable events (solid line), uncontrollable events (dashed line).

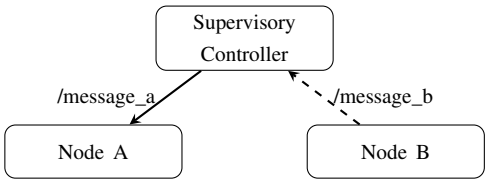

Fig. 2. Example of supervisor deployment as controller.

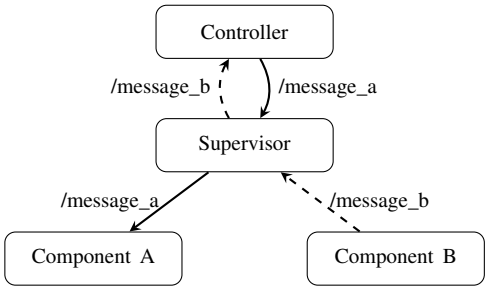

Fig. 3. Overview of the structure of the supervisory layer.

Supervisor synthesis is a generative technique. A supervisor is derived from a collection of plants and requirements. The plants describe capabilities of a system without any integrated supervisory controller. The requirements model the functions a system is supposed to perform. The goal of supervisor synthesis is to compute a supervisor that enforces the requirements, assuming the modelled behavior of the plants. Additionally, the supervisor prevents blocking, guarantees controllability (i.e., only disables controllable events), and does not restrict the system any further than is required.

A plant is modeled as a finite state automaton ([20]), see Fig. 1. The (abstract) state of a (sub)system is represented by a location, and transitions between such states are represented by edges labelled by events. Events are assumed to occur instantaneously. An automaton has an initial location and possibly a number of so-called marked locations. Marked locations indicate states where the system is stable or has finished some desired task. The events of an automaton are partitioned in controllable and uncontrollable events. Controllable events can be prevented from occurring by a supervisor, whereas uncontrollable events cannot be disabled.

An uncontrolled system is represented by a collection of finite state automata. These interact by synchronization of shared events, if any. If so desired, multiple finite state automata can be combined into a single one by repeatedly performing the so-called synchronous product [20]. Requirements can be modelled as state-based expressions ([21]). State-based expressions can specify requirements in two forms. The first form is to indicate in which states of the uncontrolled system an event is allowed. For example, we can state that the occurrence of a "set_vel" event of component Machine Controller (MC), which is used to control the speed of the wheels of a robot, can occur only if the Wheel Controller (WC) is in state $initialized$:

$$\text{MC.set\_vel } \mathbf{needs} \text{ WC.initialized} \tag{1}$$

The second form is to specify that some system states disable a certain event from occurring. For example, we can state that when the Bumper Controller (BC) is in state pressed, forward motion is disabled by the MC:

$$\text{BC.pressed } \mathbf{disables} \text{ MC.forward} \tag{2}$$

A synthesized supervisor consists of the plant automata, the requirements, and if needed additional conditions on controllable events that are needed to guarantee the reachability of a marked location (non-blocking), the disabling of controllable events only (controllability) and its maximally permissiveness.

RoboSC allows the modelling of plants and requirements. To synthesize the supervisor, it relies (transparently for the user) on an established language for SC synthesis, i.e. CIF ([22]).

B. Mapping of ROS/ROS2 concepts to SC theory

RoboSC supports a mapping between SC theory and ROS/ROS2 concepts (see [19] for a formalization).

Each node is represented as one or multiple automata. The set of states, marked states and initial states relate to the observable behavior of the nodes and are defined by the user. The set of transitions represent the topics to which a node is connected. ROS/ROS2 supports three types of communication between nodes and topics (i.e. messages, services and actions). For messages, RoboSC maps as controllable events all subscriptions to topics and as uncontrollable events all publishings to topics. For services, requests are controllable and replies are uncontrollable. For actions feedback, response and error are uncontrollable. All the remaining communication is controllable.

RoboSC allows the deployment of the supervisor in two complementary ways. The first option is to deploy the supervisor as a controller, (see Fig. 2).

In this case the supervisor directly controls the node of the application by publishing and subscribing to topics. This deployment option is relevant when a new controller with formal guarantees needs to be developed. The other option is to deploy the model as a supervision layer. In the latter case the supervisor is connected to an existing controller and ensures that the requests of the controller are allowed in the current system state (see Fig. 3). This option is relevant when legacy controllers already exists and we want to guarantee (the absence of) some behavior at system level.

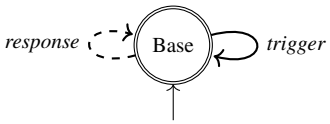

Fig. 4. Plant of a message (depending on message direction).

C. The RoboSC language

RoboSC is based on the insights reported in Sections III-A and III-B. To this end, the language defines several concepts. The first concept is a *component*, which, in general, maps to a single node. Users can define the communication modalities of each component as:

- Messages that a component can send or receive
- Services that a component offers
- Actions that a component implements

For each communication modality, the data type can be specified as a basic type (already built-in in RoboSC) or as user-specified type using custom objects.

For each component, the user must specify its automaton. State transitions are triggered by communication from and to the middleware. States can also be defined as marked states. The transitions between states can have guards, which are represented as boolean expressions, which support references to variables, component states and basic logical operators.

The state of each component can be used to specify requirements on the communication from the controller to the middleware enabling the specification of state-based requirements. When communicating, data can be passed using provide statements. Provide statements can have guards that specify when data should be sent along. These guards can define the same expressions as the regular component state transitions.

RoboSC allows to specify automata models for the three types of communication introduced earlier (i.e. messages, services and actions).

The automata model of a message is relatively straightforward, and only consists of a single state (see Fig 4). Depending on whether the component subscribes to the topic or publishes to the topic the event on the transition will be modelled as either a controllable or an uncontrollable event respectively. Such automata definition in RoboSC is reported in Listing 1.

```
outgoing message distance with
    identifier: "/distance", type: double
incoming message stop with
    identifier: "/stop", type: none
```

Listing 1. Messages concept sample code.

The plant model of a service presents more states than the one of a message (see Fig 5). The definition of a service in the RoboSC is shown in Listing 2.

```
service add_two_ints with request: TwoNumbers,
    response: integer
```

Listing 2. Service concept sample code.

The initial state is "Idle" since the service is not invoked. When the service is triggered the plant transitions to the "Await" state. When a response is received, the automaton transitions to either the "Resp." or the "Error" states depending on whether the call succeeded or not. From both states, the automaton can reach the "Idle" state again. RoboSC supports ROS and ROS2. The two versions of the middleware have different deployment strategies for services, i.e. synchronous in ROS and asynchronous in ROS2. RoboSC assumes services are asynchronous. When generating code for ROS, RoboSC creates binding code to transform services from synchronous to asynchronous. This means that when the controller invokes a service in ROS, execution will be blocked until the service has returned a response. When the response is received, the controller will immediately process this response by updating its state. For nodes running in ROS2, the service response is received asynchronously.

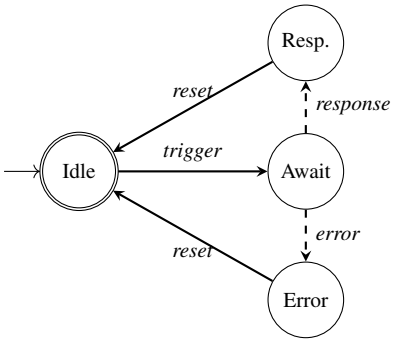

Fig. 5. Plant of a service.

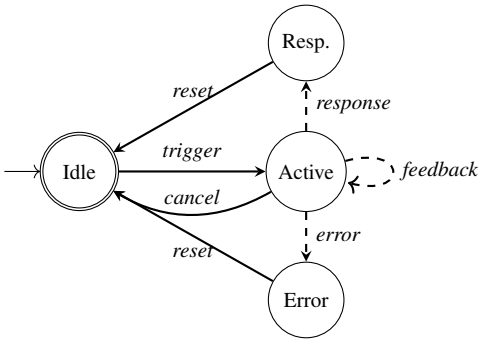

Fig. 6. Plant of an action.

The SC plant of an action (see Fig.6) is very similar to that of a service, again represented by four states: "Idle", "Active", "Response" and "Error". Differently from a service, an action has the option to receive feedback from the server during execution using an uncontrollable event. The event to cancel the action is also modelled. An action definition in RoboSC with links to external data-types is shown in Listing 3.

```
action navigate with identifier: "/navigate",
    request: Twist, response: none, feedback:
    none links twist
```

Listing 3. Action concept sample code.

All of the provide statements, which attach data to communication with the middleware, are transformed to plant models as well. The states are defined as $S = \{empty\} \cup P$,

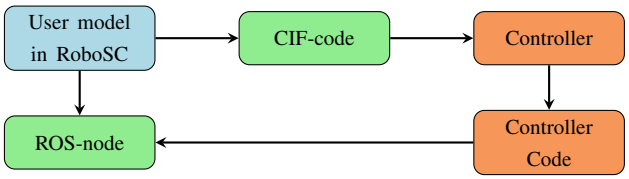

Fig. 7. Overview of the transformation chain adopted by RoboSC. Blue rectangles are user inputs, green rectangles are artifacts generated by RoboSC and orange rectangles represent artifacts generated by CIF.

with P being defined as the set of all data exchanged through a specific communication channel. In each state, edges to all other states are added, where the transition guard of each edge is defined as the condition specified in the provide-statement. The *empty* state guards represents the case that none of the other guards holds. An example of a data provisioning statement is reported in Listing 4.

```
provide halt with { linear: { x: 0.0 } }
```

Listing 4. Data provisioning sample code

RoboSC allows to specify requirements on communication between components and the middleware. All of the communication from the controller to the middleware happens when a trigger event is executed. RoboSC maps requirements to these trigger events. For actions, there is an additional requirement which enables the cancel event when one of the requirements does not hold anymore.

To foster the adoption of RoboSC, the accompanying GitHub repository[1] provides a tutorial for the synthesis of a controller in a sample scenario. The grammar of the language is also publicly available.

To facilitate debugging of the behaviour of the generated controllers, RoboSC offers debugging tools. All controllable and uncontrollable events can be logged. Additionally, there is the option of a live visualization of the automata of all components. These visualizations show the current state, values and transitions that are being taken by the controller. A video showing the execution of the scenarios and the visualization features of RoboSC is available[1].

By default RoboSC guarantees that a component plant model is always responsive to incoming uncontrollable events. It does so by automatically adding a self transition to all states of the plant in which the user did not specify a transition labelled with that uncontrollable event.

RoboSC is the front-end language for users. On the back-end a chain of transformation is performed which allows to synthesize a supervisor by relying on an external tool for SC synthesis (i.e. CIF[22]) and to generate all glue code necessary to integrate the generated supervisor's code into ROS. The transformation chain is shown in Fig. 7.

D. Experimental set-up

This section evaluates the functionality of RoboSC in two distinct scenarios: Line follower and Maze solver. More evaluation scenarios are described and available in the accompanying GitHub repository[1]. In both scenarios we test

[1]https://github.com/bartwesselink/robosc

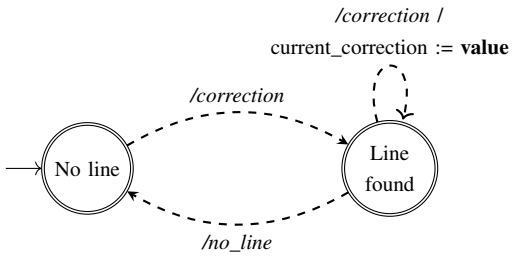

Fig. 8. Plant of the Line Follower component.

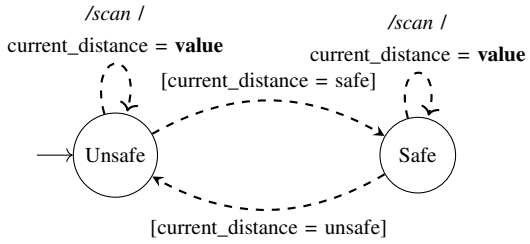

Fig. 9. States of the LiDAR sensor component.

the synthesized controller in simulation. The Line follower has been tested with real-hardware as well (i.e. a Turtlebot3 Pi). The synthesized controller is deployed on a Dell XPS 13" laptop running Ubuntu 20.04.4 (LTS) with 16 GB of RAM and an Intel® Core™ i7-10510U CPU @1.80GHz processor. The controller communicates remotely with the Turtlebot. For ROS, the Noetic distribution was used. For ROS2 the Foxy distribution was used.

1) Line follower: In this scenario the controller is deployed as a supervisor following the pattern in Fig. 3. The task of the robot is to follow a line. A node processes the camera feedback to find the center of the line when the latter is detected. The node publishes the offset of the line center to the center of the image. The latter is input to a controller node that adjusts the angular velocity of the robot.

The supervisory layer ensures the requirement that th e robot does not move when something is in front of it regardless of what the line follower controller requests. Three components are modelled with RoboSC to synthesize the supervisor. A Line Follower component models when a line has been found (see Fig. 8). It has two states, depending on whether the sensors have detected a line. A variable stores the correction which the robot can use to adjust its angular velocity to stay on track. A LiDAR sensor component has two states that indicate if obstacles are too close to the robot or not (Fig. 9). Lastly, there is a Platform component that allows the event "move" to be propagated to the system with linear and angular velocity as associated data. It is modelled as a single state with a self-transitions labelled "move".

The requirements to synthesize the supervisor are relatively straight-forward. For the controllable event "move" to be emitted with current_correction as associated data, the following requirement should hold:

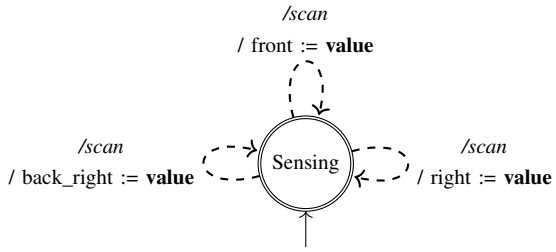

Fig. 10. States of the LiDAR sensor component.

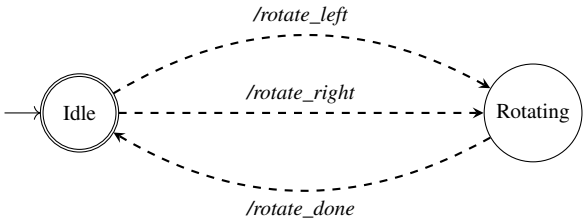

Fig. 11. States of the Rotator component.

```
requirement move needs Line_Follower.line_found
    and LiDAR_Sensor.safe
```

The move message to the middleware is provided with a velocity value based on the correction value that the controller receives from the line follower component. The implementation of this example is publicly available [1].

2) Maze solver: In this scenario the controller is deployed as a supervisory controller following the pattern in Fig. 2. The robot attempts to escape a maze by ensuring that the wall of the maze is to the right of the robot at all times. The robot uses the LiDAR sensor to measure its distance to the walls. The supervisory controller controls the robot to keep following the walls on its right side until it eventually reaches the end. Three components are modelled. First, there is a LiDAR sensor component which stores whether there is a wall at 0°, 90° and 135°. The automaton is shown in Fig. 10. The second component, the Rotator (see Fig. 11), has two states, a state where it is awaiting a command, and a state where it is executing a command, and therefore rotating to a specific position. Lastly, there is a Platform component that allows the event "move" to be propagated to the system with linear and angular velocity as associated data. It is modelled as a single state with a self-transitions labelled "move".

Issuing a move command is constrained by the following requirement:

```
requirement move needs Rotator.idle and ((
    LiDAR_Sensor.right = wall and LiDAR_Sensor.
    front = no_wall) or (LiDAR_Sensor.right =
    no_wall and LiDAR_Sensor.front = no_wall
    and LiDAR_Sensor.back_right = no_wall))
```

Issuing a rotate right command is described by the following requirement:

```
requirement rotate_right needs Rotator.idle and
    (LiDAR_Sensor.right = no_wall and
    LiDAR_Sensor.back_right = wall)
```

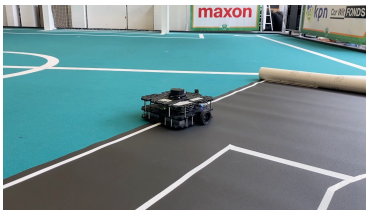

Fig. 12. The line follower running on the TurtleBot3 Pi.

Including communication with the middleware				
	Avg. (μs)	**Min.** (μs)	**Max.** (μs)	**S. D.** (μs)
Line Follower	34	11	160	25
Maze solver	40	9	660	40
Excluding communication with the middleware				
	Avg. (μs)	**Min.** (μs)	**Max.** (μs)	**S. D.** (μs)
Line Follower	5	0	73	5
Maze solver	2	0	139	7

TABLE I

EXECUTION TIME MEASUREMENTS OF A CONTROLLER CYCLE.

For the robot to turn left using the rotate left command, the following requirement should be satisfied:

```
requirement rotate_left needs Rotator.idle and
    (LiDAR_Sensor.front = wall)
```

The move message is provided with a fixed value that determines their speed. Furthermore, the rotate messages are provided with a value of 90° that represents the amount of degrees the robot should rotate when invoked.

E. Performance Criteria

For every scenario, the sequence of uncontrollable events that the supervisor receives and the controllable events that it triggers are logged. This is used to validate the qualitative correctness of the generated supervisor, by checking whether the supervisor is allowed to trigger a controllable event based on the state of the environment at that point. Each scenario is executed in simulation 3 times, where the execution time of a single control loop is measured 1000 times. This means that both the total time, including communication with the middleware, and the time the supervisory controller takes are measured separately. The measurement quantifies the execution time overhead the controller adds. Next to that, the amount of generated source lines of codes is measured. This is an indicator of the amount of manual coding time that RoboSC saves.

IV. RESULTS

The results for the generated lines of code can be found in Fig. 13. The execution times, both with communication to the middleware and without communication, can be found in Table I.

A. Line follower

A picture of the robot following the line is displayed in Fig. 12. Controllable events are represented in orange, uncontrollable events in blue.

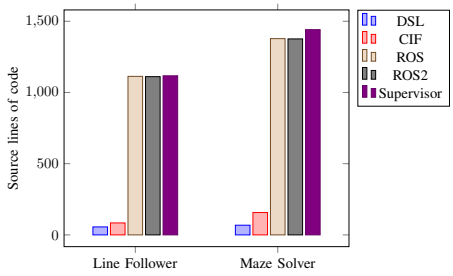

Fig. 13. Lines of code for RoboSC and generated code for all scenarios.

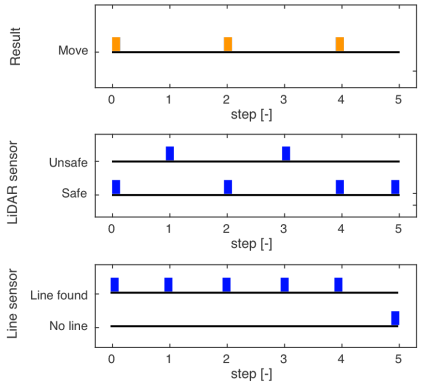

Fig. 14. Sequence of events plots for the line solver scenario.

1) Sequence of events: Fig. 14 represents the sequence of events that takes place during the scenario. Note that steps where no new event is observed are not reported. When it is safe to move (which is a variable from the LiDAR sensor) and there is a line, then movement will take place. If one of these does not hold, the supervisor will not allow the move event to happen.

2) Execution time: The values in Table I show that most of the time for a controller update is spent on communication with the middleware. The updating of the supervisory controller state takes 5 microseconds on average, whereas when including the communication with the middleware, it takes 34 microseconds. The line follower scenario uses the supervisory layer approach, meaning that the controller is updated upon a command from the controller or communication from the middleware.

3) Generated lines of code: The CIF-code that is generated for the line follower, is about 1.5 times as much as what the user specifies in the DSL. For the ROS-nodes (ROS1, ROS2 and the supervisor) this is even more, namely about 20 times as much. Note, that this also includes the controller code that is generated from the CIF-code.

B. Maze solver

1) Sequence of events: A subset of the sequence of events for the maze solver can be found in Fig. 15. When the robot detects that there is a wall in front, it will start rotating left and do nothing until it has completed the rotation. Once it has no wall on its right anymore, it starts rotating to the right. Again, the orange color represents controllable events, whereas uncontrollable events are visualized in blue.

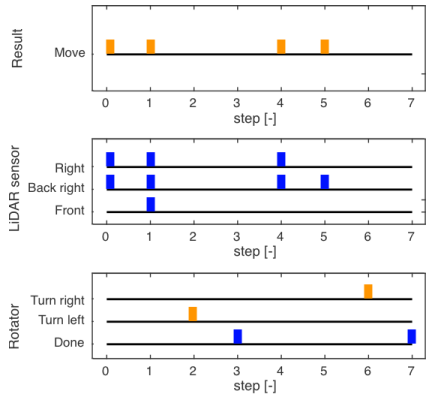

Fig. 15. Sequence of events plots for the maze solver scenario.

2) Execution time: Similar to the line follower scenario, the majority of time spent on a controller update is due to communication with the middleware. In this case, the controller update happens either on a control loop execution, or communication coming from the middleware. On average, a control loop execution takes 40 microseconds to complete, including communication with the middleware. Excluding the messages sent from the controller to the middleware, it takes 2 microseconds.

3) Generated lines of code: Starting from the RoboSC code that was specified by the user, the amount of CIF source lines of code that are generated is about twice as much. The code for the generated ROS-nodes is about 20 times more compared to the RoboSC code.

V. Discussion & Conclusion

In this paper we presented a novel tool, RoboSC, in the form of a domain-specific language, for the synthesis of supervisory controllers for ROS/ROS2 applications. The language allows the user to focus on modelling and the specification of requirements by taking care of the generation of all artifacts needed for deployment. The language was validated to show the feasibility of the approach in two scenarios, including hardware testing, to demonstrate its applicability. Additional validation scenarios are available in the public code repository accompanying the paper. RoboSC allows working at a higher level of abstraction when creating supervisory controllers for ROS applications thus reducing development time and still retaining all guarantees for the requirements. This is demonstrated by the smaller size of the RoboSC code compared to the generated implementation. In the future, we envision the usage and further validation of RoboSC for the synthesis of safe supervisory controllers particularly targeting multi-robot systems as well as human-robot interaction. In both use-cases the complexity and need of correctness guarantees are key aspects of the development.

References

[1] D. O. Johnson, R. H. Cuijpers, J. F. Juola, *et al.*, "Socially assistive robots: A comprehensive approach to extending independent living," *International journal of social robotics*, vol. 6, no. 2, pp. 195–211, 2014.

[2] J. P. Queralta, J. Taipalmaa, B. C. Pullinen, *et al.*, "Collaborative multi-robot search and rescue: Planning, coordination, perception, and active vision," *Ieee Access*, vol. 8, pp. 191 617–191 643, 2020.

[3] M. Luckcuck, M. Farrell, L. A. Dennis, C. Dixon, and M. Fisher, "Formal specification and verification of autonomous robotic systems: A survey," *ACM Computing Surveys (CSUR)*, vol. 52, no. 5, pp. 1–41, 2019.

[4] B. Lacerda, F. Faruq, D. Parker, and N. Hawes, "Probabilistic planning with formal performance guarantees for mobile service robots," *The International Journal of Robotics Research*, vol. 38, no. 9, pp. 1098–1123, 2019.

[5] R. Bohrer, Y. K. Tan, S. Mitsch, A. Sogokon, and A. Platzer, "A formal safety net for waypoint-following in ground robots," *IEEE Robotics and Automation Letters*, vol. 4, no. 3, pp. 2910–2917, 2019.

[6] M. Quigley, K. Conley, B. Gerkey, *et al.*, "Ros: An open-source robot operating system," in *ICRA workshop on open source software*, Kobe, Japan, vol. 3, 2009, p. 5.

[7] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, "Robot operating system 2: Design, architecture, and uses in the wild," *Science Robotics*, vol. 7, no. 66, eabm6074, 2022.

[8] J. Bohren and S. Cousins, "The SMACH high-level executive," *IEEE Robotics & Automation Magazine*, vol. 17, no. 4, pp. 18–20, 2010.

[9] A. Nordmann, N. Hochgeschwender, D. Wigand, and S. Wrede, "A survey on domain-specific modeling and languages in robotics," 2016.

[10] C. Heinzemann and R. Lange, "Vtsl-a formally verifiable dsl for specifying robot tasks," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2018, pp. 8308–8314.

[11] G. J. Holzmann, "The model checker spin," *IEEE Transactions on software engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[12] T. Elliott, M. Alshiekh, L. R. Humphrey, L. Pike, and U. Topcu, "Salty-a domain specific language for gr (1) specifications and designs," in *2019 International Conference on Robotics and Automation (ICRA)*, IEEE, 2019, pp. 4545–4551.

[13] B. Jobstmann, S. Galler, M. Weiglhofer, and R. Bloem, "Anzu: A tool for property synthesis," in *International Conference on Computer Aided Verification*, Springer, 2007, pp. 258–262.

[14] J. O. Ringert, R. Alexander, R. Bernhard, and W. Andreas, "Language and code generator composition for model-driven engineering of robotics component & connector systems," 2015.

[15] R. Ghosh, C. Hsieh, S. Misailovic, and S. Mitra, "Koord: A language for programming and verifying distributed robotics application," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, 2020.

[16] S. Dhouib, S. Kchir, S. Stinckwich, T. Ziadi, and M. Ziane, "Robotml, a domain-specific language to design, simulate and deploy robotic applications," in *Simulation, Modeling, and Programming for Autonomous Robots: Third International Conference, SIMPAR 2012, Tsukuba, Japan, November 5-8, 2012. Proceedings 3*, Springer, 2012, pp. 149–160.

[17] W. M. Wonham, K. Cai, *et al.*, *Supervisory control of discrete-event systems*. Springer, 2019.

[18] J. Kok, E. Torta, M. A. Reniers, J. van de Mortel-Fronczak, and M. van de Molengraft, "Synthesis-based engineering of supervisory controllers for autonomous robotic navigation," *IFAC-PapersOnLine*, vol. 54, no. 2, pp. 259–264, 2021.

[19] E. Torta, M. Reniers, J. Kok, J. van de Mortel-Fronczak, and M. van de Molengraft, "Synthesis-based engineering of supervisory controllers for ros-based applications," *Control Engineering Practice*, vol. 133, p. 105 433, 2023.

[20] M. Skoldstam, K. Aakesson, and M. Fabian, "Modeling of discrete event systems using finite automata with variables," in *IEEE Conference on Decision and Control*, IEEE, 2007, pp. 3387–3392.

[21] J. Markovski, D. A. van Beek, R. J. Theunissen, K. G. Jacobs, and J. Rooda, "A state-based framework for supervisory control synthesis and verification," in *IEEE Conference on Decision and Control (CDC)*, IEEE, 2010, pp. 3481–3486.

[22] D. A. van Beek, W. J. Fokkink, D. Hendriks, *et al.*, "CIF 3: Model-based engineering of supervisory controllers," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2014, pp. 575–580.