Synthesis-Based Engineering of Supervisory Controllers for Autonomous Robotic Navigation

J.W. Kok * E. Torta * M.A. Reniers * J.M. van de Mortel-Fronczak * M.J.G. van de Molengraft *

* Eindhoven University of Technology, The Netherlands, (e-mail: j.w.kok@student.tue.nl, e.torta, m.a.reniers, j.m.v.d.mortel and m.j.g.v.d.molengraft @tue.nl).

Abstract: When mobile robots are employed in transportation tasks involving contact with humans, their control software shall guarantee that in every possible circumstance safety and, in general, task requirements are guaranteed. When control models are manually translated into an executable implementation, it becomes cumbersome to provide such guarantees. Model-driven engineering approaches provide an answer to such a problem. Domain specific models are automatically translated into an executable implementation. Some model-driven engineering approaches exist that are specific to robotics. However, formal guarantees on correctness of the model and the generated implementation with respect to the requirements are, often, not provided. This paper investigates how a general purpose modelling language for supervisory controller synthesis can be used to formally model plants and requirements for a robotic navigation task and can generate an executable implementation that can be integrated into a leading middleware for robotic applications. The starting point is the modelling of the interface provided by existing navigation components available in the targeted middleware. We demonstrate, with simulations and real-life experiments, that the generated supervisory controller is suitable for real-time deployment and guarantees correctness of the model with respect to the requirements of the navigation task at hand. Results on the reaction time of the supervisory controller show that such reaction time is about twenty times smaller than the one of the same supervisory controller implemented with a conventional framework.

Keywords: Supervisory control and automata, Autonomous robotic systems, Guidance navigation and control, Mobile robots, Modeling

1. INTRODUCTION

Autonomous robots are increasingly often present in applications related to mobility and transportation. Examples are delivery robots in Hoffmann and Prause (2018) or hospital logistic robots in Fragapane et al. (2020). Autonomous mobile robots are increasingly sophisticated which often translates into complex and complicated control software. When mobile robots are employed in transportation tasks in which contact with humans is possible, such as last mile delivery applications (e.g. Simoni et al. (2020); Boysen et al. (2018)), their control software shall guarantee that in every possible circumstance safety requirements, such as avoiding collisions with humans, and generic requirements, such as absence of livelocks or deadlocks, are satisfied. Traditional methods to create the discrete event controller for autonomous robots follow the well-known V-shaped model as elaborated in Mathur and Malik (2010). An initial model is created, the model is then manually translated into an implementation, the implementation is tested at component level and the integrated system is finally validated to assess if requirements are satisfied. If something is discovered to be wrong, the entire cycle starts again. Such procedure, including the manual translation of models into an implementation and late requirements testing, is error prone and time consuming (Kress-Gazit et al. (2018)). Solutions to avoid manual translations of models into implementation and late requirements validation are provided by model-driven engineering approaches. With model-driven engineering (see Arne et al. (2016) and Brugali (2015) for definitions), we refer to a method in which specifications are described with domain specific models. Such models provide concepts that are specific and targeted to a certain domain. An implementation can also be generated from them. In recent years, different domain specific languages to model robotic systems have been proposed (e.g. Adam et al. (2016); Brugali and Gherardi (2016); Han et al. (2015); Estivill-Castro and Hextel (2018)). Only a few of them (e.g. Estivill-Castro and Hextel (2018); Han et al. (2015)), however, provide approaches that can use formal methods to verify the controllers of their robots. None of them provide synthesis solutions. The full controller is developed in the specific framework and properties, such as absence of livelocks or deadlocks, are verified after the model is created. The work of Han et al. (2015) is based on two model-based engineering frameworks: Reactive Blocks of Kraemer et al. (2009) and BeSpaceD of Blech and Schmidt.

2405-8963 Copyright © 2021 The Authors. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0)
and applying the Dijkstra algorithm to it to compute the navigation tasks of a single robot only. In Dulce-Galindo et al. (2019) and Gonzalez et al. ad-hoc theory targets teams of cooperative robots (i.e. Liu et al. 2019). The navigation modules interface with the communication system to coordinate the navigation behavior of a swarm of robots that performs clustering and group formation. Performance is validated with experiments. The navigation modules are implemented ad-hoc for the experiment and their complexity is limited. Another application of supervisory control theory targets teams of cooperative robots (i.e. Liu et al. 2015). For further details, refer to Reijnen et al. (2007). Requirements can be represented as a UML 2.0 activity diagram that, similarly to a Petri net, describes logic as a token flow. Java code is automatically generated from the models. The work of Estivill-Castro and Hexel (2018) defines Logic-labeled Finite State Machines (LLFSM) to control the legs of an hexapod to achieve a clockwise motion. Model checking is done with NuSMV. Both papers only present results from simulated robots. Contrary to the present work, they do not investigate synthesis solutions in combination with existing software modules provided by leading robotic motion control libraries.

In this paper we present novel work aimed at investigating the possibility of synthesizing discrete event controllers for robotic complex navigation tasks which are correct with respect to the task’s requirements using model-driven engineering approaches. We synthesize such a discrete event controller to coordinate different states of the art navigation modules provided by ROS. The theoretical framework used for synthesizing the discrete event controller is supervisory control theory. The general purpose modelling language for supervisory controller synthesis CIF is chosen to model plants and requirements, synthesize the supervisory controller and generate an implementation (Wonham et al. 2014; van Beek et al. 2014)). This paper is organized as follows. Section 2 presents an overview of previous work on supervisory controller synthesis and model-driven engineering approaches for robotics. In Section 3, an informal overview of supervisory control theory concepts is provided. Section 4 reports on the models of plants and requirements specific to the navigation use-case presented in the paper. Section 5 presents the procedure followed to synthesize the controller including the description of the set-ups used to validate the controller performance. Sections 6 and 7 respectively analyze the results of the validation experiments and draw conclusions.

2. RELATED LITERATURE

Supervisory control theory has been successfully applied to define the discrete event controllers of a number of robotic applications. For example, Lopes et al. (2014) and Lopes et al. (2016) present supervisory controllers to coordinate the navigation behavior of a swarm of robots that performs clustering and group formation. Performance is validated with experiments. The navigation modules are implemented ad-hoc for the experiment and their complexity is limited. Another application of supervisory control theory targets teams of cooperative robots (i.e. Liu et al. 2015, Furci et al. 2013, Goryca and Hill 2013). Also in these works navigation modules are created ad-hoc for the experiments and validation is performed in simulation only. In Dulce-Galindo et al. (2019) and Gonzalez et al. (2018) modelling of navigation tasks of a single robot using supervisory control theory is addressed. In Gonzalez et al. (2018), an optimal path is obtained by modelling the environment in which a robot navigates as an automaton and applying the Dijkstra algorithm to it to compute the optimal path. The approach is validated by means of simulations in MobileSim. Also Dulce-Galindo et al. (2019) present the case of a single robot that is controlled by a discrete event supervisory controller that allows obstacle avoidance and navigation in a known environment. The work uses a deliberative planner (the reactive decision tree method from OMPL 4) to plan a global path and a reactive planner to avoid obstacles (Artificial Potential Fields also from OMPL). They generate implementation for the modelled supervisory controller using UltraDES (Alves et al. 2017). Validation is done by means of simulations in the ROS Stage simulation environment. This paper addresses a similar problem as in Dulce-Galindo et al. (2019). However, their work does not model existing navigation components but rather creates them ad-hoc for the specific problem and validates the approach with simulations only.

3. SUPERVISORY CONTROL THEORY

In this section, we briefly and informally present the relevant concepts of supervisory control theory on which this paper is based. For more in depth theoretical underpinning the reader is referred to Wonham et al. (2019). Supervisor synthesis is a generative technique. A supervisor is derived from a collection of plants and requirements. The plants describe capabilities of a cyber-physical system without any integrated control. The requirements model the functions a system is supposed to perform. They represent the behavior that is allowed in the controlled system. 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 and does not restrict the system any further than is required. A plant is modeled as a finite state automaton, which can be graphically represented as shown in Figure 1. For a formal definition of automaton the reader is referred to Skoldstam et al. (2007). Requirements can be modelled as finite state automata (Wonham et al. 2019) or as state-based expressions (Markovskii et al. 2010). Events can be controllable or uncontrollable. The synthesized supervisor guarantees the reachability of the marked locations (non-blocking), disables controllable events only (controllability) and is maximally permissive. When a supervisor acts as a supervisory controller, additional considerations apply. The following properties firstly formulated by Malik (2003) and Fabian and Hellgren (1998) and later extended by Reijnen et al. (2019) need to be guaranteed to ensure non-blocking of a supervisory controller:

1. For a formal definition of automaton the reader is referred to Wonham et al. (2019).
Finite Response: the controlled system is able to reach a stable state in a finite number of transitions. A stable state is a state where the system is waiting for an uncontrolled event to happen.

Confluence: there might be multiple events that are enabled from a given state and may be chosen by the controller. When the Confluence property holds, it is guaranteed that the system will always end up in the same stable state.

Non-blocking under control: reaching a marked state should not depend on the occurrence of an uncontrollable event.

In this paper, the chosen language to model plants and requirements, synthesize the supervisory controller and generate target agnostic code is CIF\textsuperscript{2}. CIF is an automata-based modeling language that supports the entire development cycle of supervisory controllers (van Beek et al. (2014)).

4. PLANT AND REQUIREMENTS MODELLING

4.1 Plants modelling

The system to be modelled is constituted by a Turtlebot3 Waffle Pi\textsuperscript{5} and related software modules that will be coordinated by the synthesized supervisory controller. The software components with their interfaces, which are modelled as plants, are constituted by state-of-the-art navigation modules, a human machine interface (HMI) module and a Laser Distance Safety (LDS) module that monitors whether the distance between the robot and any obstacle is less than a safety threshold. Navigation modules are provided by the ROS package move base flex\textsuperscript{6}, and present the following functionalities:

- Module GetPath: it computes a global path based on a metric map of the environment using Dijkstra’s algorithm.
- Module ExecPath: it is a reactive navigation component that receives localization data and sends forward and angular velocity commands to the robot.
- Module Recovery: it provides the action of turning the robot on the spot while scanning for obstacle-free space around it.

The navigation modules interface with the communication layer provided by the ROS infrastructure according to the specifications reported in the ROS documentation\textsuperscript{7}. Following modelling approaches of supervisory control theory, we created an abstract model of the interface specifications of the navigation modules as represented in Figure 2. In order to define relevant behaviors of the supervisory controller we need to keep track of the internal state of each navigation module. We do so by modelling an observer which is updated based on the events that the modules generate. The observer model is depicted in Figure 3.

The Human-Machine Interface (HMI) and the Laser Distance Safety module (LDS) are created specifically for the experiments presented in this paper. Their interface models are represented in Figures 4 and 5, respectively. Note that we refer to them as observers since they are used to keep a representation of the status of each subsystem based on the uncontrollable events that are observed.

4.2 Requirements modelling

Requirements represent what the supervisory controller may allow to happen. We model them as state-based expressions following a convention described by Markovski et al. (2010). The full set of requirements are provided in Kok (2020) and their implementation is available in a public code repository\textsuperscript{8}. We present their instances for a particular set of modules in Table 1. We want to control the system such that two modules controlling the same hardware are not active at the same time. In our case, such modules are ExecPath and Recovery (Req.1). It is possible to issue a cancel to a navigation module only when the HMI requests it or when the distance measured by the

---

\textsuperscript{5} https://www.robotis.us/turtlebot-3-waffle-pi/

\textsuperscript{6} http://wiki.ros.org/move_base_flex

\textsuperscript{7} https://docs.ros.org/en/diamondback/api/actionlib/html/classactionlib\_1\_ISimpleActionClient.html

\textsuperscript{8} https://gitlab.tue.nl/et_projects/jk-supervisory-control
LDS is unsafe (Req.2). A cancel command by the HMI or the LDS disables all navigation modules (Req3a-c). It is desired to have only one navigation module active at a time (Req.4). Following the interface model represented in Figure 2, a client can override the goal in the active state of a navigation module before issuing a cancel. In order to maintain data consistency, we want to impose that a navigation module is always in state Idle before receiving a new navigation goal (Req.5). Executing the goal of GetPath is possible when Recovery is successful. The other possibility is when the outcome of a navigation module is not available and a goal from the HMI is available. This is captured by Req.6. We allow GetPath to be active only when GetPath succeeded and there is no outcome observed for the other modules (Req.7). We allow Recovery to be active only when ExecPath failed and there is no outcome observed for the other modules (Req.8).

5. CONTROLLER SYNTHESIS

With the plant models and the requirements introduced in Subsections 4.1 and 4.2 respectively, it is possible to synthesize a supervisory controller that guarantees fulfillment of the requirements. The design of the supervisory controller follows the synthesis-based engineering method as presented in Korssen et al. (2017) which is fully supported by the chosen modelling language (i.e. CIF). The different plant models, introduced in Subsection 4.1, define all possible sequences of events that can be observed. The supervisory controller, obtained also considering the requirements introduced in Subsection 4.2, restricts the behaviour of the system once composed with the plant models such that undesired locations cannot be reached and desired behavior can be performed. The supervisory controller is generated using data-based synthesis provided by the CIF toolset. The resulting supervisor is maximally permissive and non-blocking. This means that it is possible to reach a marked state from every reachable state. However, as mentioned in Section 3, when a supervisor is used as controller, further properties need to be verified (Reijnen (2020)). Those are finite response, confluence and non-blocking under control. We have verified that the properties finite response and confluence hold for the supervisory controller by relying on the work presented in Reijnen et al. (2019). The full report of results is available in Kok (2020). We could not check, however, for non-blocking under control because the tooling does not support a check for this property. The procedure to check this property is described in Reijnen (2020). Implementing the procedure in the CIF toolset is left to future work.

5.1 Control System Architecture

From the synthesized supervisory controller, code is generated and deployed following the architecture reported in Figure 6. The supervisor receives uncontrollable events from the software modules through the ROS middleware, it updates its internal state, and selects the appropriate control action which is represented by a controllable event. All the details related to binding the generated code for the supervisory controller to the ROS middleware are reported in Kok (2020). The full system implementation is available in a code repository.

5.2 Validation set-ups

In order to validate the effectiveness of the synthesized supervisory controller we perform experiments in three different set-ups.

Set-up 1: simulation of uncontrollable events

Uncontrollable events are triggered manually. We use this scenario to validate the sequence of controllable events generated by the supervisor for the given scenarios.

Set-up 2: integration test in a simulated environment

The integrated system constituted by the supervisory controller interacting with the software modules and with a simulator of the robot is tested in the simulator Gazebo. This setup allows to replicate multiple times different scenarios with standard conditions and is used to evaluate execution time of the supervisory controller between receiving the event (u_unsafe) and issuing the controllable event (c_cancel). The execution time of the controller is compared with the execution time of the same supervisory controller implemented with a state of the art framework where state machines for autonomous robots are manually specified (i.e. SMACH from Bohren and Cousins (2010)).

Table 1. Instances of requirements for specific modules. The full set of requirements is available at Kok (2020). Their implementation is available in a public code repository.

<table>
<thead>
<tr>
<th>ID</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Req.1</td>
<td>ExecPath.c_goal needs Recovery.Idle</td>
</tr>
<tr>
<td>Req.2</td>
<td>Recovery.c_cancel needs obs.HMI.canceled ∨ obs.LDS.unsafe_distance</td>
</tr>
<tr>
<td>Req.3a</td>
<td>obs.HMI.canceled disables GetPath.c_goal</td>
</tr>
<tr>
<td>Req.3b</td>
<td>obs.HMI.canceled disables Recovery.c_goal</td>
</tr>
<tr>
<td>Req.3c</td>
<td>obs.HMI.canceled disables ExecPath.c_goal</td>
</tr>
<tr>
<td>Req.4</td>
<td>GetPath.c_goal needs ExecPath.Idle ∧ Recovery.Idle</td>
</tr>
<tr>
<td>Req.5</td>
<td>ExecPath.Active disables GetPath.c_goal</td>
</tr>
<tr>
<td>Req.6</td>
<td>GetPath.c_goal needs (obs.GetPath.no_outcome ∧ obs.ExecPath.no_outcome ∧ obs.Recovery.no_outcome ∧ obs.HMI.goal_available)</td>
</tr>
<tr>
<td>Req.7</td>
<td>Recovery.c_goal needs (obs.GetPath.succeed ∧ obs.ExecPath.no_outcome ∧ obs.Recovery.no_outcome)</td>
</tr>
<tr>
<td>Req.8</td>
<td>Recovery.c_goal needs (obs.ExecPath.failed ∧ obs.GetPath.no_outcome ∧ obs.Recovery.no_outcome)</td>
</tr>
</tbody>
</table>
Set-up 3: real-life testing in different scenarios

A simple yet effective scenario is defined: a robot operating in a corridor. Three situations can occur: (1) the corridor is obstacle free and the robot is able to reach its goal without re-planning (left in Figure 8) (2) an obstacle is partially occluding the corridor and the robot can plan a new trajectory around it (center in Figure 8), and (3) the obstacle fully occludes the corridor and the robot cannot reach its target goal (right in Figure 8).

6. RESULTS

6.1 Set-up 1: simulation of uncontrollable events

Figure 7 shows the sequence of events that the synthesized supervisory controller triggers after receiving a navigation goal ($u_{\text{receive\_goal}}$) from the HMI. At step 2, the supervisor activates the module GetPath. At step 3, GetPath succeeds and at step 4, the supervisor activates the module ExecPath. At step 5, the uncontrollable event ($u_{\text{fail}}$) is triggered. At step 6 the supervisory controller activates the module Recovery. At step 7, the uncontrollable event ($u_{\text{success}}$) is triggered, the supervisory controller activates the module ExecPath again (step 8), the uncontrollable event ($u_{\text{success}}$) is triggered for GetPath (step 9), the supervisory controller activates the ExecPath module (step 10) which finally returns the uncontrollable event ($u_{\text{succeed}}$) (step 11). Other validated relevant scenarios can be found in Kok (2020). In all of them, the controller behaves following the requirements as expected.

6.2 Set-up 2: integration test in simulated environment

In total, 50 experiments were performed in the simulation environment Gazebo in which obstacles were placed in front of the robot at each experiment. We measure the time between the moment the LDS module detects the presence of an obstacle issuing the uncontrollable event ($u_{\text{unsafe}}$) and the moment the supervisory controller issues the uncontrollable event ($c_{\text{cancel}}$) to the module ExecPath. We compare the reaction time of the synthesized supervisor with the reaction time of the same supervisor re-implemented in a state of the art framework to manually code state machines (i.e. SMACH). Table 2 summarizes the relevant measurements. As we can see, the reaction time of the synthesized controller is about 20 times smaller than the one of the manual implementation.

6.3 Set-up 3: real-life testing in different scenarios

We tested the synthesized controller also in real-life settings with the robotic platform Turtlebot3 in the three scenarios reported in Figure 8. Qualitative representation of the obtained trajectories in all three scenarios are reported in Figure 9. Videos of the system performance in the three scenarios are available in a public repository.

In all three conditions, the navigation behavior of the robot fulfills the requirements.

7. DISCUSSION AND CONCLUSION

This paper presents a method to model the provided interface of existing complex navigation modules following modelling approaches of supervisory control theory. We formulated the problem of creating the discrete event controller to coordinate the software modules as a supervisory controller synthesis problem. We showed that tooling (i.e. CIF) exists that can support the entire development cycle of supervisory controllers for robot navigation including the generation of the deployable controller’s implementation. We also show that such synthesis-based approach can guarantee conformance of the synthesized controller to the requirements’ specifications. We tested our approach

<table>
<thead>
<tr>
<th></th>
<th>min [ms]</th>
<th>max [ms]</th>
<th>mean [ms]</th>
<th>SD [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIF</td>
<td>1.586</td>
<td>8.391</td>
<td>2.442</td>
<td>1.275</td>
</tr>
<tr>
<td>SMACH</td>
<td>30.97</td>
<td>142.3</td>
<td>49.20</td>
<td>20.88</td>
</tr>
</tbody>
</table>

Table 2. Reaction time of the CIF-based controller vs reaction time of the SMACH controller.
with simulations as well as with a real hardware platform. All of the validation experiments show the conformance of the synthesized supervisor to the requirements as well as its feasibility in terms of real-time deployment. The comparison between the reaction time of the synthesized supervisory controller and its manual re-implementation in a state of the art framework (i.e. SMACH) show that the synthesized controller has a significantly faster reaction time for the scenario investigated. The present work calls for further exploration of the potential of model-based and synthesis-based methods for the specification of supervisory controllers for robotic systems. In this work, we looked at modelling approaches for existing and relatively complex navigation modules. We could extend the approach by modelling the coordination of lower elements of a navigation algorithm. An example problem of this sort could be the selection of the right constraints to be activated in a model predictive control approach to autonomous robot navigation. With respect to further uptake of synthesis-based approaches in robotics, we argue that usability aspects should be addressed as well. For the development of this research we still had to write manual code to bind the communication middleware to the synthesized supervisor. We argue that extending existing modelling languages (or creating new ones) to provide both supervisory controller synthesis with requirements guarantees as well as middleware specific binding code generation would encourage and speed up the adoption of the formal approaches presented in this paper.

ACKNOWLEDGEMENTS

This paper was made possible thanks to dr.ir. Albert Hofkamp that provided the interface to connect CIF generated code to the ROS middleware. The finite response and confluence properties of the supervisory controller were checked by dr.ir. Ferdie F.H. Reijnen.

REFERENCES


