Supervisory Control Synthesis for Autonomous Robots
A Study Using an Autonomous Service Robot

J. Smit, Prof. dr. ir. H.P.J. Bruyninckx,
dr.ir. M.J.G. van de Molengraft, dr.ir. J.M. van de Mortel - Fronczak

***Abstract*—As shown in literature, supervisory control synthesis could provide numerous benefits in system design, e.g. reducing time-to-market, and task execution, e.g. ensuring safety requirements are met. However, supervisory control synthesis is usually not used in a dynamic environment such as autonomous robotics. The purpose of this paper, therefore, is to explore the limitations, benefits and application scope of supervisory control synthesis in autonomous robotics. Using two case studies presented in this paper, we show that supervisory control synthesis and model based engineering provides us with a new methodology for designing and programming autonomous robots. This methodology reduces development time and manually made coding errors, through the use of modular en re-usable models of the robot and requirements, specifying its required behaviour. Supervisory control synthesis is suited for high-level decision making and task- and safety requirements specification, but less suited for low level control of the robot. Therefore, the case studies also explore the application scope of supervisory control synthesis for autonomous robots. The application scope is defined as the range of abstraction levels supervisory control synthesis can be applied to in autonomous robots.**

I. INTRODUCTION

Autonomous robots are a promising next step in robotics. They could help solve many of current major societal challenges. Examples include assistance at home, in care centres, but also in industrial environments [1]. However, the complicated dynamic environment these robots operate in and the complex actions they have to perform, make this a challenging field. Advancements in computing power and hardware have made autonomous robots increasingly capable of providing the complex tasks required to operate in these environments, but at the same time increasingly complex systems.
Since the introduction of the supervisory control theory of Ramadge and Wonham [2], the benefits of supervisory control synthesis have been discussed and proven in numerous papers. Benefits relating to system design, as shown e.g. in [3], and benefits relating to task execution.

These benefits could help manage the aforementioned increase in complexity in autonomous robotics.

Improvements in system design that supervisory control synthesis can offer are illustrated in Figs. 1 and 2 from [4]. The figures show the systems engineering design process from traditional to model-based with supervisory control theory included. The traditional SE design process and MBSE design process are represented in Fig. 1 for a system with a controller S and plant P. In traditional SE the requirements are tested on the realization of the design, without using

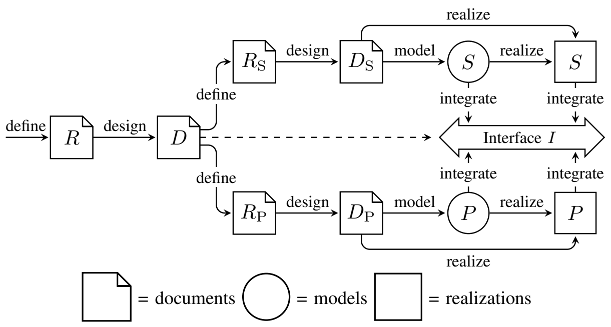

Fig. 1. Traditional and model-based systems engineering processes

models of the designs. This is represented in Fig. 1 with the *realize* arrow from design to realization. For complex systems, time-to-market and errors can be reduced by testing the design before realizing it. The process of Model Based Systems Engineering (MBSE) [5], also shown in Fig. 1, uses models of S and P for that reason, and, among others, [3] demonstrates that it indeed improves time-to-market and reduces errors. The next evolution of integrating models in the design process is using supervisory control synthesis of S from P and the models of the system requirements (Fig. 2). Where the traditional SE process relies heavily on documentation, from defining the requirements to realizing a design, the models used in MBSE provide a more systematic and unambiguous specification of S and P. This makes working on large multi-disciplinary projects faster and easier. Finally, using supervisory control synthesis on models of requirements guarantees that requirements are met, even in complex systems. This reduces the reliance on documentation and testing even more and prevents errors created by the implementation phase of the requirements.

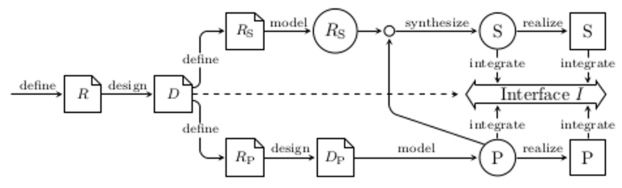

Fig. 2. Supervisory control synthesis-based systems engineering process

Besides the above mentioned benefits relating to system design, supervisory control synthesis offers benefits in task execution as well. Task execution benefits, in this paper, are a group of benefits relating to reductions in man-made errors when programming the robot. The following guarantees supervisory synthesis offers, would have to be ensured by proper implementation of logic throughout the code, when not using supervisory control synthesis. This approach, without using supervisory control synthesis, is sensitive to mistakes; incorrectly implementing the logic in just one place can cause unwanted behaviour. Also, when the system changes, the logic has to be changed everywhere. Supervisory control theory guarantees synthesis of a non-blocking, maximally permissive and controllable supervisor, as explained in Section II. These are useful when executing tasks, because it removes the ownership of these problems from the software developer to the supervisor. Another benefit provided by supervisory control synthesis, somewhat specific to this paper, is the more straightforward implementation of concurrent task execution. Traditionally, the software developer has to implement all the conditions regarding when a task can be executed, has to be interrupted or stopped, himself. With supervisory control synthesis, however, the supervisor will block all tasks (or events) that violate the requirements. Therefore, ensuring a task will never be executed untimely, as long as the requirements are specified correctly.

Despite the advantages of applying supervisory control theory to autonomous robots, there has been little research in the field. The research that has been done usually focusses on using supervisory control to govern the way robots interact with each other [6], using a human as supervisor [7] or an autonomous robot operating in a relatively static environment [1]. This is understandable, because supervisory control has three limitations that seem to make it unsuitable for autonomous robots:

- Supervisory control can only be applied to discrete-event systems.
- The synthesis of the supervisor is done off-line, so it cannot deal with the robot entering an unknown environment.
- Supervisory control theory assumes asynchronous and instantaneous task execution.

The purpose of this paper is to investigate these limitations, defining on what abstraction levels supervisory control synthesis can be applied, defined as the application scope, in autonomous robots and to explore the impact and benefits it can have. Using the Amigo robot at the Eindhoven University of Technology, two case studies are presented. These case studies showcase the benefits of using supervisory control synthesis and the application scope of the theory to autonomous robotics. A method is proposed that not only presents a new beneficial way to control autonomous robots, but also show what is and what is not (yet) possible.

The paper is structured as follows. Section II provides some necessary background on supervisory control theory and discrete-event systems. It is followed by an explanation of the case studies in Section III. The methodology and design choices are explained in Section IV. The results of applying the described method to these case studies are presented in Section V. Finally, Section VI discusses the limitations of using supervisory control synthesis for autonomous robots and possible research that can eliminate them.

II. SUPERVISORY CONTROL THEORY

Without going into too much detail, this section explains the topics of supervisory control synthesis and discrete-event systems. It contains the necessary background information to understand the research described in this paper.
Supervisory control theory applies to a subset of systems called Discrete Event Systems (DES). These systems are represented by discrete locations (states) and event-labelled transitions and can be modelled by automata. Additionally, in extended automata, also (discrete) variables can be used. An extended automaton allows to additionally augment the transitions with guards and updates.

An automaton G is formally described as follows:

$$G = (L, V, E, \rightarrow, L_m, l_0, \nu_0) \tag{1}$$

with:

L	a finite set of locations
V	a finite set of discrete variables
E	a finite set of events
$\rightarrow$	$\subseteq L \times G(V) \times E \times U(V) \times L$ the transition relation
L_m	the set of marked locations, $L_m \subseteq L$
l_0	the initial location, $l_0 \in L$
ν_0	the initial valuation of the variables, $\nu_0 : V \rightarrow \lambda$ and λ is the set of all values

An example automaton is graphically depicted in Fig. 3.

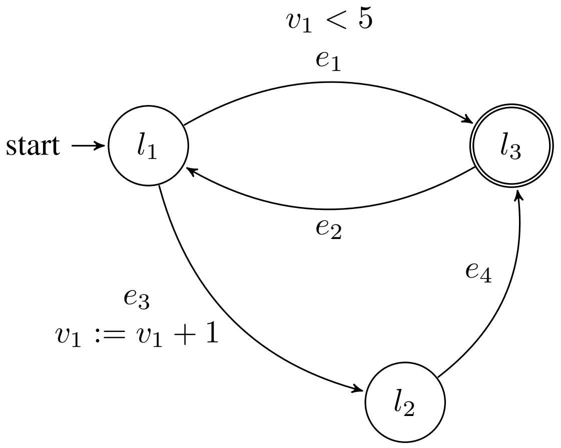

Fig. 3. Graphic depiction of an automaton

with:

l_1, l_2, l_3	the locations
v_1	the discrete variable
e_1, e_2, e_3, e_4	the events
l_3	the marked location
l_1	the initial location
$v_1 = 0$	the initial values of the variables
$v_1 := v_1 + 1$	the update of v_1 done by event e_3
$v_1 < 5$	the guard for event e_1

A model of a DES usually consists of multiple automata, sharing variables and events, called a network of automata.

In supervisory control theory, an uncontrolled system that is modelled by these automata is called a plant. A distinction is made between controllable events, which can be controlled by the supervisor, and uncontrollable events, which cannot be controlled. A button that needs to be pressed by a user, for example, is an uncontrollable event. When combined with a set of requirements, supervisory control theory provides a way to automatically synthesize a supervisor. These requirements can, for example, be a set of safety requirements or process requirements. Safety requirements prevent dangerous situations from occurring, e.g. the robot colliding with its environment or itself. Process requirements ensure a certain order of events, e.g. not releasing an object before reaching the table on which the object is supposed to be placed. The requirements can be modelled in the same way as the plant, with an automaton, and can refer to the plant variables and locations. It can also be modelled as a state-based expression, a requirement specified in terms of conditions of a state. Where a guard in an automaton based requirement would block an event from happening based on a condition of a variable, a state based expression prescribes in what state the system cannot be in, or what value a variable cannot have. While the same result can be obtained with an automaton based requirement, a state-based expression can offer a more intuitive way to describe requirements. The synthesized supervisor then disables events that would lead to a violation of these requirements.

There are a few important benefits, applicable to this research, that follow from the following.

Separation The requirements and plant can be described separately. This separation means adjustments to requirements or plant only have to be applied locally. Synthesis will take care of the knock-on effects of these changes. In conventional programming changing a simple requirement would not only involve tedious work in many different places in the code, the software developer also has to think of all the implications of the change himself and apply them correctly. This separation will make developing easier and faster.

Modular design The method proposed in Section IV uses separate modules for groups of requirements, e.g. skill requirements and task requirements. An example of a skill requirement used in this paper, is the grab skill, described in Section IV. Only when needed in a certain task this skill can be included, thus creating a modular system.

Re-usability The models of the requirements and the robot can be applied to new tasks, or even new robots, without many adjustments. Because the robot model contains general components used in robots, e.g. arms, the robot model can mostly stay the same when using a different robot. Only the intermittent communication layer between the supervisor and the robot, is robot specific.

As mentioned before, supervisory control theory does not only provide benefits related to system design (Section I), but also related to task execution. The synthesized supervisor has two useful properties related to task execution: non-blockingness and the guarantee that all events that would lead to a violation of the requirements are blocked. A non-blocking supervisor blocks the execution of every event that would lead to a system state from which no marked state L_m can be reached.

These properties are useful, because without the use of supervisory control synthesis, the software developer would have to create these properties with logic in the code. Removing the ownership of this issue to the supervisor can prevent unwanted behaviour when changes have to be made to related parts of the system. The developer could forget to change the logic in all the related parts and the system can reach a blocking state or violate the requirements.

Supervisory control synthesis also guarantees a maximally permissive and controllable supervisor. Maximal permissiveness assures that only those events that could lead to a violation of the requirements or the non-blockingness principle, are blocked by the supervisor. Controllability refers to the fact that the supervisor only blocks controllable events.

When synthesizing one supervisor for large systems with many components, an effect occurs called state-space explosion. This kind of supervisor is called a monolithic supervisor and since it has to account for all possible state combinations of the components and block according to all the requirements, it becomes very large and takes a long time to synthesize. There are different synthesis techniques that can circumvent this problem, i.e. modular [8] or hierarchical [9] synthesis and although they were not necessary for the case studies in this research, they could prove useful when increasing the scope and complexity of the supervisor.

III. CASE STUDIES

Section I states the main contribution and purpose of this paper: to investigate the limitations, benefits and application scope of supervisory control synthesis for autonomous robots. To this end, two case studies were designed to showcase what supervisory control can contribute in the field of autonomous robotics.

As discussed in Section I, the benefits of supervisory control synthesis can be classified into two main categories: system design and task execution. Further on in Section II, these categories were expanded on. For systems design, aspects related to separation, modularity and re-usability were discussed. The properties of the synthesized supervisor, non-blockingness and no violation of the requirements, are benefits in the task execution category. The case studies presented here show the extent to which these benefits and principles can be applied to autonomous robots, using the method described in Section IV.

The robot used for the case studies, is the Amigo robot, see [10]–[12]. This is an autonomous service robot developed and maintained at the Eindhoven University of Technology in the Control Systems Technology Group. It competes in the RoboCup@Home league and has a functional system architecture and code, which can be found in the `tue_robotics` repository [13]. This robot was chosen for the case studies, because the low-level control design was already in place and from there, it was possible to use supervisory control synthesis to control the robot with high-level commands. Also, the robot is controlled at task and skill level by a finite-state machine implementation in ROS using the SMACH library [14], which was thought to make interaction with the synthesized supervisor more straightforward. The robot model, its requirements and the synthesized supervisor are created with the CIF 3 tool set [15]. CIF 3 is also developed and maintained at the Control Systems Technology Group. CIF 3 can generate a C library [16] implementation of the supervisor and this is leveraged to fuse the existing SMACH finite-state machines with the supervisor. When the robot is being used without supervisor, a task is described by a SMACH finite state-machine. In the case studies presented here, this task finite state-machine is replaced by a communication layer between the supervisor and the robot system architecture, called a wrapper. Figure 4 depicts the system architecture used for the case studies. Section IV provides a more detailed description of this architecture.

A. *CASE STUDY I: PICK AND PLACE*

In this case study, Amigo navigates to a table with an object on it. It then grabs the object, navigates to a different table and puts the object down. Figure 5 provides a, somewhat simplified, flowchart representation of the case study. The supervisor here dictates when it is safe to execute the actions,

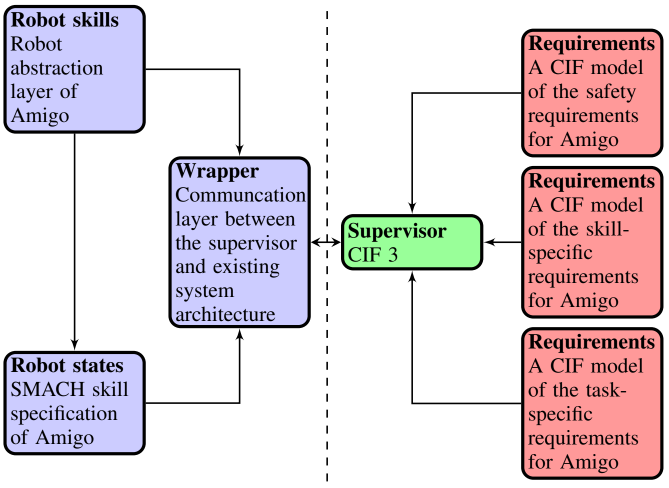
Robot skills
Robot abstraction layer of Amigo
Wrapper
Communcation layer between the supervisor and existing system architecture
Robot states
SMACH skill specification of Amigo
Supervisor
CIF 3
Requirements
A CIF model of the safety requirements for Amigo
Requirements
A CIF model of the skill-specific requirements for Amigo
Requirements
A CIF model of the task-specific requirements for Amigo

Fig. 4. System architecture

move the arm, grab and navigate.

The system design benefits supervisory control synthesis provides in this case study, can be found in the way the code is structured. As can be seen in Fig. 4, a clear distinction is made between safety, skill and task requirements. Apart from the model of the robot, there are 3 separate models of requirements, each with their own purpose:

Safety requirements This set of requirements is always necessary for the robot to function correctly and needs to be included in the supervisory control synthesis for every task; e.g. collision prevention. See Fig. 9 in Appendix A and Listing 2 in Appendix B.

Skill requirements This set of requirements is specific to a certain skill of the robot and needs to be included in the supervisor control synthesis of a task that uses this skill; e.g. the grab skill. See Listing 3 in Appendix B.

Task requirements This set of requirements is specific to a certain task and defines the (order of) skills that need to be executed during a task. It needs to be designed for every task and placed on top of the safety and skill requirements; e.g. the pick-up task. See Fig. 10 in Appendix A and Listing 1 in Appendix B.

This separation makes the system very robust to changes. When, in the future, a new safety requirement needs to be implemented, the model of safety requirements is the only thing that needs to be changed. The newly synthesized supervisor will handle the rest, where without the use of supervisory control synthesis changes need to be made throughout the code.

One of the task execution benefits supervisory control synthesis provides in this case study, is that the individual skills are executed concurrently where possible. As mentioned in Section II, supervisory control synthesis can remove ownership of certain problems from the software developer to the synthesized supervisor. In this case study, this means that the software developer only has to define the order of the skills needed for the case study in a CIF task requirement automaton.This requirement automaton describes the case study skill order as shown in Fig. 5. The supervisor then decides, based on these task requirements, the skill order. It handles all skill scheduling and allows skills to execute concurrently where the requirements allow it.

Furthermore, this case study shows how the supervisor can handle unexpected situations. When a collision is found on the path, the arm stops moving and re-plans the path to the object. In Section IV, a more in-depth explanation is given of the way the supervisor handles this.

B. *CASE STUDY II: DYNAMIC GRAB*

This second case study is designed to show the *modularity* and *re-usability* of the models used for supervisory control synthesis. Applying most of the same models, a new task with the same properties (i.e. no collision) is designed with minimal effort.

In this case study, the robot starts in front of a table with two objects placed on it. The robot then picks up both objects. In SMACH it is only specified that the robot has to pick up both objects, the grab-skill-requirement guarantees that a supervisor is synthesized that decides what arm is used to grab which object. Fig. 6 shows the flowchart of the case study.

The safety requirements still apply, so collisions are still avoided (Fig. 9), but the task requirements of Section III-A are removed.

IV. METHOD

This section explains how supervisory control synthesis is used to perform the case studies described in Section III. First the existing architecture to control the robot is explained. This control, already in place, is used as the basis on which the supervisory control, presented in this paper, is added. The system architecture for this supervisory control implementation is discussed next. The design choices made to achieve a working supervisor implementation on the robot, that provides the benefits laid out throughout the paper, are discussed as well.

As shown in Fig. 4, the system architecture used in the case studies can be divided in two parts: existing robot control implementation and the supervisory control implementation. Since this research focuses on the supervisory control of autonomous robots in general, the existing robot control implementation is only explained where it interacts with the supervisor.

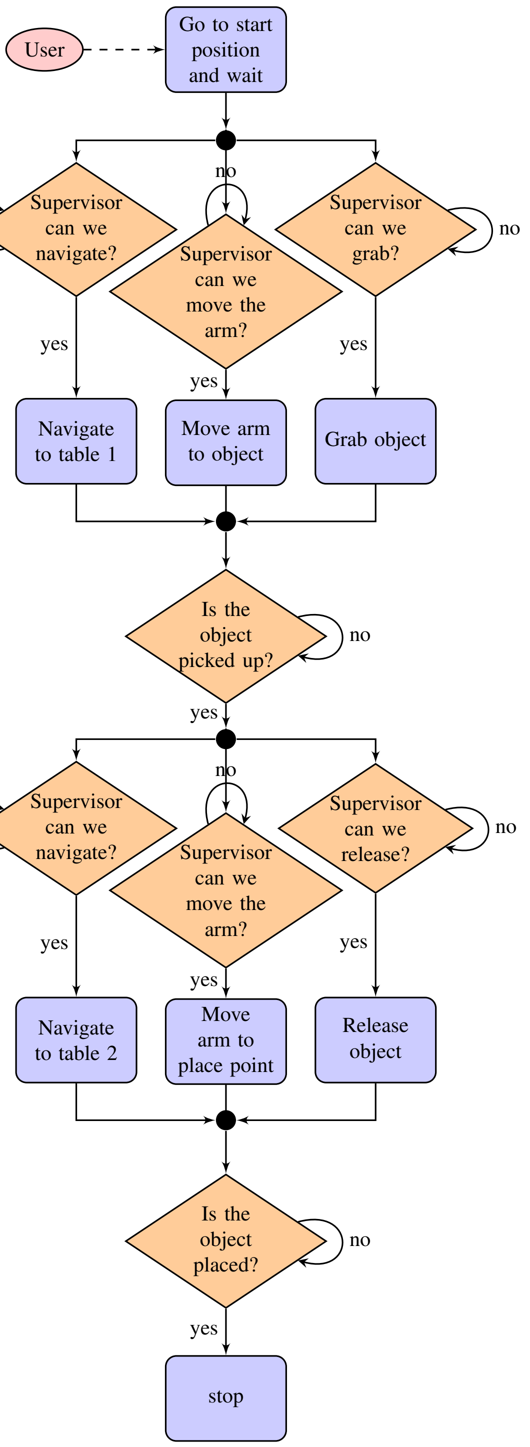

Fig. 5. Decision flowchart of Case Study I

The low-level control of the robot is written in `C++` and leveraged by a set of `robot_skills` written in `Python`. The `robot_skills` *"provide interfaces to all parts of a robot: its base, arms, head, perception, worldmodel, speech system, etc."* [13]. The robot abstraction layer provided by `robot_skills` is used by `robot_states`, which in turn uses SMACH finite-state machines to describe skills

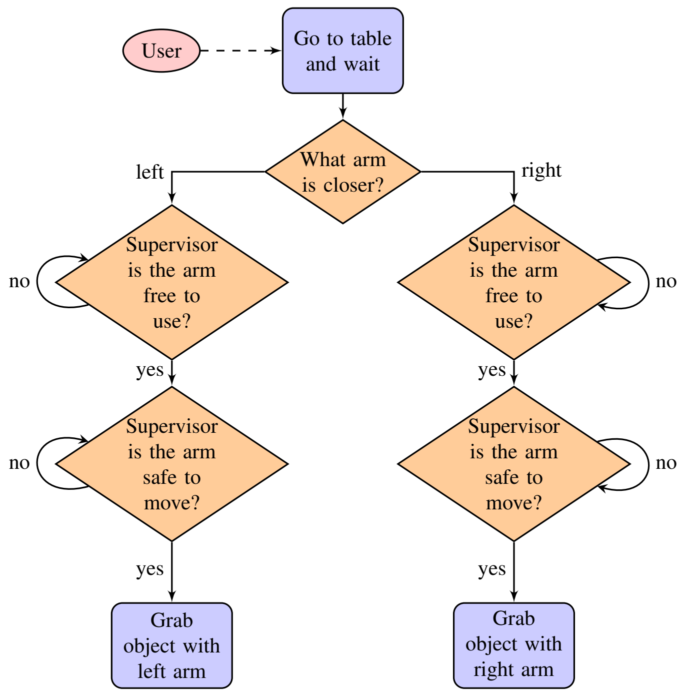
User
Go to table and wait
What arm is closer?
left
right
Supervisor is the arm free to use?
no
yes
Supervisor is the arm free to use?
no
yes
Supervisor is the arm safe to move?
no
yes
Supervisor is the arm safe to move?
no
yes
Grab object with left arm
Grab object with right arm

Fig. 6. Decision flowchart of Case Study II

regarding navigation, manipulation and so forth. Finally, the skills in `robot_states` are used in a SMACH-based task description, called a challenge. An example of this interaction is shown in Fig. 7, taken from the `tue_robotics` repository. The code in the `tue_robotics` repository [13] contains further information about the system architecture of the existing robot control implementation.

The model of the robot and requirements are created in CIF 3 [15] and based on them a supervisor was synthesized. The CIF 3 language is designed to efficiently describe a plant, in this case Amigo, and its requirements, with automata or state-based expressions. An example plant automaton is shown in Listing 4 of Appendix B. One recognizes the locations, including the initial and marked keywords, transitions represented by edges and declarations of (un-)controllable events as mentioned in Section II. This CIF automaton is shown graphically in Fig. 8. An example of a requirement automaton in CIF and its graphical representation are shown in, respectively, Listing 2 of Appendix B and Fig. 9. This safety requirement prevents collisions of the arms. When a collision is detected on the path planned by the robot, the supervisor cancels the current plan and plans a new one. A more detailed explanation of these models can be found in [17]. For this paper a set of models and requirements was created, that contained all the necessary components to execute the case studies, this can be expanded on in later research.

CIF uses the algorithm [18] to synthesize a maximal permissive, controllable and non-blocking supervisor. The output of the CIF synthesis procedure is a supervisor described in the CIF modelling language, but the CIF tool set provides conversions to other languages as well [19]. To integrate this supervisor with the robot and the existing control implementation, the supervisor was converted to `C` code, utilizing the tool from [19]. The CIF converted `C` code is available to the user as a library, which is used in this paper as follows: the CIF generated supervisor library exposes two entry point functions to the user: `void EngineFirstStep()` and `void EngineTimeStep()`. `void EngineFirstStep()` initializes all the data and executes the events that are not blocked by the supervisor, before the first time step. `void EngineTimeStep(double delta)`, which needs to be called regularly by the user, updates time-dependent equations and variables in the model with time `delta` and then executes the events that are not blocked by the supervisor. Guards for these events can depend on variables that can be set by the user, by calling a function with the name of the variable. Looking at the example in Figs. 8 and 9, the variables that can be set from the calling program are: `collision_on_path` (cp), `path_valid` (pv) and `goal_reached` (gr). The function names would be, respectively, `void collision_on_path()`, `void path_valid()` and `void goal_reached()`.
The supervisor needs to interact with the robot, so the robot actually executes the actions imposed by the events of the supervisor. To this end, the CIF generated supervisor library exposes a callback function `void InfoEvent(Event_ event, BoolType pre)`. This callback function needs to be defined in the program using the library and is called every time an event is about to, or has been executed. In the case studies described in this paper, the function is defined in python and will execute the `robot_states` or `robot_skills` relevant to the event. This function definition and other communication functions between the supervisor and the existing robot control are implemented in a file called the python wrapper. It is important to note that to ensure that the supervisor knows if the robot is actually executing, or has completed, the task requested by the supervisor, variables need to be included so the robot can report back its current state. CIF provides the `input` variable type, to facilitate this. These are variables that the robot needs to assign to, after every time step, using the function `AssignInputVariables`

The complete code containing the CIF models, the supervisor and the python wrapper of the case studies, and a detailed plan on how to compile them, can be found in the code repository [17] of the project described in this paper. The repository also contains a detailed usage description of the CIF models, the supervisor, the python wrapper and the SMACH implementation, so the method can easily be integrated in further research.

Concerning the application scope of supervisory control in

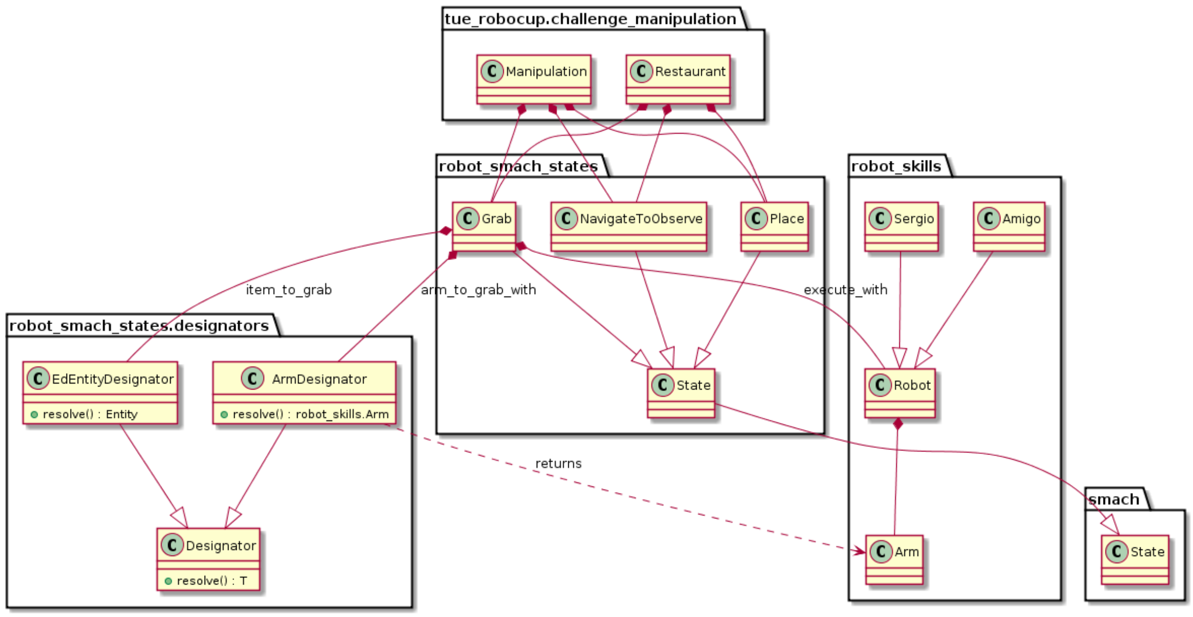

Fig. 7. Example interaction robot layers

the case studies, the supervisor can be applied to different abstraction levels. Applying supervisory control to the low-level control of the robot is difficult, because the control is mainly continuous and not discrete-event based. The lowest level of abstraction supervisory control was applied to in this research was on the skill level, using the skill requirements. They control one component of the robot, but only the discrete events of that component. This abstraction level was chosen, because the skills can still be described by finite-state machines and are discrete-event based. One level of abstraction higher are the safety requirements. They can span multiple components and block events from one component based on information from another. The highest abstraction level supervisory control was applied to in this research was on the task level, using the task requirements. They only control the order of skill events the robot executes and control nothing of the skills themselves.

A. LIMITATIONS

The presented case studies and method show the benefits that supervisory control synthesis can offer to autonomous robots. There are, however, properties or benefits supervisory control synthesis can offer, that could not be translated to the dynamic environment of autonomous robots.

Non-blockingness, for example, can be used to ensure that a system is able to reach a certain state. In robotics this could mean that a robot would never execute an action that would block a path, the robot later needs to navigate. Another example where the non-blockingness property could prove useful, is a low-level implementation of supervisory control synthesis for control of the arms. One can imagine it could be beneficial to make sure the arms never block each other when reaching for an object. Both of these examples are impossible to implement with the method described in this paper, because of the dynamic environment the robot operates in. This is caused by the fact that the supervisor is synthesised off-line and therefore cannot predict what happens when the robot enters an unknown environment. There exist on-line supervisory control synthesis techniques [20]–[22], that could prove useful in future research.

Another limitation is that CIF is a clear and concise language, but also somewhat limited. Strings are not allowed as variables when performing supervisory control synthesis, for example, which is a big limitation when trying to describe a list of items to be picked up.

V. RESULTS

The case studies presented in Section III, using the method presented in Section IV, were implemented on the Amigo robot. Due to the system architecture already in place, the case studies could only be partly implemented. It was thought that the existing SMACH finite-state machines of the robot system architecture, would provide an easy platform on which to integrate the supervisor. However, the synchronous nature of these finite-state machines and their complexity, made it difficult to integrate them with the asynchronous callback structure of the supervisor. An example of this is the grab skill SMACH finite-state machine in the Amigo repository. The state machine navigates the robot in front of the specified object, updates the world model, opens and closes the grabber and more. During the execution of this state machine the supervisor cannot call other events, but without all these steps the object will not be grabbed correctly. These are just limitations on the robot side, however, and not limitations to the applicability and benefits of supervisory control synthesis to autonomous robot. Furthermore, future research can solve these issues by, for example, making the callbacks multi-threaded.

Because of these limitations case study II could not be implemented on the robot, because the current architecture only allows planning of one arm at the same time. Case study I was implemented for the most part.

More information on what is implemented and what is not, is provided in this paper's repository [17]. This implementation confirmed the following benefits the method in this paper discusses:

Concurrency In case study I the robot moves one of the arms and navigates to a table at the same time.

Modularity and Re-usability Both case studies combine a robot model, safety requirements, skill requirements and a task requirement from which a supervisor is synthesized. When the task requirement describing case study I is replaced with the task requirement describing case study II, synthesis provides a supervisor that executes this new task.

And to a lesser extent:

Separation Because the safety requirements were the same for both case studies, the effect of changing a requirement was not shown. Because of the modular design, however, this property should still hold.

VI. Concluding Remarks

In this paper, the limitations, the impact and benefits supervisory control synthesis can have in autonomous robots, are explored. To this end, two case studies are described, using the Amigo robot at the Eindhoven University of Technology. A method is presented that can be used as a basis for implementation of supervisory control synthesis in autonomous robots. This method defines on what abstraction levels supervisory control synthesis can be applied in autonomous robots and presents a framework for future research in the field. The two case studies show that applying this method to autonomous robots indeed provides benefits in both system design and task execution.
Further research can increase the impact of supervisory control synthesis, by removing limitations mentioned in this paper. On-line synthesis of the supervisor [20]–[22] increases its capability of reacting to changes in the dynamic environment of the robot. Furthermore, using a non-monolithic supervisor is useful when the complexity and size of the system increases e.g. modular [8] or hierarchical [9] synthesis. Further research can also expand the implementaton of the case studies described in this paper, to provide further insight into the application of supervisory control synthesis to autonomous robots.

Acknowledgement

The authors would like to thank Albert Hofkamp, as he has been an enormous help in understanding the CIF tool set and providing a way to convert a CIF specification into a working supervisor implementation in C.

References

[1] A. G. C. Gonzalez, M. V. S. Alves, G. S. Viana, L. K. Carvalho, and J. C. Basilio, "Supervisory control-based navigation architecture: A new framework for autonomous robots in industry 4.0 environments," *IEEE Transactions on Industrial Informatics*, vol. PP, no. 99, pp. 1–1, 2017, ISSN: 1551-3203. DOI: 10.1109/TII.2017.2788079.

[2] P. J. Ramadge and W. M. Wonham, "Supervisory control of a class of discrete event processes," *SIAM Journal of Control and Optimization*, vol. 25, no. 1, pp. 206–230, 1987.

[3] S. T. J. Forschelen, J. M. van de Mortel-Fronczak, R. Su, and J. E. Rooda, "Application of supervisory control theory to theme park vehicles," *Discrete Event Dyn Syst*, 2012.

[4] J. C. M. Baeten, J. M. van de Mortel-Fronczak, and J. E. Rooda, "Integration of supervisory control synthesis in model-based systems engineering," in *Complex Systems: Relationships between Control, Communications and Computing*, G. M. Dimirovski, Ed. Cham: Springer International Publishing, 2016, pp. 39–58.

[5] J. A. Estefan *et al.*, "Survey of model-based systems engineering (mbse) methodologies," *Incose MBSE Focus Group*, vol. 25, no. 8, pp. 1–12, 2007.

[6] M. Furci, A. Paoli, and R. Naldi, "A supervisory control strategy for robot-assisted search and rescue in hostile environments," in *2013 IEEE 18th Conference on Emerging Technologies Factory Automation (ETFA)*, Sep. 2013, pp. 1–4. DOI: 10.1109/ETFA.2013.6648162.

[7] X. Zhang, Y. Zhu, and H. Lin, "Performance guaranteed human-robot collaboration through correct-by-design," in *2016 American Control Conference (ACC)*, Jul. 2016, pp. 6183–6188. DOI: 10.1109/ACC.2016.7526641.

[8] M. H. de Queiroz and J. E. R. Cury, "Modular control of composed systems," in *Proceedings of the 2000 American Control Conference. ACC (IEEE Cat. No.00CH36334)*, vol. 6, 2000, 4051–4055 vol.6. DOI: 10.1109/ACC.2000.876983.

[9] K. C. Wong and W. M. Wonham, “Hierarchical control of discrete-event systems,” *Discrete Event Dynamic Systems*, vol. 6, no. 3, pp. 241–273, Jul. 1996, ISSN: 1573-7594. DOI: 10.1007/BF01797154. [Online]. Available: https://doi.org/10.1007/BF01797154.

[10] (Mar. 2018). Amigo robot description on ros website, [Online]. Available: http://wiki.ros.org/Robots/AMIGO.

[11] (Mar. 2018). Amigo robot description on robotic open platform website, [Online]. Available: http://roboticopenplatform.org/wiki/AMIGO.

[12] J. Lunenburg, T. Clephas, N. Dirkx, B. Willems, J. Elfring, J. Sandee, and M. van de Molengraft, “Tech united eindhoven team description 2011,” 2011.

[13] TU/e. (Mar. 2018). Github repository tu/e robotics, [Online]. Available: https://github.com/tue-robotics.

[14] (Mar. 2018). Smach wiki on ros.org, [Online]. Available: http://wiki.ros.org/smach.

[15] D. van Beek, W. Fokkink, D. Hendriks, A. Hofkamp, J. Markovski, J. van de Mortel-Fronczak, M. Reniers, E. Abraham, and K. Havelund, “Cif 3: Model-based engineering of supervisory controller,” *TACAS*, 2014.

[16] (Mar. 2018). Cif 3 language tutorial, [Online]. Available: http://cif.se.wtb.tue.nl/lang/tut/index.html.

[17] J. Smit. (Mar. 2018). Github repository of this paper, [Online]. Available: https://github.com/amigo-supervisor.

[18] L. Ouedraogo, R. Kumar, R. Malik, and K. Åkesson, “Nonblocking and safe control of discrete-event systems modeled as extended finite automata,” *IEEE Transactions on Automation Science and Engineering*, vol. 8, pp. 560–569, 2011.

[19] (Mar. 2018). Cif 3 c99 conversion manual, [Online]. Available: http://cif.se.wtb.tue.nl/tools/codegen/c99.html.

[20] J. H. Prosser, M. Kam, and H. G. Kwatny, “Online supervisor synthesis for partially observed discrete-event systems,” *IEEE transactions on automatic control*, vol. 43, 1988.

[21] L. Grigorov and K. Rudie, “Near-optimal online control of dynamic discrete-event systems,” *Discrete Event Dynamic Systems*, vol. 16, 2006.

[22] R. Zhang and K. Cai, “Online computation of supremal relatively observable sublanguage of discrete-event systems,” *Chinese Control Conference*, vol. 34, 2015.

Appendix A
CIF Automata: Graphical

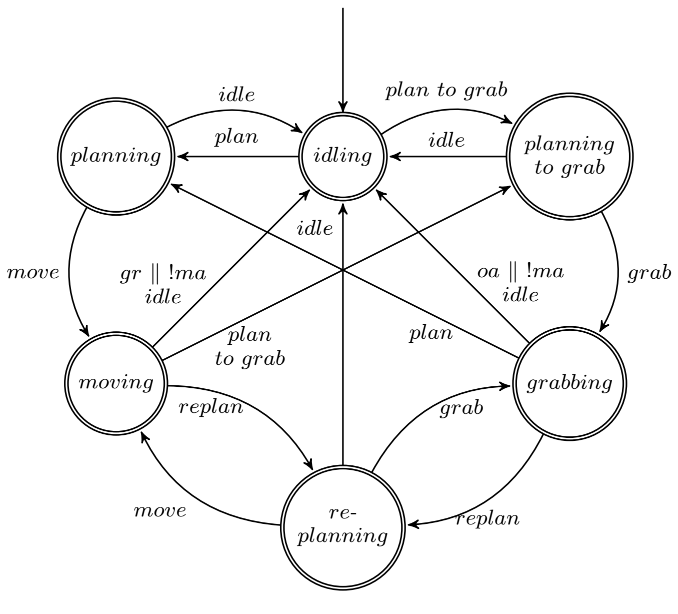

Fig. 8. Automaton of left arm

with:

gr	goal reached
ma	moving arm
oa	object in arm

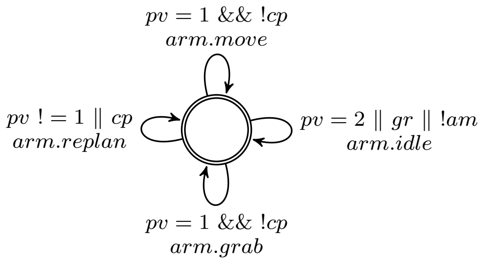

Fig. 9. Automaton of safety requirement left arm

with:

pv	plan viable
cp	collision on path
gr	goal reached
$!am$	arm.moving-arm = false, while arm is in location moving

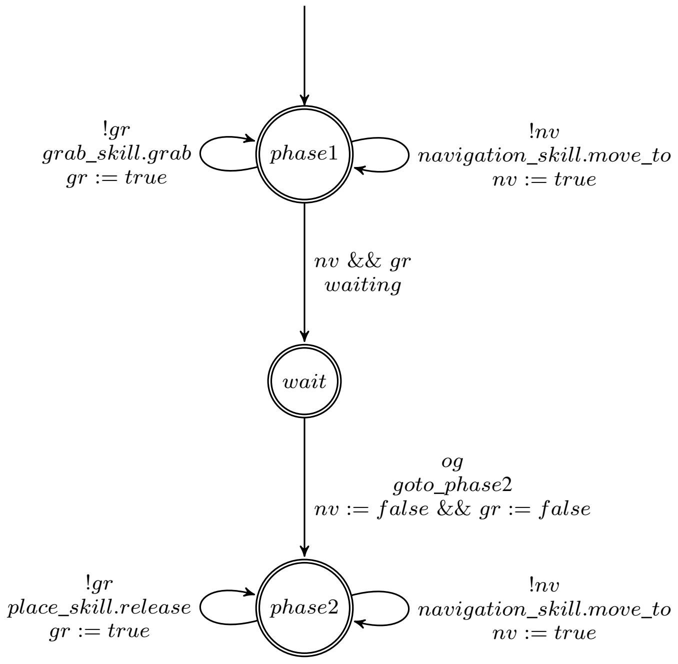

Fig. 10. Automaton of pick-up task

with:

nv	navigating
gr	grabbing
og	object grabbed

Appendix B
CIF Automata: Code

```
requirement task:
   controllable waiting, goto_phase2;

    input bool object_grabbed;
    disc bool navigating = false;
    disc bool grabbing = false;

    location phase1:
    initial;
    marked;
    edge navigation_skill.move_to when navigating = false do navigating := true;
    edge grab_skill.grab when grabbing = false do grabbing := true;
    edge waiting when navigating = true and grabbing = true do navigating := false and grabbing := false goto wait;

    location wait:
    marked;
    edge goto_phase2 when object_grabbed = true goto phase2;

    location phase2:
    marked;
    edge navigation_skill.move_to when navigating = false do navigating := true;
    edge place_skill.release when grabbing = false do grabbing := true;
end
```

Listing 1. CIF model of pick-up task requirement

```
requirement def SafeArmMovement(Arm arm):
    location:
    initial;
    marked;
    edge arm.replan when arm.collision_on_path or arm.plan_viable != 1;
    edge arm.move when arm.plan_viable = 1 and not arm.collision_on_path;
    edge arm.grab when arm.plan_viable = 1 and not arm.collision_on_path;
    edge arm.idle when arm.plan_viable = 2 or arm.goal_reached or arm.moving_arm != true; // Set plan_viable to 0
end

requirement_arm_left: SafeArmMovement(arm_left);
requirement_arm_right: SafeArmMovement(arm_right);
```

Listing 2. CIF model of safety requirement arms

```
requirement grab_skill:
    controllable grab, determine_grab_arm;

    input int [0..2] use_arm; //0 nothing, 1 left, 2 right

    location idling:
    initial;
    marked;
    edge grab goto grabbing;

    location grabbing:
    marked;
    edge determine_grab_arm when use_arm = 0;
    edge left_grab_exec.start when use_arm = 1 goto idling;
    edge right_grab_exec.start when use_arm = 2 goto idling;
end

requirement def grab_exec(Arm arm):
    controllable check_in_WS, start, complete, idle;

    input int [0..2] object_in_WS;

    location idling:
    initial;
    marked;
    edge start goto starting;

    location starting:
    marked;
    edge check_in_WS when object_in_WS = 0;
    edge arm.plan when object_in_WS = 1;
    edge arm.plan_to_grab when object_in_WS = 2;
    edge complete when arm.object_in_arm goto completing;

    location completing:
    marked;
    edge idle goto idling;

end

left_grab_exec: grab_exec(arm_left);
right_grab_exec: grab_exec(arm_right);
```

Listing 3. CIF model of grab skill requirement

```
plant automaton def Arm():
    // Events
    // Stop all movement, move arm to plan, plan to waypoint
    controllable idle, move, plan, replan, plan_to_grab, grab;

    // Variables
    input int [0..2] plan_viable; //{0,1,2} = {empty,true,false}
    input bool collision_on_path;
    input bool moving_arm;
    input bool goal_reached;
    input bool object_in_arm;

    location idling:
    marked;
    initial;
    edge plan goto planning;
    edge plan_to_grab goto planning_to_grab;

    location planning:
    // entry point to start moving to something
    marked;
    edge move goto moving;
    edge idle goto idling;

    location planning_to_grab:
    marked;
    edge grab goto grabbing;
    edge idle goto idling;

    location replanning:
    marked;
    edge move goto moving;
    edge grab goto grabbing;
    edge idle goto idling;

    location moving:
    marked;
    edge plan_to_grab goto planning_to_grab;
    edge replan goto replanning;
    edge idle when (goal_reached or moving_arm != true) goto idling;

    location grabbing:
    marked;
    edge plan goto planning;
    edge replan goto replanning;
    edge idle when (object_in_arm or moving_arm != true) goto idling;

end

arm_left: Arm();
arm_right: Arm();
```

Listing 4. CIF model of Amigo Arms