

Compositional Design of Multi-Robot Systems Control Software on ROS

STEFANO SPELLINI, University of Verona

MICHELE LORA, Singapore University of Technology and Design

FRANCO FUMMI, University of Verona

SUDIPTA CHATTOPADHYAY, Singapore University of Technology and Design

This paper presents a methodology that relies on Assume-Guarantee Contracts to decompose the problem of synthesizing control software for a multi-robot system. Initially, each contract describes either a component (e.g., a robot) or an aspect of the system. Then, the design problem is decomposed into different synthesis and verification sub-problems, allowing to tackle the complexity involved in the design process. The design problem is then recomposed by exploiting the rigorousness provided by contracts. This allows us to achieve system-level simulation capable to be used for validating the entire design. Once validated, the software synthesized during the process can be integrated into Robot Operating System (ROS) nodes and executed using state-of-the-practice packages and tools for modern robotic systems.

We apply the methodology to generate a control strategy for an autonomous goods transportation system. Our results show a massive reduction of the time required to obtain automatically the control software implementing a multi-robot mission.

1 INTRODUCTION

The recent technological advances in the field of autonomous systems pushed the introduction of robots in many human activities. Autonomous robotics is now widely used to clean domestic environments, as well as to support industrial processes within production plants. The increasing adoption of these technologies, especially in safety-critical applications, implies the rising importance of reliable, and structured flows along all the steps of the design process, from the requirement specification to the system validation.

Traditionally, the software in charge of controlling robots, as well as other families of cyber-physical systems, is designed on top of dynamical model simulators, and its validation strongly relies on extensive simulation [6]: a practice not providing the rigorousness required by safety-critical applications. Recently, formal methods have been introduced in this field [13, 21], trying to develop correct-by-construction design flows able to synthesize the control software from high-level requirements. However, dealing with such a problem holistically leads to serious complexity issues, especially when considering the constantly increasing size of the applications being designed. Problem decomposition, together with abstraction, will play a central role for future system-level design methodologies [19]. Considering the problem depicted in Figure 1, where a set of system requirements must be implemented by a set of collaborating robots, while respecting some

This is a **preprint** of an article that will appear as part of the ESWEEK-TECS special issue and will be presented in the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2019.

Authors' addresses: Stefano Spellini, stefano.spellini@univr.it, University of Verona, Department of Computer Science, Strada le Grazie, 15, Verona, Italy, 37134; Michele Lora, michele_lora@sutd.edu.sg, Singapore University of Technology and Design, Information Systems Technology and Design (ISTD), 8, Somapah Rd. Singapore, Singapore, 487372; Franco Fummi, franco.fummi@univr.it, University of Verona, Department of Computer Science, Strada le Grazie, 15, Verona, Italy, 37134; Sudipta Chattopadhyay, sudipta_chattopadhyay@sutd.edu.sg, Singapore University of Technology and Design, Information Systems Technology and Design (ISTD), 8, Somapah Rd. Singapore, Singapore, 487372.

2019. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *ACM Transactions on Embedded Computing Systems*, <https://doi.org/10.1145/nnnnnnn>.

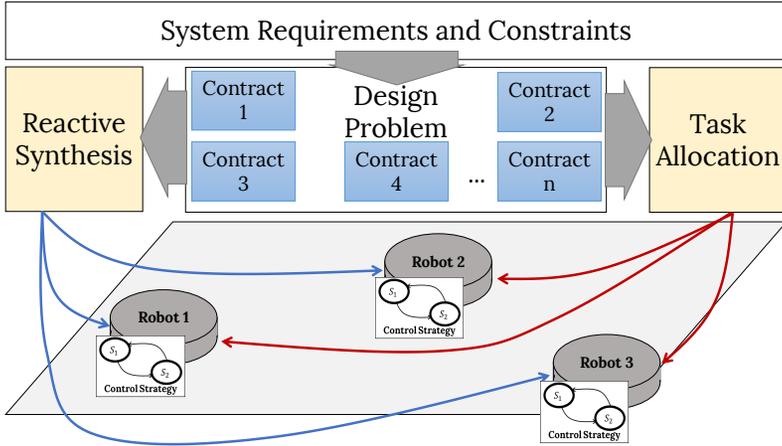


Fig. 1. Overview of problem tackled by the presented approach, and proposed problem decomposition.

constraints, abstraction and problem decomposition may be exploited in different ways. When designing the control software for the agents composing such a system one may consider going straight creating the coordination for the entire ensemble of robots. Otherwise, a designer may structure its design flow by defining the tasks composing a mission. Then, allocating the tasks to the single robots, and creating a control strategy for each of them independently. Of course, this requires to assure that the composition of the robots' behaviors, guided by their task allocations, is an *implementation* of the intended mission. Thus, the design flow needs to be supported by a formalism to describe system components and their abstractions.

The *Assume-Guarantee (A/G) Contracts* theory [3] has been developed to deal with these aspects. A/G contracts allow to decompose system design among the different components involved (*i.e.*, horizontal decomposition), and among different levels of abstraction (*i.e.*, vertical decomposition). An A/G contract formally represents a component as a pair of sets of behaviors defined over its variables: the *assumptions* and the *guarantees* [3]. A system can be represented as a composition of components (*i.e.*, *horizontal contracts*), while each component may be modeled at different levels of abstraction and according to diverse points of view (*i.e.*, *vertical contracts*) [31].

We present a *compositional approach to generate the control software* for multi-robot systems. The main innovation of our methodology is the structured decomposition, supported by the A/G contract formalism, of the design process. As Figure 1 shows, it starts from a set of requirements and constraints characterizing a mission of the system. The requirements and the constraints are formalized as A/G contracts, to partition the design problem into multiple sub-problems. Then, we propose to use a subset of the contracts to *synthesize a control strategy* for every single robot in the system. Meanwhile, a subset of the requirements is used to perform *tasks allocation* among the different robots. The information contained in the contracts are also used to automatically generate an executable model of the system for simulation purposes. We exploit system-level simulation to validate the results of the design flow. Once the synthesized control strategies are shown to be correct, they can be translated into the software concretely controlling the robots. In particular, this paper presents an approach able to generate code for the popular ROS middleware [35]. More specifically, each agent's control software is wrapped into a ROS node. A set of well-defined messages and topics is created to handle the communication between the generated nodes. We validate the results of the presented methodology by simulating the final system using *Gazebo*, a

robotic oriented 3D CAD tool and simulation environment, able to emulate the behavior of robots controlled by ROS software.

The approach is presented through a running example based on a multi-robot goods transportation system. We evaluate the effectiveness of the approach by applying it to scenarios of increasing complexity. The experimental results show the scalability that may be achieved by systematically decomposing the design problem.

The contributions of this work can be summarized as follows:

- Definition of a problem decomposition strategy that allows to address compositionally the design of control software for multi-robot systems. In particular, we exploit the A/G contracts to automatically partition the requirements and the specifications of the system into smaller sub-problems to be addressed individually, and obtain a set of control strategies for the components of the system. Then, we define a simulation-based method to validate the composition of the sub-problems solutions with respect to the main problem requirements.
- A technique to automatically generate software code implementing the control strategies generated, and validated, for the components of the system. In particular, the generated code is based on the popular ROS framework.

This paper is organised as follows. Section 2 analyzes the related work. It reports some key concepts of the A/G contracts theory, the General Reactivity (1) (GR(1)) fragment and ROS. A case study is defined, to illustrate the application of the proposed approach. Section 3 provides an overview of the suggested methodology. The contract-based specification of the robotic system is detailed in Section 4. The decomposed system is synthesized and validated as described in Section 5. Section 6 presents the automatic generation of code for ROS. Section 7 shows the applicability of obtained implementations and the scalability of the entire methodology. We finally draw some conclusions in Section 8.

2 PRELIMINARIES AND RELATED WORK

The increasing complexity of robotic systems led to the introduction of multiple robot software frameworks [18] aiming at easing the development of robotic software. Notable examples are ROS [35] and OROCOS [7]. These frameworks typically act as middleware, abstracting away the low level details about the physical system and providing high-level primitives to develop functionalities. Usually, designers develop the software on top of such frameworks. Then, they evaluate the correctness of the implemented behavior by using simulators able to reproduce the physical behavior of the system. Meanwhile, the main focus of the robotic software research community is shifting toward the integration of cognitive functionalities to make robots everyday more autonomous [1]. However, all these works rely on extensive simulation to reason about the correctness of the system being controlled by the developed software. Thus, none of them is able to provide strong guarantees about the correctness of the software being produced.

Formally specifying high-level behaviors of robotic systems to obtain a provably correct implementation is not a new task in the autonomous robotics research area. The work presented in [38] proposes the idea of using A/G contracts, in order to reduce the complexity required to synthesize a control strategy for autonomous multi-robot systems. Even though it introduces the guiding ideas supporting the presented work, it presents only preliminary results without providing a structured methodology. It decomposes the design problem only horizontally: it decomposes the problem over the different components only; it does not decompose over the different aspects and abstractions of the same problem (*i.e.*, vertical decomposition [30]). Furthermore, it does not target the generation of actual code able to run on real systems. On the contrary, the presented contribution exploits the ability of A/G contracts to decompose the problem also vertically. Moreover, the different

parts resulting from the decomposition are used to solve different sub-problems in a structured methodology, able to generate ROS code that can be run on actual robots.

Another end-to-end methodology to generate executable code that implements high-level robot behaviors has been proposed in [25]. The methodology is applied to the DARPA Robotic Challenge (DRC), consisting in a semi-autonomous mission executed by a ground robot operating in a dangerous environment for humans. This approach assumes that each sub-component of the system is already defined in a centralized way, while concentrating on the reactive mission plan applied to a high-level view of available actions and behaviors. The approach has been exploited by the same team [26] to specify system capabilities that are mapped onto the task definition and execution. The integration of robotic control-software obtained by formal specifications and GR(1) synthesis in ROS is addressed in [39]. The authors developed a framework to automatically synthesize a ROS node implementing GR(1) specifications. The approach relies on the reactive synthesis algorithms by *Slugs* [10]. Finucane *et al.* [14] developed a toolbox (*i.e.*, *LTLMoP*) to develop and test high-level robot controllers from problem specification in natural English. This tools collection is also able to set up a visual simulation of the robot mission execution. All these contributions rely on the automatic synthesis of control strategies from Linear Temporal Logic (LTL) specifications [22]. However, none of the previous works addressed and exploited a structured problem decomposition to reduce the complexity of the design process.

Our methodology has similar objectives to the previously described works. Furthermore, we also rely on the same theoretical pillars, such as the automatic generation of control strategies from GR(1) specifications [4]. However, the work presented in this paper is able to keep manageable larger and more detailed definitions of the system being designed thanks to the compositional reasoning enabled by A/G contracts formalism. In addition, the robotics systems we consider are composed of multiple agents interacting with each other, producing an extremely complex problem to specify and synthesize in practice. As we will discuss, traditional synthesis from temporal logic specifications is not feasible, taking into account the necessity for an optimized specification method and synthesis algorithms. In the following, we provide some key concepts to understand the background of the proposed approach.

2.1 A/G Contracts

A contract C for a component M is a triple (V, A, G) , where V is the set of the component variables, and A and G are *assertions*, each representing a set of behaviors over V [3]. A represents the *assumptions* that M makes on its environment, and G represents the *guarantees* provided by M under the environment assumptions.

A component M satisfies a contract C whenever M and C are defined over the same set of variables, and all the behaviors of M satisfy the guarantees of C in the context of the assumptions, meaning that M is an implementation of C . Moreover, a component E can also be associated with a contract C as an environment for the contract. We say that E is a legal environment of C , whenever E and C have the same variables and the behaviors implemented by E are a subset of A . The A/G contract theory [3] defines a set of operations, those used in this work are:

- *Composition*: Contracts associated to different components can be combined according to different rules. *Parallel composition* builds complex contracts from simpler ones.
- *Compatibility and Consistency*: C is *compatible* if there exists a legal environment E for it. A contract is *consistent* when the set of implementations satisfying it is not empty.
- *Refinement*: A contract C refines a contract C' , written $C \leq C'$, if and only if $A \supseteq A'$ and $G \subseteq G'$.

2.2 Linear Temporal Logic (LTL)

LTL formulas are perfectly suited to model the evolution of a system over time. A component behaviour is expressed considering present and future paths, *i.e.* a condition that will eventually hold in the future. Specifically related to A/G contracts, both assumptions A and guarantees G of a contract C can be specified as LTL formulas [28]: a component M satisfies a contract C if it fulfils the logical implication $A \rightarrow G$, while it is a legal environment for C if it satisfies the formula A [31].

LTL Syntax. Let AP be a set of atomic propositions where $\pi \in AP$ is a Boolean variable. LTL formulas are constructed from atomic propositions $\pi \in AP$ according to the following grammar:

$$\varphi ::= \pi \mid \neg\varphi \mid \varphi \vee \varphi \mid \bigcirc\varphi \mid \varphi\mathcal{U}\varphi \quad (1)$$

where \neg (“not”) and \vee (“or”) are Boolean operators, and \bigcirc (“next”) and \mathcal{U} (“until”) are temporal operators. Given “next” (\bigcirc) and “until” (\mathcal{U}) operators, additional temporal operators can be derived such as “eventually”: $\diamond\varphi = \text{True}\mathcal{U}\varphi$ and “always”: $\Box\varphi = \neg\diamond\neg\varphi$.

LTL Semantics. Semantics of an LTL formula φ are defined on an infinite sequence $\sigma = \sigma_1\sigma_2\dots$ of truth assignments to the atomic propositions $\pi \in AP$, where σ_i denotes the set of atomic propositions that are True at position i . Whether σ satisfies LTL formula φ at position i (denoted $\sigma, i \models \varphi$) is recursively defined as:

- $\sigma, i \models \varphi$ iff $\pi \in \sigma_i$,
- $\sigma, i \models \neg\varphi$ iff $\sigma, i \not\models \varphi$,
- $\sigma, i \models \varphi_1 \vee \varphi_2$ iff $\sigma, i \models \varphi_1$ or $\sigma, i \models \varphi_2$,
- $\sigma, i \models \bigcirc\varphi$ iff $\sigma, i+1 \models \varphi$
- $\sigma, i \models \varphi_1\mathcal{U}\varphi_2$ iff there exists $k \geq i$ such that $\sigma, k \models \varphi_2$, and for all $i \leq j < k$, $\sigma, j \models \varphi_1$.

The formula $\bigcirc\varphi$ expresses that φ is True in the next “step” (the next position in the sequence) and the formula $\varphi_1\mathcal{U}\varphi_2$ expresses the property that φ_1 is True until φ_2 becomes True.

The sequence σ satisfies formula φ if $\sigma, 0 \models \varphi$. The sequence σ satisfies formula $\Box\varphi$ if φ is True in every position of the sequence, and satisfies the formula $\diamond\varphi$ if φ is True at some position on the sequence.

2.3 General Reactivity (GR(1))

LTL formulas can be used as specifications of reactive systems where atomic propositions are divided between the *environment* (*i.e.*, the system input) and the *system* (*i.e.*, the system output). The realizability of LTL is 2-EXPTIME-complete, which makes it practically infeasible [37]. However, for the GR(1) fragment of the LTL, there is an algorithm able to decide the realizability in N^3 [27]. GR(1) synthesis specifications contain assertions over initial states, safety constraints relating the current and next state, and goals requiring that an assertion holds infinitely often during a computation. More specifically, a GR(1) synthesis problem is defined as a game between a system player and an environment player, with the following game structure:

- \mathcal{X} input variables controlled by the environment
- \mathcal{Y} output variables controlled by the system
- θ^e assertion over \mathcal{X} characterizing initial states of the environment
- θ^s assertion over $\mathcal{X} \cup \mathcal{Y}$ characterizing initial states of the system
- $\rho^e(\mathcal{X} \cup \mathcal{Y}, \mathcal{X})$ transition relation of the environment
- $\rho^s(\mathcal{X} \cup \mathcal{Y}, \mathcal{X} \cup \mathcal{Y})$ transition relation of the system
- $\varphi = GFJ^e \rightarrow GFJ^s$ winning condition as implication between justice goals J^e of the environment and J^s of the system.

The acceptance condition is finally defined as:

$$(\theta^e \wedge G\rho^e \wedge GFJ^e) \rightarrow (\theta^s \wedge G\rho^s \wedge GFJ^s) \quad (2)$$

where $G\rho^e$ and $G\rho^s$ are safety conditions over the environment and the system while GFJ^e and GFJ^s are liveness properties over the environment and the system.

In the literature, many different tools implement reactive synthesis from the GR(1) fragment [10, 13]. These tools accept a GR(1) LTL specifications and return a Mealy Machine implementing a control strategy that allows the system player to win over the environment player. In this paper, we are going to use the GR1C tool of the Temporal Logic Planning (TuLiP) toolbox [13] to perform reactive synthesis from GR(1) specifications.

2.4 The Robot Operating System (ROS)

ROS [35] is a collection of packages for developing robotic software, from low-level device control to communication protocols between robot components. Typically, a robotic system is organized in ROS as a set of distributed processes connected to each other. As such, a ROS-based system is usually represented by a graph depicting its architecture where nodes represent the processes implementing the computation in the system. For the same reason, in the ROS terminology, the term *node* is synonym of process. Each node of the graph corresponds to a component of the system, *i.e.*, a sensor, an actuator, or an algorithm implementing a functionality. Each node is usually connected to other nodes using different styles of communication, such as synchronous RPC-style services or asynchronous data streaming over *topics*. More specifically, a topic is an abstraction used to identify a specific message subject or content, while ROS messages through nodes can be considered as particular data structures, composed by multiple fields with specific field types (*i.e.*, *int8* or *string*). A *message_generation* package is available to define custom system-level messages: by specifying a set of data fields and types, a C++ header file is generated, which can be used by nodes implementations to pack and unpack topic's messages structures. A ROS node is generally implemented by using a well-defined *client library*. Such a library provides an application programming interface in different languages, such as C++ or Python. In particular, the *roscpp* library provides interfaces and standard functions to create a ROS node in C++, capable of interacting with topics and services. A less runtime-performant but more intuitive method of designing nodes is provided by the *rospy* client library, which is similar to *roscpp* in terms of functionalities.

The ROS ecosystem is completed by some tools for analysis and simulation. *Gazebo* [33] is a 3D CAD tool, that allows designing the mechanical parts of robots. Furthermore, it provides a robust 3D simulation environment. It includes a solid physical engine and multiple extension packages, such as *gazebo_ros_pkgs*, that allow the integration of newly defined ROS nodes into Gazebo logic and visual models. More specifically, Gazebo interacts with ROS by using a set of defined messages and services, which acts as an interface between a standalone simulation client and various ROS nodes representing a robotic system.

2.5 Case study: goods transportation system

For the sake of clarity, we exemplify the concepts described on the paper on a running example. The case study is inspired by autonomous multi-robot goods transportation systems. Figure 2 gives a schematic representation of the system being considered. A set of robots moves in a two-dimensional space, such as a building floor. The robots are required to move a set of objects to specific positions in space. Such positions are called *targets*. They are indicated in Figure 2 by the star icons. The order in which the targets must be reached is imposed by the system requirements, *i.e.*, the requirements define a partial order over the set of targets. Each robot may move in four directions (*i.e.*, up, down, left and right). Robots must avoid crashing into eventual obstacles and

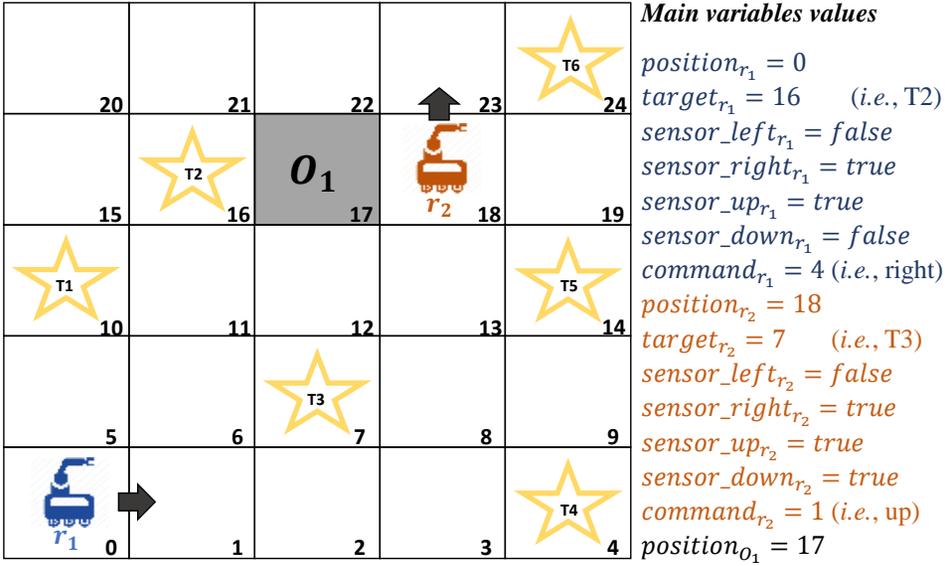


Fig. 2. Running example: an autonomous goods transportation system, composed of two robots, coordinating to reach six different targets.

into each other. Each robot is equipped with proximity sensors, providing to the robot information about the physical objects present in the robot’s immediate surrounding environment. In practice, at each instant the robot is aware of the space available in the four directions it can move. Figure 2 depicts the case in which two robots (i.e., R1, and R2) must collaborate to complete all six tasks. The right side of Figure 2 reports the main variables used in the formalization described in Section 4, and their values according to the state depicted in the left side of the figure.

It is important noticing that the methodology can be extended to many classes of robotic systems. In fact, it is trivial to extend the formalization described in Section 4 to consider three-dimensional spaces. Such an extension requires to simply add a further variable modeling the third dimension for every variable referring to the position of a system component. The extension allows to extend the field of applicability to autonomous unmanned vehicles [9]. However, using a three-dimensional case study would not add any information about the methodology, while complicating the formulas used in the formalization step.

Furthermore, the approach can be extended to deal with robots composed of multiple independent mobile parts moving in the three-dimensional space. In such a case, the formalization would have multiple components (i.e., the mobile parts of the robot) rather than multiple robots. Indeed, a set of logical assertions must be added, during the formalization step, modeling the constraints about the relative positions of the components. As long as the different mobile parts act independently to one another, the approach can be used to generate a ROS node for each mobile part to be controlled.

3 OVERVIEW

Given a multi-robot system and a set of targets to be reached by the system, the final objective of our work is to automatically generate the control software for each robot composing the system. The generated software is targeted to the ROS framework, and it allows the ensemble of robots to accomplish all the required targets. Figure 3 summarizes all the phases composing the proposed flow, enumerated within gray boxes.

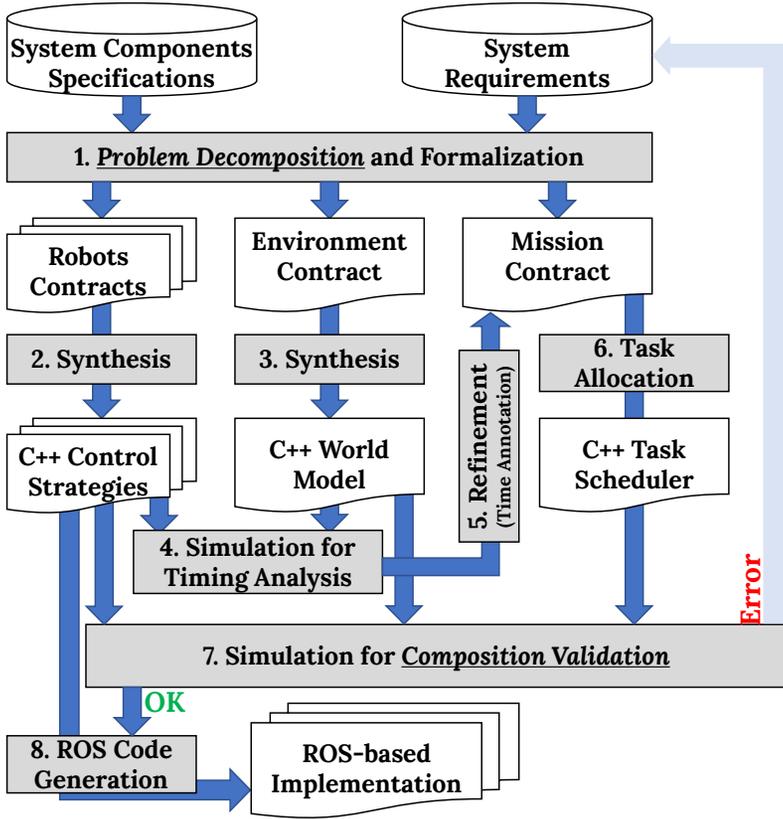


Fig. 3. General schema of the proposed methodology. Requirements and specifications are formalized and decomposed as a set of *A/G contracts* (Step 1). After solving the sub-problems (Steps 2 to 6), their composition is validated through simulation (Step 7). In case the validation results in a positive outcome, ROS code implementing the control strategy of each robot is automatically produced (Step 8).

Initially, the design problem is characterized by a set of requirements and specifications. The *system components specifications* define all the resources available in the system, as well as the constraints on their use. On the other hand, the *system requirements* specify which are the objectives of the system, such as the operations that must be accomplished by the the robots and the timing constraints on such operations. Requirements may differ between different systems, and they can be captured in different ways. Many different approaches have been proposed to specify and formalize requirements and components for robotics and cyber-physical systems in general [12, 34]. Requirements management methodologies allow to automatically cast the formalization of requirements and components within a specific theory or design flow [32]. For instance, a structured management of requirements is helpful when aiming at automatically partitioning a complex design problem by using *A/G contracts* [2, 29]. While requirements management is not in the scope of this paper, we assume that the design process starts from requirements managed by such a kind of techniques. In the case study introduced in Section 2.5, the requirements are the targets to be reached and the timing within these must be reached. Its components are indeed the robots, but also the two-dimensional physical space in which the robots move. As the physical space is a component of the system, then the constraints imposed by the physics are part of the system component

specifications. Thus, for instance, the presence of physical obstacles, as well as the reaction time, or the features of the robots' sensors are part of the system components specifications.

The *Problem decomposition and formalization step* (i.e., Step 1) formalizes the specifications and requirements composing the design problem by using A/G contracts. Alternatively, the design problem may be represented by a single contract having all the identified requirements as guarantees, and assuming all the identified constraints, as many state-of-the-art approaches do [22]. Theoretically, such a holistic contract may also be processed to perform reactive synthesis, and to generate a control software for the entire system [13]. However, the problem would be computationally intractable [29]. Conversely, the presented approach partitions the design problem into multiple sub-problems, each of them represented by a contract. As such, the design problem formalization is partitioned into the following contracts:

- multiple *Robot Contracts*, one for each robot in the system, describing each robot constraints and behavior. Each contract assumes the characteristics of the environment, while guaranteeing to react to external stimuli properly.
- one *Environment Contract* describing the constraints imposed by the system, and assuming the behavior of the robots, i.e., it assumes the robots properly reacting to environmental stimuli. It must guarantee the assumptions made by the robot contracts.
- one *Mission Contract* describes the tasks the robots must accomplish. It assumes the ability of the robots of reaching each target within a given time limit. It guarantees the existence of a sequence of tasks for each robot, such that the entire ensemble of robots completes all the required tasks.

The defined contracts are used in different manners. Each robot contract is used to synthesize a *control strategy* for each robot by applying reactive synthesis (Step 2). Thus, for each robot contract it is produced a Mealy Machine implementing the contract. The generated Mealy Machine is translated into an equivalent *C++ executable model of the control strategies* for the robot. The same flow is applied to the environment contract (Step 3), thus producing a *C++ model of the world* containing the robots (i.e., the environment).

The implementations obtained by synthesizing the robot and environment contracts are merged and compiled together to create a custom simulator for the system. Such a custom simulator provides higher simulation speed by integrating the simulation engine and all the models into a single executable [24], and it is used to perform multiple *simulations* necessary to evaluate the time required by each robot to reach each target (Step 4). The information retrieved by the simulation is used to *refine the mission contract*: for each robot and each task, the mission contract assumes the timing value retrieved from the simulations as the time the robot needs to reach the target.

The mission contract is given to a *task allocation* algorithm that takes care of allocating the tasks to the single robot (Step 6). The procedure generates a *C++ task scheduler* assigning to each robot the tasks to perform. After this step, we have both a strategy for each robot and a task allocation for the entire system. However, each sub-problem has been solved by using different assumptions. As such, it is necessary to verify that the composition of the solution is still a solution for the initial specification. Proving the correctness of the composition using formal methods leads to intractable complexity. For this reason, the presented approach performs *simulation for composition validation*. The C++ models of the control strategies, environment (i.e., the world model), and the C++ task scheduler can be compiled together to create an executable specification of the entire system, thus composing the solutions of the sub-problems into a fast custom simulator. The composed system is simulated for all the initial conditions that are admissible according to the initial specification assumptions (Step 7). In the case none of the simulations violates the initial requirements, then the

C++ control strategy of each robot is automatically translated into a ROS-based implementation (Step 8).

On the contrary, in the case the simulation for composition validation step fails, then the designer may either try to generate an alternative task allocation, or to relax the system requirements and iterate the design flow. In either cases, the trace generated by the failing simulation may be helpful to identify the conflicting requirements [8, 23]. However, a more in-depth investigation of this strategy is beyond the scope of this paper.

4 DESIGN PROBLEM SPECIFICATION

For each instance of the design problem, three contract-based specifications have been created: a contract representing the entire problem holistically, *i.e.*, the *System Contract* (C_S); a contract specifying a single robot instance, *i.e.*, the *Robot Contract* (C_R); a contract that describes the environment in which various robots move, *i.e.* the *Environment Contract* (C_E); and a contract specifying the tasks to be performed to complete the mission, *i.e.*, the *Mission Contract* (C_M). Contracts C_R , C_E and C_M are obtained by decomposing C_S . An implementation of the composition of the multiple instances of C_R (one for each robot in the system), C_E and C_M must implement the initial contract C_S .

Such a partition decomposes the problem *horizontally* (*i.e.*, among the different components of the system) by dividing the problem among the different robots composing the system. It also decomposes the problem *vertically* (*i.e.*, among different levels of abstraction), as the Mission Contract may be seen as *an abstraction of the composition* of robots and environment. In fact, while satisfying the mission contract requires the existence of a task assignment, satisfying the composition of the Robot Contracts and the Environment Contract requires to identify a control strategy for each problem that is able to implement the solution of the Mission Contract. The following of this section provides the details of these specifications.

4.1 System Contract (C_S)

The representation of the two-dimensional space is discretized and represented as an occupancy grid [11]. Each robot r is characterized by four boolean variables, expressing the four directional sensors (*i.e.* *sensor_up*, *sensor_down*, *sensor_left*, *sensor_right*), an integer variable for the position (*i.e.* *position = n*), and an integer variable representing the robot command (*i.e.* *command = c*). A boolean value for each cell of the occupancy grid is used to store whether a position is free or not (*i.e.* *free_p* where p is the index of the cell). Target positions are given as input to each robot. Finally, each robot stores an integer value to represent the number of steps it performed (*i.e.*, how many times its position changed). The system specification is within the GR(1) fragment.

The *initialization assumptions* predicates about the robots' initial positions and sensor values:

- two robots cannot have the same position, for each grid cell n :

$$\neg(\text{position}_{r_i} = n \wedge \text{position}_{r_j} = n \wedge i \neq j)$$

- For every position n , a conjunction over all the robots defines the *occupation status* of the cells:

$$\text{free}_n = \left(\bigwedge_{r_i} \neg(\text{position}_{r_i} = n) \right)$$

- Sensors values are set according to cells adjacent in the occupancy grid and the grid boundaries. For instance, the assertion:

$$\text{position}_{r_i} = 18 \rightarrow (\text{sensor_left}_{r_i} = \text{free}_{17})$$

Compositional Design of Multi-Robot Systems Control Software on ROS

specifies that the value of the left sensor of a robot in position 18, depends on the occupancy state of position 17. C_S specifies one assertion such as this for each position and each sensor.

The *initialization guarantees* set the initial command of every robot to stay, and its steps counter to zero. Initialization assumptions have to hold also at each time step, so they are also duplicated as safety guarantees.

The *safety assumption* properties define the robots motion:

$$\Box((\text{position}_r = n \wedge \text{command}_r = c) \rightarrow \bigcirc(\text{position}_r = m))$$

where m is the position adjacent to the position n , in the direction specified by the command c . Then, safety assumptions model the global time of the system advancing at every evaluation step:

$$\Box(\text{timer} = t \rightarrow \bigcirc \text{timer} = t + 1)$$

The *safety guarantees* specify that the robot decide the next command based on the sensor values. Thus, considering the running example in Section 2.5, and the command *up* the system guarantees are as follows:

$$\Box(\bigcirc(\text{command}_{r_i} = \text{up}) \rightarrow \text{sensor_up}_{r_i})$$

The same guarantee must be inserted for each robot, and each command modifying the status of the system, in the case study: *up*, *down*, *left* and *right*. C_S guarantees also the correct evolution of the step counters:

$$\Box(((\text{position}_{r_i} \neq \bigcirc \text{position}_{r_i}) \wedge \text{steps}_{r_i} = n) \rightarrow (\bigcirc \text{steps}_{r_i} = n + 1))$$

The only *goal* of the system is that each robot r_i target should be reached within a certain amount of time T , i.e.,

$$\Box \diamond (\text{position}_{r_i} = \text{target}_{r_i} \wedge \text{timer} \leq T)$$

4.2 Robot Contract (C_R)

The A/G contract of every robot instance is an abstraction of C_S (i.e., $C_S \leq C_R$) obtained by increasing the set of assumptions by assuming an environment that allows the robot to move freely (i.e. $A_S \subset A_R$). The resulting A/G contract responds with the appropriate command value to input sensors values provided by the environment. For every position n , assumptions are as follows:

$$\Box(\text{position}_{r_i} = n \rightarrow (\text{sensor_up}_{r_i} \wedge \text{sensor_down}_{r_i} \wedge \text{sensor_left}_{r_i} \wedge \text{sensor_right}_{r_i}))$$

However, it is possible to define different configurations inserting assumptions about obstacles. Considering the example in Figure 2, it is possible to assume the existence of the obstacle O_1 by inserting assumptions that forces different values for the sensors, such as:

$$\Box(\text{position}_{r_i} = 12 \rightarrow (\neg \text{sensor_up}_{r_i} \wedge \text{sensor_down}_{r_i} \wedge \text{sensor_left}_{r_i} \wedge \text{sensor_right}_{r_i}))$$

However, C_R does not consider the other robots in the system, thus assuming that other robots do not act as obstacles.

The guarantees expressed in the C_R are the subset of guarantees of C_S that generates the value for the next command to send according to the sensors values.

In particular, if the target position is reached, the only possible command is stay:

$$\Box((\text{position}_r = \text{target}_r) \rightarrow \bigcirc(\text{possible_command} = \text{stay}))$$

Finally, if the output command is different than stay, the robot has moved. Thus, the time required by the agent to reach the desired target is incremented by 1.

4.3 Environment Contract (C_E)

The environment contract is an abstraction of the general system contract modeling the physical environment in which the robots act. The occupancy grid is represented by a set of boolean variables, one for each cell. Each variable values true if and only if the corresponding block is free. The environment contract assumes the behavior of the robots. Thus, its behavior is based on the command values assumed the robots would generate. Its output is the set of positions for the robots, as well as the sensor values for each robot. More specifically, for each robot r , for each position n and each command c , an assertion is in charge of computing the next position of the robot. Assertions describing robot movements are:

$$\square(\text{position}_r = n \wedge \text{command}_r = c \rightarrow \bigcirc(\text{position}_r = m))$$

The environment contract also guarantees some trivial constraints of the physical system, *i.e.*, two robots can never share the same position. Thus, for each pair of robots r_i and r_j :

$$\neg(\text{position}_{r_i} = n \wedge \text{position}_{r_j} = n \wedge i \neq j)$$

4.4 Mission Contract (C_M)

The mission contract formalizes system requirements specifying the timing constraints of the target to be reached by the robots, as well as their ordering. It assumes the main features of the system, *i.e.* the number of cooperating robots, as well as the time needed by each robot to complete each target. Furthermore, it assumes that no obstacle can create a situation such that a robot gets stuck in some positions in the two-dimensional space. Meanwhile, the contract guarantees a certain time-bound within every task of the system will be completed. That is, the mission will be completed within N steps.

The maximum time required by each robot to move to every target is hard to be computed. Retrieving such time requires computing, for each robot, all the possible paths between every pair of targets in the system. For this reason, system simulation is used to enrich the contract with the travel time information for each robot and each target. The execution of synthesized robot controllers into the generated environment model ensures that a robot r_n can reach a target t_m in k steps. The contract is defined as follows:

- *Assumptions*: each robot possible path to each available target is structured as a triple (I, D, T) where I is the initial robot position, D is the destination cell and T is the time required to reach that cell.
- *Guarantees*: mission requirements can be fulfilled since a set of targets can be reached in a predictable time limit.

The consistency of the mission contract provides a possible allocation of tasks to the robots that may potentially realize initial requirements and constraints. This contract is an abstraction of C_S , as it defines a larger set of assumptions: it assumes for each robot, and each pair of targets, the maximum travel time required by the robot to travel between the two targets. As such, while the solution obtained satisfy C_M , it may not satisfy all of its refinements such as C_S . Thus, the solution must be validated. The following section details how the allocation is generated and validated.

5 SYNTHESIS AND VALIDATION

After decomposing the design problem, the sub-problems represented by the contracts defined above must be solved. Then, their solutions must be combined to obtain a solution to the design problem. Thus, each contract must undergo different steps, as introduced above in Section 3. This section details the different steps.

5.1 Code generation and simulation

Contracts describing the robots and the environment of the system undergo reactive synthesis. They are formalized within the GR(1) fragment of the LTL. Thus, each of them can be used to synthesize a control strategy by using a reactive synthesis tool. In our case, we rely on GR1C [13]. After being synthesized, each strategy is expressed as a Mealy Machine in JSON format.

An automatic tool has been developed to generate executable C++ code starting from each of the Mealy Machine descriptions. It includes a JSON parser creating an intermediate representation of a Mealy Machine object. Then, the tool generates a C++ class implementing the Mealy Machine by exploiting automatic homogeneous code generation [15]. This allows creating an executable specification for each A/G contract emulating the behavior of the component specified by the contract.

Composing the executable models of robots and environment enables early simulation for the system. Each C++ class representing a robot is instantiated by a top-level component, together with the class generated by the environment contract. The inputs of the robots are the outputs of the environment model and vice versa. A scheduling procedure [15] is in charge of reproducing the concurrency, and managing the communication and synchronization among the different components of the system.

5.2 Task allocation

The simulation environment defined above can be used to retrieve the set of times required by robots to reach all the available targets. This is done by simulating the system multiple times. At each simulation, one robot moves between a pair of targets. The process is repeated for each robot, and each pair of targets to be reached consecutively. The information gathered by such simulation phase is then used to generate a tasks allocation over the available robots.

The task allocation problem can be resolved in multiple ways, and much research has been already carried on in this field [16]. Furthermore, the solution generated by performing reactive synthesis from the system contract C_S , without decomposing the design problem, already embeds in itself the solution of the task allocation sub-problem. The framework we implemented to evaluate the positive impact of problem decomposition relies on a procedure made in-house. However, it is important noticing that any other technique for task allocation can be used to address this step. For the sake of completeness, we report the details about our solution to better clarify how to connect other solutions to our methodology.

The procedure encodes the problem as a *directed weighted graph* $G = (V, E)$. Vertices V represent either robots initial positions and target positions. Edges E and their associated weight represent *cost of paths* connecting nodes. A partial or total ordering relation may be defined over the targets by the mission specification, *i.e.*, a set of constraints about the order in which the tasks must be performed. Furthermore, the mission may impose that two consecutive tasks are carried on by the same robot. This may be useful to represent a robot moving an object from one target to another. Figure 4 shows the graph extracted by the case study described in Section 2.5.

Algorithm 1 takes the graph representation as input, to produce a task allocation while minimizing the overall mission cost, *i.e.*, the sum of all the robots actions. It also aims at keeping balanced the number of actions performed by each robot. Thus, trying to increase concurrency, and decreasing the overall mission duration. Line 1 to 3 initialize some lists: the list of n robots, the costs of each robot, and a list that will be used to store temporary values, one for each robot. Then, the list of the allocations is initialized empty: it will contain a list of robot/task pairs, each indicating that a task must be performed by the assigned robot. The procedure iterates over the targets in the `task_list` (Line 6). Then for each robot, the `min_path` function, based on the Dijkstra shortest path algorithm,

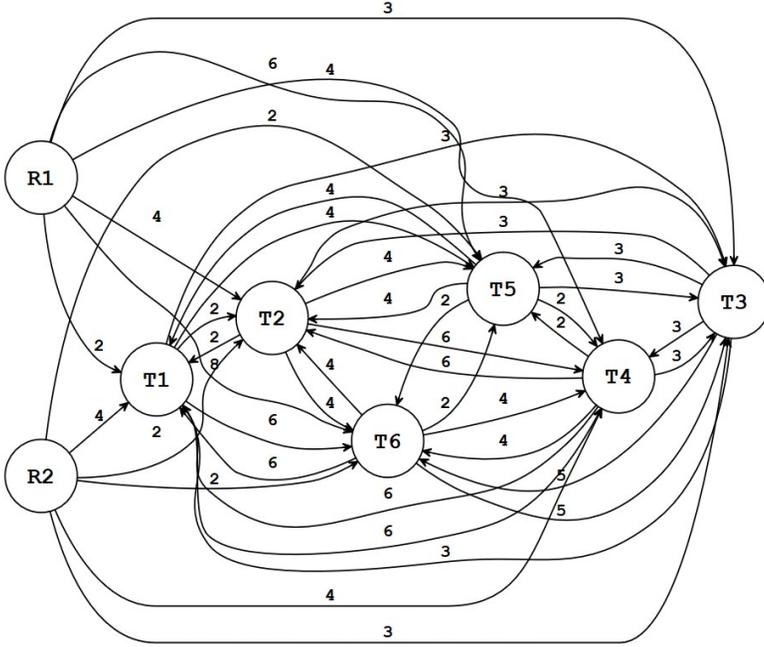


Fig. 4. Weighted graph extracted from the case study. It represents the costs of all the paths from robots to targets.

returns the cost for the robot of the shortest path to the given target. Then, it allocates the target to the robot having the minimum value in costs, after summing the cost to reach the considered target (Lines 10-11). A pair is added to the allocation list, to indicate which robot will perform the task (Line 12). Then, the `remove_edges` sub-routine updates the graph by removing all the edges entering the node representing the allocated task. Then, the algorithm iterates to the following target in `task_list`. Finally, the procedure will return the list of allocated tasks (Line 16).

5.3 Simulation for system validation

Generating a control strategy for each robot of the system by synthesizing the complete system contract C_S provides a control strategy that already contains the solution of all the design sub-problems. Meanwhile, in our approach, every sub-problem is solved by using a set of assumptions that is larger than the set of assumptions of C_S . The task allocation is generated under the assumptions of the Mission Contract (C_M), that is an abstraction of the System Contract (C_S). Thus, C_M makes more assumptions with respect to C_S . For this reason, the solution of C_M may or may not be a solution also for C_S . Consequently, it is necessary to validate the generated task allocation. The final step of the design flow is a System-level validation through simulation. The objective is to verify that task allocation to each robot is feasible and the control software will be able to complete the given mission while respecting the constraints imposed by C_S . The simulation relies on the executable models generated by the approach in Section 5.1, while being driven by the generated task allocation. The allocation is encoded as a C++ static scheduler assigning the tasks to the different robots instantiated by the C++ executable model of the environment.

The different C++ models are integrated and compiled together into an executable system-level model of the system. Such a model is simulated for all the possible combination of initial conditions

Algorithm 1 Task allocation**Require:** graph of costs, task_list**Ensure:** minimize total mission cost and robot cost differences

```

1: robots  $\leftarrow [r_1, r_2, \dots, r_n]$ 
2: costs  $\leftarrow [0, 0, \dots, 0]$ 
3: tmp_costs  $\leftarrow [0, 0, \dots, 0]$ 
4: allocation  $\leftarrow \emptyset$ 
5: procedure ASSIGN_TASKS
6:   for task in task_list do
7:     for r in robots do
8:       tmp_costs[r]  $\leftarrow \text{min\_path}(\text{graph}, r, \text{task}) + \text{costs}[r]$ 
9:     end for
10:    best_robot  $\leftarrow \text{min\_cost\_move}(\text{tmp\_costs})$ 
11:    costs[best_robot] += tmp_costs[best_robot]
12:    allocation.insert(best_robot, task)
13:    remove_edges(graph, task)
14:    tmp_costs  $\leftarrow [0, 0, \dots, 0]$ 
15:  end for
16: return allocation
17: end procedure

```

admissible according to the system contract. Finally, if the system-level validation returns a positive outcome, then the control strategies generated are *correct under the assumptions defined by the initial requirements*, and formalized by the system contract C_S . It is important to keep in mind that if the actual system does not respect assumptions specified in the requirements, then the generated implementation may be unable to control the system properly.

Once validated, the C++ implementations of the control strategies for the robots may be used as control software for the multi-robot system. Next Section describes how the validated C++ control strategies are used to produce ROS nodes.

6 ROS CODE GENERATION

The integration into ROS of the generated implementations requires to create a ROS node for each component of the defined system (*i.e.*, a robot). The set control strategies generated and validated for each robot in the system is the starting point to create multiple ROS nodes, each enclosing the control strategy of one component. The structure of such control strategy is depicted in the listing 1. In particular, it outlines the code implementation of the FSM that realizes the control strategy for such an agent: at each executeMachine call (Line 5), the next state is elaborated according to the current state (Lines 9, 28) and input parameters (Lines 11 and 19) while appropriately setting output variables (Lines 15-17 and 23-25).

The listing 2 describes the standard structure of a robot node implementation in C++, using the *roscpp* library. Input and output messages have a precise structure of type `msgs::robot_output` and `msgs::env_output` (Lines 1, 21). The first is composed of a set of boolean values, one for each directional sensor, and two integer variables, one representing the current position and one for the assigned target. The output message is characterized by two integer values, one for the produced command and one for the executed movements count.

Initially the ROS node is initialized (Line 8) and its handle is defined (Line 9), with its update rate (Line 10). At Line 12, the Robot controller is declared and instantiated. Then, a subscriber

```

1 String state;
2 int32_t steps_out;
3 int32_t target_out;
4
5 void Robot::executeMachine( bool up_in,
6   bool down_in, bool left_in, bool right_in,
7   int32_t target_in, int32_t position_in )
8 {
9   if (state == std::string("0"))
10  {
11    if (up_in == false && down_in == false
12      && left_in == false && right_in == false
13      && target_in == 0L && position_in == 0L)
14    {
15      state = "20";
16      steps_out = 1L;
17      command_out = 0L;
18    }
19    else if (up_in == true && down_in == false
20      && left_in == false && right_in == false
21      && target_in == 0L && position_in == 0L)
22    {
23      state = "28";
24      steps_out = 1L;
25      command_out = 0L;
26    }
27    ...
28  } else if (state == std::string("1"))
29  {
30    ...
31  }
32  ...
33 }

```

Listing 1. Code structure of the synthesized control software using the proposed methodology. The control logic is implemented by the *executeMachine* function, that models the behavior of an Finite State Machine (FSM).

and a publisher are created to manage the message passing protocol: the publisher (Line 14) sends messages of type `msgs::robot_output` to the `/r01/robot/output` topic, while the subscriber component listens to `/r01/environment/output` for `msgs::env_output` messages, calling a callback function (Line 3) to handle the received message and to fill the internal inputs structure. The *while* loop at Line 21 simulates the robot controller with input parameters gathered from the received message (Lines 22-24) and publishes an output message representing the computed values from the controller (Lines 26-30).

A node subscription, as well as publication, is not constrained to a single topic: the environment node, in our case, is composed of N subscribers and publishers, attached to N topics, where N is the number of robots acting in the system. The mission execution is handled by another similarly defined node: it subscribes to each robot position, checking whether the current target is reached. In that case, it publishes to the environment node the next target.

The execution of each node is started by publishing an initial message to each robot, containing the initial conditions of the entire robotic system. It is terminated when each robot reaches all the targets the mission nodes has assigned to it.

Compositional Design of Multi-Robot Systems Control Software on ROS

```
1 msgs::env_output inputs;
2
3 void callback(const msgs::env_outputConstPtr &msg) {
4     inputs = *msg;
5 }
6
7 int main(int argc, char** argv) {
8     ros::init(argc, argv, "robot_controller");
9     ros::NodeHandle nh("~");
10    ros::Rate r(1);
11
12    Robot robot_ctrl;
13
14    ros::Publisher p = nh.advertise<msgs::robot_output>
15        ("/r01/robot/output", 1);
16    ros::Subscriber s = nh.subscribe<msgs::env_output>
17        ("/r01/environment/output", 1, &callback);
18
19    msgs::robot_output output;
20
21    while (ros::ok()) {
22        robot_ctrl.executeMachine(inputs.up0, inputs.down0,
23            inputs.left0, inputs.right0, inputs.target0,
24            inputs.position0);
25
26        output.steps = robot_ctrl.steps_out;
27        output.command = robot_ctrl.command_out;
28        output.target = inputs.target0;
29
30        p.publish(output);
31
32        ros::spinOnce();
33        r.sleep();
34    }
35
36    ros::shutdown();
37
38    return 0;
39 }
```

Listing 2. Code structure used to implement and instantiate the control software into a ROS node. At each update, input message parameters are gathered and passed to the control strategy instance. This produces new output values, that are published to the destination topic.

7 EXPERIMENTAL RESULTS

Every phase of the proposed design flow has been automated. We set up a proof-of-concept tool-chain created specifically to bind A/G contract reasoning and simulation-based techniques. Each sub-component or aspect of the problem is specified through an A/G contract in the SPIN syntax of LTL [17]. Consistency checking of each contract is performed by using *GRIC* [13]. The tool is also able of performing reactive synthesis to produce a Mealy Machine implementing the given specification. *GRIC* produces a JSON description of the state machine whenever the specification is realizable. The JSON specification is parsed by a tool we built on top of the *HIFSuite* APIs. Relying on the *HIFSuite* we can exploit its automatic C++ generation tool [5] to generate the executable model of the original specification. The task planning algorithm has been implemented in Python.

Table 1. Comparison between the time needed to obtain the final control strategy using the holistic system contracts, and decomposing the design problem. The experiments have been carried on by varying the three main dimensions of the problem.

Problem Dimension			Non-decomposed system formalization			Decomposed system formalization			
# Blocks	# Robots	# Targets	Synthesis time (s)	Code Generation (s)	Total Time (s)	Synthesis time (s)	Code Generation (s)	Simulation for Validation (s)	Total Time (s)
9	2	2	20.36	20.47	40.83	6.85	37.31	0.01	44.17
16	1	2	31.48	25.83	57.31	20.54	40.46	0.03	61.21
16	2	2	3924.18	1906.16	5831.37	33.13	244.48	0.02	277.63
16	2	4	3924.22	1910.79	5835.01	33.17	247.97	0.04	281.18
16	2	6	3924.60	1913.95	5838.55	33.17	247.96	0.07	281.20
16	3	6	Time Out (6 hours)	Time Out (6 hours)	Time Out (6 hours)	71.59	380.86	0.04	452.49
16	4	6	Time Out	Time Out	Time Out	184.35	810.34	0.05	994.74
16	4	8	Time Out	Time Out	Time Out	184.35	814.56	0.06	998.97
25	2	4	Time Out	Time Out	Time Out	55.54	292.94	0.08	348.56
25	2	6	Time Out	Time Out	Time Out	55.54	295.08	0.09	350.71
25	3	9	Time Out	Time Out	Time Out	142.79	504.25	0.12	647.16
25	4	12	Time Out	Time Out	Time Out	238.83	943.75	0.17	1182.75
25	5	15	Time Out	Time Out	Time Out	312.64	1351.49	0.21	1664.34

Finally, we developed a backend producing the final ROS implementation of the control software as described in Section 6.

A series of Python scripts has been produced to generate various scenarios of the problem. The scenarios we vary the number of deployed robots and the size of the environment. For all the scenarios, we produced the holistic (*i.e.*, non-decomposed) specification of the system to be solved by state-of-the-art techniques based on reactive synthesis [22]. Then, we generated the decomposed specification as discussed in Section 4, and we apply the proposed approach.

In Section 7.1, we compare the time necessary to obtain a valid control strategy using our approach, against the state-of-the-art reactive synthesis of the non-decomposed system specification. For each specification, we set a timeout of six hours. Then, we compare qualitatively the code generated by the two approaches.

We also evaluate the applicability of the proposed methodology by building virtual models of the scenarios we considered using the Gazebo simulator [20]. Then, we deployed the ROS code generated by our methodology to control the dynamical models of the robots instantiated in Gazebo. Then, we monitor the behavior of the system to evaluate if it meets the initial requirements, and it is thus properly controlled by the generated code. Contrary to the system-level simulation relying on the code synthesized by the contracts (*i.e.*, steps 4 and 7 of Figure 3), the simulation performed using Gazebo is not a step of the methodology. However, it allows evaluating the properties of the generated code once deployed in a real system. Section 7.2 provides the details about the Gazebo simulations.

7.1 Methodology evaluation

Table 1 reports the time required by the various design phases, *i.e.*, the realizability, synthesis, the control software generation and its execution based on the finite state machines produced by reactive synthesis tools. The different entries have been obtained by varying the three main design problem dimensions, *i.e.*, the number of blocks in the two-dimensional space representation, number of robots, and the number of targets. The *Non-decomposed formalization* columns report the time required to synthesize the control strategy from the contract representing the system “holistically”. The last four columns report the time required by applying the presented approach. In both cases, we reported the time necessary to perform reactive synthesis, code generation, and the total time required. In the case of the decomposed system formalization, we report also

Table 2. Qualitative comparison of the generated code.

Blocks	Robots	Target	Synthesized States		Number of ROS Messages	
			Non-decomp.	Decomp.	Non-decomp.	Decomp.
9	2	2	10704291	2664 * 2	15	19
16	1	2	3435704	8432	35	39
16	2	2	72938223	8432 * 2	20	24
16	2	4	72938223	8432 * 2	40	48
16	2	6	72938223	8432 * 2	55	67

the simulation time required for validating the generated code. It is worth noticing that the code generated from the non-decomposed system formalization does not require further validation, as its synthesis relies on a state-of-the-art, and proved to be correct-by-construction [4], synthesis methodology.

The state-of-the-art approach [22] using the non-decomposed system formalization suffers the computational complexity of the reactive synthesis algorithm. Thus, the synthesis process requires more than six hours for the majority of our benchmarks. The complexity rises significantly by increasing the number of robots in the system. This can be explained by the substantial number of safety invariants needed to model the set of guarantees of the holistic representation. By increasing the granularity of the grid representation, the problem's complexity rises more gradually, since most of safety invariants regarding the environment shape and size compose the set of assumptions, that are more easily managed. On the other hand, the approach proposed in this paper allows synthesizing also the instances that are otherwise intractable. In particular, it shows good scalability also when increasing the number of robots in the system, as highlighted by the experiments using the 16 and 25 blocks occupancy grids.

Notice that whenever the set of targets of the mission is changed, then it will be possible to perform only the task allocation while maintaining the previously synthesized strategies for the single robots. In this case, the time required by the task allocation procedure is negligible in comparison to the entire design flow, as the algorithm required at most 1.3 seconds in our experiments.

Table 2 provides a qualitative comparison between the code produced by our approach, and the code generated by using the non-decomposed system specification. We compare the number of states composing the synthesized Mealy Machines. Then, for each scenario we simulate the generated code using Gazebo, we monitor their behavior and we quantify it by counting the ROS messages used to control the system. It is important noticing that in ROS messages are the main software primitive.

Using the non-decomposed specification leads the generated Mealy Machines to grow exponentially. Meanwhile, using the presented approach, *i.e.*, synthesizing multiple Mealy Machines from the multiple contracts composing the decomposed system specification, allows generating smaller Mealy Machines. Thus, the proposed methodology provides a more compact implementation for the same given set of requirements. Meanwhile, the generated software requires slighter more messages to control the system. This is due to the fact that using our methodology, each robot needs to broadcast the information about the target aimed by the robot. Thus, for each target, two additional messages must be broadcast to all the robots in the system. Additional details about the Gazebo simulation are provided below.

Table 3. Results obtained by the Gazebo simulation.

Blocks	Robots	Targets	Steps	ROS Messages	Simulation Time (s)
9	2	2	3	19	20.53
16	1	2	7	39	98.21
16	2	2	4	24	34.78
16	2	4	8	48	85.67
16	2	6	11	67	110.24
16	3	6	8	52	75.27
16	4	6	7	47	60.02
16	4	8	10	66	82.86
25	2	4	12	68	104.13
25	2	6	15	87	119.44
25	3	9	17	103	133.26
25	4	12	20	124	94.86
25	5	15	19	125	70.32

Overall, this set of experiments shows that the proposed decomposition strategy allows managing systems otherwise intractable.

7.2 Software deployment: Gazebo simulation

The concrete applicability of the methodology has been evaluated by deploying the code being generated to control the mechanical models of real robots running in Gazebo [20]. In this work we use different instances of the open source robot Turtlebot3 [36]. Gazebo provides great adherence to systems physical reality, as well as integration with ROS. These features, make Gazebo simulation a widely used practice to evaluate the behavior of a robot system in absence of the final hardware systems.

Additional topics must be defined to set-up a 3-dimensional simulation of robots physical behavior. The `turtlebot3_simulation` package exposes a specific node that is controllable publishing on the already defined topic `cmd_vel`. It also provides information about the robot position in the simulated Gazebo environment through its odometry functionality. The messages of the `cmd_vel` topic are of *Twist* type, that expresses both linear and angular velocity through vectors. Moreover, the odometry messages are constructed by a pose point with x , y and z components, together with a quaternion that models the actual orientation of the robot.

For each robot, we create a new node to instantiate the control strategy generated by the presented approach for the robot, and to interface the generated control strategy with the robot hardware. First, cell positions are mapped into a range of x and y coordinates. Then, since the contract-based specification does not take into account the rotation of the robot, the `turtlebot3` has to be rotated to face the direction of the target cell before moving: from odometry messages the destination position and rotation can be calculated using traditional algebraic formulas. More specifically, the angle facing the target is determined in radians by calling the $\text{atan2}(x, y)$ function. Moreover, the actual Euler rotation angle of the robot is collected by its odometry module, applying the `euler_from_quaternion` function to the rotation quaternion sent on the odometry topic.

Figure 5 shows the simulation environment emulating the case study used throughout the paper. We monitored the Gazebo simulation of different scenarios used to evaluate the methodology. For each scenario we used one of the possible initial conditions. Table 3 reports for each scenario, the

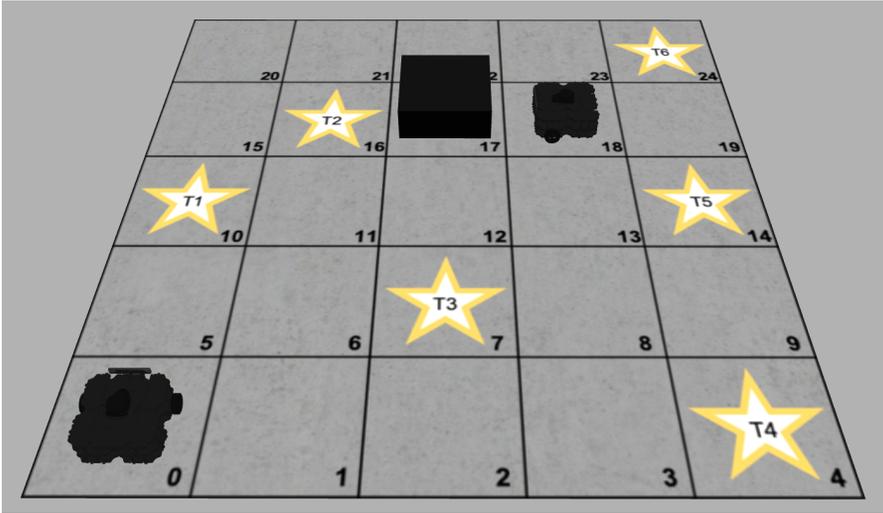


Fig. 5. The case study analyzed in this paper represented in a Gazebo simulation. Through publisher/subscriber architecture, each robot subscribes for messages from the environment controller, which publishes the set of sensor values and each agent position. A robot command is then produced by the implemented controller, wrapped in another ROS node.

number of steps performed by the robots (*i.e.*, the sum of commands generated by the control strategies of all the robots), the number of ROS messages passing in the system, the time needed to simulate the scenario in Gazebo. In all the monitored cases, the system is implementing the requirements without violating any constraint. The simulation time using Gazebo is many order of magnitude higher than the system-level simulation used for validation (Table 1). This is due to the fact that Gazebo simulates every detail of the systems’ physical behavior. Meanwhile, the system-level simulation emulates the details interesting the control strategies of the different robots, while relying on a coarse abstraction of the system’s kinematics. It is also worth noticing that the time required by Gazebo to simulate does not depend on the scenario’s parameters. The time is affected by the parallelism due to the presence of multiple robots, as well as the equations to be solved for emulating the dynamical models.

In conclusion, the proposed decomposition allows producing more compact ROS code, able to fulfill the initial design requirements, while slightly increasing the number of messages.

8 CONCLUSIONS

In this paper, we proposed an approach to exploit assume-guarantee reasoning to decompose, and make more efficient, the design of robotic control software. The approach decomposes the main design problem into multiple sub-problems, formalized as A/G contracts. Then, the composition of the sub-problems solutions is validated through simulation to verify that the solution implements the initial requirements. The experimental results show the advantage of decomposing the design problem, highlighting a substantial reduction of the required design time.

ACKNOWLEDGMENTS

This work has been supported by the following grants: the Singaporean National Robotics Programme (NRP) – *Robotics Enabling Capabilities and Technologies* (RECT), Grant no. 172 25 00022;

the Singaporean Ministry of Education (MoE), Grant no. MOE2018-T2-1-098; the Italian Ministry of Education, University and Research (MIUR) “Dipartimenti di Eccellenza” 2018-2022 grant.

REFERENCES

- [1] Houssam Abbas, Indranil Saha, Yasser Shoukry, Rudiger Ehlers, Georgios Fainekos, Rajesh Gupta, Rupak Majumdar, and Dogan Ulus. 2018. Special Session: Embedded Software for Robotics: Challenges and Future Directions. In *Proceedings of the International Conference on Embedded Software, (EMSOFT) 2018*.
- [2] Nikunj Bajaj, Pierluigi Nuzzo, Michael Masin, and Alberto Sangiovanni-Vincentelli. 2015. Optimized selection of reliable and cost-effective cyber-physical system architectures. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. EDA Consortium, 561–566.
- [3] Albert Benveniste et al. 2018. Contracts for system design. *Foundations and Trends® in Electronic Design Automation* 12, 2-3 (2018), 124–400.
- [4] Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Yaniv Saar. 2012. Synthesis of reactive (1) designs. *J. Comput. System Sci.* 78, 3 (2012), 911–938.
- [5] Nicola Bombieri, Giuseppe Di Guglielmo, Michele Ferrari, Franco Fummi, Graziano Pravadelli, Francesco Stefanni, and Alessandro Venturelli. 2010. HIFSuite: Tools for HDL code conversion and manipulation. *EURASIP Journal on Embedded Systems* 2010, 1 (2010), 436328.
- [6] Davide Bresolin, Luigi Di Guglielmo, Luca Geretti, Riccardo Muradore, Paolo Fiorini, and Tiziano Villa. 2012. Open problems in verification and refinement of autonomous robotic systems. In *Proceedings of Euromicro Digital System Design (DSD) 2012*. IEEE, 469–476.
- [7] H. Bruyninckx. 2001. Open robot control software: the OROCOS project. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, Vol. 3. IEEE, 2523–2528. <http://ieeexplore.ieee.org/document/933002/>
- [8] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2003. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM)* 50, 5 (2003), 752–794.
- [9] Ankush Desai, Indranil Saha, Jianqiao Yang, Shaz Qadeer, and Sanjit A Seshia. 2017. Drona: A Framework for Safe Distributed Mobile Robotics. In *2017 ACM/IEEE 8th International Conference on Cyber-Physical Systems (ICCPs)*. IEEE, 239–248.
- [10] Rüdiger Ehlers and Vasumathi Raman. 2016. Slugs: Extensible GR(1) Synthesis. In *Proceedings of International Conference on Computer Aided Verification (CAV)*. Springer, 333–339.
- [11] Alberto Elfes. 1989. Using occupancy grids for mobile robot perception and navigation. *Computer* 6 (1989), 46–57.
- [12] Yishai A Feldman and Henry Broodney. 2016. A cognitive journey for requirements engineering. In *INCOSE International Symposium*, Vol. 26. Wiley Online Library, 430–444.
- [13] Ioannis Filippidis, Sumanth Dathathri, Scott C Livingston, Necmiye Ozay, and Richard M Murray. 2016. Control Design for Hybrid Systems with TuLiP: The Temporal Logic Planning Toolbox. In *Proceedings of IEEE Conference on Control Applications (CCA) 2016*. 1030–1041.
- [14] C. Finucane and H. Kress-Gazit. 2010. LTLMoP: Experimenting with language, Temporal Logic and robot control. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 1988–1993.
- [15] Franco Fummi, Michele Lora, Francesco Stefanni, Dimitrios Trachanis, Jahn Vanhese, and Sara Vinco. 2014. Moving from Co-Simulation to Simulation for Effective smart systems design. In *Proceedings of IEEE/ACM Design Automation and Test in Europe (DATE) 2014*. 286.
- [16] Brian P Gerkey and Maja J Matarić. 2004. A formal analysis and taxonomy of task allocation in multi-robot systems. *The International Journal of Robotics Research* 23, 9 (2004), 939–954.
- [17] G.J. Gerard J Holzmann. 1997. The model checker SPIN. *IEEE Transactions on Software Engineering* 23, 5 (1997), 279. <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=588521>
- [18] Pablo Iñigo-Blasco, Fernando Diaz-del Rio, Ma Carmen Romero-Tertero, Daniel Cagigas-Muñiz, and Saturnino Vicente-Diaz. 2012. Robotics software frameworks for multi-agent robotic systems development. *Robotics and Autonomous Systems* 60, 6 (jun 2012), 803–821.
- [19] Kurt Keutzer, A Richard Newton, Jan M Rabaey, and Alberto Sangiovanni-Vincentelli. 2000. System-Level Design: Orthogonalization of Concerns and Platform-Based Design. *IEEE transactions on computer-aided design of integrated circuits and systems* 19, 12 (2000), 1523–1543.
- [20] N. Koenig and A. Howard. 2004. Design and Use Paradigms for Gazebo, An Open-Source Multi-Robot Simulator. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robotics and Systems (IROS)*, Vol. 3. IEEE, 2149–2154. <http://ieeexplore.ieee.org/document/1389727/>
- [21] Hadas Kress-Gazit. 2011. Robot challenges: Toward development of verification and synthesis techniques. *IEEE Robotics & Automation Magazine* 18, 3 (2011), 22–23.

Compositional Design of Multi-Robot Systems Control Software on ROS

- [22] Hadas Kress-Gazit, Georgios E Fainekos, and George J Pappas. 2009. Temporal-logic-based reactive mission and motion planning. *IEEE transactions on robotics* 25, 6 (2009), 1370–1381.
- [23] Daniel Kroening, Alex Groce, and Edmund Clarke. 2004. Counterexample guided abstraction refinement via program execution. In *International Conference on Formal Engineering Methods*. Springer, 224–238.
- [24] Michele Lora, Sara Vinco, and Franco Fummi. 2019. Translation, Abstraction and Integration for Effective Smart System Design. *IEEE Trans. Comput.* (2019).
- [25] S. Maniopoulos, P. Schillinger, V. Pong, D. C. Conner, and H. Kress-Gazit. 2016. Reactive high-level behavior synthesis for an Atlas humanoid robot. In *IEEE International Conference on Robotics and Automation (ICRA)*. 4192–4199.
- [26] J. W. McDonald Hayhurst and D. C. Conner. 2018. Towards Capability-Based Synthesis of Executable Robot Behaviors. In *SoutheastCon 2018*. 1–8.
- [27] Amir Pnueli Nir Piterman and Yaniv Saar. 2006. Synthesis of Reactive(1) Designs. In *Proceedings of VMCAI 2006*.
- [28] P. Nuzzo, A. Iannopolo, S. Tripakis, and A. Sangiovanni-Vincentelli. 2014. Are interface theories equivalent to contract theories?. In *2014 Twelfth ACM/IEEE Conference on Formal Methods and Models for Codesign (MEMOCODE)*. 104–113.
- [29] Pierluigi Nuzzo, Michele Lora, Yishai A Feldman, and Alberto Sangiovanni-Vincentelli. 2018. CHASE: Contract-based requirement engineering for cyber-physical system design. In *Proceedings of IEEE/ACM Design Automation and Test in Europe (DATE) 2018*. 839–844.
- [30] Pierluigi Nuzzo and Alberto L Sangiovanni-Vincentelli. 2018. Hierarchical System Design with Vertical Contracts. In *Principles of Modeling*. Springer, 360–382.
- [31] Pierluigi Nuzzo, Alberto L Sangiovanni-Vincentelli, Davide Bresolin, Luca Geretti, and Tiziano Villa. 2015. A platform-based design methodology with contracts and related tools for the design of cyber-physical systems. *Proc. IEEE* 103, 11 (2015), 2104–2132.
- [32] Pierluigi Nuzzo, Alberto L Sangiovanni-Vincentelli, and Richard M Murray. 2015. Methodology and Tools for Next Generation Cyber-Physical Systems: The iCyPhy Approach. In *INCOSE International Symposium*, Vol. 25. Wiley Online Library, 235–249.
- [33] Open Source Robotics Foundation (OSRF). 2018. Gazebo Simulator. (2018). <https://www.openrobotics.org/>
- [34] Roberto Passerone, Imene Ben Hafaiedh, Susanne Graf, Albert Benveniste, Daniela Cancila, Arnaud Cuccuru, Sebastien Gerard, Francois Terrier, Werner Damm, Alberto Ferrari, et al. 2009. Metamodels in Europe: Languages, tools, and applications. *IEEE Design & Test of Computers* 26, 3 (2009), 38–53.
- [35] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. 2009. ROS: an open-source Robot Operating System. In *IEEE International Conference on Robotics and Automation (ICRA)*, Vol. 3. Kobe, Japan, 5.
- [36] Robotis. 2017. Turtlebot3. (2017). <http://www.robotis.us/turtlebot-3/>
- [37] Roni Rosner. 1991. Modular Synthesis of Reactive Systems. *PhD thesis. The Weizmann Institute of Science* (1991).
- [38] S. Spellini, M. Lora, S. Chattopadhyay, and F. Fummi. 2018. Work-in-Progress: Introducing Assume-Guarantee Contracts for Verifying Robotic Applications. In *2018 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. 1–2.
- [39] K. W. Wong and H. Kress-Gazit. 2017. From High-Level Task Specification to Robot Operating System (ROS) Implementation. In *2017 First IEEE International Conference on Robotic Computing (IRC)*. 188–195.