A Contract-based Approach to Designing an Autonomous Taxi System

Yuening Zhang, Vasumathi Raman, Samira Farahani and Richard Murray

Abstract— **Contract-based methodology emphasizes decomposing a complex problem into more manageable sub-problems by using contracts to define assume-guarantee relationships between the components in a system. Contract-based approaches have been proven successful in several applications such as aircraft electric power system and automotive applications. However, there has not yet been a generalized contract-based design framework for robotic problems. In this paper, a control system for an autonomous taxi center is designed and we propose a contract-based platform architecture for the design structure of this system. This structure enables independent implementation of system topology (i.e. interconnection among elements) and control protocol by using a contract-based approach. The architecture includes a centralized top-level dispatcher, middle-level route planner and a low-level navigation planner and the specifications are expressed using the formalisms of linear temporal logic, signal temporal logic ~~and arithmetic constraints on Boolean variables~~. We synthesize controllers for each platform and the platforms are interconnected to transfer necessary information among each other, while obeying their assume-guarantee contracts. The simulation results show the effectiveness of contract-based method in promoting problem segregation and supporting platform-based architecture.**

I. Introduction

In recent years, there is increasing interest in the field of robotics and automaton. Researchers have developed robotic applications that fulfill many complicated tasks [1], [2]. The control systems of those applications are designed using the state-of-art methods by groups of expert researchers. As the control systems become more complex, system design focuses more on having a decomposed design structure. The reasons are manifold: It can be hard for a group of designers to design a controller that encompasses all aspects of the problem. More often, a large project requires collaboration of several teams in charge of different aspects of the problems. It is better to limit the influence of a mistake within the local controller, or even better, to have a correct-by-construction system in the first place. Furthermore, the limitations of the existing automatic synthesis tools, including their vulnerability to combinatorial explosion and the limited types of models and specifications they allow, force the designers to divide the problems to fit the applicable range of different tools. Some decomposed design structures that are explored include layered design, component-based design, V-model design and model-based design[3].

In this paper, we propose contract-based design for autonomous robotic system. Contract-based methodology can facilitate the design of complex or distributed cyber-physical systems by defining the interconnection of the components. It can potentially reduce the cost for re-design, thus shortening the development cycle, and promote re-use and optimization of functionalities. Among the different interpretations of contracts, a more recent and active one is the contract theory based on assume-guarantee relationships. [4] described the contract-based design in cyber-physical system and [5] successfully applied the method in aircraft electric power system design. Combined with platform-based design [6], a design methodology for decomposing the system, contract-based design has been given increased attention because of its successful application in many embedded system domains, including automotives [7] and integrated circuit [8].

However, there has not yet been a generalized contract-based design framework for many robotics problems and the state-of-the-art robotics design today are mostly specialized for the problems. In this paper, we use the contract-based design method to design an autonomous taxi system in an effort to establish a contract-based framework to be used in distributed autonomous vehicle systems. We follow the platform-based paradigm to decompose the system and use contracts to define the legal composition of the different components of the system. We try to use existing computational tools and communication platforms to implement a problem, in order to promote utilizing the pre-existing modules in system design to ease design difficulties of robotics problems.

II. Autonomous Taxi System

We look at the problem of designing an autonumous taxi system. The applications of similar systems can be used in hotel shuttle services.

A. Problem Formulation

A taxi center is in charge of carrying passengers. The taxi center has multiple taxis that can be assigned to different requests. Each taxi needs to pick up the assigned passengers at the requested locations and drop them off at the destination. Each request corresponds to a specific location for pick up. In our examples, all passengers will eventually go to the same destination for simplicity of the problem, but there is no conceptual barrier to extending this to different destinations. Each taxi can take more than one passengers at a time until it reaches its capacity constraint.

B. System Requirements

Given the requests from the passengers, the goal of the autonomous taxi system is to accommodate those requests, i.e. assign taxis to pick up the passengers and send them to the designated destination. While designing the system, the following requirements should be considered.

1) Service: Service specifications specify that under all possible requests that are allowed, the taxi center should eventually serve those requests.

2) Optimality: Optimality specifications specify that the system should pick the optimal solution among all the solutions that satisfy the service requirements. For example, the center should use the least number of taxis possible, or should assign the closest taxi to the pick-up location, or the taxi should take the shortest path to save fuel.

3) Safety: Safety specifications specify that the taxis should avoid obstacles in the path and the multiple taxis should not crash into each other.

III. Contract-based Design Methodology

A contract, used in the context of decomposed design structures, specifies the assume-guarantee composition of ~~components~~ of the system. The set of contracts restrict the legal composition of different components of a system. The essence of contract-based method is therefore based on decomposing a complex problem into more manageable sub-problems while making sure that the solutions can be recomposed. Contract-based design allows setting up the design structure at an early stage, and the design can then progress through each component smoothly. A generic use is to merge contract-based design with platform-based design, which supports a decomposed design structure that combines both component-based design and layered design.

Platform-based design provides a meet-in-the-middle process where top-down refinements of the specifications are mapped onto bottom-up implementations of the behavior [6]. A platform is an abstraction layer with a library of components to be utilized, which includes computational blocks to carry out computation and give performance estimation of the platform abstraction, and communication components to ensure the correct interconnections of the platforms and provide interface to these sub-systems.

A platform hides several possible refinement and implementations of the specifications. There are two ways to separate the platforms: dividing a task horizontally into multiple sub-tasks, and refining a high-level task vertically into lower-level layers. Contracts serve as the communication components that define how these platforms interact with each other. Figure 1 shows a sample platform-based architecture of the taxi problem.

A detailed explanation for the elements in contract-based design and platform-based design is given in [4]. The platform-based method using contracts will improve the design in several ways: 1) It achieves modularity for different functionalities. 2) It improves the scalability of the problem with finer division of the tasks.

IV. Platform-based Flow for Autonomous Taxi System Design Using Contracts

In the platform-based paradigm, the design structure is decomposed into a platform-based architecture, with hierarchical layering and one or more components at each layer. The establishment of the architucture is set up with respect

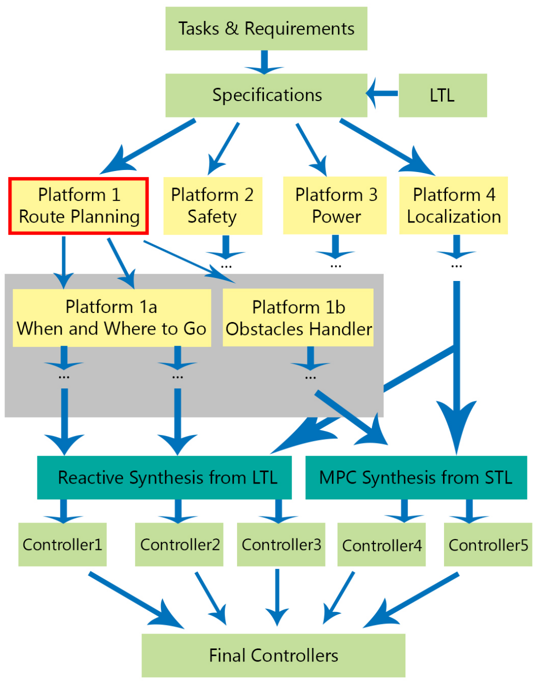


Fig. 1. Sample platform-based architecture for motion planning problems

to the functionality of each platform, separability of the platforms, and the library tools available at each platform. Platforms can be divided for distinct functionalities. For example, a strategic route planner can be naturally separated from hybrid controller that control the movements of a vehicle. The separability of the platforms, or more specifically, whether separating the functionality of a task into sub-tasks will introduce shared variables in these low-level platforms, and whether there are compatible contracts among these platforms, are other important factors to consider for system decomposition, especially during refinements of a high-level platform. As library tools may be different, seeing that there is an available computational tool for a platform can also be considered as a guideline for separating the platforms. This establishes the topology of the system.

Once the system topology is established, the design progresses through each component to carry out controller design. Controllers are designed for each platform while satisfying the boundary conditions of the contract of that platform. We express the control system specifications in the formalism of temporal logic. Temporal logic captures the timing aspect of a problem, and has many branches such as linear temporal logic (LTL) [9] or signal temporal logic (STL) [10]. For the purpose of our paper, we utilized some formal synthesis methods that solve temporal logic specifications. The reason for using synthesis tools is that it eliminates the need for manually designing the controller, and provides an easier way for the designers to apply the state-of-the-art tools. While there can be different behaviors of a platform that all satisfy the specifications, controller design gives one acceptable implementation of the problem, and in some cases with optimality guarantees.

A. *Reactive Control Synthesis for LTL Specifications*

Reactive control refers to control protocols that constantly adjust according to environment states. The LTL specification for a reactive control protocol takes the form of $\psi = (\psi_e \rightarrow \psi_s)$, where ψ_e is the environment specification and ψ_s is the system specification. Given a contract (A, G), the assumption A can be specified in environment specification ψ_e and system guarantee G can be specified in system specification ψ_s.

Application area: LTL specifications considers the ordering of events. LTL operators include "always" $\square$, "eventually" $\diamondsuit$, "next" $\bigcirc$, "until" $\mathcal{U}$, and their combinations. The design space for using LTL specifications grows exponentially. However, GR(1) [12], a subset of LTL, can be solved in polynomial time. GR(1) specifications allow three types of specifications: initial condition ψ_0, which is satisfied at initial state, safety condition $\square\psi$, which is satisfied at all time, and progress condition $\square\diamondsuit\psi$, which is satisfied infinitely often. The reactive synthesis for specification $\psi_e \rightarrow \psi_s$ treats the environment and the system as opponents that take steps in turn. Therefore, it can solve finite-state automata for two-player games, or more generally, discrete strategy planning problems.

Available tools: Temporal logic planning toolbox (TuLiP) [13] uses a collection of tools to solve reactive control system that is correct-by-construction. It considers discrete systems, or alternatively, it takes a continuous system and discretize it using proposition preserved partition. The output is a Mealy machine that provides a strategy for satisfying the specification in uncertain environments.

B. *Optimization-based Reactive Synthesis Using MILP*

Reactive synthesis for STL specifications can be solved as a series of optimization problems using mixed-integer linear encoding [11]. STL can specify the absolute timing of events, meaning the time bound for which the specifications need to be satisfied. The use of model predictive control (MPC) [11] can observe the current plant at every timestep and compute the controller signal up to a horizon H such that the specifications are satisfied within every time bound.

Application area: STL specifications include operators such as "always" $\square$, "eventually" $\diamondsuit$, "until" $\mathcal{U}$ with a time bound $[t_0, t_1]$. This method guarantees optimality of the solution and is suitable for solving real-time problems with timing constraints.

Available tools: BluSTL [14] is developed as a Matlab toolbox that solves STL specifications. Given the system dynamics, STL specifications and the cost function, it yields the control sequence according to the environment input during simulation.

V. System Topology Design

Unfortunately, there is no automatic synthesis tool for topology design that automatically breaks the problems into pieces and gives the topology of the system. Therefore, deriving the topology of the system depends on the creativity of the designer.

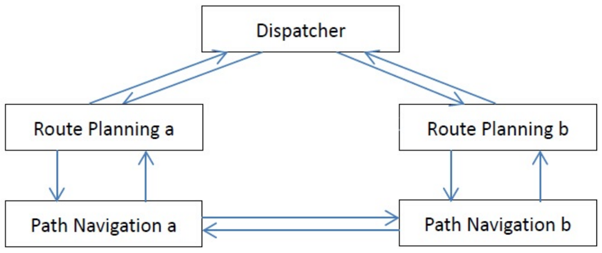


Fig. 2. Platform architecture for taxi center problem

A proposed design architecture is shown in Figure 2, which divides the planner into strategic level and navigation level. Strategically, the taxi center system should have a dispatcher to assign taxis to requests and route planners to plan the route of the taxis so that all the passengers assigned to them are picked up and eventually dropped off. The navigation level navigates the taxis along the designated route.

Considering functionality, the dispatcher is a centralized controller whose main task is to monitor the distributed system of multiple taxis, and the route planners are local controllers for the taxis. The navigation planner takes the dynamics of the car into consideration and provides the actual controller signal. The functionality of a platform with respect to the whole system will be represented as the contract of the platform. The allowed operation environment and the tasks or specifications that the platform will satisfy becomes the assumption A and the guarantee G respectively in the assume-guarantee relation (A, G) of the contract.

With respect to separability, the dispatcher does not need to know the details of how the routes are planned, and the route planner does not need to know the information about other taxis. This significantly reduces the design space for the sub-systems. There may be, however, some shared variables between the platforms. For example, variables that indicate the current status of a taxi, e.g. being full, may be controlled by a local controller such as route planner, and passed into a centralized controller such as dispatcher. The dispatcher also needs to pass the variable to the route planner that indicates which taxi is assigned to which request. Therefore, the contracts for different platforms follow that the behavioral guarantee of one platform will be captured by other platforms that use the same shared variables in its environment assumption. A real-valued problem below shows an example pattern in which a high-level contract C has two output variables y, z shared with the input of two lower-level conrtacts C_1, C_2:

Example 5.1:

$$C\begin{cases}\text{variables}: \begin{cases}\text{input}: x\\ \text{output}: y, z\end{cases}\\ \text{type}: x, y, z \in \mathbb{R}\\ \text{assume}: x > 0\\ \text{behavior}: y = f_1(x)\\ \qquad\qquad z = f_2(x)\end{cases}$$

$$C_1\begin{cases}\text{variables}: \begin{cases}\text{input}: y\\ \text{output}: p\end{cases}\\ \text{type}: y, p \in \mathbb{R}\\ \text{assume}: y \in \text{range of } f_1(x)\\ \text{behavior}: p = g(y)\end{cases}\quad C_2\begin{cases}\text{variables}: \begin{cases}\text{input}: z\\ \text{output}: q\end{cases}\\ \text{type}: z, q \in \mathbb{R}\\ \text{assume}: z \in \text{range of } f_2(x)\\ \text{behavior}: q = h(z)\end{cases}$$

Considering library tools, the computational blocks of a platform may solve problems in one domain but not the other domains. Therefore, each platform should limit its functionality within the use of its computational blocks. As strategic level mainly concerns discrete state space problems and navigation level mainly concerns continuous state space problems, they may need different platforms that have computational blocks that target at their corresponding problems.

As a result of the platform decomposition, we give contracts for these platforms as below. ~~The dispatcher should take requests from passengers, considers the availabilities of the taxis, and at some point in time assigns a taxi to each of the requests. The route planner receives taxi assignment from dispatcher, considers if the taxi is ready to make the next move, and specifys the next target location for the taxi such that in the end, all the passengers are picked up and dropped off. It also monitors the availability status of the taxi. The path navigation receives target location from route planner and executes the route. Written in contract form,~~

$$C_d\begin{cases}\text{variables}: \begin{cases}\text{input}: \text{requests, taxi availability}\\ \text{output}: \text{taxi assignment}\end{cases}\\ \text{type}: \text{all in Boolean}\\ \text{assumption}: \text{A passenger will keep requesting until}\\ \qquad\qquad \text{a taxi is assigned.}\\ \qquad\qquad \text{Each taxi is not always full.}\\ \text{behavior}: \text{An available taxi will eventually be}\\ \qquad\qquad \text{assigned to a request.}\end{cases}$$

$$C_p\begin{cases}\text{variables}: \begin{cases}\text{input}: \text{taxi assignment, ready}\\ \text{output}: \text{target location, taxi availability}\end{cases}\\ \text{type}: \text{all in Boolean}\\ \text{assumption}: \text{When t[illegible]s full, there will be no new}\\ \qquad\qquad \text{request assigned.}\\ \text{behavior}: \text{Determine whether the taxi is full.}\\ \qquad\qquad \text{Specify the next target location, or the}\\ \qquad\qquad \text{route that satisfies requests.}\end{cases}$$

$$C_n\begin{cases}\text{variables}: \begin{cases}\text{input}: \text{target location}\\ \text{output}: \text{ready}\end{cases}\\ \text{type}: \text{all in Boolean}\\ \text{assumption}: \text{No new target location will be assigned}\\ \qquad\qquad \text{until the taxi reaches the current target.}\\ \text{behavior}: \text{Navigate the taxi to target location.}\\ \qquad\qquad \text{Signal when the taxi is ready.}\end{cases}$$

VI. System Controller Design

With the system topology established, we synthesize the controllers for the platforms. For each platform, we consider its contract and specify the behavior of the platform using temporal logic specifications.

A. Dispatcher

The dispatcher considers the strategy of the taxi center, and therefore we use TuLiP to synthesize the controller. Using GR(1) specifications, we provide a refinement of the contract by specifying the environment variables, system variables, environment specfications and system specifications as follows:

Environment variables: The environment input includes *request$_i$* for $i \in \{1, 2, ...n\}$, which indicates the n possible requests from the passengers, and *full$_j$* for $j \in \{1, 2, ...m\}$, which indicates the availability of the m taxis of the taxi center. Since here we only monitor if a taxi is full or not, we use "full" instead of "available". However, it is possible to use "available" in the future to capture all possible failure of the taxi. If *full* is true, then the taxi is not available.

System variables: The system output includes *taxi$_{j,i}$* for $j \in \{1, 2, ...m\}$ and for $i \in \{1, 2, ...n\}$, where *taxi$_{j,i}$* being true indicates taxi *j* is assigned to request *i*.

Environment specifications: The assumption that a passenger will keep requesting until a taxi is assigned can be specified as

$$\bigwedge_n \square\left(request_i \wedge \neg \bigvee_m taxi_{j,i} \Rightarrow \bigcirc request_i\right)$$

The assumption that each taxi is not always full can be specified as

$$\bigwedge_m \square\left(\neg full_j \wedge \bigwedge_n \neg taxi_{j,i} \Rightarrow \bigcirc \neg full_j\right)$$

and

$$\bigwedge_m \square\Diamond\neg full_j$$

The assumption on full only specify a partial transitional relation, regardless of what the maximum number of people each taxi can take in reality and how the number is counted.

System specifications: To ensure that every request will get a taxi, we specify LTL specification

$$\bigwedge_{n} \Box \left(request_i \Rightarrow \Diamond \bigvee_{m} taxi_{j,i} \right)$$

which can be transformed into GR(1) specifications.

We also want to eliminate unreasonable behaviors such as assigning taxis when there is no request, assigning unavailable taxis, and assigning more than one taxi to a request; therefore, we write

$$\bigwedge_{n} \Box \left(\neg request_i \Rightarrow \bigwedge_{m} \neg taxi_{j,i} \right)$$

$$\bigwedge_{m} \Box \left(full_j \Rightarrow \bigwedge_{n} \neg taxi_{j,i} \right)$$

$$\bigwedge_{n} \bigwedge_{m} \Box \left(taxi_{j,i} \Rightarrow \bigwedge_{m \text{ except j}} \neg taxi_{j,i} \right)$$

The variables $taxi_{j,i}$ for $i \in \{1, 2, ..., n\}$ are defined to be mutex.

B. Route planner

In the same way, we use TuLiP to synthesize the controller for route planner. For our paper, we simply abstract the route to be in one of the pick-up locations. However, given a continuous map, we can use TuLiP to compute an "overapproximate" discrete abstraction of the map that is reasonably divided based on vehicle dynamics and properties, but assuming no obstacles. This alleviates the burden of the navigation level as the navigation level should be guaranteed success unless it a serious obstacle comes up. While contracts define the input and output variables, there may be other local variables used to help specify the behavior of the platform.

Environment variables: The environment input of the route planner for taxi *j* are *request*$_i$ for $i \in \{1, 2, ..., n\}$, which takes the output value of *taxi*$_{j,i}$ for $i \in \{1, 2, ..., n\}$ from the dispatcher, and *ready* from the path navigation platform, which indicates if the taxi has reached the current target location and is ready to receive a new target.

System variables: The system output *location*$_i$ for $i \in \{1, 2, ..., n\}$, corresponding to the pick-up locations for the n requests, and *des* indicate where the next target location is. *full* is another system output passed into dispatcher as *full*$_j$. Assume the maximum number of requests that the taxi can take is *l*, then there are local system variables including *wait*$_{k,i}$ for $k \in \{1, 2, ..., l\}$ and $i \in \{1, 2, ..., n\}$ and *seat*$_{k,i}$ for $k \in \{1, 2, ..., l\}$ and $i \in \{1, 2, ..., n\}$ that are used to indicate the two phases of the assigned requests: if *request*$_i$ has been assigned but the passenger is still waiting to be picked up, and if the passenger has been picked up and is seated in the taxi until dropped off at the destination.

Environment specifications: We first specify the assumption on $ready$ to avoid when $ready$ is never true and the goal is impossible to be satisfied,

$$\Box \Diamond ready$$

The assumptions on assignment that there will not be a new request assignment when the taxi is full, and there is only one request at a time are specified as

$$\Box \left(full \Rightarrow \bigcirc \bigwedge_{n} \neg request_i \right)$$

$$\bigwedge_{n} \Box \left(request_i \Rightarrow \bigwedge_{n \text{ except } i} \neg request_i \right)$$

System specifications: We first specify that the assigned requests cannot be in both phases. Then, the variable $full$ is defined as all assigned requests are in either of the two phases,

$$\Box \left(full \Leftrightarrow \bigwedge_{l} \left(\bigvee_{n} wait_{k,i} \vee \bigvee_{n} seat_{k,i} \right) \right)$$

When the taxi is not ready, we specify that the target location should not change.

$$\bigwedge_{n} \Box \left(location_i \Rightarrow \bigcirc \left(\neg ready \Rightarrow location_i \right) \right)$$

To make sure one and only one passenger seat is given to each request, we specify the following together with additional initial conditions:

$$\bigwedge_{n} \Box \left(\bigcirc request_i \Rightarrow \bigvee_{l} \left(\neg wait_{k,i} \wedge \bigcirc wait_{k,i} \right) \right)$$

$$\bigwedge_{n} \bigwedge_{l} \Box \Bigg(\neg wait_{k,i} \wedge \bigcirc wait_{k,i} \Rightarrow \bigwedge_{l \text{ except } k} \left(\neg wait_{k,i} \Rightarrow \bigcirc \neg wait_{k,i} \right) \Bigg)$$

The rest of the transition relations of $wait_{k,i}$ and $seat_{k,i}$ are specified. As an example, the specifications on $wait_{k,i}$ are given as follows,

$$\bigwedge_{l} \bigwedge_{n} \Box \left(wait_{k,i} \Rightarrow \bigcirc \left(\neg location_i \Rightarrow wait_{k,i} \right) \right)$$

$$\bigwedge_{l} \bigwedge_{n} \Box \left(wait_{k,i} \Rightarrow \bigcirc \left(location_i \Rightarrow \neg wait_{k,i} \wedge seat_{k,i} \right) \right)$$

$$\bigwedge_{l} \Box \left(\bigwedge_{n} \neg wait_{k,i} \Rightarrow \bigcirc \left(\bigwedge_{n} \neg request_i \Rightarrow \bigwedge_{n} \neg wait_{k,i} \right) \right)$$

In the end, we want to specify that eventually the taxis go to the required locations to finish the requests,

$$\bigwedge_n \Box \left(\bigvee_l wait_{k,i} \Rightarrow \Diamond location_i \right)$$

$$\Box \left(\bigvee_l \bigvee_n seat_{k,i} \Rightarrow \Diamond des \right)$$

The variables $wait_{k,i}$ for $i \in \{1, 2, ..., n\}$, variables $location_i$ for $i \in \{1, 2, ..., n\}$ and des, variables $seat_{k,i}$ for $i \in \{1, 2, ..., n\}$ are defined to be mutex. Also, there should always be one and only one $location_i$ or des that is true.

C. Path navigation

Path navigation considers the dynamics of the car, and navigates the taxi to follow the route designated by route planner. BluSTL can be used for controller synthesis for this platform. For the purpose of this paper, we consider the following simple dynamics,

$$\dot{x} = u_1$$
$$\dot{y} = u_2$$

where u_1 and u_2 are controller signals that control the velocity of the taxi in two directions x and y. Since BluSTL allows synthesis for linear, switched linear models, we can use any exact or approximate dynamics that fit these categories (e.g. piecewise affine approximation of Dubins dynamics).

In BluSTL, the system variables at time *t* are captured by state variables *X* and input variables *U*, and environment variables are captured by disturbance signal *W*. While we can plan the entire strategy beforehand, when considering the navigation level, it is more realistic to synthesize the controller on-line. In the same way, we specify environment variables, system variables, environment specifications and system specifications.

Environment variables: The disturbance signal includes the Boolean variable *close(t)* $\in$ *W* that indicates if at time *t*, the sensor on the taxi detects a close object, and real-valued *xpos(t)* $\in$ *W* and *ypos(t)* $\in$ *W*, which indicate x and y position of the other taxis.

System variables: The system variables are $x(t)$, $y(t)$ $\in X$ and $u_1(t)$, $u_2(t) \in U$.

Environment specifications: In adversarial setup, we can specify the disturbance signal to stay within certain bounds below and above a reference value. In our case, the environment input is deterministic in real time. Therefore, we do not need environment assumption, or the environment assumption is true.

System specifications: We first specify that the taxi does not exceed an environment boundary, written in STL specification,

$$\Box_{[0,\text{Inf}]} \left(x(t) > x_{\text{min}}(t) \wedge x(t) < x_{\text{max}}(t) \wedge y(t) > y_{\text{min}}(t) \wedge y(t) < y_{\text{max}}(t)\right)$$

Then, we specify avoiding the static obstacles for safety requirement,

$$\Box_{[0,\text{Inf}]} \left(x(t) < x_{\text{min}}(t) \vee x(t) > x_{\text{max}}(t) \vee y(t) < y_{\text{min}}(t) \vee y(t) > y_{\text{max}}(t)\right)$$

To satisfy the realizability requirement, when the target location is $location_i$ or des, we specify that

$$\Box_{[0,\text{Inf}]} \Diamond_{[0,t_0]} \left(x(t) > x_{\text{min}}(t) \wedge x(t) < x_{\text{max}}(t) \wedge y(t) > y_{\text{min}}(t) \wedge y(t) < y_{\text{max}}(t)\right)$$

In the end, in order to avoid other taxis, we use a buffer value e that is slightly greater than the radius of the vehicle, and specify that

$$\Box_{[0,\text{Inf}]}(close(t) > 0.5 \Rightarrow x(t) < xpos(t) - e \vee x(t) > xpos(t) + e \vee y(t) < ypos(t) - e \vee y(t) > ypos(t) + e)$$

D. Communication Platform

After synthesizing controllers for all components, the controller need to be connected. The contracts have specified the ports for different controllers, and during implementation, there are two major problems to be considered: which communication platform can host the communication between the components, and how to control the synchronous or asynchronous communication between the components.

In our case, because the distributed taxi systems run concurrently, Robotic Operating System (ROS) [15] can serve as the communication platform. We examine here how ROS can be used to integrate the controllers synthesized by different synthesis tools.

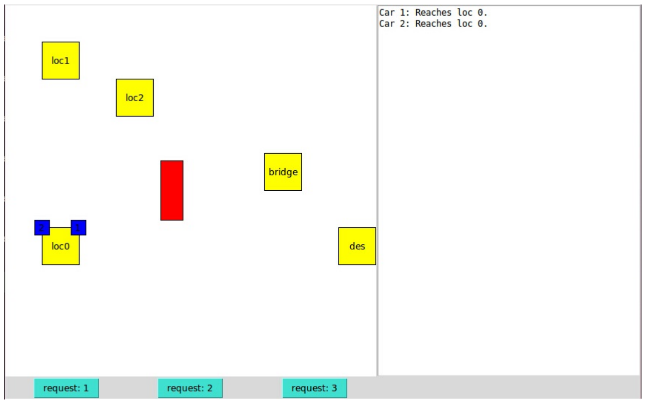
TuLiP and BluSTL give different forms of controller output, and ROS interfaces for these two are rospy, which is the python API of ROS, and Matlab package Robotic System Toolbox [16] that allows easy interface with ROS, respectively.

ROS provides two ways of communication: publisher and subsriber, client and server. A publisher actively publishes messages, while a server passively responds upon request by the client. We use a combination of the two to allow asynchronous communication of the components. Every controller can be a ROS node of the final integrated system. Using publisher and subscriber, we can specify active input variables, meaning variables that trigger the controller state transition, to be different topics. Platforms whose contracts include these active input variables can subscribe to the corresponding topics. If new values are published as messages to one of those topics, the subscribed platforms will generate the new controller signal. Using client and server, we can record the values of the status variables, which indicate what state the system is in, in the server, and return the value when requested by the client. If a contract considers more than one variables to be environment input, for status variables, the platform can be the client to request their values whenever state transition occurs, for more than one active variables, if they are asynchronous, default values can be associated with them for when they are not triggering the state transition, or if synchronous, we can write a new node as a synchronizer for these variables.

In our problem, the dispatcher treats requests as active input and taxi availability as status variables. The route planner treats taxi assignment and ready as active input. The path navigation treats every change in target location as active input.

VII. Design Example and Result

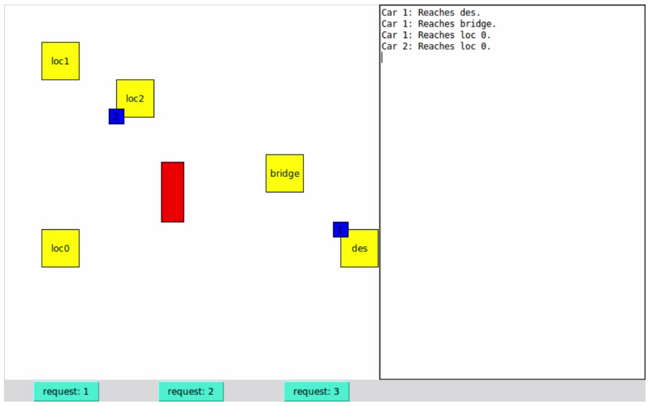
To demonstrate the results, we solve a concrete problem as an example. For a simple version, we specify that there are totally three different requests, two taxis available and the maximum number of passengers in each taxi is two. The different locations are connected to the destination by a bridge. Figure 3 shows the visualization of the problem.



```
Car 1: Reaches loc 0.
Car 2: Reaches loc 0.
```

Fig. 3. Problem visualization, where red block indicates static obstacle, blue blocks indicate the two taxis, yellow blocks indicate different locations

We run the simulation and give three requests at the same time to the system. Figure 4 shows an intermediate state when the first taxi has dropped off the first passenger from location 0 and the second taxi has reached location 2 to pick up the third passenger.

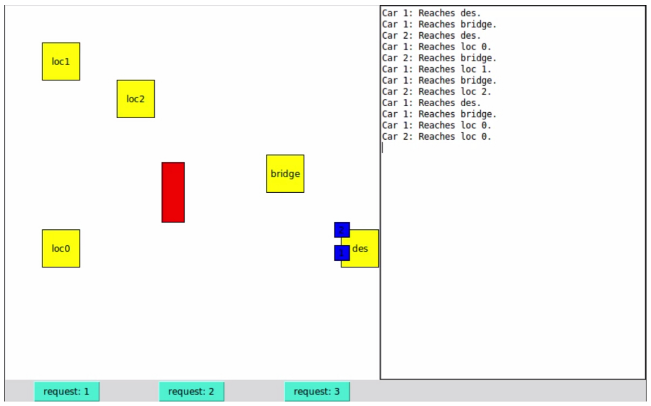


```
Car 1: Reaches des.
Car 1: Reaches bridge.
Car 1: Reaches loc 0.
Car 2: Reaches loc 0.
```

Fig. 4. Visualization of the intermediate state of the simulation

Figure 5 shows the end state of the simulation. The record of the location history of the taxis on the right column shows that the requests are satisfied. The final locations of the taxis also show that they have avoided each other.

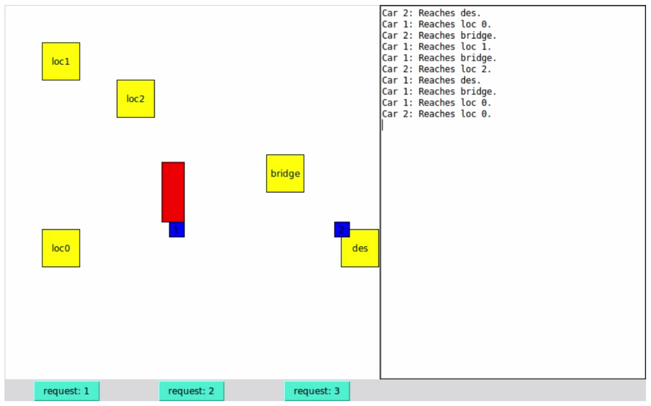
During simulation, the taxis successfully avoid the static obstacle. Figure 6 shows an intermediate state when the taxi



```
Car 1: Reaches des.
Car 1: Reaches bridge.
Car 2: Reaches des.
Car 1: Reaches loc 0.
Car 2: Reaches bridge.
Car 1: Reaches loc 1.
Car 1: Reaches bridge.
Car 2: Reaches loc 2.
Car 1: Reaches des.
Car 1: Reaches bridge.
Car 1: Reaches loc 0.
Car 2: Reaches loc 0.
```

Fig. 5. Visualization of the end state of the simulation

was going from location 0 to bridge. When it was close to the obstacle, it stayed outside the boundary of the obstacle, and after passing the obstacle, it went upwards to go to the bridge.



```
Car 2: Reaches des.
Car 1: Reaches loc 0.
Car 2: Reaches bridge.
Car 1: Reaches loc 1.
Car 1: Reaches bridge.
Car 2: Reaches loc 2.
Car 1: Reaches des.
Car 1: Reaches bridge.
Car 1: Reaches loc 0.
Car 2: Reaches loc 0.
```

Fig. 6. Visualization of static obstacle avoidance

Given the results, we discuss the performance of the designed system with respect to the system requirements.

- The service requirement is satisfied as the synthesized TuLiP strategy satisfies the service-level specification of the problem and the low-level controllers follow the strategy to navigate the taxis.
- The optimality is satisfied on the low-level navigation platform, but the overall optimality is not guaranteed as optimality in reactive setting is not configurable. TuLiP gives an optimal solution in the sense that each request is fulfilled in as few discrete steps as possible, but it does not support a customized cost function that allows the designer to specify the direction for optimization.
- The safety requirement is satisfied by having different BluSTL sections update controller while continuously monitoring other taxis. It allows a more flexible and less conservative solution than specifying mutual exclusion requirements on the taxi locations.

VIII. Conclusions

In this paper, we proposed using contract-based design with a platform-based paradigm in solving an autonomous taxi system. More specifically, we proposed the workflow for using platform-based design in the problem, which includes the independent implementation of system topology and controller synthesis. We established a promising framework for task and motion planning problems such as this one, which includes the use of synthesis tools and communication platform. We examined in detail the implementation of the autonomous taxi system and illustrated the design process following the proposed design flow. The example problem and its simulation show positive results about utilizing contract-based design in complex system design.

Future work on exploring contract-based design can include developing contract synthesis tools that formalize the design process of system topology, enriching the framework in designing distributed autonomous vehicle systems, and generalizing a platform-based architecture for robotic problems using contracts.

References

[1] T. Wongpiromsarn, U. Topcu and R. Murray, 'Receding Horizon Temporal Logic Planning', IEEE Transactions on Automatic Control, vol. 57, no. 11, pp. 2817-2830, 2012.

[2] C. McGhan, R. Murray, R. Serra, M. Ingham, M. Ono, T. Estlin and B. Williams, 'A Risk-Aware Architecture for Resilient Spacecraft Operations', Submitted, 2015 IEEE Aerospace Conference.

[3] A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J.-B. Raclet, P. Reinkemeier, A. Sangiovanni-Vincentelli, W. Damm, T. Henzinger, and K.G. Larsen. 'Contracts for System Design', Technical Report RR-8147, INRIA, November 2012.

[4] A. Sangiovanni-Vincentelli, W. Damm and R. Passerone, 'Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems*', European Journal of Control, vol. 18, no. 3, pp. 217-238, 2012.

[5] P. Nuzzo, Huan Xu, N. Ozay, J. Finn, A. Sangiovanni-Vincentelli, R. Murray, A. Donze and S. Seshia, 'A Contract-Based Methodology for Aircraft Electric Power System Design', IEEE Access, vol. 2, pp. 1-25, 2014.

[6] A. Sangiovanni-Vincentelli and G. Martin, 'Platform-based design and software design methodology for embedded systems, Design & Test of Computers, IEEE, vol.18, no.6, pp.23,33, Nov/Dec 2001.

[7] W. Damm, 'Contract-based analysis of automotive and avionics applications: The SPEEDS approach', In: Cofer, D.D., et al. (eds.) FMICS. LNCS, vol. 5596, p. 3. Springer, 2008.

[8] P. Nuzzo, A. Sangiovanni-Vincentelli, Xuening Sun and A. Puggelli, 'Methodology for the Design of Analog Integrated Interfaces Using Contracts', IEEE Sensors J., vol. 12, no. 12, pp. 3329-3345, 2012.

[9] C. Baier and J. Katoen, Principles of model checking. Cambridge, Mass.: MIT Press, 2008.

[10] V. Raman, M. Maasoumy, A. Donze, R. M. Murray, A. Sangiovanni-Vincentelli, and S. A. Seshia. Model predictive control with signal temporal logic specifications. In Proc. of the IEEE Conf. on Decision and Control, 2014.

[11] V. Raman, A. Donze, D. Sadigh, R. M. Murray, and S. A. Seshia. 'Reactive synthesis from signal temporal logic specifications'. In Proceedings of the international conference on Hybrid Systems: Computation and Control, HSCC 2015, 2015.

[12] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli and Y. Sa'ar, 'Synthesis of Reactive(1) designs', Journal of Computer and System Sciences, vol. 78, no. 3, pp. 911-938, 2012.

[13] T. Wongpiromsarn, U. Topcu, N. Ozay, H. Xu, and R. M. Murray, 'TuLiP: a software toolbox for receding horizon temporal logic planning', International Conference on Hybrid Systems: Computation and Control, 2011.

[14] A. Donze and V. Raman, 'BluSTL: Controller Synthesis from Signal Temporal Logic Specifications', 2nd International Workshop on Applied Verification for Continuous and Hybrid Systems (ARCH 2015), April, 2015.

[15] Ros.org, 'ROS.org — Powering the world's robots', 2015. [Online]. Available: http://www.ros.org/. [Accessed: 27- Jul- 2015].

[16] Mathworks.com, 'Robotics System Toolbox - Simulink and MATLAB', 2015. [Online]. Available: http://www.mathworks.com/products/robotics/?refresh=true. [Accessed: 30- Aug- 2015].