Failure-Tolerant Contract-Based Design of an Automated Valet Parking System using a Directive-Response Architecture

Josefine B. Graebener, Tung Phan-Minh, Jiaqi Yan, Qiming Zhao and Richard M. Murray

Abstract—Increased complexity in cyber-physical systems calls for modular system design methodologies that guarantee correct and reliable behavior, both in normal operations and in the presence of failures. This paper aims to extend the contract-based design approach using a directive-response architecture to enable reactivity to failure scenarios. The architecture is demonstrated on a modular automated valet parking (AVP) system. The contracts for the different components in the AVP system are explicitly defined, implemented, and validated against a Python implementation.

I. INTRODUCTION

Formally guaranteeing safe and reliable behavior for modern cyber-physical systems is becoming challenging as standard practices do not scale [1]. Managing these highly complex architectures requires a design process that explicitly defines the dependencies and interconnections of system components to enable guaranteed safe behavior of the implemented system [2]. A leading design methodology to develop component-based software is contract-based design, which formalizes the design process in view of component hierarchy and composition [3], [4], [5]. Contract-based design reduces the complexity of the design and verification process by decomposing the system tasks into smaller tasks for the components to satisfy. From the composition of these components, overall system properties can be inferred or proved. This contract-based architecture has been demonstrated for several applications [6], [7], [8], [9]. Our goal here is to adapt and extend this framework to model a directive-response architecture on an automated valet parking system with the following features:

1) Discrete and continuous decision making components, which have to interact with one another.
2) Different components have different temporal requirements.
3) A natural hierarchy between the different components in our system that may be thought of as different layers of abstraction.
4) The system involves both human and non-human agents, the number of which is allowed to change over time.
5) Industry interest in such a system.

One example of industry efforts to commercialize such a system is the automated valet parking system developed by Bosch in collaboration with Mercedes-Benz, which has been demonstrated in the Mercedes-Benz Museum parking garage in Stuttgart, Germany. Bosch and Daimler also later announced in 2020 that they would set up a commercially operating AVP at the Stuttgart airport [10]. Another commercial AVP system is supposed to be set up by Bosch in downtown Detroit as a collaboration with Bedrock and Ford [11]. Other examples include efforts by Siemens [12] and DENSO [13]. The contributions of this paper include the formulation of a formal contract structure for an automated valet parking system with multiple layers of abstraction with a directive-response architecture for failure-handling. By implementing this system in Python, we aim to bridge the large gap between abstract contract metatheory and such non-trivial engineering applications. In addition, we incorporate error handling into the contracts and demonstrate the use of this architecture and approach towards writing specifications in the context of the automated valet parking example. Finally, we prove that the composed implementation satisfies the composite contract, adding this example of a larger scale control system, involving a dynamic set of agents that are allowed to fail, to the small and slowly growing list of examples of formal assume-guarantee contract-based design.

II. THEORETICAL BACKGROUND

A. Contract Theory Background

Contract-based design is a formal modular design methodology originally developed for component-based software systems [14]. A component’s behavior can be specified in terms of a guarantee that must be provided when its environment satisfies a certain assumption. This pairing of an assumption with a guarantee provides the basis for defining a contract. A contract algebra can be developed in which different contract operations can be defined which enable comparison between and combinations of contracts, formalizing modularity, reusability, hierarchy etc. [15]. A comprehensive meta-theory of contracts is presented in [1]. In the following, we will introduce a variant of assume-guarantee contracts that incorporates a directive-response architecture.

B. Directive-Response Architecture

In a centralized approach for contingency management, recovery from failures is achieved by communicating with nearly every module in the system from a central module, hence increasing the system’s complexity and potentially making it more error-prone [16]. The Mission Data system (MDS), developed by JPL as a multi-mission information and control architecture for robotic exploration spacecraft, was an approach to unify the space system software design architecture. MDS includes failure handling as an integral part of the design [17], [18]. It is based on the state analysis framework,
a system engineering methodology that relies on a state-based control architecture and explicit models of the system behavior. Fault detection in MDS is executed at the level of the modules, which report if they cannot reach the active goal and possible recovery strategies. Resolving failures is one of the tasks the system was designed to be capable of and not an unexpected situation [17], [19]. Another architecture based on the state analysis framework is the Canonical Software Architecture (CSA) used on the autonomous vehicle Alice by the Caltech team in the DARPA Urban Challenge in 2007. The CSA enables decomposition of the planning system into a hierarchical framework, respecting the different levels of abstraction at which the modules are reasoning and the communication between the modules is via a directive-response framework [20]. This framework enables the system to detect and react to unexpected failure scenarios, which might arise from changes in the environment or hardware and software failures in the system [16]. In this paper we are trying to capture the MDS and CSA approaches by incorporating directive-response techniques into a contract framework.

C. Directive-Response Contract Framework

In this paper, we propose a contract-based design framework incorporating a directive-response architecture to enable reactivity to failures in the system. System components can be abstracted as black boxes constrained by assume-guarantee contracts that specify the behavior of the integrated system. Components communicate with one another by exchanging directives and responses, potentially acting according to a contingency plan that specifies how to react to possible failures. The higher module sends a directive, and the lower module chooses its responses according to its status in achieving the directive’s intended goal. The system components are composed to satisfy the overall system requirements while interacting with the environment, such as safety and liveness specifications.

III. Motivating Example

The motivating example that we are developing in this paper is automated valet parking (AVP), as introduced in the previous section. The goal of this system is to automate the parking and retrieving process for multiple cars concurrently, while providing efficient operations in a safe environment.

A. Overall Specification

To be a successful operation, the AVP system needs to provide guarantees to customers regarding their safety and that their car will eventually be returned. These specifications can be written in linear temporal logic (LTL) [21]. For a detailed discussion on LTL, see [22]. The □ symbol represents the “always” operator and the ◇ represents “eventually”. These are operators on predicates or traces. An example of the specification is the following:

Property 1 (Safety): □¬collision (Always no collision.)

and

Property 2 (Liveness): □healthy ⇒ ◇Returned (Healthy car will eventually be returned.)

where the predicate collision is True if more than one car or pedestrian occupy the same space, and healthy and Returned are predicates which correspond to the status of the car, where healthy is True if the car does not have a failure and Returned is True once the control of the car has been given back to the customer. These specifications have to be satisfied for any implementation of the system and we will show this in our proof of the correctness of the composed system.

IV. Mathematical Formulation

To provide a formal description of the contracts and the components, we will introduce the mathematical background in this section. We will provide definitions regarding the geometry of the path planning, introduce the variables of our AVP world, and define the directive response framework and components.

A. Geometry

Definition 1 (Path): A path is a continuous map \( p : [0, 1] \rightarrow \mathbb{R}^2 \). For each path \( p \), let \( p_h : [0, 1] \rightarrow (-180, 180) \) be such that \( p_h(s) \) is the heading angle measured in degrees from the abscissa to \( p'(s) \), the derivative vector of \( p \) with respect to \( s \). For \( t \in [0, 1] \), let \( \hat{p}(t) \) denote the element \( p(t) \times p_h(t) \) of \( \mathbb{R}^3 \).

We will denote the set of all paths by \( \mathbb{P} \) and, by abuse of notation, we will also use \( p \) to denote \( p([0, 1]) \), the image of \( [0, 1] \) under \( p \).

Definition 2 (Curvature feasibility): Given \( \kappa > 0 \) and a path \( p \), \( \kappa \)-feasible \( p \) is set to True if and only if \( p \) is twice differentiable on \([0, 1]\), and its curvature \( \left| \frac{\text{det}(p'(s), p''(s))}{\|p'(s)\|} \right| < \kappa \) for all \( s \in [0, 1] \).

Definition 3 (δ-corridor): Let \( \mathcal{B} := \{\text{True}, \text{False}\} \). If \( p \in \mathbb{P} \) and \( \delta : \mathbb{P} \times [0, 1] \times \mathbb{R}^3 \rightarrow \mathcal{B} \) is such that the
corresponding subset:
\[ \Gamma_{\delta}(p) := \bigcup_{s \in [0,1]} \Gamma_{\delta}(p, s), \]
where \( \Gamma_{\delta}(p, s) := \{(x, y, \theta) \in \mathbb{R}^3 \mid \delta(p, s, (x, y, \theta)) = \text{True}\} \) such that \( \Gamma_{\delta}(p, s) \) is open and contains \( p(s) \) then we say \( \Gamma_{\delta}(p) \) is a \( \delta \)-corridor for \( p \).

B. AVP World

Building Blocks: In this section we will introduce naming symbols for objects that exist in the AVP world.

Definition 4 (AVP World): The AVP world consists of the following
1) A distinguished set of indexing symbols \( T := \{t, t', t'', \ldots \} \) denoting time.
2) A set of typed variables \( U \) to denote actions, states, channels, etc.
3) The following set of constants: \( C, G \) where
   a) \( C \), a set of symbols, is called the customer set.
   b) \( G \), a set of symbols, is called the garage set containing the following constant values
      i) \( G.d{}rivable.area \subseteq \mathbb{R}^3 \), the set of configurations that vehicles are allowed to be in.
      ii) \( G.walk{}able.area \subseteq \mathbb{R}^2 \), the area that pedestrians are allowed to walk on.
      iii) \( G.entry.configurations \subseteq \mathbb{R}^3 \), a set of configurations that the customers can deposit their car in.
      iv) \( G.return.configurations \subseteq \mathbb{R}^3 \), a set of configurations that the car should be returned in.
      v) \( G.parking.spots \subseteq \mathbb{N} \), the number of parking spots available in the parking lot.
      vi) \( G.interior \subseteq \mathbb{R}^2 \), the area inside the parking garage.

Directive-Response Message Types: Each channel in the system is associated with a unique message type. The following are all the message types in our AVP system.

\( A(\cdot) \), directive types:
1) \( A(\text{CustomerInterface}) := \{\text{Park, Retrieve}\} \).
2) \( A(\text{Supervisor}) := \mathbb{R}^6 \).
3) \( A(\text{Planner}) = \mathbb{P} \).
4) \( A(\text{Tracker}) := I \subseteq \mathbb{R}^2 \), the set of all control inputs.

\( B(\cdot) \), response types:
1) \( B(\text{CustomerInterface}) := \{\text{Failed}\} \).
2) \( B(\text{Supervisor}) := \{\text{Rejected, Accepted, Returned}\} \).
3) \( B(\text{Planner}) = B(\text{Tracker}) := \{\text{Blocked, Failed, Completed}\} \).

For each type \( T \) we will denote by \( \bar{T} \) the product type \( T \times C \) which will be used to associate a message of type \( T \) with a specific customer in \( C \). In addition, we will use \( \text{Id} \) to denote the set of message IDs.

Behavior: For each variable \( u \in U \), we denote by \( \text{type}(u) \) the type of \( u \), namely, the set of values that it can take. The types of elements of \( T \) are taken to be \( \mathbb{R}_{\geq 0} \).

Definition 5 (Behavior): Let \( Z \) be an ordered subset of variables in \( U \). A \( Z \)-behavior is an element of \( B(Z) := (\prod_{z \in Z} \text{type}(z))^{\mathbb{R}_{\geq 0}} \). Given \( \sigma_z \in B(Z) \) and \( \tau \in T \), we will call \( \sigma_z(\tau) \) the valuation of \( Z \) at time \( \tau \). If \( z \in Z \), we will also denote by \( z(\tau) \) the value of \( z \) at time \( \tau \). Note that each behavior in \( Z \subseteq U \) can be “lifted” to a set of behaviors in \( U \) by letting variables that are not contained in \( Z \) assume all possible values in their domains. Additionally, the set of behaviors \( B(Z) \) can be lifted to a set of behaviors in \( B(U) \) in a similar way. To ease notational burden for the reader, we will take the liberty of not explicitly making any reference to the “lifting” operation in this paper when they are in use unless there is any ambiguity that may result from doing so.

Definition 6 (Constraint): A constraint \( k \) on a set of variables \( Z \) is a function that maps each behavior of \( Z \) to an element of \( \mathbb{R} \), the Boolean domain. In other words, \( k \in \mathbb{B}^Z \).

Note that by “lifting”, a constraint on a set of variables \( Z \) is also a constraint on \( U \).

Definition 7 (Channel variables): For each component \( X \) and another component \( Y \), we can define two types of channel variables:
- \( X_{\rightarrow Y} \), denoting an incoming information flow from \( Y \) to \( X \).
- \( X_{\rightarrow Y} \), denoting an outgoing information flow from \( X \) to \( Y \).

In this work, we assume that \( X_{\rightarrow Y} \) is always identical to \( Y_{\rightarrow X} \). Each channel variable must have a well-defined message type and each message \( m \) has an ID denoted by \( \text{id}(m) \in \text{Id} \). If the message has value \( v \), then we will denote it by \([v, \text{id}(m)]\), but we will often refer to it as \([v]\) whereby we omit the ID part to simplify the presentation. Intuitively, given a behavior, a channel variable \( x \) is a function that maps each time step to the message the associated channel is broadcasting at that time step.

Definition 8 (System): A system \( M \) consists of a set of each of the following
1) internal variables/ constants \( \text{var}_{X}^M \),
2) output channel variables \( \text{var}_{Y}^M \),
3) input channel variables \( \text{var}_{U}^M \),
4) constraints \( \text{con}_M \) on \( \text{var}_{X}^M \cup \text{var}_{Y}^M \cup \text{var}_{U}^M \).

A behavior of a system \( M \) is an element of the set of behaviors that correspond to \( \text{var}_{X}^M \cup \text{var}_{Y}^M \cup \text{var}_{U}^M \) subject to \( \text{con}_M \). This is denoted by \( B(M) \).

Directive-response: Before introducing directive-response systems, for any predicates \( A \) and \( B \), we define the following syntax:
\[ A \leadsto B := \forall t :: A(t) \Rightarrow \exists t' \geq t :: B(t'). \quad \text{("leads to")} \]
\[ A \preceq B := \forall t :: B(t) \Rightarrow \exists t' \leq t :: A(t'). \quad \text{("precedes")} \]
\[ \Box_{\geq t} A := \forall t' \geq t :: A(t'). \quad \text{("always from t")} \]
\[ \text{starts.at}(A, t) := A(t) \land \forall t' < t :: \neg A(t'). \quad (1) \]
If $M$ is a set-valued variable, then we define:
\[
persistent(M) := \forall t :: \forall m :: m \in M(t) \Rightarrow \square_{\geq t}(m \in M).
\] (2)

**Definition 9 (Directive-response system):** A directive-response system $M$ is a system such that for each output (resp., input) channel variable $\text{chan}$ there is an internal variable $send_{\text{chan}}$ (resp., $receive_{\text{chan}}$) whose domain is a collection of sets of messages that are of the type associated with $\text{chan}$. If $\text{chan}$ is an output channel variable, there is a causality constraint $k_{\text{chan}} \in \text{con}_M$ defined by:
\[
k_{\text{chan}} := m \in send_{\text{chan}} \leq m = \text{chan}.
\] (3)
That is, a message must be sent before it shows in the channel. Otherwise if $\text{chan}$ is an input channel variable:
\[
k_{\text{chan}} := m = \text{chan} \leq m \in receive_{\text{chan}}.
\] (4)
Namely, a message cannot be received before it is broadcasted.

**Definition 10 (Lossless directive-response system):** A lossless directive-response system is a directive-response system such that if $\text{chan}$ is an output channel then
\[
persistent(send_{\text{chan}}) \land (m \in send_{\text{chan}} \Rightarrow m = \text{chan}).
\] (5)
and if $\text{chan}$ is an input channel
\[
persistent(receive_{\text{chan}}) \land (m = \text{chan} \Rightarrow m \in receive_{\text{chan}}).
\] (6)

**Definition 11 (Assume-guarantee contracts):** An assume-guarantee contract $C$ for a directive-response system $M$ consists of a pair of behaviors $A, G$ of $M$ and denoted by $C = (A, G)$. An environment for $C$ is any set of all behaviors that are contained in $A$ while an implementation of $C$ is any set of behaviors that is contained in $A \Rightarrow G$. $C$ is said to be saturated if the guarantee part satisfies $G = (\neg A \lor G) = (A \Rightarrow G)$.

Note that any contract can be converted to the saturated form without changing its sets of environments and implementations. The saturated form is useful in making contract algebra less cumbersome in general. If $M$ is a system, then we say $M$ satisfies $C$ if $B(M) \subseteq (A \Rightarrow G)$. Furthermore, the system composition $M_1 \times M_2$ of $M_1$ and $M_2$ is a system whose behavior is equal to $B(M_1) \cap B(M_2)$.

**Definition 12 (Customer):** A customer is an element of $C$. Corresponding to each $c \in C$ is a set of $\mathcal{U}$ variables $\text{var}(c)$ that include $c.x, c.y$ (the coordinates of the customer him/herself), $c.car.x$, $c.car.y$, $c.car.\theta$ (the coordinates and heading of the customer’s car), $c.car.\text{healthy}$, whether the car is healthy, $c.controls.v, c.controls.\varphi$ (the velocity and steering inputs to the vehicle), $c.car.\ell$ (the length of the car), $c.car.towed$ (whether the car is being towed). We will use the shorthand $c.car.state$ to mean the 3-tuple $(c.car.x, c.car.y, c.car.\theta)$.

For each behavior in $B(\mathcal{U})$, we require each $c \in C$ for which $c.car.towed$ is False to satisfy the following constraints that describe the Dubins car model:
\[
\begin{align*}
\frac{d(c.car.x)}{dt}(t) &= c.controls.v(t) \cos(c.car.\theta(t)) \\
\frac{d(c.car.y)}{dt}(t) &= c.controls.v(t) \sin(c.car.\theta(t)) \\
\frac{d(c.car.\varphi)}{dt}(t) &= \frac{c.controls.v(t)}{c.car.\ell} \tan(c.controls.\varphi(t))
\end{align*}
\] (7)

C. AVP System

By treating the CustomerInterface as an external component, the AVP system consists of three internal components: Supervisor, Planner, and Tracker. These systems are described below.

1) CustomerInterface: The environment in which the system shall operate consists of the customers and the pedestrians, which we will call a CustomerInterface. A customer drops off the car at the drop-off location and is assumed to make a request for the parked car back from the garage eventually. The pedestrians are also controlled by the environment. When a pedestrian was generated by the environment, they start walking on the crosswalks. Pedestrians are confined to the pedestrian path, meaning they will not leave the crosswalk and walkway areas and their dynamics are continuous, meaning no sudden jumps. The cars move according to their specified dynamics. This includes a breaking distance depending on their velocity and maximum allowed curvature. For a formal description, refer to Table I. Below are some constraints we impose on this module.

For each behavior in $B(\mathcal{U})$, we require each $c \in C$ for which $c.car.towed$ is False to satisfy the following constraints that describe the Dubins car model:
\[
\begin{align*}
\frac{d(c.car.x)}{dt}(t) &= c.controls.v(t) \cos(c.car.\theta(t)) \\
\frac{d(c.car.y)}{dt}(t) &= c.controls.v(t) \sin(c.car.\theta(t)) \\
\frac{d(c.car.\varphi)}{dt}(t) &= \frac{c.controls.v(t)}{c.car.\ell} \tan(c.controls.\varphi(t))
\end{align*}
\] (7)

2) Supervisor: A Supervisor component is responsible for the high level decision making in the process. It receives the CustomerInterface requests and processes them by sending the appropriate directives to the Planner to fulfill a task. A Supervisor determines whether a car can be accepted into the garage or rejected. It also receives responses from the Planner. A Supervisor is to be aware of the reachability, the vacancy, and occupied spaces in the lot, as well as the parking lot layout. Formally, a Supervisor is a lossless directive-response system described by Table I.
TABLE II
Supervisor DIRECTIVE-RESPONSE SYSTEM.

| Internal variables/constants var_\(X\) | 
|-----------------------------|
| \(G, *\) | All \(G\) objects. |
| num\(_\text{active_cusomers}\) | The number of cars currently being served in the parking lot. |

Outputs vary:

- Supervisor \(\rightarrow\) CustomerInterface: An output channel of type \(\tilde{B}(\text{CustomerInterface})\). 
- Supervisor \(\rightarrow\) Planner: An output channel of type \(\tilde{A}(\text{Supervisor})\). 

Inputs vary:

- Supervisor \(\leftarrow\) CustomerInterface: An input channel of type \(A(\text{CustomerInterface})\). 
- Supervisor \(\leftarrow\) Planner: An input channel of type \(\tilde{B}(\text{Planner})\). 

Constraints \(con_M\):

- Parking lot topology: Any specific geometric constraints on \(G,*\). 
- Number of active customers: \(num\_\text{active\_customers}\) must be equal to the number of cars that have been accepted but not yet left the parking lot.

TABLE III
Planner DIRECTIVE-RESPONSE SYSTEM.

| Interval variables/constants var_\(X\) | 
|------------------------------------------|
| \(G, *\) \{c.car.x,c.car.y,c.car.\(\theta\) | \(c \in \mathbb{C}\} | All \(G\) objects. |
| \(\kappa\) | The configurations of all cars in AVP world. |
| Maximum allowable curvature. |

Outputs vary:

- Planner \(\rightarrow\) Supervisor: An output channel of type \(\tilde{B}(\text{Planner})\). 
- Planner \(\rightarrow\) Tracker: An output channel of type \(\tilde{A}(\text{Planner})\). 

Inputs vary:

- Planner \(\leftarrow\) Supervisor: An input channel of type \(\tilde{A}(\text{Supervisor})\). 
- Planner \(\leftarrow\) Tracker: An input channel of type \(\tilde{B}(\text{Tracker})\). 

Constraints \(con_M\):

- Parking lot topology: Any specific geometric constraints on \(G,*\). 
- \(\kappa\): Maximum allowable curvature given car dynamics and input constraints.

3) Planner: A Planner system receives directives from the Supervisor to make a car reach a specific location in the parking lot. A Planner system has access to a planning graph determined from the parking lot layout, and thus can generate executable trajectories for the cars to follow. The Planner is aware of the locations of the agents and the obstacles in the parking lot from the camera system. A Planner is a lossless directive-response system described by Table III.

4) Tracker: A Tracker system is responsible for the safe control of cars that are accepted into the garage by a Supervisor. It receives directives from a Planner consisting of executable paths to track and send responses based on the task status to a Planner. See Table IV.

V. AVP CONTRACTS

In this section we will define the contracts for each of the modules in our system. These contracts are the guidelines for the implementation, and will be used to verify each of the components, as well as the composed system. In Figure 2 the green arrows represent directive-response assume-guarantee contracts, solid black arrows represent communication, and dashed black arrows represent passive information flow (observing movement of the agents). The \(\Delta\) in the car component represents the possibility of failure and uncertainty.

**Contract 1 (C_{CustomerInterface}):** The following is the contract for the CustomerInterface.

- Assumes
  - If the CustomerInterface sends a request to the Supervisor, then they will receive a response from the Supervisor:
    \[
    \forall c \in \mathbb{C} :: ([m,c] \in \text{send}_{\text{CustomerInterface}}_{\rightarrow}\text{Supervisor}) \rightarrow \exists r \in \tilde{B}(\text{Supervisor}) :: [r,c] \in \text{receive}_{\text{CustomerInterface}}_{\rightarrow}\text{Supervisor}. \tag{10}
    \]
  - If the car is healthy and accepted by the garage, it will be returned after being summoned:
    \[
    \forall c \in \mathbb{C} :: \Box_{\mathbb{D}_{\text{\(c.car.healthy\)}}} \land \Box ([\text{Accepted},c] \in \text{send}_{\text{CustomerInterface}}_{\rightarrow}\text{Supervisor}) \land \Box ([\text{Returned},c] \in \text{receive}_{\text{CustomerInterface}}_{\rightarrow}\text{Supervisor}) \rightarrow [\text{Returned},c] \in \text{receive}_{\text{CustomerInterface}}_{\rightarrow}\text{Supervisor}. \tag{11}
    \]

- Guarantees
  - When the request is accepted, the CustomerInterface should not tamper with the car controls until the car is returned (i.e., control signals should match the...
When the CustomerInterface is not receiving any new input signal, then it keeps the control inputs at zero:

\[ \forall c \in C :: (\exists t :: \forall (v, \varphi) \in I :: (\neg ((\text{Accepted}, c) \in \text{receive}_{\text{CustomerInterface}} \rightarrow_{\text{Supervisor}} \text{Supervisor}(t)) \land \neg ((\text{Rejected}, c) \in \text{receive}_{\text{CustomerInterface}} \rightarrow_{\text{Supervisor}} \text{Supervisor}(t))) \implies \exists t' < t :: ((v, \varphi), c) \notin \text{receive}_{\text{CustomerInterface}} \rightarrow_{\text{Tracker}} \text{Tracker}(t') \land c.\text{controls} \cdot v(t) = v \land c.\text{controls} \cdot \varphi(t) = \varphi). \]

(12)

- From sending a request until receiving a response, the car must stay in the deposit area:

\[ \forall c \in C :: \exists t :: \forall (v, \varphi) \in I :: (\neg ((\text{Accepted}, c) \in \text{receive}_{\text{CustomerInterface}} \rightarrow_{\text{Supervisor}} \text{Supervisor}(t))) \implies (\exists t' < t :: ((v, \varphi), c) \in \text{receive}_{\text{CustomerInterface}} \rightarrow_{\text{Tracker}} \text{Tracker}(t')) \implies c.\text{controls} \cdot v(t) = 0 \land c.\text{controls} \cdot \varphi(t) = 0). \]

(13)

- After the car is deposited, the customer will eventually summon it:

\[ \forall c \in C :: (\text{Accepted}, c) \in \text{receive}_{\text{CustomerInterface}} \rightarrow_{\text{Supervisor}} \text{Supervisor} \implies (\exists t :: \text{send}_{\text{CustomerInterface}} \rightarrow_{\text{Supervisor}} \text{Supervisor}(t)) \land (\exists t :: \text{send}_{\text{CustomerInterface}} \rightarrow_{\text{Supervisor}} \text{Supervisor}(t)). \]

(15)

- Pedestrians will only walk on “walkable” area:

\[ \forall c \in C :: \exists t :: (\forall (x, y) \in \text{G.walkable_area}) \implies (\forall (x, y) \in \text{G.walkable_area} \land \neg c.\text{car}.x \in \text{G.walkable_area} \land \neg c.\text{car}.y \in \text{G.walkable_area} \implies (\forall (x, y) \in \text{G.walkable_area} \land \neg c.\text{car}.x \in \text{G.walkable_area} \land \neg c.\text{car}.y \in \text{G.walkable_area}). \]

(16)

- Pedestrians will not stay on crosswalks forever:

\[ \forall c \in C :: (\forall (x, y) \in \text{G.walkable_area} \land \neg c.\text{car}.x \in \text{G.walkable_area} \land \neg c.\text{car}.y \in \text{G.walkable_area} \implies (\forall (x, y) \in \text{G.walkable_area} \land \neg c.\text{car}.x \in \text{G.walkable_area} \land \neg c.\text{car}.y \in \text{G.walkable_area}). \]

(17)

- If the car is not healthy and not towed, it cannot move:

\[ \forall c \in C :: (\forall (x, y) \in \text{G.walkable_area} \land \neg c.\text{car}.x \in \text{G.walkable_area} \land \neg c.\text{car}.y \in \text{G.walkable_area} \implies (\forall (x, y) \in \text{G.walkable_area} \land \neg c.\text{car}.x \in \text{G.walkable_area} \land \neg c.\text{car}.y \in \text{G.walkable_area}). \]

(18)

- Sending a Retrieve message must always be preceded by receiving an Accepted message from the Supervisor:

\[ \forall c \in C :: (\forall c.\text{car}.healthy \land (c.\text{car.towed} \implies c.\text{controls} \cdot v = 0 \land c.\text{controls} \cdot \varphi = 0)). \]

(19)

- If a customer receives Rejected or Returned from the Supervisor, then they must leave the lot forever:

\[ \forall c \in C :: \forall t :: (\forall (v, \varphi) \in I :: (\neg ((\text{Rejected}, c) \in \text{receive}_{\text{CustomerInterface}} \rightarrow_{\text{Supervisor}} \text{Supervisor}(t))) \land \neg ((\text{Returned}, c) \in \text{receive}_{\text{CustomerInterface}} \rightarrow_{\text{Supervisor}} \text{Supervisor}(t))) \implies \exists t' > t :: (\neg (\forall (x, y) \in \text{G.interior})). \]

(20)

\[ \text{Contract 2 (C}_{\text{Supervisor}}) \text{: } \text{The contract for the Supervisor is as follows.} \]

- Assumes
  - Towing eventually happens after the Supervisor is alerted of car failure:
    \[ \forall c \in C :: \forall t :: (\exists ! p \in \text{R}^3 :: (\exists ! \text{send}_{\text{Supervisor}} \rightarrow_{\text{Planner}} \text{Planner}(t)) \implies p = 0 \land (\forall p' \in \text{R}^3 :: (\exists ! \text{send}_{\text{Supervisor}} \rightarrow_{\text{Planner}} \text{Planner}(t') \implies p = 0 \land (\forall p' \in \text{R}^3 :: (\exists ! \text{send}_{\text{Supervisor}} \rightarrow_{\text{Planner}} \text{Planner}))). \]

(21)

- If a car fails, then the Planner reports Failed:

\[ \forall c \in C :: \forall t :: (\forall (x, y) \in \text{G.interior} ; (\exists ! \text{send}_{\text{Supervisor}} \rightarrow_{\text{CustomerInterface}} \text{CustomerInterface}(t)) \implies (\forall (x, y) \in \text{G.interior} ; (\exists ! \text{send}_{\text{Supervisor}} \rightarrow_{\text{CustomerInterface}} \text{CustomerInterface}(t))). \]

(22)

- Cars making requests are deposited correctly by the customer:

\[ \forall c \in C :: (\forall (x, y) \in \text{G.interior} ; (\exists ! \text{send}_{\text{Supervisor}} \rightarrow_{\text{CustomerInterface}} \text{CustomerInterface}(t)) \implies (\forall (x, y) \in \text{G.interior} ; (\exists ! \text{send}_{\text{Supervisor}} \rightarrow_{\text{CustomerInterface}} \text{CustomerInterface}(t))). \]

(23)

- If a car is healthy and summoned, then it will eventually appear at the return area and the Planner will send a Completed signal to the Supervisor:

\[ \forall c \in C :: (\text{G.interior} \land (\forall (x, y) \in \text{G.interior} ; (\exists ! \text{send}_{\text{Supervisor}} \rightarrow_{\text{CustomerInterface}} \text{CustomerInterface}(t)) \implies (\forall (x, y) \in \text{G.interior} ; (\exists ! \text{send}_{\text{Supervisor}} \rightarrow_{\text{CustomerInterface}} \text{CustomerInterface}(t))). \]

(24)

- Guarantees
  - All requests from customers will be replied:
    \[ \forall c \in C :: (\forall (x, y) \in \text{G.interior} ; (\exists ! \text{send}_{\text{Supervisor}} \rightarrow_{\text{CustomerInterface}} \text{CustomerInterface}(t)) \implies (\forall (x, y) \in \text{G.interior} ; (\exists ! \text{send}_{\text{Supervisor}} \rightarrow_{\text{CustomerInterface}} \text{CustomerInterface}(t))). \]

(25)

- The Supervisor cannot send a Returned message to the CustomerInterface unless it has received a Completed message from the Planner and the car is in the return area:

\[ \forall c \in C :: (\text{G.interior} \land (\forall (x, y) \in \text{G.interior} ; (\exists ! \text{send}_{\text{Supervisor}} \rightarrow_{\text{CustomerInterface}} \text{CustomerInterface}(t)) \implies (\forall (x, y) \in \text{G.interior} ; (\exists ! \text{send}_{\text{Supervisor}} \rightarrow_{\text{CustomerInterface}} \text{CustomerInterface}(t))). \]

(26)

- If a car is healthy and a Retrieve message is received, then the last thing sent to the Planner should be a directive to the return area (the second configuration should be one of the return configurations):

\[ \forall c \in C :: (\text{G.interior} \land (\exists ! \text{send}_{\text{Supervisor}} \rightarrow_{\text{Planner}} \text{Planner}(t)) \implies (\forall (x, y) \in \text{G.interior} ; (\exists ! \text{send}_{\text{Supervisor}} \rightarrow_{\text{Planner}} \text{Planner}(t)) \implies (\forall (x, y) \in \text{G.interior} ; (\exists ! \text{send}_{\text{Supervisor}} \rightarrow_{\text{Planner}} \text{Planner}(t))). \]

(27)

- If the car is healthy and if it is ever summoned, then the Supervisor will send a Returned message to its owner:

\[ \forall c \in C :: (\forall (x, y) \in \text{G.interior} ; (\exists ! \text{send}_{\text{Supervisor}} \rightarrow_{\text{CustomerInterface}} \text{CustomerInterface}(t)) \implies (\forall (x, y) \in \text{G.interior} ; (\exists ! \text{send}_{\text{Supervisor}} \rightarrow_{\text{CustomerInterface}} \text{CustomerInterface}(t))). \]

(28)
- If there is a not-yet-responded-to Park request and the parking lot capacity is not yet reached, then the Supervisor should accept the request:

\[
\forall c \in C :: \exists ![Park, c] \\
\in receive_{\text{Supervisor} \rightarrow \text{CustomerInterface}} (t) \\
\land \forall' \leq t :: [\text{Rejected}, c] \not\in send_{\text{Supervisor} \rightarrow \text{CustomerInterface}} (t') \\
\land [\text{Accepted}, c] \not\in send_{\text{Supervisor} \rightarrow \text{CustomerInterface}} (t') \\
\land num_{\text{active_customers}} (t) < G.parking_{\text{spots}} \\
\Rightarrow \exists t'' > t :: [\text{Accepted}, c] \\
\in send_{\text{Supervisor} \rightarrow \text{CustomerInterface}} (t'').
\] (29)

- For every Accepted to or Retrieve from the CustomerInterface or Blocked from the Planner, the Supervisor sends a pair of configurations to the Planner, the first of which is the current configuration of the car and such that there exists a path of allowable curvature:

\[
\forall c \in C :: [\text{Accepted}, c] \in \text{send}_{\text{Supervisor} \rightarrow \text{CustomerInterface}} \\
\lor \exists [\text{Retrieve}, c] \in \text{receive}_{\text{Supervisor} \rightarrow \text{CustomerInterface}} \\
\lor \exists [\text{Blocked}, c] \in \text{receive}_{\text{Supervisor} \rightarrow \text{CustomerInterface}} \\
\exists k_0, k_1 \in \mathbb{R}^3 :: (k_0, k_1) = c \\
\land k_0 = c.car.state \land k_1 \epsilon \mathbb{P} :: \kappa\text{-feasible}(p) \land \\
p(0) = k_0 \land p(1) = k_1.
\] (30)

**Contract 3 (C_{Planner}):** The contract for the Planner is as follows:

- Assumptions
  - When the Tracker completes its task according to the corridor map δ, it should send a report to the Planner:

\[
\forall c \in C :: \exists p \in \mathbb{P} :: p' \epsilon \mathbb{P} :: \\
[p', c] \in \text{send}_{\text{Planner} \rightarrow \text{Tracker}} \land \\
[\text{Completed}, c] \in \text{receive}_{\text{Planner} \rightarrow \text{Tracker}}.
\] (31)

- If the Tracker sees a failure, it should report to the Planner:

\[
\forall c \in C :: \neg c.car.healthy \\
\Rightarrow [\text{Failed}, c] \in \text{receive}_{\text{Planner} \rightarrow \text{Tracker}}.
\] (32)

- Guarantees
  - When receiving a pair of configurations from the Supervisor, the Planner should send a path to the Tracker such that the starting and ending configurations of the path match the received configurations or if this is not possible, send Blocked to the Supervisor:

\[
\forall c \in C :: \exists (p_0, p_1) \in \mathbb{R}^6 :: \\
[p_0, p_1] \in \text{receive}_{\text{Planner} \rightarrow \text{Supervisor}} \\
\Rightarrow (\exists p \in \mathbb{P} :: p(0) = p_0 \land p(1) = p_1) \\
\land [p, c] \in \text{send}_{\text{Planner} \rightarrow \text{Tracker}} \\
\lor [\text{Blocked}, c] \in \text{send}_{\text{Planner} \rightarrow \text{Supervisor}}.
\] (33)

  Only send safe paths with ω-feasible curvature:

\[
\forall c \in C :: \exists p \epsilon \mathbb{P} :: p, c \in \text{send}_{\text{Planner} \rightarrow \text{Tracker}} \\
\Rightarrow \kappa\text{-feasible}(p) \land \Gamma(d)(p) \subseteq G.drivable_{\text{area}}.
\] (34)

- If receiving a task status update from the Tracker, eventually forward it to the Supervisor:

\[
\forall c \in C :: [m, c] \in \text{receive}_{\text{Planner} \rightarrow \text{Tracker}} \\
\land m \in \{\text{Failed, Completed}\} \\
\Rightarrow [m, c] \in \text{send}_{\text{Planner} \rightarrow \text{Supervisor}}.
\] (35)

- If the Planner receives a Blocked signal from the Tracker, it attempts to fix it, otherwise forwards it to the Supervisor:

\[
\forall c \in C :: \forall t :: [\text{Blocked}, c] \in \text{receive}_{\text{Planner} \rightarrow \text{Tracker}} (t) \\
\Rightarrow \exists p \epsilon \mathbb{P} :: [p, c] \in \text{send}_{\text{Planner} \rightarrow \text{Tracker}} (t + \epsilon) \\
\land \forall' \leq t + \epsilon :: [p, c] \not\in \text{send}_{\text{Planner} \rightarrow \text{Tracker}} (t') \\
\lor [\text{Blocked}, c] \in \text{send}_{\text{Planner} \rightarrow \text{Supervisor}} (t + \epsilon).
\] (36)

**Contract 4 (C_{Tracker}):** The contract for the tracking component is as follows:

- Assumptions
  - Any path command from the Planner is always κ-feasible, the corresponding corridor is drivable, and the car configuration upon receiving the command is in the initial portion of the corridor:

\[
\forall c \in C :: \forall p \epsilon \mathbb{P} :: \forall t :: \text{starts}_{\text{at}}([p, c] \\
\in \text{receive}_{\text{Tracker} \rightarrow \text{Planner}}, t) \\
\land \kappa\text{-feasible}(p) \land c.car.state \epsilon \Gamma_3(p, 0) \land \\
\Gamma_3(p) \subseteq G.drivable_{\text{area}}.
\] (37)

- Commands are not modified by the CustomerInterface:

\[
\text{See [12] and [13].}
\] (38)

- Guarantees
  - Make sure car stays in the latest sent p’s corridor Γ_3(p):

\[
\forall c \in C :: \forall t :: \exists p \in \mathbb{P} :: p' \in \mathbb{P} :: \\
\[p', c] \in \text{send}_{\text{Planner} \rightarrow \text{CustomerInterface}} \\
\Rightarrow v_{\text{min}} \leq v \land v \leq v_{\text{max}} \land \phi_{\text{min}} \leq \phi \land \phi \leq \phi_{\text{max}}.
\] (39)

  - Never drive into a dynamic obstacle (customer or car):

\[
\forall c_1, c_2 \in C :: \exists \theta :: (c_1 \neq c_2 :: \\
|| (c_1, \text{car}x, c_1, \text{car}y) - (c_2, \text{car}x, c_2, \text{car}y) || \geq \min_{\text{car}} \\
\land || (c_1, \text{car}x, c_1, \text{car}y) - (c_2, \text{car}x, c_2, \text{car}y) || \geq \min_{\text{people}}).
\] (40)

  - If a car fails, it must report to the Planner:

\[
\forall c \in C :: \neg c.car.healthy \\
\Rightarrow [\text{Failed}, c] \in \text{send}_{\text{Tracker} \rightarrow \text{Planner}}.
\] (41)

- If a car is healthy then it must “track” the last sent path from the Planner:

\[
\forall c \in C :: \exists p_0, p_1 \epsilon \mathbb{P} :: p' \epsilon \mathbb{P} :: \\
[p', c] \in \text{receive}_{\text{Tracker} \rightarrow \text{Planner}} \\
\Rightarrow \exists p :: c.car.state \epsilon \Gamma_3(p, 1).
\] (42)
– When the Tracker completes its task according to a corridor map δ, it should send a report to the Planner module:

$$\text{See } (31). \quad (44)$$

– If a car is blocked (i.e., there is a failed car in its current corridor), then the Tracker must report Blocked to the Planner:

$$\forall c \in C :: \forall t :: \exists p \in P :: ([p, c] \in \text{receive}_{\text{Tracker} \to \text{Planner}}(t)) \implies \forall p' \in P :: [p', c] \in \text{receive}_{\text{Tracker} \to \text{Planner}}(t') \implies \exists [p, c] \in \text{receive}_{\text{Tracker} \to \text{Planner}}(t') \implies (\exists \bar{c} \in C :: c' \neq c \wedge \neg c'.\text{car.healthy} \wedge c'.\text{car.state}(t) \in \Gamma_p(p) \wedge \neg [\text{Blocked}, c] \in \text{send}_{\text{Tracker} \to \text{Planner}}). \quad (45)$$

VI. SYSTEM DESIGN

A. Simulation Environment and Implementation

The proposed design framework was demonstrated via simulation of an automated valet parking (AVP) system [23]. It consists of the layout of a parking lot (Fig. 1), as well as multiple cars that arrive at the drop-off location of the parking lot and are parked in one of the vacant spots by the AVP system. Once the customer requests their car, it is returned to the pick-up location. The asynchronicity is captured by modeling each component as a concurrent process using Python async library Trio [24]. The communication between the layers is implemented using Trio’s memory_channel. In particular, each channel is a first-in-first-out queue which ensures losslessness. The architecture is described in Figure 2. In this setup, the cars may experience failures and report them to the Tracker module. The failures considered in this demonstration are a blocked path, a blocked parking spot, and a total engine failure resulting in immobilization.

The benefit of the directive-response architecture becomes apparent when failures are introduced into the system. Upon experiencing a failure, a component that is higher in the hierarchy will be alerted through the response it receives. If possible, the failure will be resolved, e.g., through the re-planning of the path or assigning a different spot. Every layer has access to its contingency plan, consisting of several predetermined actions according to the possible failure scenarios and corresponding responses it receives. In some cases (e.g., complete blockage of a car), when no action can resolve the issue, the cars have to wait until the obstruction is removed. We assume that only broken cars can be towed, and when a car breaks down, it will take a specified amount of time until it is towed.

B. CustomerInterface Modeling

In our simulation, customers are responsible for driving their cars into the parking garage and depositing them at the drop-off area with an admissible configuration before sending a Park directive to the Supervisor and stay there until they get a response. This is satisfied as long as the customer drops off their vehicle behind the green line such that the heading of the vehicle is within the angle bounds ω and 3ω as shown in Figure 4 with the projection w of the vehicle onto the green edge of the blown-up entrance box shown in Figure 5. Thus, CustomerInterface satisfies $G_{\text{(14)}}$. If the Park directive is Rejected by the Supervisor, the customer is assumed to be able to leave the garage safely (satisfying $G_{\text{(15)}}$). If the car is Accepted then the customer will leave the control of the car to the Tracker (satisfying $G_{\text{(16)}}$ and $G_{\text{(17)}}$). The customer is assumed to eventually send a Retrieve directive to the Supervisor, after their car is Accepted (satisfying $G_{\text{(18)}}$ and $G_{\text{(19)}}$). Once the vehicle is Returned, the customer is assumed to be able to pick it up and drive safely away. All pedestrians in the parking lot are customers, and they are constrained to only walk on the walkable area and never stay on a crosswalk forever (thus satisfying $G_{\text{(20)}}$ and $G_{\text{(21)}}$). When a car fails, it becomes immobilized until it is towed ($G_{\text{(22)}}$). From this, it follows that CustomerInterface satisfies $C_{\text{CustomerInterface}}$.

C. Supervisor Implementation

At any time, the Supervisor knows the total number of cars that have been accepted into the garage, which is represented by the variable num_active_customers, and is designed to accept new cars when this number is strictly less than the total number of parking spots $G_{\text{parking spots}}$. This implies $G_{\text{(23)}}$ is satisfied. Overall, this ensures all directives will get a response, yielding $G_{\text{(24)}}$. Whenever the Supervisor receives a Completed signal, it will check if the car is the return area. If it is, then the Supervisor will send a Returned signal to the CustomerInterface in compliance with $G_{\text{(25)}}$. If the Supervisor ever accepts a new car, or receives a Blocked signal from the Planner, or a Retrieve request it will send a start configuration compatible with the car’s current state as well as an end configuration to one of the parking spaces in the former case and to a place in the return area in the latter. This guarantees $G_{\text{(26)}}$.

Proposition 1: $M_{\text{Supervisor}}$ satisfies $C_{\text{Supervisor}}$

Let $M$ denote our implementation of the Supervisor and $\sigma \in M$. We want to show that

$$\sigma \in \bigcap_{i \in [24]} A_{(i)} \implies \sigma \in \bigcap_{i \in [30]} G_{(i)}.$$  

From the description of the Supervisor implementation, we conclude $\sigma \in G_{\text{(24)}} \land G_{\text{(25)}} \land G_{\text{(26)}} \land G_{\text{(27)}} \land G_{\text{(28)}}$. Since $\sigma \in A_{\text{(24)}}$ and because in our implementation whenever the Supervisor receives a Completed signal it will alert the customer of the corresponding status, our implementation satisfies $G_{\text{(29)}}$.

Proof:
D. Planner Implementation

The Planner computes paths that cover the parking spots, as well as the entry and exit areas of the parking garage, which are \( \kappa \)-feasible for a car that satisfies (7) such that the corresponding \( \delta \)-corridor is on \( G_{\text{drivable area}} \). Given a maximum allowable curvature, a grid discretization scheme is based on a planning grid whose size is computed to provide full lot coverage and satisfy the curvature bounds, as depicted in Figure 3. For every specified grid size, the algorithm will check if the planning graph is appropriate by determining how well the parking lot is covered. Only a grid size that provides full coverage of the lot is chosen for path planning. The dynamical system specified in (7) is differentially flat [25]. In particular, it is possible to compute all states and inputs to the system, given the outputs \( x, y \), and their (in this case, up to second order) derivatives. Specifically, the steering input is given by

\[
\varphi(t) = \arctan(f \kappa(t)), \quad (46)
\]

where \( \kappa(t) \) is the curvature of the path traced by the midpoint of the rear axle at time \( t \) given by

\[
\kappa(t) = \frac{\ddot{y}(t) \dot{x}(t) - \ddot{x}(t) \dot{y}(t)}{(\dot{x}^2(t) + \dot{y}^2(t))^{\frac{3}{2}}}. \quad (47)
\]

The task of tracking a given path can be shown to depend only on how \( \varphi(t) \) is constrained. For practical purposes, let us assume \( |\varphi(t)| \leq B \) for some \( B > 0 \). Then by Equation (46), tracking feasibility depends on whether the maximum curvature of that path exceeds \( \frac{\tan(B)}{\ell} \). For our implementation this is assumed to be 0.2 \( m^{-1} \). This problem has been studied in [26] in the context of rectangular cell planning. We apply the algorithm described therein for a Type 1 path (CBTA-S1) to a rectangular cell while constraining the exit configuration to a heading difference of \( \pm 5^\circ \) and a deviation of \( \pm 0.5 \, m \) from the nominal path. The setup and the resulting initial configuration, for which traversal is guaranteed, are shown in Figure 3 and Figure 5. The initial car configuration can be anywhere on the grid segment entry edge, as long as it is between the lower bound \( \alpha \) and the upper bound \( \bar{\alpha} \). By passing through this initial funnel segment, the car will transition itself onto the planning grid. Therefore, it remains to be verified that each path generated from the grid is guaranteed to have a maximum curvature that is smaller than \( \kappa \). An example path through the parking lot and corresponding curvature and initial grid segment layout is shown in Figure 5.

The Planner guarantees that all trajectories generated meet this maximum curvature requirement. In addition to satisfying \( G_{\text{grid}} \), any execution of the Planner also satisfies \( G_{\text{Car}} \) and \( G_{\text{Seg}} \) because either the Planner can generate a feasible path or it will send a Blocked signal to the Supervisor. When the Planner receives a Blocked signal from the Tracker it will either attempt to find a different path on the planning graph or report this to the Supervisor. This satisfies \( G_{\text{Path}} \).

E. Tracker Implementation

The Tracker receives directives from the Planner consisting of trackable paths and sends responses according to the task status to the Planner. The Tracker sends instructions to the vehicles in \( G_{\text{interior}} \) and guarantees no collisions by sending a brake signal when necessary to ensure a minimum safe distance is maintained at all times. The tracking algorithm that we use is an off-the-shelf MPC algorithm from [27].

To ensure that the vehicles stay in the \( \delta \)-corridors, given knowledge of the vehicle’s dynamics, we can synthesize motion primitives that are robust to a certain disturbance set \( \Delta_{\text{Car}} \) (see Figure 2). Algorithms for achieving this have been proposed and implemented, for example, in [28] for nonlinear, continuous-time systems and for affine, discrete-time systems in [29]. In our implementation we used a simplified approach, which ensures that a backup controller for the car gets activated if the car approaches the boundary of the \( \delta \)-corridor and ensures that the car will merge onto the path again. Once it reaches the original path, the tracking of the remaining path will continue.
By $A_{[3]}$, any new path command $[p, c]$ sent down from the Planner module is assumed to be $\kappa$-feasible and have a drivable $\delta$-corridor, the initial portion of which contains $c.car$ at that time. In our implementation, we ensure that every time this happens, $c.car$ is stationary. And under this condition, we were able to confirm by testing that a car controlled by the MPC algorithm can track the corresponding $\delta$-corridors of a diverse enough set of paths, thus satisfying $G_{[30]}$ and $G_{[32]}$. The MPC algorithm is configured to output properly bounded control inputs, thus satisfies $G_{[33]}$. In addition, our implementation satisfies $G_{[34]}$, $G_{[35]}$ and $G_{[36]}$ by construction. And finally, we can guarantee $G_{[37]}$ by Property [49].

VII. CORRECTNESS OF THE COMPOSED SYSTEM

In this section, we will show that our implementation of the AVP is correct and satisfies the overall system specification by leveraging the modularity provided by the contract based design. We start by composing the AVP components, namely the Supervisor, the Planner, and the Tracker and then computing the quotient of the overall specification and the composed contract. Then we will show that our contract for the CustomerInterface is a refinement of this quotient.

A. Contract Composition

As part of the final verification step, we will be taking the composition of the component contracts and showing that our overall system implementation satisfies this composition. This will imply that the composition is consistent.

Given two saturated contracts $C_1$ and $C_2$, their composition $C_1 \otimes C_2 = (A, G)$ given by [1]:

$$G = G_1 \land G_2$$

$$A = \bigvee_{A' \in A} A'$$

where

$$A = \begin{cases} A' | A' \land G_2 \Rightarrow A_1 \\ A' \land G_1 \Rightarrow A_2 \end{cases}$$

A nice property of the composed contract is that if $M_1$ satisfies $C_1$ and $M_2$ satisfies $C_2$ then $M_1 \times M_2$ satisfies $C_1 \otimes C_2$. Using the fact that the composition operator $\otimes$ is associative and commutative, a straightforward calculation yields the following more explicit form for the composition of $N$ saturated contracts $(A_i, G_i)_{i=1}^N$.

$$G = \bigwedge_{i=1}^N G_i$$

$$A = \bigvee_{i=1}^N A_i \lor \neg \bigwedge_{i=1}^N G_i$$

If $A \neq \emptyset$, then the composed contract is compatible. The contract is consistent if there exists an implementation for it, namely $G \neq \emptyset$ if it is saturated. For our AVP system, we will show that our composed implementation also satisfies the composed contract in a non-vacuous way, meaning it satisfies all guarantees of the component contracts simultaneously. In the composition, an acceptable behavior satisfies the following properties, namely the operation of the car inside the garage:

1) The Supervisor rejects the car due to the lack of reachable, vacant spots. The car will not enter the garage.

2) A car which was dropped off correctly in the deposit area is accepted by the Supervisor by $G_{[39]}$.
   a) Accepted, no contingency: The Tracker by $A_{[38]}$ takes over control. After this, the Supervisor must send a directive in the form of a pair of configurations to the Planner $G_{[30]}$, which in turn must send to the Tracker a safe and feasible path (satisfying $A_{[31]}$) such that the starting and ending configurations of the path match the received configurations $(G_{[32]}, G_{[33]})$. Upon receiving the path from the Planner, the Tracker ensures that the car stays in the corridor of the path $G_{[34]}$ and ensures that it will make progress on that path (this satisfies $G_{[35]}$). It will accomplish this while sending compatible inputs to the customer’s car $G_{[36]}$ and not driving it into people and other cars $G_{[37]}$. When the CustomerInterface sends a Retrieve command, the above process repeats with the Supervisor, which ensures that the last configuration is in the return area, thus satisfying $G_{[38]}$. If this is the last sent path, then upon reaching the end of the path, it should notify the Planner module that it has completed the task by $G_{[39]}$ which satisfies $A_{[39]}$, $A_{[40]}$, and $A_{[41]}$. The Supervisor alerts the CustomerInterface of the completed return by $G_{[40]}$.

3) Accepted, with problems: If the car is accepted and at any time during the above process:
   a) The car fails (hence, cannot move by $G_{[40]}$), the Tracker will send a Failed message to the Planner by $G_{[41]}$ satisfying $A_{[42]}$, and $G_{[42]}$ will be forwarded to the Supervisor. This satisfies $A_{[43]}$, which together with $A_{[44]}$, will imply that the failed car will eventually be towed.
   b) The car is Blocked, the Tracker will report to the Planner by $G_{[45]}$, which will try to resolve or alert the Supervisor satisfying $G_{[46]}$.

B. Contract Quotient

For saturated contracts $C$ and $C_1$, the quotient is defined in [15] as follows:

$$C/C_1 := (A \land G_1 \land A_1 \land G \lor \neg(A \land G_1))$$

Quotienting out the composed contract from the overall system specification should yield the required customer behavior. The composed system was computed to be

$$C_{AVP} = ((A_S \land A_P \land A_T) \lor \neg(G_S \land G_P \land G_T)), G_S \land G_P \land G_T$$

with the assertions $G_1$ and $A_i$ of the Supervisor, Planner and Tracker contract $C_i$ in saturated form. With the contract for the overall system defined as:

Contract 5 ($C_{System}$): The contract for the overall system is as follows:
• Assumes
  – Any circumstances.
• Guarantees
  – Never any collisions (safety).
  – Always healthy cars will eventually be returned (liveness).

When computing the quotient of the overall system specification and the composed system, the resulting assumptions and guarantees are the following. Assuming that the AVP components work correctly (e.g. provide their respective guarantees), the customer must guarantee that all assumptions that the AVP components make on the customer are valid, while ensuring safety and progress. Meaning the customer need to provide the following guarantees:

• Guarantees:
  – The customer will drop off the car correctly satisfying $A$, $A'$.
  – The customer will not interfere with the car controls after the drop-off satisfying $A'$.
  – The customer needs to ensure progress by not blocking the path forever, and eventually requesting and picking up the car.
  – The customer will not take any action towards collision ensuring safety.

Our customer contract refines the contract with the above mentioned guarantees. $A$ and $A'$ are satisfied by $G$ and $G'$. The safety property is guaranteed by the customer staying in the walkable area by $G$. Progress is ensured by $G$, $G'$, and $G'$. Our CustomerInterface contract includes the guarantees generated from the quotient and thus is a refinement of this contract.

We will now show specifically that the composed system satisfies the safety and progress properties ($G$ and $G'$):

Property 3 (Safety):

$$\forall c_1,c_2 \in C : (c_1 \neq c_2 \Rightarrow ||(c_1.car.x,c_1.car.y) - (c_2.car.x,c_2.car.y)|| \geq \varepsilon_{\text{min},\text{car}}) \land
\Vert (c_1.car.x,c_1.car.y) - (c_2.x,c_2.y) \Vert \geq \varepsilon_{\text{min},\text{people}})$$

(49)

Proof: (Sketch) For each vehicle in the parking lot, the following invariance is maintained. There will be no collisions, as the Tracker checks the spatial region in front of the car and brings it to a full stop in case the path is blocked by another agent (car or pedestrian). The minimum distance to an obstacle is determined by a minimum braking distance. Furthermore, the environment does not take actions, which will lead to an inevitable collision due to the constraints on the pedestrian dynamics.

Property 4 (Liveness):

$$\forall c \in C : (\Box c.car.healthy \land \Box [\Box \neg \Box \neg \text{Blocked},c])$$

(50)

Proof: (Sketch) Consider the parking lot topology shown in Figure 1. Let $c \in C$ and $c.car.healthy$. Assume that $c$ sends a Retrieve message to the Supervisor. For each $t$, let us define $f(t)$ to be the number cars between $c.car$ and its destination. Clearly, $f(t) \geq 0$ for any $t$ and $f(t)$ is well-defined because for the topology being considered, we can trace out a line that starts from the entrance area, going to any one of the parking spots and ending at the return area without having to retrace our steps at any time. We will show that there exists a $t' \geq t$ such that $f(t') = 0$, implying that there is no longer any obstacle between $c$ and its destination. Next, we claim that $\forall t,t' :: t' > t :: f(t) \geq f(t')$. This is true because:

• The parking lot topology and the safety measures do not allow for overtaking.
• The area reservation strategy implemented in the Supervisor prevents an increase in $f$ upon re-routing to avoid a failed car. A notable detail is that if $c.car$ is trying to back out of a parking spot, a stream of cars passing by can potentially block it forever. This is resolved by having $c.car$ reserve the required area so that once any other car has cleared this area, $c.car$ is the only one that has the right to enter it.

Finally, we will show that $\forall t,t' :: t' > t :: f(t) > f(t')$. Let $c'$ be such that $c'.car$ is between $c.car$ and its destination. By the dynamical constraint on pedestrians and by assumptions $A$ and $A'$, they will not block cars forever. Our algorithm guarantees that one of the following will happen at some time $t' > t$:

1) $c'.car$ is picked up by $c'$.
2) $c'.car$ is parked and $c.car$ drives past it
3) $c'.car$ drives past $c.car$’s destination.
4) $c'.car$ breaks down and by $A$ is eventually towed.

It is easy to see that each of these events implies that $f(t) > f(t')$. Since $f$ is an integer and cannot drop below 0, the result follows.

VIII. SUMMARY AND FUTURE WORK

We have formalized an assume-guarantee contract variant with communication via a directive-response framework. We then used it to write specifications and verified the correctness of an AVP system implementation [23]. This was done separately for each module and everything together as a complete system.

The application of this framework in the AVP can be extended to more agent types, for example, human-driven cars and pedestrians that do not necessarily follow traffic rules at all times. A contract between the valet driven cars and the human-driven cars will be needed to ensure the safe operation of the parking lot, and in the event that a human-driven car violates the contract, cars controlled by the system need to be able to react to this situation safely. More failure scenarios such as communication errors (message loss, cyberphysical attacks etc.) may also be included.

IX. ACKNOWLEDGMENTS

This research was supported by DENSOL International America, Inc and National Science Foundation award CNS-1932091.
S. S. Bauer, A. David, R. Hennicker, K. G. Larsen, A. Legay, A. Yamazaki, Y. Izumi, K. Yamane, T. Nomura, and Y. Seike, Siemens, “Improving autonomous valet parking with simulation and Bosch, “Ford, bedrock and bosch are exploring highly automated Bosch, “Automated valet parking service,” mar 2020.

I. Romeo, A. Sangiovanni-Vincentelli, C.-W. Lin, and E. Kang, “Quotient for assume-guarantee contracts,” in Proceedings of the 16th ACM-IEEE International Conference on Formal Methods and Models for System Design, pp. 67–77, IEEE Press, 2018.

T. Wongpiromsarn and R. M. Murray, “Distributed mission and contingency management for the darpa urban challenge,” in International Workshop on Intelligent Vehicle Control Systems (IVCS), vol. 5, 2008.

D. Dvorak, R. Rasmussen, G. Reeves, and A. Sacks, “Software architecture themes in jpl’s mission data system,” in 2000 IEEE Aerospace Conference. Proceedings, vol. 7, pp. 259–268, IEEE, 2000.

M. D. Ingham, R. D. Rasmussen, M. B. Bennett, and A. C. Moncada, “Engineering complex embedded systems with state analysis and the mission data system,” Journal of Aerospace Computing, Information, and Communication, vol. 2, no. 12, pp. 507–536, 2005.

R. D. Rasmussen, “Goal-based fault tolerance for space systems using the mission data system,” in 2001 IEEE Aerospace Conference Proceedings (Cat. No. 01TH8542), vol. 5, pp. 2401–2410, IEEE, 2001.