University of Nevada, Reno

Composing Controllers for Team Coherence Maintenance with Collision Avoidance

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science
with a major in Computer Science.

by

Andrew Kimmel

Dr. Kostas E. Bekris, Thesis Advisor

August 2012

THE GRADUATE SCHOOL

We recommend that the thesis prepared under our supervision by

Andrew Kimmel

entitled

Composing Controllers for Team Coherence Maintenance with Collision Avoidance

be accepted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Dr. Kostas E. Bekris, Ph.D., Advisor

Dr. Eelke Folmer, Ph.D., Committee Member

Dr. Frederick Harris, Ph.D., Committee Member

Dr. Yasuhiko Sentoku, Ph.D.,Graduate School Representative

Marsha H. Read, Ph.D., Associate Dean, Graduate School

August 2012

Abstract

Many multi-agent applications may involve a notion of spatial coherence. For instance, simulations of virtual agents often need to model a coherent group or crowd. Alternatively, robots may prefer to stay within a pre-specified communication range. This work proposes an extension of a decentralized, reactive collision avoidance framework, which defines obstacles in the velocity space, known as Velocity Obstacles (VOs), for coherent groups of agents. The extension, referred to in this work as a Loss of Communication Obstacle (LOCO), aims to maintain proximity among agents by imposing constraints in the velocity space and restricting the set of feasible controls. If the introduction of LOCOs results in a problem that is too restrictive, then the proximity constraints are relaxed in order to maintain collision avoidance. A weighted velocity selection scheme is utilized in order to steer agents towards their goals, as well as agents which are farther away and thus might be violating proximity constraints. The approach is fast and integrates nicely with the Velocity Obstacle framework. It is shown to yield improved coherence for groups of robots connected through an input constraint graph, while moving with constant velocity. The approach is evaluated for collisions, computational cost and proximity constraint maintenance through a series of simulated environments, which have single and multiple team variations. The experiments show that improved coherence is achieved while maintaining collision avoidance, at a small computational cost and path quality degradation.

In order to experimentally validate new algorithms, such as the LOCO approach, having a software infrastructure capable of running a plethora of algorithms on many agents is needed. When moving from virtual agents to robotic systems, these algorithms take the form of a controller. Composing various controllers together to

create a diverse testbed environment is also essential for the development process of new algorithms. Thus, this work additionally describes a software infrastructure for developing controllers and planners for robotic systems, referred to here as `PRACSYS`. At the core of the software is the abstraction of a dynamical system, which, given a control, propagates its state forward in time. The platform simplifies the development of new controllers and planners and provides an extensible framework that allows complex interactions between one or many controllers, as well as motion planners. For instance, it is possible to compose many control layers over a physical system, to define multi-agent controllers that operate over many systems, to easily switch between different underlying controllers, and plan over controllers to achieve feedback-based planning. Such capabilities are especially useful for the control of hybrid and cyber-physical systems, which are important in many applications. The software is complementary and builds on top of many existing open-source contributions. It allows the use of different libraries as plugins for various modules, such as collision checking, physics-based simulation, visualization, and planning. This work describes the overall architecture, explains important features and provides use-cases that evaluate aspects of the infrastructure.

Acknowledgements

This work has been supported by:

- The National Science Foundation under grant: CNS 0932423

I would like to thank my advisor Dr. Kostas Bekris, who provided me much assistance in editing my thesis. I would also like to thank my committee members for their help. Finally, I would like to give special thanks to my family and to my colleagues from the PRACSYS group for all of their support and assistance.

Contents

Abstract **i**

Acknowledgements **iii**

List of Figures **v**

List of Tables **vii**

1 Introduction **1**
1.1 Team Coherence and Collision Avoidance 1
1.2 Composition and Evaluation of Controllers 4

2 Background **7**
2.1 Team Coherence and Collision Avoidance 7
2.1.1 Virtual Agent Applications 7
2.1.2 Coupled Multi-Robot Path Planning 7
2.1.3 Decoupled Multi-Robot Path Planning 8
2.1.4 Formations 8
2.1.5 Reactive Obstacle Avoidance 9
2.1.6 Control-Based Obstacle Avoidance 9
2.1.7 Contribution 10
2.2 Controller Composition and Simulation Environments 10

3 Decentralized Team Coherence Maintenance under the Velocity Obstacle Framework **12**
3.1 Problem Statement 12
3.2 Velocity Obstacle Framework 13
3.3 Loss of Communication Obstacles 15
3.4 Integration of VOs and LOCOs 17
3.5 Velocity Selection 18

4 Creating an Extensible Architecture for Composing Controllers and Planners **21**
4.1 Architecture 21
4.2 Descriptions 22
4.2.1 Ground-truth Simulation and Controller Architecture 22

4.2.2 Planning 27
4.2.3 Visualization 30
4.2.4 Other PRACSYS Packages 30

5 Evaluation **31**
5.1 LOCO Results 31
5.1.1 Single Team 32
5.1.2 Multiple Teams 33
5.2 PRACSYS Use Cases 36
5.2.1 Showing Scalability for Multiple Agents 36
5.2.2 Planning over Controllers using LQR Trees 37
5.2.3 Controller Composition in Physics-based Simulation 38
5.2.4 Integration with Octave, OMPL, and MoCap Data 39

6 Discussion and Future Work **40**
6.1 LOCO 40
6.2 PRACSYS 41

List of Figures

1.1 Two agents navigating around a static obstacle: (a) the agents split to reach their goals without collisions, while in (b) the agents move together as a coherent team. The second behavior must be achieved in a decentralized manner. 2

3.1 Agents a and b share a proximity constraint d_{prox}. Agent a moves with velocity v_a where $\|v\| = s$ and can sense all agents within d_{sense}. . . . 13

3.2 Construction of $VO^{\infty}_{a|b}$ for an infinite time horizon. The Minkowski sum of a and b, $a \oplus b$ is used to define a cone in velocity space, which is then translated by v_b. The shaded region represents all velocities v_a, which lead a into collision with b. 14

3.3 Construction of $\texttt{LOCO}^{\tau}_{a|b}$ and $\texttt{VVC}^{\tau}_{a|b}$. The circular region represents the viable velocities to maintain $d(a, b, t) \leq d_{prox}$ for time horizon τ. The shaded region outside the disk represents invalid velocities for agent a. 15

3.4 The conservative approximation of $\texttt{VVC}^{\tau}_{a}$ in the velocity space of agent a for two (left) and three (right) neighbors. The white circle corresponds to velocities that are guaranteed to maintain connectivity with the neighbors for time τ. 17

3.5 An example of how changing the horizon affects the set of feasible controls. The left image has a larger value for τ while the right has a smaller value of τ. Larger values of τ provide stronger guarantees for communication maintenance, but make finding a feasible control more difficult. 18

4.1 Package interactions. `ROS` nodes communicate via message passing: *simulation*, *visualization*, and *planning*. The *common* and *utilities* packages are dependencies of the previous three. 21

4.2 A view of the class inheritance tree for the `PRACSYS` *system*. 23

4.3 The core interface of a system. 24

4.4 The general structure of the *planning* modules. The *task planner* contains multiple *motion planners*. The *task planner* also contains a *world model* and communicates with the *simulation* node. 28

5.1 The environments on which the experiments were executed 32

5.2 Examples of input proximity graphs used in the experiments. 33

5.3 Coherence over time for 24 agents in the wedge grid experiment. . . . 34

5.4 An example of the paths taken by 24 agents in the `PACHINKO` environment for the `RVO` (left) and `LOCO` (right) approach. 35

5.5 A plot of simulation steps vs number of plants. 36
5.6 3000 plants running in an environment. 37
5.7 A visual representation of the controller composition for controlling a bipedal robot 38

List of Tables

5.1 Results for a single team. 33
5.2 Results for multiple teams. 35

Chapter 1 Introduction

This chapter introduces the two major contributions of this work: team coherence and controller composition. The aim of this chapter is to help familiarize readers with the material and to provide an outline of the structure of the thesis.

1.1 Team Coherence and Collision Avoidance

There are many practical applications of decentralized collision avoidance techniques. For instance, crowd simulations often have hundreds of agents moving in a dynamically changing environment. While it is important for agents to not collide with each other and the obstacles, in order to run the simulation in real time, it is also equally important to reduce all communication overhead. However, sometimes only providing decentralized collision avoidance may not be sufficient, as some applications may involve a secondary objective, where teams of agents need to maintain a certain level of coherence. Coherence often implies that the agents should remain within a certain distance of one another. In games and simulations, the agents may need to remain within a certain distance because of implied social interactions, or because they need to reach their destination together so as to be more effective in completing an objective at their goal. For instance, a team of agents in a game will be more effective in attacking an enemy if all the units move together against the opponent and do not split into multiple groups. In mobile sensor networks, robots may have to respect radial communication limits.

The Velocity Obstacle (`VO`) formulation [14, 43, 51] is a framework for reactive collision avoidance. It is fast as it operates directly in the velocity space of each agent. It is also a decentralized approach as each agent reasons independently about its controls, as long as it can compute the position and velocity of its neighbors. There is also anonymity between robots, as the framework does not have to reason about specific agents, which is an important distinction for computational efficiency.

Current work in Velocity Obstacles does not directly address the issue of coherence. Consider the situation in Figure 1.1, where two agents are required to avoid an obstacle and reach their desired destination. An application of the basic `VO` framework may result in the two agents splitting and passing the obstacle from opposite ends. It would be desirable, however, for the agents to select, in a decentralized manner, a single direction to follow so as to avoid the obstacle and reach their goal while maintaining coherence. The decentralized nature of the solution will be especially helpful in robotic applications because no communication will be required between robots. It will also provide improved scalability in simulations and games.

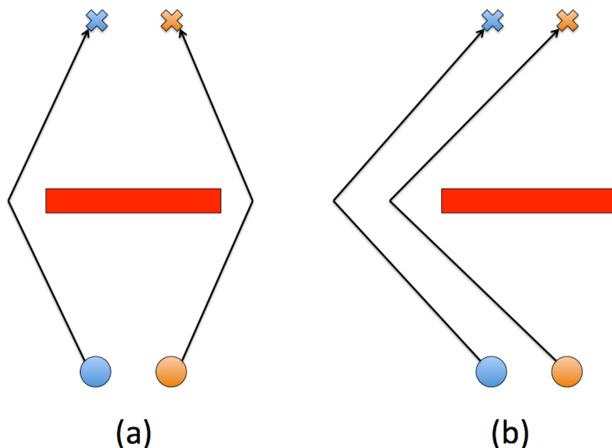

Figure 1.1: Two agents navigating around a static obstacle: (a) the agents split to reach their goals without collisions, while in (b) the agents move together as a coherent team. The second behavior must be achieved in a decentralized manner.

This work proposes a method for maintaining team coherence within the `VO` framework in a decentralized manner. The desired coherence for a problem can be defined as a graph of dependencies between agents. An edge in the graph implies that the corresponding agents should remain within a predefined distance as they move towards their goal. Given this input graph, an agent constructs an additional obstacle in the velocity space for each neighbor with which it wants to maintain connectivity. This construction is referred to in this work as a Loss of Communication Obstacle (`LOCO`).

Building on top of the VO framework allows LOCOs to focus on coherence maintenance, rather than obstacle avoidance. In order to construct the LOCO, the assumption is that a neighbor will maintain its current control, as in the original VO framework. Then, a LOCO defines the set of velocities that will lead the two agents to be separated beyond a desired distance within a certain time horizon. A scheme is proposed for integrating information from the multiple VOs defined for collision avoidance and the LOCOs defined for coherence maintenance. If the set of velocities that satisfy both constraints is not empty for a satisfactory time horizon, then the velocity in this set that brings the agent closer to its goal is chosen. A valid velocity implies that, given the neighbor does not change control, following this velocity will not lead the agent into a collision or violation of proximity constraints for the given horizon. If the set of valid velocities is empty, then the objective of maintaining coherence is dropped in favor of guaranteeing collision avoidance, which should be satisfiable, unless oscillations appear in the selected controls of agents. If two agents that are supposed to retain proximity end up violating the distance constraint, the proposed method makes them move in a direction that will allow them to reconnect.

This work describes the LOCO method and provides simulations which show that, by reasoning about distance constraints, it is possible to solve decentralized collision avoidance problems while improving the coherence of a team. The approach is built on top of and compared against a formulation of velocity obstacles proposed in the literature for teams of agents that execute the same protocol, referred to as Reciprocal Velocity Obstacles [43, 51]. The experiments consider disk-shaped agents that move with a constant speed in a holonomic manner, i.e., the agents can freely choose to follow any direction instantaneously. First, a series of static environments are tested with a single team of agents navigating while maintaining coherence. Different types of proximity constraints between team members are considered. Then, tests for multiple teams of agents navigating in the same environment while maintaining coherence are performed.

1.2 Composition and Evaluation of Controllers

Developing and evaluating control or motion planning methods can be significantly assisted by the presence of an appropriate software infrastructure that provides basic functionality common among many solutions. At the same time, new algorithms should be thoroughly tested in simulation before being applied on a real system. Physics-based simulation can assist in testing algorithms in a more realistic setup so as to reveal information about the methods helpful for real world application. These realizations have led into the development of various software packages for physics-based simulation, collision checking and motion planning for robotic and other physical systems, such as Player/Stage/Gazebo [20], OpenRAVE [10], OMPL [19], PQP [17], USARSim [7]. At the same time many researchers interested in developing and evaluating controllers, especially for systems with interesting dynamics, often utilize the extensive set of MatLab libraries.

There are numerous problems which require the integration of multiple controllers or the integration of higher-level planners with control-based methods. For instance, controlling cyber-physical systems requires an integration of discrete and continuous reasoning, as well as reasoning over different time horizons. Similarly, a problem that has attracted attention corresponds to the integration of task planners with motion planners so as to solve challenges that are more complex than the traditional Piano Mover's Problem. At the same time, higher-dimensional and more complex robotic platforms, including humanoid systems and robots with complex dynamics, are increasingly becoming the focus of the academic community.

This work builds on top of many existing contributions and provides an extensible control and planning framework that allows for complex interactions between different types of controllers and planners, while simplifying the development of new solutions. The focus is not on providing implementations of planners and controllers but defining an environment where new algorithms can be easily developed, integrated in an object-oriented way and evaluated. In particular, the proposed software

platform, referred to here as PRACSYS, offers the following benefits:

- **Composability:** PRACSYS provides an extensible, composable, object-oriented abstraction for developing new controllers and simulating physical systems, as well as achieving the integration of such solutions. The interface is kept to a minimum so as to simplify the process of learning the infrastructure.
- **Ease of Evaluation:** The platform simplifies the comparison of alternative methodologies with different characteristics on similar problems. For instance, it is possible to evaluate a reactive controller for collision avoidance against a replanning sampling-based or search-based approach.
- **Scalability:** The software is built so as to support lightweight, multi-robot simulation, where potentially thousands of systems are simulated simultaneously and where each one of them may execute a different controller or planner.
- **New Functionality**: PRACSYS builds on top of existing motion planning software. In particular, the OMPL [19] library focuses on single-shot planning but PRACSYS allows the use of OMPL algorithms on problems involving replanning, dynamic obstacles, as well as extending into feedback-based planning.
- **ROS Compatibility**: The proposed software architecture is integrated with the Robotics Operating System (ROS) [55]. Using ROS allows the platform to meet a standard that many developers in the robotic community already utilize. ROS also allows for inter-process communication, through the use of message passing, service calls, and topics, all of which PRACSYS takes advantage of.
- **Pluggability:** PRACSYS allows the replacement of many modules through a plugin support system. The following modules can be replaced: collision checking (e.g., PQP [17]), physics-based simulation (e.g., Open Dynamics Engine [47]), visualization (e.g., OpenSceneGraph [32]), as well as planners (e.g., through OMPL [19]) or controllers (e.g., Matlab implementations of controllers).

After reviewing related contributions, this thesis outlines the software architecture and details the two main components of PRACSYS, simulation and planning. The thesis also provides a set of use-cases that illustrate some of the features of the software

infrastructure and gives examples of various algorithms that have been implemented with the assistance of `PRACSYS`.

Chapter 2 Background

This chapter gives crucial background information on the two major parts of this work: decentralized collision avoidance with team coherence and controller composition.

2.1 Team Coherence and Collision Avoidance

There are many related works that arise when dealing with notions of teams and providing collision avoidance. These works are categorized and described in the following subsections.

2.1.1 Virtual Agent Applications

The need to move multiple agents as a coherent team arises in many virtual agent applications [36,46], ranging from crowd simulation [35], to pedestrian behavior analysis [34,37], to shepherding and flocking behaviors [27,53].

Many methods make use of "steering behaviors", with the objective of having agents navigate in a life-like and improvisational manner [40]. These steering behaviors are separated into three categories: separation alignment, and cohesion. Using combinations of these steering behaviors, it is possible to achieve higher-level goals, such as "get to the goal and avoid all obstacles", "follow this path", or "join a group of characters". These higher-level behaviors are particularly useful for objectives such as simulating realistic crowd interactions [39]. An alternative method is the social force model [18], which shares a similar goal of trying to achieve realistic interactions.

2.1.2 Coupled Multi-Robot Path Planning

There is also extensive literature on motion coordination and collision avoidance in robotics. The multi-robot path planning problem can be approached by either a coupled approach or a decoupled one [23,24]. The coupled approach plans for the composite robot, which has as many degrees of freedom as the sum of degrees of freedom

of each individual robot. Integrated with complete/optimal planners, the coupled algorithm achieves completeness/optimality. Nevertheless, it becomes intractable due to its exponential dependency on the number of degrees of freedom.

2.1.3 Decoupled Multi-Robot Path Planning

Decoupled approaches plan for each agent individually. In prioritized schemes, paths are computed sequentially and high-priority agents are treated as moving obstacles by low-priority ones [12]. Searching the space of priorities can assist in performance [5]. Such decoupled planners tend to prune states in which higher priority agents allow lower priority ones to progress, which may eliminate the only viable solutions. Search-based decoupled approaches consider dynamic prioritization and windowed search [42], as well as spatial abstraction for improved multi-agent heuristic computation [45, 54]. In particular, mobile robotic sensor networks require that robots move while maintaining communication. Techniques which attempt to tackle this issue have to balance a trade-off between centralized and decentralized planning. There are techniques that create networks of robots to compute plans in a centralized manner across distributed systems [8] or in a fully decentralized manner [4].

2.1.4 Formations

There is a significant amount of work on formation control, which is a way of moving multiple agents as a coherent team. One direction is to use a virtual rigid body structure to define the shape of a formation, and then plan for this rigid body [25]. Other techniques attempt to have more flexible structures, where interactions between robots are modeled as flexible joints [2]. An alternative is to first generate a feasible trajectory for the group's leader according to its constraints and then use feedback controllers for the followers [3, 13]. This work, however, does not share the same objectives as standard formation approaches.

2.1.5 Reactive Obstacle Avoidance

Many techniques attempt to solve the problem of collision avoidance using reactive methods. One technique used in robotics is the Dynamic Window approach, which operates directly in the velocity space of a robot, reasoning over the achievable velocities within a small time interval [15]. An alternative approach, known as the Velocity Obstacle (VO), assumes that neighboring agents will keep following their current control. Based on this assumption, it defines conic regions in velocity space, which are invalid to follow, as they lead to a collision with the neighbor at some time in the future [14]. If the future trajectory of other robots is known, non-linear VOs can be constructed [22]. The basic VO formulation can result in oscillations in motion when multiple agents execute the same algorithm. The reciprocal nature of other robots can be taken into account in order to avoid these oscillations, which leads to the definition of Reciprocal Velocity Obstacles [51]. Using this idea of reciprocity, the robots can attempt to optimally steer out of collision courses with other robots using an extension of this framework called Optimal Reciprocal Collision Avoidance (ORCA) [50]. This technique was extended to 3D cases using simple-airplane systems [44]. Further work extends the VO formulation to work with acceleration constraints as well as many kinematically and dynamically constrained systems [52].

2.1.6 Control-Based Obstacle Avoidance

A common control-based method is the Receding Horizon technique, which has been applied to this problem [26]. Another technique employed to retain safety is the use of a roundabout policy, which ensures collision avoidance between robots that follow the same policy. Such policies have been proven to be safe under certain conditions, where the robots are modeled as hybrid systems [48]. A generalized roundabout policy has been proven safe for an arbitrarily large number of robots has been proposed, but the formal verification of liveness for the policy has not been provided [33]. To address the liveness issue of this policy, further work adds minimal communication between the robots to derive a dynamic priority scheme, ensuring that at least one robot is

making progress towards its goal at all times [21].

2.1.7 Contribution

A major part of this work involves using a reactive technique to define new obstacles in the velocity space, called Loss of Communication Obstacles (LOCO). LOCOs are computed quickly and, when integrated with Velocity Obstacles, allow robots to reactively avoid static and dynamic obstacles while maintaining a better sense of coherence. The new technique does not impose communication requirements, yet maintains connectivity in a decentralized manner. This approach utilizes a weighted velocity selection method in order to move agents towards their goals, while also pushing agents towards each other if they start to move farther apart. LOCOs do not introduce any new requirements or assumptions. The approach requires that robots are able to sense the position and velocity of other robots in the scene, as well as the position of static obstacles, which are all previous requirements and assumptions from the VO framework.

2.2 Controller Composition and Simulation Environments

The Robot Operating System (ROS) [55] is an architecture that provides libraries and tools to help software developers create robotic applications. It provides hardware abstractions, drivers, visualizers, message-passing and package management. PRACSYS builds on top of ROS and utilizes its message-passing and package management. ROS was inspired by the Player/Stage combination of a robot device interface and multi-robot simulator [16]. Gazebo is focusing on 3D simulation of multiple systems with dynamics [20]. Although PRACSYS shares similar objectives with Gazebo, the former focuses mostly on a control and planning interface that is not provided by Gazebo.

There is a series of alternative simulators, such as USARSim [7], the Microsoft Robotics Developers studio [29], UrbiForge [49], the Carmen Navigation Toolkit [6], Delta3D [9] and the commercial package Webots [28]. Most of these systems focus on modeling complex systems and robots and not on defining a software infrastructure

for composing and integrating controllers and planners for a variety of challenges.

Other software packages provide support for developing and testing planners. For instance, Graspit! [30] is a library for grasping research, while OpenRAVE [10] is an open-source plugin-based planning architecture that provides primitives for grasping and motion planning for mobile manipulators or full-body humanoid robots. The current project shares certain objectives with tools, such as OpenRAVE. Nevertheless, the definition of an extensible, object-oriented infrastructure for the integration of controllers, as well as the integration of planners with controllers to achieve feedback-based planning, are unique features of `PRACSYS`. Furthermore, multiple aspects of OpenRAVE, such as the work on kinematics, are complementary to the objectives of `PRACSYS` and could be integrated into the proposed architecture. The same is true for libraries focusing on providing prototypical implementations of sampling-based motion planners, such as the Motion Strategy Library (MSL) [41] and the Open Motion Planning Library (OMPL) [19]. In particular, OMPL has already been integrated with `PRACSYS` and is used to provide concrete implementations of motion planners. The proposed infrastructure, however, allows the definition of more complex problems than the typical single-shot motion planning challenge, including challenges such as planning among moving obstacles.

The second contribution of this work involves the development of a platform for composing and evaluating controllers and motion planners. This platform, called `PRACSYS` utilizes a framework which allows both high level and low level controllers to be composed in a way that provides complex interactions between many agents. In addition to the `LOCO` algorithm being written and implemented in `PRACSYS`, a set of use cases highlighting the scalability, pluggability, robustness, and composability of the platform is provided.

Chapter 3 Decentralized Team Coherence Maintenance under the Velocity Obstacle Framework

This chapter contains information about the `LOCO` algorithm. It provides a definition of the problem, a description of the VO framework, the details of the `LOCO` algorithm, how the VOs and `LOCO`s are integrated, and finally the velocity selection scheme.

3.1 Problem Statement

Consider n planar, holonomic disks moving with constant speed s. Let the set of all agents be A. Each disk agent $a \in A$ has radius r_a and can instantaneously move with a velocity vector v_a that has magnitude s. Agents are assumed to be capable of sensing the position and velocity of other agents in the environment within a sensing radius d_{sense}. Furthermore, the agents have available a map M of the environment that includes the static obstacles. The configuration space for each agent is $Q = \mathbb{R}^2$, and it can be partitioned into two sets, Q_{free} and Q_{obst}, where Q_{free} represents the obstacle free part of the space, and Q_{obst} is the part of the space with obstacles. Each agent follows a trajectory $q_a(t)$, where t is time. Initially an agent is located at a configuration $q_a(0) = q_a^{init}$ and has a goal location q_a^{goal}.

Consider the distance $d(a, b, t)$ between agents a and b at time t (Figure 3.1). If $d(a, b, t) < r_a + r_b$, then agents a and b are said to be in collision at time t. Collisions with static obstacles occur when $q_a(t) \in Q_{obst}$. An input graph $G(A, E)$ is provided that specifies which agents need to be in close proximity. The vertices of graph G correspond to the set of agents A and an edge (a, b) implies that agents a and b must satisfy $d(a, b, t) \leq d_{prox}$, where d_{prox} is a proximity constraint.

The objective is for the agents to move from q^{init} to q^{goal} without any collisions with obstacles or among them, while satisfying as much as possible the proximity constraints specified in the input graph G. More formally, all agents should follow

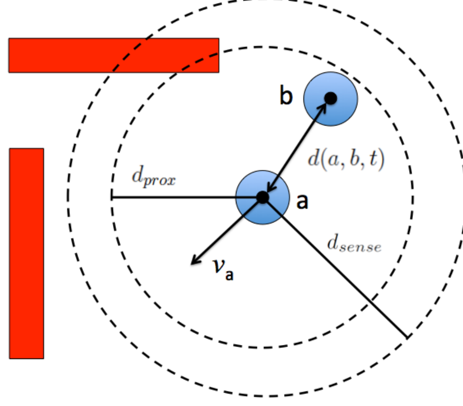

Figure 3.1: Agents a and b share a proximity constraint d_{prox}. Agent a moves with velocity v_a where $\|v\| = s$ and can sense all agents within d_{sense}.

trajectories $q_a(t)$ for $0 \leq t \leq t_{final}$, so that $\forall\ a \in A$:

- $q_a(0) = q_a^{init}$,
- $q_a(t_{final}) = q_a^{goal}$,
- $q_a(t) \in Q_{free}$, $\forall\ t \in [0, t_{final}]$,
- and $\forall\ b : r_a + r_b < d(a, b, t) < D$, where
  - $D = d_{prox}$, if $\exists (a, b) \in E$ of graph G,
  - and $D = \infty$ otherwise.

3.2 Velocity Obstacle Framework

Velocity obstacles are defined in the relative velocity space of two agents. $VO_{a|b}^{\infty}$ can be geometrically constructed as in Figure 3.2. Agent a's geometry is reduced to a point by performing the Minkowski sum of agent a and agent b: $a \oplus b$. Then, tangent lines to the Minkowski sum disk $a \oplus b$ are constructed from agent a. These tangent lines bound a conic region. This region represents the space of all velocities v_a of agent a that would eventually lead into collisions with agent b assuming that b has zero velocity. Given that agent b has a velocity v_b, the conic region needs to be translated by the vector v_b. This construction assumes an infinite time horizon. In practice, it is often helpful to truncate the `VO` based on a finite time horizon τ. This `VO` represents all velocities v_a, which will lead into a collision within time $t \leq \tau$, given

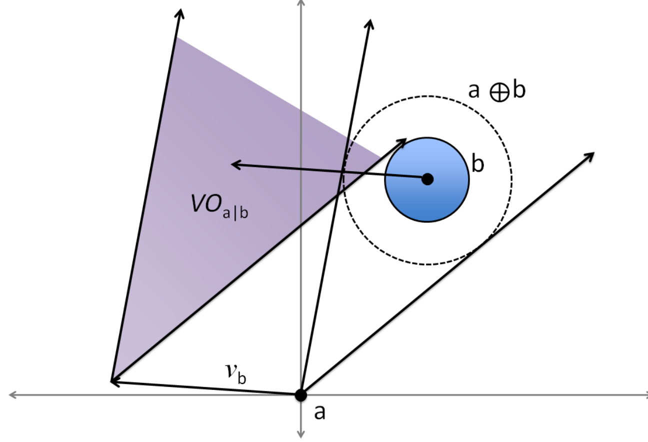

Figure 3.2: Construction of $VO_{a|b}^{\infty}$ for an infinite time horizon. The Minkowski sum of a and b, $a \oplus b$ is used to define a cone in velocity space, which is then translated by v_b. The shaded region represents all velocities v_a, which lead a into collision with b.

that agent b keeps moving with velocity v_b.

This work adopts a modification of the basic VO framework, which deals with the case that the two agents are reciprocating [51]. Reciprocal Velocity Obstacles (RVOs) are created by translating the VO according to a weighted average of the agents' velocities, $(\alpha \cdot v_A) + ((1 - \alpha) \cdot v_B)$ where α is a parameter representing the level of reciprocity between agents A and B. If both agents are equally reciprocal, then $\alpha = 0.5$.

Given this framework, an algorithm for calculating valid velocities for decentralized collision avoidance can be defined. Agent a will have a set of reachable velocities and the task is to select one such velocity, which is collision-free. For holonomic disk agents moving at constant speed s, the set of reachable velocities would be $V_{reach} = \{v | \|v\| = s\}$. Let the set of velocities which are invalid according to all the VOs for neighbors of a be defined as follows: $V_{inv} = \{v | \ \exists \ \mathtt{VO}_{a|b} \ s.t. \ v \in \mathtt{VO}_{a|b}, b \in A, a \neq b\}$. Then, the set of feasible velocities which are reachable but not in the invalid set is defined as $V_{feas} = V_{reach} \setminus V_{inv}$. The selected velocity should be feasible, $v \in V_{feas}$, and typically minimizes a metric relative to the preferred velocity v_a^{pref}, e.g., a velocity vector that points to the goal. More details will be provided regarding the specific velocity selection scheme used in this work, in section 3.5.

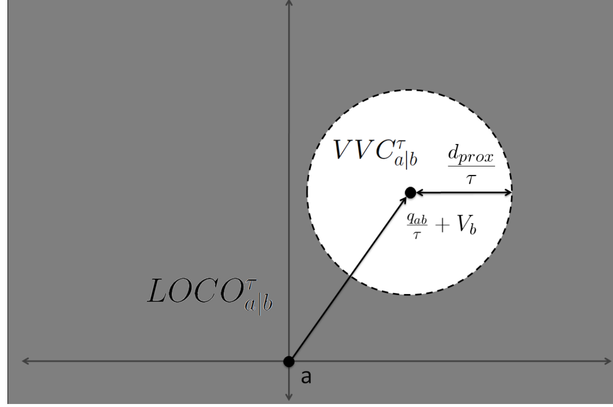

Figure 3.3: Construction of $\texttt{LOCO}^{\tau}_{a|b}$ and $\texttt{VVC}^{\tau}_{a|b}$. The circular region represents the viable velocities to maintain $d(a,b,t) \leq d_{prox}$ for time horizon τ. The shaded region outside the disk represents invalid velocities for agent a.

3.3 Loss of Communication Obstacles

The proposed approach extends the `VO` framework by creating new obstacles in the velocity space. These obstacles aim to prevent loss of communication, or more generally, to satisfy proximity constraints between agents in the form $d(a,b,t) \leq d_{prox}$ for agents a and b for at least a finite time horizon τ. The `LOCO` imposed by agent b on agent a for a horizon τ, denoted as $\texttt{LOCO}^{\tau}_{a|b}$, is the set

$$\texttt{LOCO}^{\tau}_{a|b} = \{v| \ \forall \ t \in [0,\tau] : d(a,b,t) \leq d_{prox}\},$$

under the assumption that agent b follows its current velocity v_b for at least time τ.

Assume agents a, $b \in A$ for which $(a,b) \in E$ of the input graph G. The relative position of agent b for agent a will be denoted as $q_{ab} = q_b - q_a$. If the relative position at time t is $q_{(ab)}(t)$, then at time $t+\tau$ the relative position of the two agents is going to be:

$$q_{ab}(t+\tau) = q_{(ab)}(t) + \tau * (V_b - V_a).$$

For the two agents to be able to communicate at time $t+\tau$, it has to be that:

$$(q^X_{ab}(t+\tau))^2 + (q^Y_{ab}(t+\tau))^2 \leq d^2_{prox} \Rightarrow$$

$$(q^X_{ab}(t) + \tau * (V^X_b - V^X_a))^2 + (q^Y_{ab}(t) + \tau * (V^Y_b - V^Y_a))^2 \leq d^2_{prox} \Rightarrow$$

$$(\frac{q_{ab}^X(t)}{\tau} + V_b^X - V_a^X)^2 + (\frac{q_{ab}^Y(t)}{\tau} + V_b^Y - V_a^Y)^2 \leq \frac{d^{prox^2}}{\tau^2} \Rightarrow$$
$$(V_a^X - \frac{q_{ab}^X(t)}{\tau} - V_b^X)^2 + (V_a^Y - \frac{q_{ab}^Y(t)}{\tau} - V_b^Y)^2 \leq (\frac{d_{prox}}{\tau})^2$$

The last expression implies that the velocity V_a of agent a has to be within a circle with center $(\frac{q_{ab}}{\tau} + V_b)$ and radius $\frac{d_{prox}}{\tau}$. This circle will be referred to as the valid velocity circle ($\texttt{VVC}^{\tau}_{a|b}$) and is the complement of the $\texttt{LOCO}^{\tau}_{a|b}$. Figure 3.3 gives an example of a VVC circle.

An agent a, however, may have multiple neighbors leading to the definition of multiple LOCOs and VVCs. A choice that is made for simplicity is to consider the same time horizon τ for the definition of all the LOCOs for all the neighbors. Then, the set of velocities v_a that will not allow a to maintain connectivity with at least one neighbor b in the graph G is the union of individual LOCOs:

$$\texttt{LOCO}^{\tau}_{a} = \bigcup_{\forall\ b\ |\ \exists (a,b) \in E} \texttt{LOCO}^{\tau}_{a|b}$$

There are two complications arising from this definition. Firstly, it is not straightforward to compute the longest horizon for which this union is not the entire plane, i.e., the longest horizon for which the intersection of VVCs is not empty. The problem is that both the centers and the radii of the VVCs are changing for different time horizons. Secondly, the resulting region is rather complex to describe, as it corresponds to multiple circle intersections. This representation can impose significant computational overhead when the LOCOs are integrated with VOs.

Instead of computing the exact intersection of VVCs, this work proposes a conservative approximation that is easier to represent and beneficial for computational purposes. The approximation of the valid set of velocities corresponds to a circle inside the intersection of VVCs. Figure 3.4 illustrates the procedure for two and three neighbors.

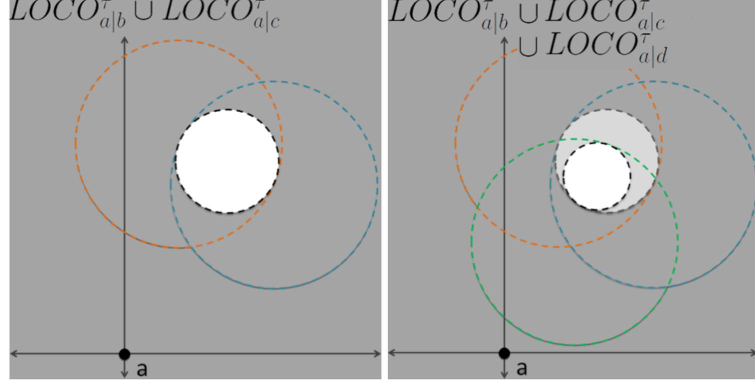

Figure 3.4: The conservative approximation of $\texttt{VVC}^{\tau}_{a}$ in the velocity space of agent a for two (left) and three (right) neighbors. The white circle corresponds to velocities that are guaranteed to maintain connectivity with the neighbors for time τ.

Given two circles with centers C_b and C_c and radii r_b and r_c, the inscribed circle of their intersection has the following radius and center:

$$r_{bc} = \frac{r_b + r_c - ||C_b, C_c||}{2}$$

$$C_{bc} = C_b + (r_b - r_{bc})\frac{C_c - C_b}{||C_b, C_c||}$$

The procedure works in an incremental manner. First it computes the inscribed circle (C_{bc}, r_{bc}) of the intersection of two VVCs, and then computes the inscribed circle of (C_{bc}, r_{bc}) with another VVC and so on.

3.4 Integration of VOs and LOCOs

The final step for computing the $\texttt{LOCO}^{\tau}_{a}$ is to select a suitable time horizon. A tuning approach over consecutive simulation steps is used. Given the horizon τ from the previous time step, the $\texttt{LOCO}^{\tau}_{a}$ is computed as in Figure 3.4. Then the set $V_{valid} = V_{reach} \setminus \texttt{LOCO}^{\tau}_{a}$ is computed, which considers the reachable controls that do not violate the LOCO constraints. This operation can be done in an efficient manner especially for systems with constant speed s, as it corresponds to a circle to circle intersection. In the general case for systems with varying velocity there are still computational advantages, as the circular representation of the VVC greatly increases the speed in which V_{valid} can be computed. Once V_{valid} is available for a given τ, the measure

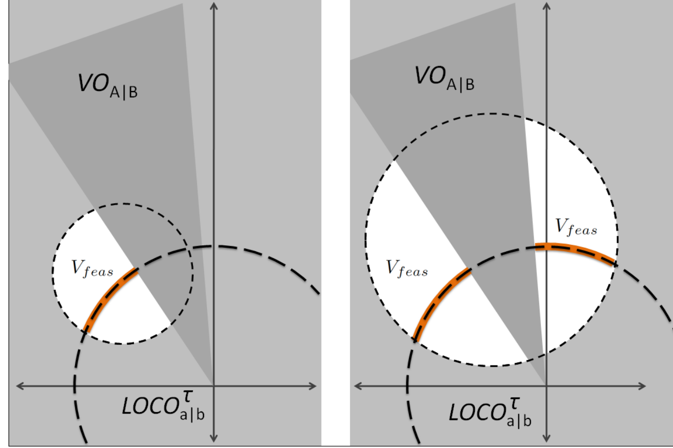

Figure 3.5: An example of how changing the horizon affects the set of feasible controls. The left image has a larger value for τ while the right has a smaller value of τ. Larger values of τ provide stronger guarantees for communication maintenance, but make finding a feasible control more difficult.

of $|V_{valid}|$ is compared against a predefined threshold $|V_{thresh}|$. If $|V_{valid}| < |V_{thresh}|$, then τ is decreased and V_{valid} is recomputed. A smaller τ implies that a bigger set of valid velocities for proximity maintenance will be returned that provides guarantees for a shorter horizon. Alternatively, if $|V_{valid}| > |V_{thresh}|$, then τ is increased. This will return a smaller set of valid velocities but will provide guarantees for a longer horizon. This tuning process continues until V_{valid} comes close to V_{thresh}, but this takes place over multiple simulation steps and adapts on the fly to changes in the relative configuration of agents. The effects of tuning τ can be seen in Figure 3.5.

Once $\texttt{LOCO}_a^\tau$ has been computed, it must then be included in the list of constraints in order to correctly compute V_{feas}, which is now redefined to be $V_{feas} = V_{valid} \setminus V_{inv}$, as in Figure 3.5. Sometimes, the additional constraints imposed by `LOCO`s, cause V_{feas} to become empty. In this situation, the `LOCO` constraints are ignored, as attempting to maintain communication may be perilous to the agent's safety.

3.5 Velocity Selection

Each agent has a preferred velocity it would like to follow, denoted as v_a^{pref} for agent a. Ignoring the proximity constraints and in obstacle-free environment, the preferred

velocity should be in the direction of the goal configuration $v_{goal} = q_a^{goal} - q_a$. If there are obstacles, however, setting the goal velocity in the same manner can lead the agents into local minima, causing the agent to become stuck behind the obstacle. This work avoids this issue by computing a discrete wave-front function in environments with obstacles. The goal velocity is computed as the direction to the cell with the minimum distance to the goal within a 3×3 region from the current cell of the agent.

In order to account for agents violating proximity constraints, a weighted velocity selection scheme is employed that takes neighbors into account. For some agent a, let d_i be the distance from agent a to another agent i and X_i be the state of agent i, where agent i is part of the proximity graph of agent a. Then, for n agents in the proximity graph of agent a, the average weighted configuration can be computed as:

$$q_{avg} = \frac{\sum_i^n \frac{d_i}{d_{prox}} q_i}{\sum_i^n \frac{d_i}{d_{prox}}}$$

Then, $d_{avg} = ||q_a - q_{avg}||$ is the distance between agent a and q_{avg}. Let $v_{avg} = q_{avg} - q_a$ be the vector pointing to q_{avg}, and let v_{goal} be the vector towards the goal computed through the wavefront. Then, agent a's preferred velocity is computed as follows:

$$v_a^{pref} = \frac{d_{avg}}{d_{prox}} v_{avg} + \frac{d_{prox} - d_{avg}}{d_{prox}} v_{goal}$$

Thus, agents which are farther away from their proximity constraints (i.e., have violated constraints) will be inclined to shorten this distance, whereas agents that have not violated any proximity constraints move towards their goals.

Once v_a^{pref} is computed, it can be checked for validity given the set V_{feas}. If v_a^{pref} is feasible, then agent a will use it. In the case that v_a^{pref} is not in V_{feas}, a different feasible control must be computed. In the general case, the region V_{feas} defines an area in velocity space which must be searched to find a control. The control found is the velocity $v \in V_{feas}^a$ which minimizes distance between v and v_a^{pref}.

The distance metric used in this approach is the Euclidean distance in the velocity space. Other metrics for finding the distance between velocities in this space are possible [52]. A list of intersections of the boundaries of the `RVO`s with V_{reach}^a is

generated and a rotational sweep algorithm determines which points are valid. These points define the boundaries of the V^a_{feas} regions. They are checked to find which one of them minimizes the distance to the preferred velocity : $min_{v \in V^a_{feas}}[d(v, v_a^{pref})]$. In the general case, V^a_{feas} will be a non-convex region or possibly disjoint non-convex regions in the velocity space where a variant of the simplex algorithm can be used to find the optimum.

In the event that there is no valid velocity available, the agent will select its current control. The reasoning behind this is that in the VO framework, after computing VOs for their neighbors, agents assume that these neighbors will not change their control. Thus, choices that keep the current velocity are preferable for this scheme.

Chapter 4 Creating an Extensible Architecture for Composing Controllers and Planners

4.1 Architecture

The proposed architecture is composed of several modules, following the architecture of the Robotic Operating System (`ROS`) [55]. `ROS`'s architecture has separate nodes launched as executables which communicate via message passing and are organized into packages, stacks, and nodes. A package is a collection of files, while a stack is a collection of such packages. Nodes are executables. `PRACSYS` is a stack and the nodes launched from `PRACSYS` are associated with a single package. `PRACSYS` also allows developers to integrate additional plugins into the architecture. There are three packages which run as nodes: the *simulation*, *planning*, and *visualization* packages. See Figure 4.1 for a visual representation of the interactions between different packages of `PRACSYS`. The advantage of having separate nodes is that it makes the jump to distributed computation such as on a computing grid easier.

The *common* package contains useful data structures, as well as mathematical tools. The *utilities* package contains useful algorithms, such as graph search, as well as abstractions for planning. Both the *common* and *utilities* packages use the Boost library to facilitate efficient implementations.

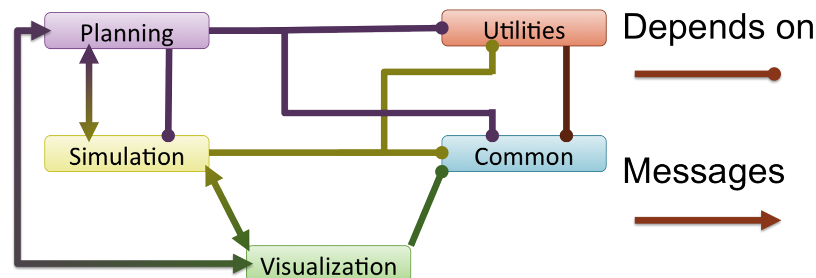

Figure 4.1: Package interactions. `ROS` nodes communicate via message passing: *simulation*, *visualization*, and *planning*. The *common* and *utilities* packages are dependencies of the previous three.

The higher-level packages include *simulation* , *visualization* , and *planning*. The *simulation* package has *common* and *utilities* as dependencies. The *simulation* package is responsible for simulating the physical world in which the agents reside and contains integrators and collision checking. The *simulation* package also contains many controllers which operate over short time horizons. Controllers are part of the main pipeline and are not in the *planning* package because they only operate over a single simulation step. The *planning* package is primarily concerned with controlling agents over a longer horizon, using the *simulation* package internally. The *visualization* package gives meaningful I/O between the user and the *simulation*, such as selecting agents and providing manual control. The *planning* package can have different state than the ground truth simulator, which is useful for applications such as planning under uncertainty.

`PRACSYS` also has an optional input package with files for loading simulations in YAML [56] format. There is also a package containing external package dependencies used by `PRACSYS`, such as the Approximate Nearest Neighbors library [1].

The following discussion details the capabilities of these packages, starting with the most fundamental of the three: the *simulation* package.

4.2 Descriptions

This section discusses the different packages of `PRACSYS` in further detail.

4.2.1 Ground-truth Simulation and Controller Architecture

The *simulation* node is the primary location for the development and testing of new controllers, and contains a set of features which are useful for developers. The following sections will go over each of these features individually.

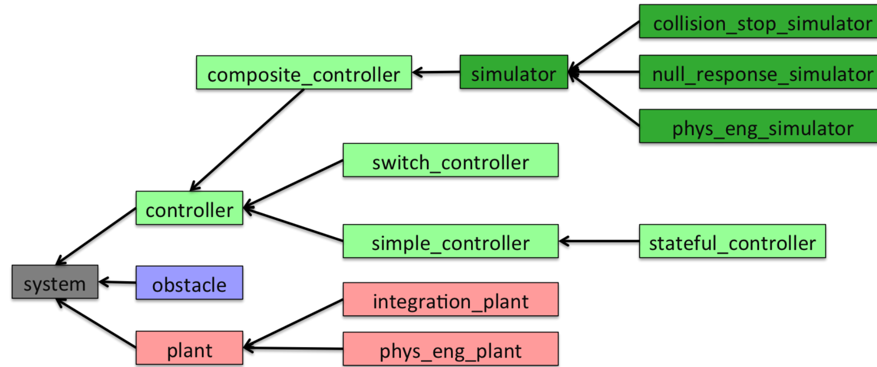

Figure 4.2: A view of the class inheritance tree for the `PRACSYS` *system*.

Composability

The simulator contains several classes which are interfaces used for the development of controllers. The fundamental abstraction is the *system* class - all controllers and plants are *systems* in `PRACSYS`, as shown in Figure 4.2. This functionality allows for other nodes, such as planning, to reason over one or more *systems* without knowing the specifics of the system. The interface is the same whether planning happens over a physical plant or a controlled system, which simplifies feedback-based planning.

The interaction between systems is governed by the pipeline shown in Figure 4.3, which ensures that every system properly updates its state and control. The functions in the pipeline are responsible for the following:

1. **copy state**: receive a state from a higher-level system, potentially manipulate this state, and pass it down to lower-level systems.

2. **copy control**: receive a control from a higher-level system, potentially manipulate this control, and pass it down to lower-level systems.

3. **propagate**: propagates a system according to its dynamics (if it is a plant), or sends a propagate signal down to lower-level systems (if it is a controller).

4. **get state**: receives the state from a lower-level system. This allows higher-level systems to query for the full state of the simulation

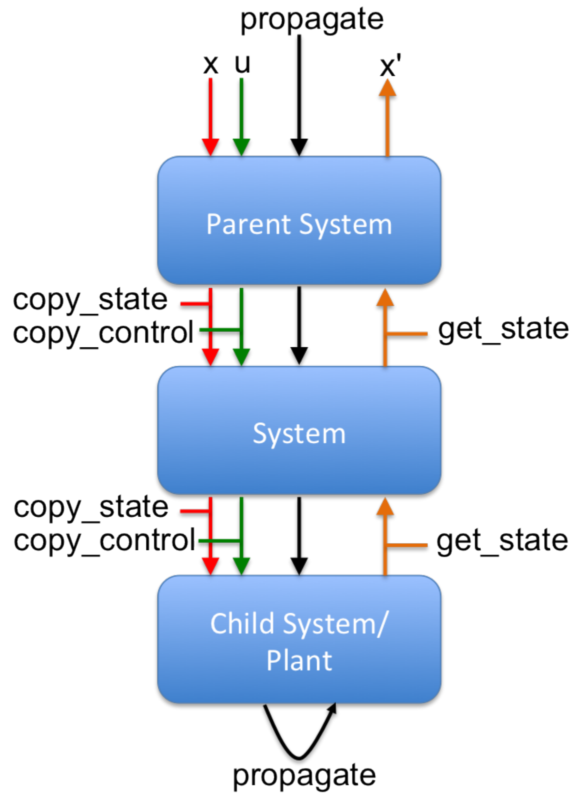

Figure 4.3: The core interface of a system.

The *system* class uses the *space* abstraction, which provides a way for users to define abstract spaces in a general way, allowing for scopes beyond a Euclidean space. A space is a rectangular-bounded region of n dimensions where each dimension can be Euclidean, rotational, or one component of a quaternion. The abstraction automatically provides a way to compute metrics between different points within the space. A *space point* stores the parameters of each dimension of the *space*, and can be used to represent states and controls in the *system* class.

Note that a *system* does not need to get controls from lower-level *systems*. The *system* class contains other interface functions, which primarily fall under the category of initialization, set functions, and get functions. Systems are broken into two major categories: physical plants and controllers. Physical plants are responsible for simulating the physical agents and how they move through the environment. They store geometries and have functionality to update their configurations based on states

set through copy state. Furthermore, physical plants are governed by state-update equations of the form $\dot{x} = f(x, u)$ implemented by the propagate function.

Controllers are classified into four major types: simple, stateful, composite, and switch. These are classes that extend the abstract controller class. A **simple controller** contains a single subsystem, which is most often the plant itself, but can also be another controller. Simple controllers are useful for creating a chain of controllers, allowing for straightforward compositions. A controller which computes a motion vector along a potential field for a holonomic disk system is an example of a simple controller on top of a physical plant. In more complex compositions, controllers will also have the need to keep an internal state, separate from its subsystem. A **stateful controller** allows for an internal state, and one such example of a *stateful controller* is the consumer controller. The *consumer controller* simply supplies controls to its subsystem from a pre-computed plan, where its state is the point in time along the plan to extract the control. **Composite controllers**, which can have many subsystems, provide the necessary interface for controlling multiple subsystems. The simulator, for example, which is responsible for propagating systems, is a composite controller containing all controllers and plants. The final major type of controller is the **switch controller**, which behaves quite similar to a C/C++ switch statement. A switch controller operates over an internal controller state, called a mode, which determines which of its subsystems is active. For example, a switch controller could be used to change the dynamics being applied to a plant depending if it was in a normal environment or an icy environment, in which case an inactive slip-controller would be "switched" active. Developing controllers which utilize these archetypes allows for an easy way to create more complex interactions.

High-level *simulation* Abstractions

In addition to the *system* abstraction, there are several high-level abstractions which give users additional control over the function of the simulation. One such abstraction is the **collision checking** abstraction, which consists of an actual collision checker

and a collision list. The collision list simply describes which pairs of geometries should be interacting in the simulation, while the collision checker is actually responsible for performing the checks and reporting when geometries have come into contact. The **simulator** is an extension of the composite controller, but it has additional functionality and has the unique property of always being the highest-level *system* in the simulation. The abstraction which users are most likely to change is the **application**. The application class contains the simulator and is responsible for defining the type of problem that the user wants to solve.

Interaction

The *simulation* node can communicate with other nodes via `ROS` messages and service calls. For example, moving a robot on the visualization side involves a `ROS` service call. Similar to how a propagate signal is sent between systems, an update signal is used to change geometry configurations. Once all systems have appended to this update signal, a `ROS` message is constructed from it and sent to *visualization* in order to actually move the physical geometry. For non-physical geometries, such as additional information of a system (i.e., a robot could have a visualized vector indicating its direction), each system is responsible for making the appropriate `ROS` call. If a user needs additional functionality and interaction between the nodes, they only need to implement their function in the communication class and create the appropriate `ROS` files.

Plugin Support

Adding a new physics engine as a plugin simply involves extending the simulator, plant, obstacle, and collision checker classes. If a user does not require the use of physics simulation, they can simply disable this by omitting it from the input file. For collision checking, `PRACSYS` currently provides support for PQP, as well as the use of no-collision checking. Similarly, if a user would like to add a new collision checker, they only need to extend the collision checking class.

4.2.2 Planning

The *planning* package is responsible for determining sequences of controls for one or many agents over a longer horizon than a single simulation step. This excludes methods which use reactive control. Furthermore, *planning* also reasons over higher-level planning processes, such as task coordination. The *planning* package is divided among several modules in order to accomplish these tasks. The high-level task coordination is provided by *task planners*, which contain *motion planners* for generating sequences of controls given a *world model*. The *world model* is a system with a simulator as a subsystem and is responsible for providing information to the planners about the state of the simulator as well as providing additional functionality. The *motion planners* are the individual motion planning algorithms which compute controls. The current version of `PRACSYS` has certain sampling-based motion planners implemented, which use some basic modules to accomplish their specified tasks, including local planners and validity checkers. `PRACSYS` also integrates existing motion planning packages such as `OMPL` by providing an appropriate interface. The current focus of the *planning* package has been sampling-based methods; however, it is not limited to these types of planners and can easily support search-based or combinatorial approaches.

High-Level Abstractions

All of the abstractions described in this section interact according to Figure 4.4.

Task planners are responsible for coordinating the high-level planning efforts of the node. *Task planners* contain at least one instance of a motion planner and use planners to accomplish a given task. Ultimately, the goal of *planning* is to come up with valid trajectories for one or many systems, which bring them from some initial state to a goal state, but the *task planner* may be attempting to accomplish a higher-level task such as motion coordination. In this sense, the task planners are responsible for defining the objective of the planning process, while the motion planners actually generate the plan. One example is the single-shot task planner, which allows a planner to plan until it reports it has found a specified goal. Then

the single-shot task planner forwards the plan to the *simulation* node. Other tasks include single-shot planning, replanning, multi-target planning and velocity tuning or trajectory coordination among multiple agents.

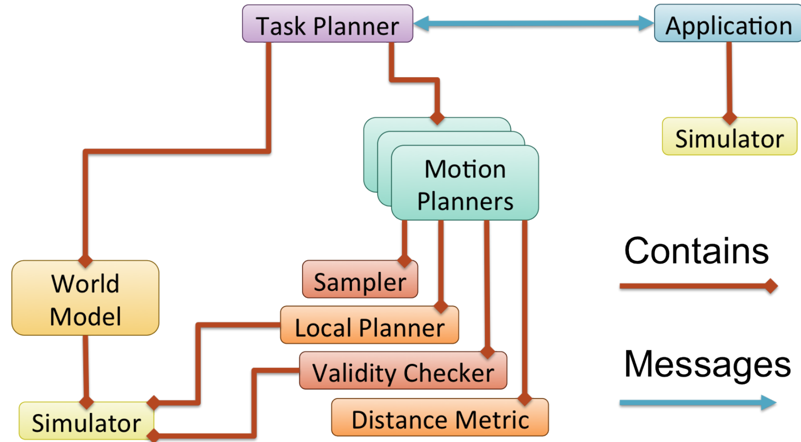

Figure 4.4: The general structure of the *planning* modules. The *task planner* contains multiple *motion planners*. The *task planner* also contains a *world model* and communicates with the *simulation* node.

A **World Model** represents the physical world as observed or known to the planner, and also extends the *system* abstraction. A *world model* contains a simulator as a subsystem, exposing the functionality of the simulator to the *motion planners*. World models can be used to hide dimensions of the state space from the motion planners, introduce and model the uncertainty an agent has about its environment, or removing certain agents from collision checking. Having the capability to remove dimensions from the state space is useful for planning purposes because the planning process has complexity which depends on the dimensionality of the space. This reduction will make the planning process more efficient, and is related to being able to remove some systems from collision checking. These two functions together allow a full simulation to be loaded, while allowing planning to plan for agents individually in a decoupled manner for greater efficiency.

Motion planners are responsible for coming up with trajectories for individual agents or for groups of agents. The flexibility of `PRACSYS` allows for planners to easily be changed from performing fully decoupled planning to any range of coupling,

including fully coupled problems. *Motion planners* employ the use of modules, used as a black box by the *motion planner*, but may have a wide variety of underlying implementations available to accomplish the task. Furthermore, because of the flexibility of the *system* class, planners can plan over controllers as well. This allows for interesting new planning opportunities, and one example of this use is given in section 5. In this case, planners are able to essentially create trajectories through parameter space of controllers. The sampling-based planners in PRACSYS make use of four modules: local planners, samplers, distance metrics, and validity checkers. The distance metric module and the sampler module are provided by the *utilities* package.

Sampling-Based Motion Modules

Local planners propagate the agents according to their dynamics. PRACSYS offers two basic types of local planners, an approach local planner which uses a basic approach technique to extend trajectories toward desired states, and a kinematic local planner which connects two states exactly, but only works for agents which have a two-point boundary problem solver and kinematic agents.

Validity checkers provide a way to determine if a given state is valid. The most basic implementation of a validity checker, which is provided with PRACSYS simply takes a state, translates it into its geometrical configuration, and checks if there is a collision between the geometry of the agents and the environment.

Samplers are able to generate samples within the bounds of an abstract *space*. Different samplers will allow for different methods of sampling, such as uniform randomly, or on a grid.

Distance metrics are responsible for determining the distance of points in a *space*. These modules may use simple interpolating methods or may be extended to be more complex and take into account invalid areas of the space.

Interaction

The *planning* package communicates primarily with *simulation*. A *planning* node can send messages to the *simulation* such as computed plans for the agents. The *planning* package can further send trajectory and planning structure information to visualization so users can see the results of an algorithm. The *planning* node also receives control signals from the *simulation* node, such as when to start planning. `OMPL` is a software package developed for planning purposes [19]. Because of the plugin system of `PRACSYS`, a simple wrapper is provided around the existing `OMPL` implementation in order to utilize the `OMPL` planners within `PRACSYS`.

4.2.3 Visualization

The *visualization* node is responsible for visualizing any aspect needed by the other nodes. Users interact with the simulation environment through the *visualization*. This includes, but not limited to, camera interaction, screen shots and videos, and robot tracking. The *visualization* provides an interface to develop alternative implementations, in case users do not want to use the provided implementation based on OpenSceneGraph (OSG) [32].

4.2.4 Other `PRACSYS` Packages

The remaining packages provide functionality useful across the infrastructure, such as geometric calculations, configuration information, and interpolation. An important concept is the idea of a *space* as provided by the *utilities* package, which was described earlier in Section 4.1. The *input* package is an optional package which includes sample input for use with `PRACSYS`. Files are in YAML or `ROS` .launch format. `PRACSYS` also comes with an *external* package for carrying along external software packages, such as the Approximate Nearest Neighbors (`ANN`) package [1], which is useful for motion planning.

Chapter 5 Evaluation

This chapter contains the experimental results obtained from evaluating the LOCO controller in a variety of scenarios against the most relevant competing algorithm, Reciprocal Velocity Obstacles [51]. Additionally, several use cases are presented which highlight the usability and robustness of PRACSYS.

5.1 LOCO Results

The approach was implemented using a simulation software platform. Experiments were run on computers with a 3.06 GHz Intel Core 2 Duo processor and 4GB of RAM. The experiments are organized in the following manner. First, experiments were conducted using a single team of agents moving in an environment with obstacles. Then, experiments with multiple teams were run both in an obstacle-free world and in an environment with obstacles. A team of agents corresponds to a connected component of the input graph G. The experiments conducted compare the LOCO formulation against RVOs. The reason for comparing against RVOs, rather than other formation techniques, is due to the decentralized nature of the LOCO algorithm, which requires no additional information outside the VO framework. In addition, LOCOs utilize the notion of coherence, which is more abstract in nature than formations. Thus, RVOs are the most relevant technique to experiment against. Each experiment was measured with regards to the following metrics:

- total number of collisions during the entire experiment,
- computation time per frame,
- total time to solve the problem,
- average ratio of respected proximity links per frame,
- and number of successful runs.

For each variation of the environment, input graph G and algorithm, there were 10 runs executed. Each run had a random initial q_{init} and goal configuration q_{goal}, which,

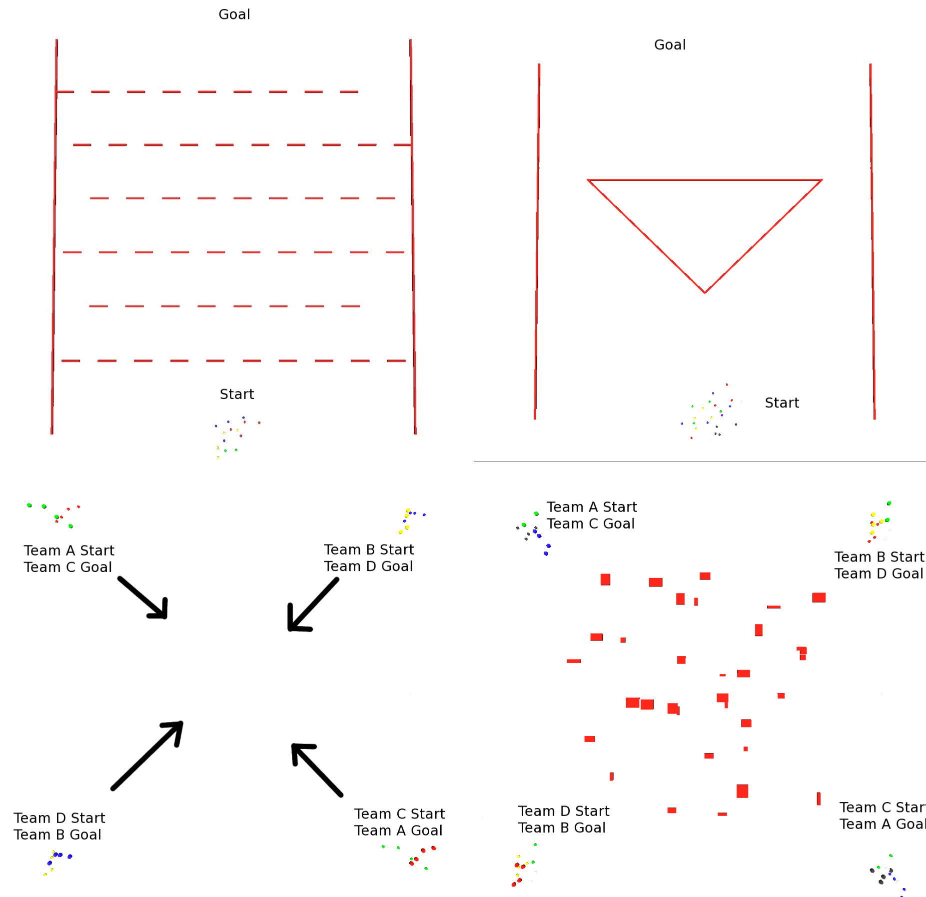

Figure 5.1: The environments on which the experiments were executed

satisfied the proximity link in the graph G.

5.1.1 Single Team

For the single team scenarios, two different environments were tested with varying numbers of agents. The `PACHINKO` environment uses a series of walls with small gaps (Figure 5.1 top left), and a team of 10 agents was considered in this case. The proximity graph imposed on the team is shown in Figure 5.2 top left. The `WEDGE` environment (Figure 5.1 top right) attempts to split the agents into two lanes. The proximity graph imposed on the team is shown in Figure 5.2 top right.

Table 5.1 shows that there is a significant improvement in terms of the percentage of links maintained by the `LOCO`-based approach relative to `RVO`s. Another interesting

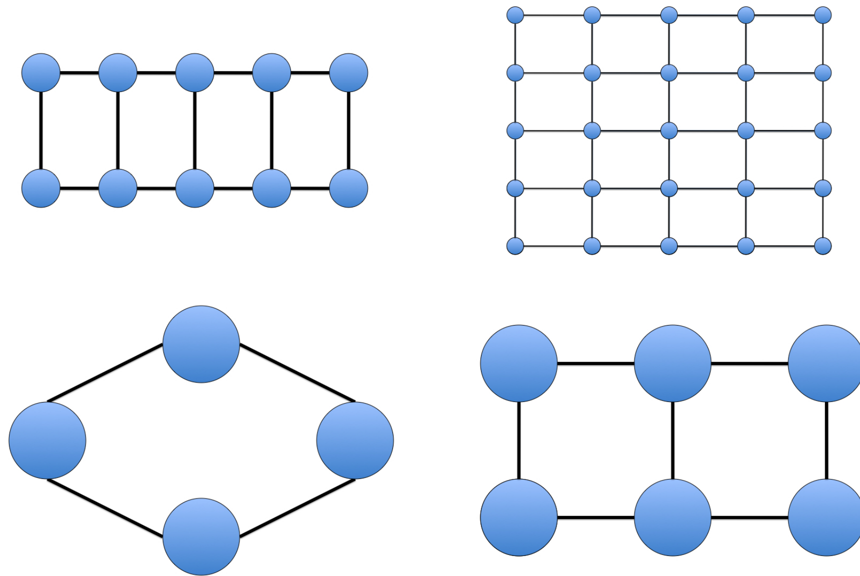

Figure 5.2: Examples of input proximity graphs used in the experiments.

statistic to examine is the coherence of agents over time, shown in Figure 5.3, which the LOCO-based approach also improves. This comes at the cost of a slightly increased computational cost and increased time taken by the algorithm to bring all of the agents to their goals. An example of the paths taken by 24 agents in the PACHINKO environment is shown in Figure 5.4. Both approaches were able to solve all of the problems and without any collision, with the exception of one experiment for the WEDGE environment, where a single collision was reported.

	time per step (s)		collisions		maintained links		steps		success	
	LOCO	RVO	LOCO	RVO	LOCO	RVO	LOCO	RVO	LOCO	RVO
wedge grid	0.03964	0.03488	0.1	0	92%	65%	4279.1	3376.9	100%	100%
pachinko train	0.04474	0.0355	0	0	96%	46%	3881.2	3555	100%	100%

Table 5.1: Results for a single team.

5.1.2 Multiple Teams

The performance of LOCOs was evaluated against RVOs in a scenario involving four teams of agents navigating in an obstacle-free environment as shown in Figure 5.1 bottom left (CROSSROADS) and for the connectivity graph for each team shown in

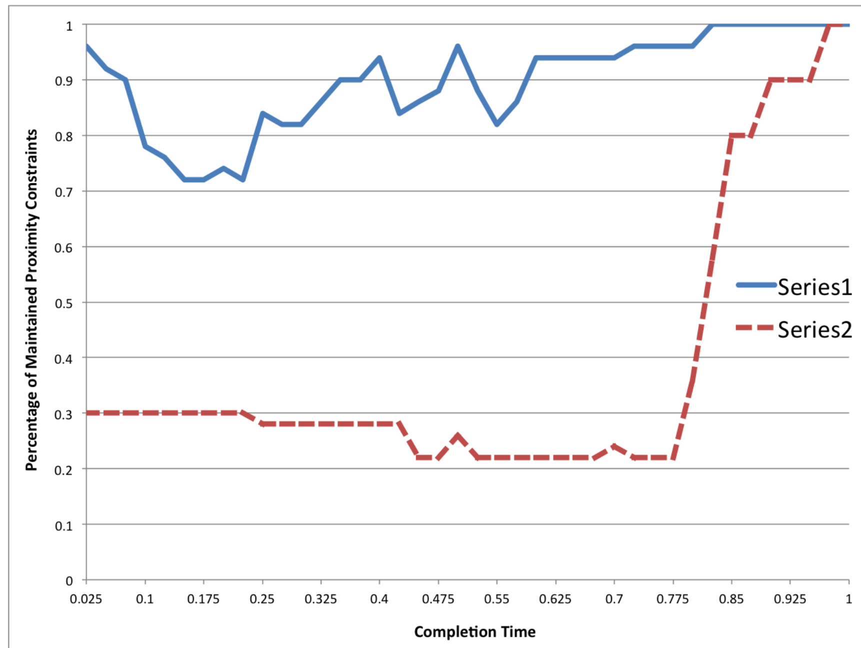

Figure 5.3: Coherence over time for 24 agents in the wedge grid experiment.

Figure 5.2 bottom left. There were four agents in each team. The last setup involved again four teams of agents, each one of which had six agents connected as in Figure 5.2 bottom right. This time the environment involved a random set of rectangular obstacles as in Figure 5.1 bottom right. For each of the 10 runs the rectangles were randomly placed. The results are shown in Table 5.2.

In both the random and crossroads examples, `LOCOs` outperform `RVOs` as far as connections are concerned, though oftentimes `LOCOs` take more time to solve the specified problem. On different problems, both approaches can take some extra time to complete the problem for two different reasons. The `VOs` take extra time as agents will sometimes get caught on obstacles or are pushed away from their goals by agents on other teams. `LOCOs` may take extra time as they spend effort navigating agents in densely-packed situations that occur at the center of the environment as they cross over. Furthermore, `LOCOs` exhibit a flocking behavior which sometimes causes agents to overshoot their goals.

In the random obstacle environment, `LOCOs` imposed a very small computational overhead. The crossroads scenario resulted in a more significant increase, because all

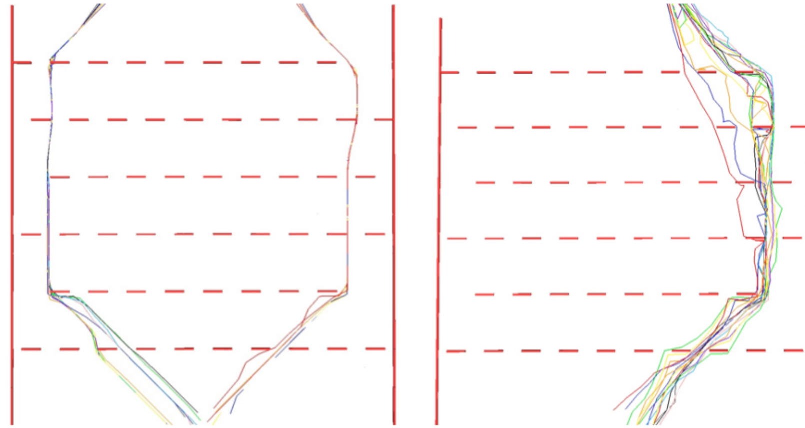

Figure 5.4: An example of the paths taken by 24 agents in the PACHINKO environment for the RVO (left) and LOCO (right) approach.

	time per step (s)		collisions		maintained links		steps		success	
	LOCO	RVO	LOCO	RVO	LOCO	RVO	LOCO	RVO	LOCO	RVO
random	0.03005	0.02919	0	0.1	96%	72%	2840.78	2416.3	90%	100%
crossroads	0.00564	0.0034	0	0	98%	71%	2871.1	2596	100%	100%

Table 5.2: Results for multiple teams.

agents meet in the center of the environment nearly at the same point in time. This increases computational cost, as LOCOs must tune their horizon and reconnect broken proximities more frequently. In both environments, LOCOs and RVOs resulted in no collisions, with the exception of one run of RVOs in the random environment. The main focus of these results, however, is the improvement in maintained links. The LOCO-based approach maintained 95% of the proximity links, which was an improvement over RVOs. LOCOs cause a small degradation in path quality, which is measured in the number of steps it took for agents to solve the environment. In addition to this, LOCO failed to solve one of the randomly generated environments, because agents tend to spread to the edge of their proximity constraints, which may cause problems when agents move around obstacles. Overall, the results seem to validate the initial approach, as the goal of LOCOs is to maintain safety while providing better proximity maintenance is experimentally supported.

5.2 PRACSYS Use Cases

This section provides specific examples of the features offered by PRACSYS.

5.2.1 Showing Scalability for Multiple Agents

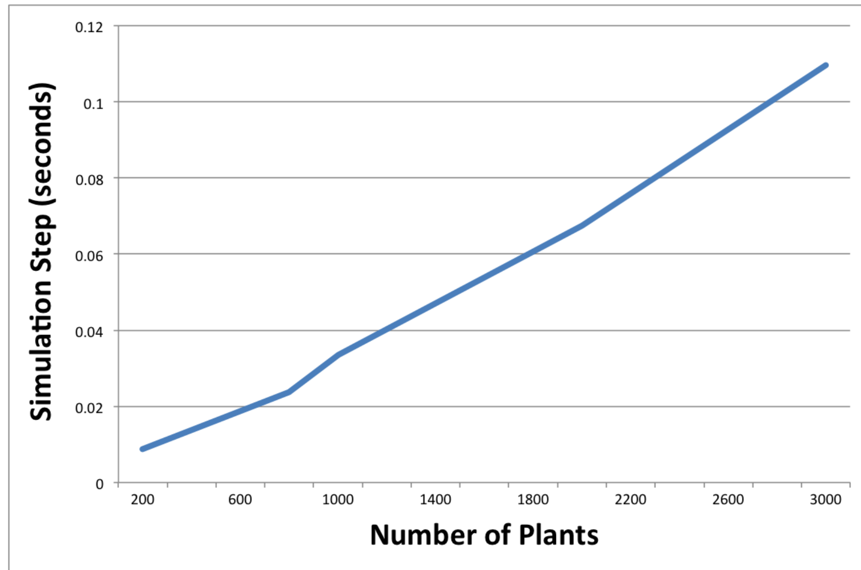

Figure 5.5: A plot of simulation steps vs number of plants.

Scalability is important for simulation environments in which multiple agents are to be simulated at once. The PRACSYS system structure was designed with multi-agent simulation in mind. Given a very simple controller, multiple physical plants were simulated and the time for a simulation step was tracked against number of agents, as shown in Figure 5.5. The trend shows a linear increase in simulation step duration with the number of agents, even with simulations of thousands of agents. Figure 5.6 shows 3000 agents being simulated in PRACSYS.

The Velocity Obstacle (VO) Framework was introduced as a lightweight reactive obstacle avoidance technique [14]. The basic framework has been expanded upon, with such extensions as Reciprocal Velocity Obstacles (RVO) [51]. The basic VO framework as well as several extensions have been implemented in PRACSYS and have also been used in large-scale experiments.

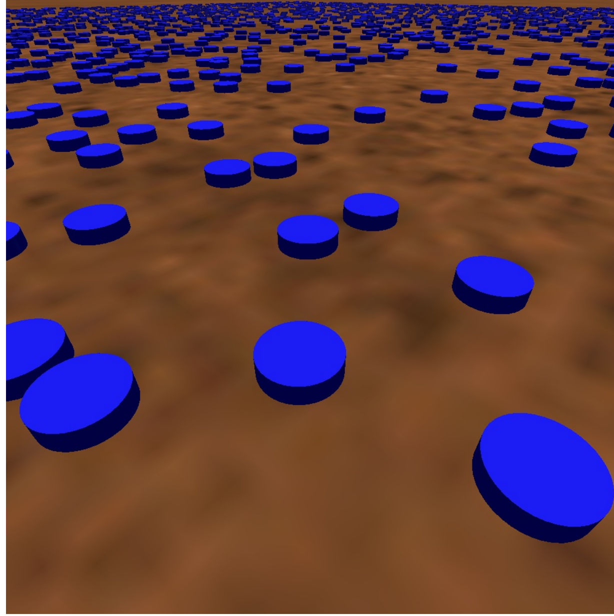

Figure 5.6: 3000 plants running in an environment.

5.2.2 Planning over Controllers using LQR Trees

Using a C/C++ interface to Octave and its control package [11], PRACSYS can utilize the optimal control guarantees of linear quadratic regulators (LQR). Octave is an open-source Matlab clone which PRACSYS can invoke Octave, meaning it can run software developed in Matlab with little or no conversion time effort. An implementation of LQR-Trees has been developed in PRACSYS [38]. This algorithm is a prototypical example of "planning over controllers" so as to provide feedback-based solutions. The created LQR-tree is sent to the *simulation* node for execution. The process of incorporating the LQR code into PRACSYS took a matter of minutes.

The LQR-Tree is built incrementally similarly to RRT and its variants. It begins by computing an LQR that is based around the goal region, and then using sampling trajectories until new basins can be created using time-varying LQR over trajectories which enter existing basins. The technique has been shown to probabilistically cover the space, and stores a full description of the LQR controller used to create the basin of attraction at each tree node.

The *planning* node can send the controller information to the *simulation* node. This implementation illustrates the use of `LQR` controllers inside a planning structure. With this kind of framework, many more complex applications can be implemented and studied. This also shows the integration of a matrix-based interpreted language, `Octave`, into `PRACSYS`.

5.2.3 Controller Composition in Physics-based Simulation

`PRACSYS` offers a unique capability of composing systems. This scheme gives users flexibility by allowing the decomposition of individual steps into separate controllers, so that they can reused and re-combined to create new functionality.

For example, the framework given in the SIMBICON project [57], in which controllers are created for controlling bipedal robots has been implemented in `PRACSYS`. The hierarchy of controllers for this implementation is shown in Figure 5.7. A breakdown of this hierarchy is as follows:

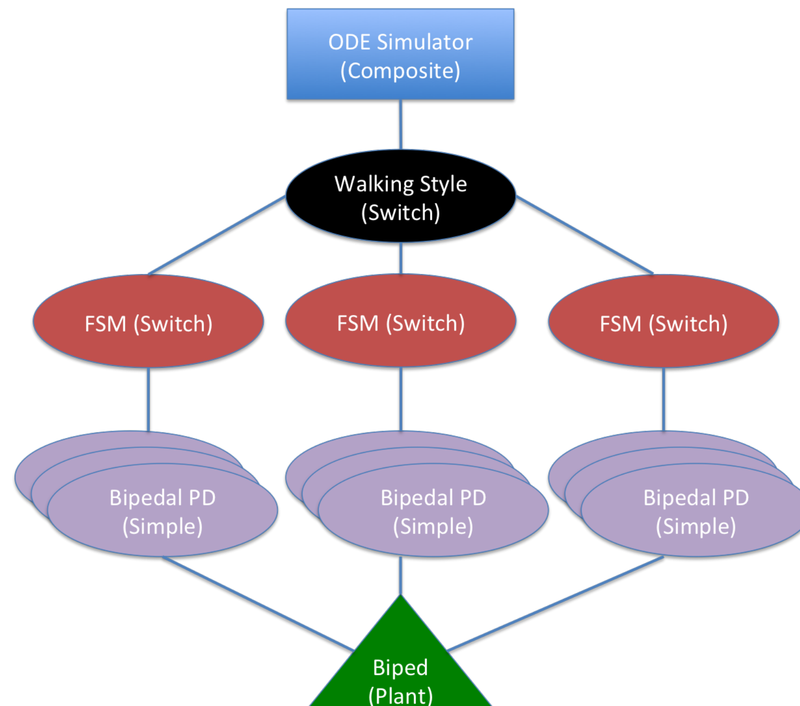

Figure 5.7: A visual representation of the controller composition for controlling a bipedal robot

ODE Simulator: The simulator, using the Open Dynamics Engine. **Finite State Machine (FSM):** Several FSMs, implemented with *switch controllers*, sit Below the simulator. Each FSM corresponds to a particular bipedal gait, such as running or skipping. Changes in the state of the simulation eventually causes a switch in the used gait. **Bipedal PD Controller:** Several PD controllers sit underneath each FSM, and represents a specific part of a gait, such as being mid-stride or having both feet planted. **Bipedal Plant:** This is the physical representation of the robot, and contains the geometry and joint information, as well as the actual dynamics of the plant.

Because ODE focuses on quick simulation for real-time applications, interactive applications can be created while sacrificing as little realism as possible. This lightweight implementation allows users to interact with this physically simulated world in interesting ways, such as manually controlling a plant among many other plants being controlled through various means. An example as shown in the submission's video shows a toy car controlled by a user interacting with the bipedal *system*, described previously.

5.2.4 Integration with `Octave`, `OMPL`, and MoCap Data

`PRACSYS` has been integrated with several other software packages in order to extend its functionality. `PRACSYS` has been integrated with `Octave`, `OMPL`, and motion capture data from CMU. **Motion Capture** data is used to animate characters in a realistic manner, and `PRACSYS` has integrated motion capture data from CMU. A controller which reads motion capture data has been utilized, and it reads and assigns the data to a control space point, where it is passed to the plant with copy control. The plant connected to this controller simulates a skeleton, which interpolates the configuration of each of its bones.

Chapter 6 Discussion and Future Work

The work presented in this thesis has a wide variety of possible extensions. This chapter presents a set of directions to consider for future work, in addition to discussing the two contributions.

6.1 LOCO

This work provides the formal definition for a new kind of obstacle in the velocity space of moving agents, referred to as a Loss of Communication Obstacle (LOCO), which correspond to proximity constraints with neighboring agents. These obstacles can be easily computed and integrated into the existing framework of Velocity Obstacles for decentralized collision avoidance. Additionally, an approach for tuning the time horizon parameter for these obstacles over multiple simulation steps is provided. These additional constraints increase the overall coherence of teams of agents while navigating through environments with static obstacles and other moving bodies. LOCOs can be dropped if it is determined that it is too difficult to maintain proximity without jeopardizing safety. The implementation of the technique shows improved coherence for agents who share communication links without sacrificing safety at a small computational overhead.

A natural extension is to consider more challenging systems, including kinematically and dynamically constrained systems. Adapting LOCOs to work in these cases is possible, as there have already been extensions on using VOs with dynamics. Since these extensions are in an orthogonal direction to the LOCO algorithm, it is possible to consider an integrated solution towards decentralized coherence maintenance for dynamic systems. Further extending the work to applications of real robots will require introducing a method for handling sensor errors, as well as creating a more robust reconnection strategy built on this error model. One drawback of reactive techniques is that they may get stuck in local minima. It is interesting to better

study the integration with the wavefront approach or another global planner so as to guarantee that agents will make progress towards their goal while guaranteeing safety and connectivity. A global planner can also improve the performance of the reconnection strategy. Currently, the direction to the agent with which connectivity has been lost ignores the existence of local obstacles. Another interesting direction is to study reciprocity tuning. More constrained agents, such as those who create a bridge for two otherwise disconnected agents, may be less able to reciprocate than others. Further reducing the computational overhead of the technique would also have great benefits, since larger-scale tests could be performed and have practical application areas, as in crowd simulation and video games.

6.2 PRACSYS

PRACSYS is an extensible environment for developing and composing controllers and planners with a broad range of applications. It can support multi-agent simulations, physics-based tools, and can incorporate Matlab code, the OMPL library [19] and MoCap data. There are multiple important future steps for PRACSYS. One direction is to utilize principles of software engineering to improve the generality of the *system* class. For example, the *system* class could be modeled using the blackboard architectural model, which is designed to handle complex, ill-defined problems. Another alternative is the use of design patterns, such as the composite and module structures, which would further increase the extensibility of the *system* class. One of the current pursuits is the development of a communication node, which simulates communication protocol parameters and failures between agents by employing a discrete event network simulator, such as ns3 [31]. This will allow the simulation of distributed planning involving communication on a computing cluster. Furthermore, a sensing node is developed for simulating sensor data in place of a physical sensor. This objective, as well as allowing algorithms coded on PRACSYS to run on physical systems, will be assisted by a tighter integration with the latest versions of Gazebo [20], OpenRave [10] and by utilizing existing ROS functionality [55].

Bibliography

[1] S. Arya and D. M. Mount. Approximate nearest neighbor searching. In *Proc. 4th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 271–280, 1993.

[2] T. Balch and M. Hybinette. Social potentials for scalable multi-robot formations. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 73–80, 2000.

[3] T. D. Barfoot and C. M. Clark. Motion planning for formations of mobile robots. *Journal of Robotics and Autonomous Systems*, 46(2), 2004.

[4] K. E. Bekris, K. I. Tsianos, and L. E. Kavraki. Safe and distributed kinodynamic replanning for vehicular networks. *Mobile Networks and Applications*, 14(3), 2009.

[5] M. Bennewitz, W. Burgard, and S. Thrun. Finding and Optimizing Solvable Priority Schemes for Decoupled Path Planning Techniques for Teams of Mobile Robots. *Robotics and Autonomous Systems*, 41(2):89–99, 2002.

[6] Carmen Robot Navigation Toolkit. http://carmen.sourceforge.net/home.html.

[7] S. Carpin, M. Lewis, J. Wang, S. Balakirsky, and C. Scrapper. USARSim: A Robot Simulator for Research and Education. In *IEEE ICRA*, pages 1400–1405, 2007.

[8] C. M. Clark, S. M. Rock, and J.-C. Latombe. Motion Planning for Multiple Robots using Dynamic Networks. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 4222–4227, 2003.

[9] Delta3D. http://www.delta3d.org/, 2006.

[10] R. Diankov and J. J. Kuffner. Openrave: A planning architecture for autonomous robotics. Technical report, CMU-RI-TR-08-34, The Robotics Institute, Carnegie Mellon University, 2008.

[11] J. W. Eaton. *GNU Octave Manual.* Network Theory Limited, 2002.

[12] M. Erdmann and T. Lozano-Perez. On multiple moving objects. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 1419–1424, 1986.

[13] R. Fierro, C. Belta, J.P. Desai, and V. Kumar. On controlling aircraft formations. In *Proceedings of the 40th IEEE Conference on Decision and Control, 2001.*, volume 2, pages 1065 –1070 vol.2, 2001.

[14] P. Fiorini and Z. Shiller. Motion Planning in Dynamic Environments Using Velocity Obstacles. *International Journal of Robotics Research (IJRR)*, 17(7):760–772, 1998.

[15] D. Fox, W. Burgard, and S. Thrun. The dynamic window approach to collision avoidance. *IEEE Robotics and Automation Magazine*, 4(1), 1997.

[16] B. Gerkey, R. T. Vaughan, and A. Howard. The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor and Systems. In *In Proceedings of the 11th International Conference on Advanced Robotics (ICAR)*, pages 317–323, 2003.

[17] S. Gottschalk, M. C. Lin, and D. Manocha. OBBTree: A Hierarchical Structure for Rapid Interference Detection. In *SIGGRAPH*, pages 171–180, http://gamma.cs.unc.edu/SSV/, Aug 1996.

[18] D. Helbing, L. Buzna, A. Johansson, and T. Werner. Self-organized pedestrian crowd dynamics: Experiments, simulations and design solutions. *Transportation Science*, 2005.

[19] Kavraki Lab Group: The Open Motion Planning Library (OMPL). http://ompl.kavrakilab.org.

[20] Koenig, N. and Hsu, J. and Dolha, M. - Willow Garage, Gazebo. http://gazebosim.org/.

[21] Athanasios Krontiris and Kostas E. Bekris. Using minimal communication to improve decentralized conflict resolution for non-holonomic vehicles. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, September 2011.

[22] F. Large, C. Laugier, and Z. Shiller. Navigation among moving obstacles using the NLVO: Principles and applications to intelligent vehicles. *Autonomous Robots*, 19(2), 2005.

[23] J.-C. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, Boston, MA, 1991.

[24] S. LaValle. *Planning Algorithms*. Cambridge, 2006.

[25] M. Anthony Lewis and Kar-Han Tan. High precision formation control of mobile robots using virtual structures. *Auton. Robots*, 4:387–403, October 1997.

[26] W. Li and C. Cassandras. A cooperative receding horizon controller for multi-vehicle uncertain environments. *IEEE Transactions on Automatic Control*, 51(2):242–257, 2006.

[27] J. M. Lien, S. Rodriguez, J. P. Malric, and N. Amato. Shepherding behaviors with multiple shepherds. In *Intern. Conf. on Robotic and Automation*, 2005.

[28] O. Michel. Webots: Professional Mobile Robot Simulation. *International Journal of Advanced Robotic Systems (IJARS)*, 1(1):39–42, 2004.

[29] Microsoft Robotics Developer Studio. http://www.microsoft.com/robotics/.

[30] A. Miller. *Graspit!: A Versatile Simulator for Robotic Grasping*. PhD thesis, Columbia University, http://www.cs.columbia.edu/ cmatei/graspit/, 2001.

[31] NS3. http://www.nsnam.org/.

[32] OpenSceneGraph. http://www.openscenegraph.org/.

[33] L. Pallotino, V. G. Scordio, E. Frazzoli, and A. Bicchi. Decentralized cooperative policy for conflict resolution in multi-vehicle systems. *IEEE Transactions on Robotics*, 23(6):1170–1183, 2007.

[34] S. Paris, J. Pettre, and S. Donikian. Pedestrian reactive navigation for crowd simulation: A predictive approach. *Computer Graphics Forum*, September 2007.

[35] N. Pelechano, J. Allbeck, and N. Badler. Controlling individual agents in high-density crowd simulation. In *ACM SIGGRAPH / Eurographics Symposium on Computer Animation (SCA)*, volume 3, pages 99–108, San Diego, CA, August 3-4 2007.

[36] N. Pelechano, J. Allbeck, and N. Badler. *Virtual Crowds: Methods, Simulation and Control*. Morgan and Claypool Publishers, 2008.

[37] J. Pettre, J. Ondrej, A.-H. Olivier, A. Cretual, and S. Donikian. Experiment-based modeling, simulation and validation of interactions between virtual walkers. In *Symposium on Computer Animation*. ACM, 2009.

[38] P. Reist and R. Tedrake. Simulation-based LQR-trees with input and state constraints. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 5504 –5510, 2010.

[39] C. W. Reynolds. Interaction with groups of autonomous characters. In *Proc. of the Game Developers Conference (GDC)*, pages 449–460, San Francisco, California.

[40] C. W. Reynolds. Steering behaviors for autonomous characters. In *Proc. of the Game Developers Conference (GDC)*, pages 763–782, San Jose, CA, 1999.

[41] S. LaValle, Motion Strategy Library. http://msl.cs.uiuc.edu/msl/.

[42] David Silver. Cooperative pathfinding. In *The 1st Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE'05)*, pages 23–28, 2005.

[43] J. Snape, S. J. Guy, J. van den Berg, S. Curtis, S. Patil, M. C. Lin, and D. Manocha. Independent navigation of multiple robots and virtual agents. In *Proc. of the 9th Int. Conf. on Autonomous Agents and Multiagents Systems (AAMAS 2010)*, Toronto, Canada, May 2010.

[44] Jamie Snape and Dinesh Manocha. Navigating multiple simple-airplanes in 3d workspace. In *Proc. of the IEEE Int. Conf. on Robotics and Automation (ICRA)*, May 2010.

[45] N. Sturtevant and M. Buro. Improving collaborative pathfinding using map abstraction. In *The Second Artificial Intelligence for Interactive Digital Entertainment Conference (AIIDE'06)*, pages 80–85, 2006.

[46] D. Thalmann. Populating virtual environments with crowds. In *Int. Conf. on Virtual Reality Continuum and Its Applications*. ACM, 2006.

[47] The Open Dynamics Engine (ODE), Russell Smith, 2007. http://ode-wiki.org/wiki/.

[48] C. J. Tomlin, I. Mitchell, and R. Ghosh. Safety Verification of Conflict Resolution Maneuvers. *IEEE Transactions on Intelligent Transportation Systems*, 2(2):110–120, June 2001.

[49] UrbiForge. http://www.urbiforge.org/.

[50] J. van den Berg, S. J. Guy, M. Lin, and D. Manocha. Reciprocal n-body collision avoidance. In *Proc. Int. Symposium of Robotics Research*. International Foundation on Robotics Research, aug. 2009.

[51] J. van den Berg, M. Lin, and D. Manocha. Reciprocal velocity obstacles for real-time multi-agent navigation. In *Proc. of the IEEE Int. Conf. on Robotics and Automation (ICRA)*, 2008.

[52] J. Van den Berg, J. Snape, S. Guy, and D. Manocha. Reciprocal Collision Avoidance with Acceleration-Velocity Obstacles. In *IEEE International Conf. on Robotics and Automation (ICRA)*, May 2011.

[53] C. Vo, J. F. Harrison, and J.-M. Lien. Behavior-based motion planning for group control. In *Intern. Conf. on Intelligent Robots and Systems*, St. Louis, MO, 2009.

[54] K.-H. C. Wang and A. Botea. Fast and memory-efficient multi-agent pathfinding. In *International Conference on Automated Planning and Scheduling (ICAPS)*, pages 380–387, Sydney, Australia, 2008.

[55] Willow Garage, Robot Operating System (ROS). http://www.ros.org/wiki/.

[56] YAML Ain't Markup Language (YAML). http://yaml.org/.

[57] K. Yin, K. Loken, and M. van den Panne. SIMBICON: Simple Biped Locomotion Control. *ACM Transactions on Graphics*, 26(3), 2007.