PRACSYS: An Extensible Architecture for Composing Motion Controllers and Planners

Andrew Kimmel, Andrew Dobson, Zakary Littlefield,
Athanasios Krontiris, James Marble, and Kostas E. Bekris*

Computer Science Department, Rutgers University, Piscataway, NJ, 08554, USA,
kostas.bekris@cs.rutgers.edu

Abstract. This paper describes a software infrastructure for developing controllers and planners for robotic systems, referred 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 paper describes the overall architecture, explains important features and provides use-cases that evaluate aspects of the infrastructure.

1 Introduction

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 before 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 [1], OpenRAVE [2], OMPL [3], PQP [4], USARSim [5]. 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, however, which require the integration of multiple controllers or the integration of higher-level planners with control-based methods. For

*This work has been supported by NSF CNS 0932423. Any conclusions expressed here are of the authors and do not reflect the views of the sponsor.

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, interest is moving towards higher-dimensional and more complex robotic platforms, including humanoid systems and robots with complex dynamics.

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, PRACSYS[1], 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 methods 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 [3] 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) [6]. 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 [4]), physics-based simulation (e.g., Open Dynamics Engine [7]), visualization (e.g., OpenSceneGraph [8]), as well as planners (e.g., through OMPL [3]) or controllers (e.g., Matlab implementations of controllers).

After reviewing related contributions, this paper outlines the software architecture and details the two main components of PRACSYS, simulation and planning. The paper 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.

[1] SourceForge package: http://sourceforge.net/projects/pracsys/.

2 Related Work

The Robot Operating System (ROS) [6] is an architecture that provides libraries and tools to help software developers create robot 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 [9]. Gazebo is focusing on 3D simulation of multiple systems with dynamics [1]. PRACSYS shares objectives with Gazebo but focuses mostly on a control and planning interface that is not provided by Gazebo.

There is a series of alternative simulators, such as USARSim [5], the Microsoft Robotics Developers studio [10], UrbiForge [11], the Carmen Navigation Toolkit [12], Delta3D [13] and the commercial package Webots [14]. 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! [15] is a library for grasping research, while OpenRAVE [2] 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) [16] and the Open Motion Planning Library (OMPL) [3]. 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 problems like replanning.

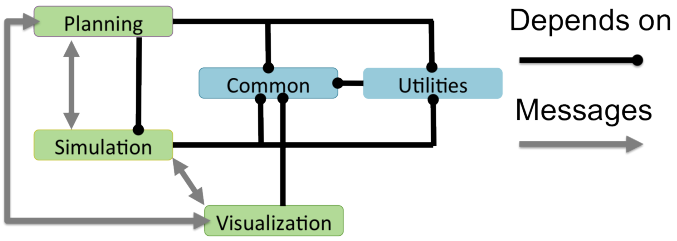


Fig. 1. Package interactions. ROS nodes communicate via message passing: *simulation*, *visualization*, and *planning*. The *common* and *utilities* packages are dependencies of the previous three.

3 General Architecture of PRACSYS

The proposed architecture is composed of several modules, following the architecture of the Robotic Operating System (ROS) [6]. ROS's architecture has separate nodes launched as executables which communicate via message passing and are organized into packages and stacks. A package is a collection of files, while a stack is a collection of such packages. PRACSYS is a stack and each node launched from PRACSYS is 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 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 some 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[2] library to facilitate efficient implementations.

The higher-level packages include *simulation*, *visualization*, and *planning*. The *simulation* package has *common* and *utilities* as dependencies, while it is responsible for simulating the physical world in which the agents reside and contains integrators and collision checking. The same 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 provides an interface between the user and the *simulation*, such as selecting agents and providing manual control. The state of systems simulated in the *planning* package can be different than the state in the ground truth simulator, which is useful for applications such as planning under uncertainty.

PRACSYS makes use of a package for loading simulations from files YAML [17] format. There is also a set of dependencies to external software packages, such as the Approximate Nearest Neighbors library [18].

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

4 Description of the Packages

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

4.1 Ground-truth Simulation and Controller Architecture

The *simulation* package 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.

[2] Boost is a set of libraries that extend the functionality of the C++ programming language.

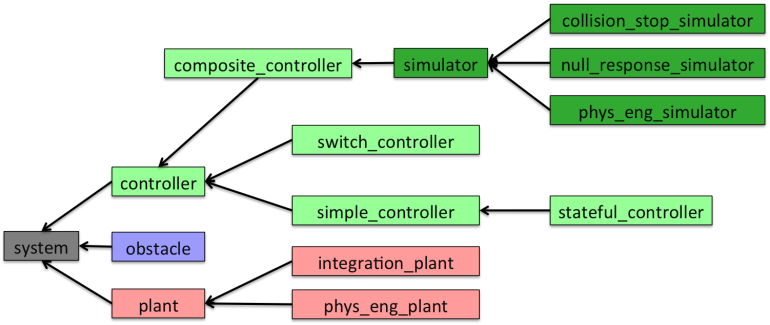


Fig. 2. A view of the class inheritance tree for the PRACSYS *system*. All classes are abstract, with the exception of the switch controller and the three concrete simulator classes.

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 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 3, which ensures that every system properly updates its state and control. The functions in the pipeline are responsible for the following:

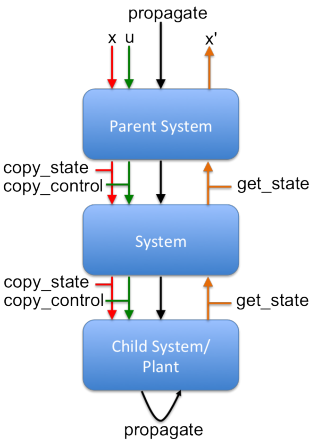


Fig. 3. The core interface of a system: x and x' are states, while u is a control.

copy state: receive a state from a higher-level system, potentially manipulate this state, and pass it down to lower-level systems.
copy control: receive a control from a higher-level system, potentially manipulate this control, and pass it down to lower-level systems.
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).
get state: receives the state from a lower-level system. This allows higher-level systems to query for the full state of the simulation.

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 box constrained region in 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 functions, which primarily fall under the categories of initialization, set functions, and get functions. Systems are broken into: physical plants, obstacles 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 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-based simulation, they can simply disable this functionality by omitting it from the input configuration 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 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. OMPL is a software package developed for planning purposes [3]. 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.

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 has computed a path to 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.

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.

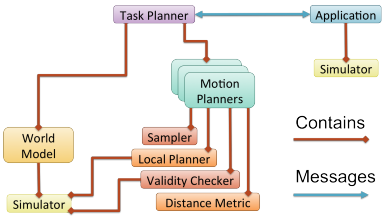


Fig. 4. The general structure of the *planning* modules. The *task planner* contains multiple *motion planners*. The *task planner* also contains a *world model* and communicates with the *simulation* node.

Motion planners are responsible for coming up with trajectories for individual or 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 a set of black-box modules, which may have a wide variety of underlying implementations. Furthermore, because of the flexibility of the *system* class, planners can plan over controllers as well. In this case, planners are essentially able to 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. 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.

4.3 Visualization

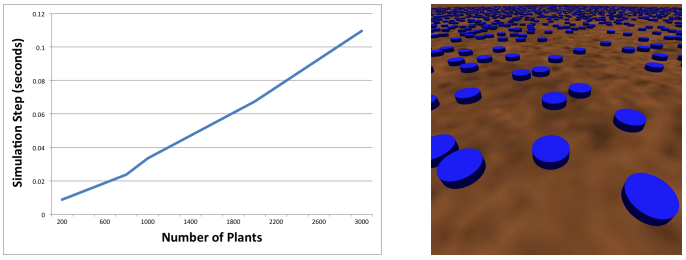
The *visualization* node is responsible for visualizing any aspect needed by the other nodes. Users interact with the simulation environment through the *visualization*. This includes, but is not limited to, camera interaction, screen shots and videos, and robot tracking. The *visualization* provides an interface to develop alternative implementations, in case users do not want to use the provided implementation based on Open Scene Graph (OSG) [8].

4.4 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. Configuration 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 [18], which is useful for motion planning.

5 Use-Cases

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



(a) (b)

Fig. 5. Figure on the left shows a plot of simulation steps vs number of plants, figure on the right shows 3000 plants running in the environment.

5.1 Showing Scalability for Multiple Agents

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. The trend shows a linear increase in simulation step duration with the number of agents, even with simulations of thousands of agents. The Velocity Obstacle (VO) Framework was introduced as a lightweight reactive obstacle avoidance technique [19]. The basic VO framework as well as several extensions have been implemented in PRACSYS and have also been used in large-scale experiments.

5.2 Planning over Controllers using LQR Trees

Through a C/C++ interface to Octave and its control package [20], PRACSYS can utilize the optimal control guarantees of linear quadratic regulators (LQR). Octave is an open-source Matlab clone. In this way, PRACSYS can run software developed in Matlab with little effort for the conversion. An implementation of LQR-Tree has been developed in PRACSYS [21]. 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 the Rapidly-exploring Random Tree algorithm (RRT) and its variants. It computes 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 high-level language, primarily intended for numerical computations into PRACSYS.

5.3 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 be reused and re-combined to create new functionality.

For example, the framework given in the SIMBICON project [22], 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 6. A breakdown of this hierarchy is as follows: **ODE Simulator:** A simulator built on top of 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.

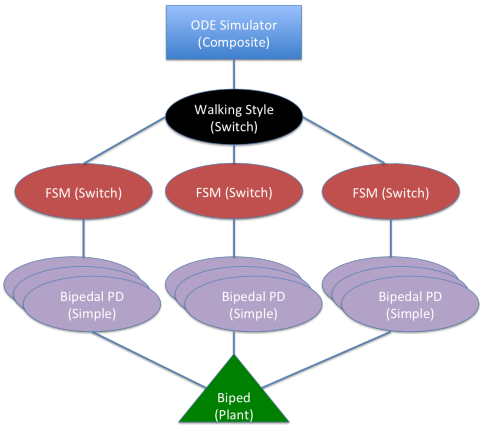


Fig. 6. A visual representation of the controller composition for controlling a bipedal robot

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.

5.4 Integration with Octave, OMPL, and MoCap Data

PRACSYS has been integrated with several other software packages in order to extend its functionality, such as Octave, OMPL, and motion capture data from CMU. **Motion Capture** (MoCap) data is used to animate characters in a realistic manner. 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.

6 Discussion

PRACSYS is an extensible environment for developing and composing motion controllers and planners. It supports multi-agent simulations, physics-based tools, and incorporates Matlab code, the OMPL library [3] and MoCap data. There are multiple important future steps for PRACSYS. A current pursuit 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 ns-3 [23]. 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 [1], OpenRAVE [2] and by utilizing existing ROS functionality [6].

Bibliography

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

[2] Diankov, R., Kuffner, J.J.: OpenRAVE: A Planning Architecture for Autonomous Robotics. Technical report, CMU-RI-TR-08-34, The Robotics Institute, CMU (2008)

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

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

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

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

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

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

[9] Gerkey, B., Vaughan, R.T., Howard, A.: The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor and Systems. In: ICAR. (2003) 317–323

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

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

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

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

[14] Michel, O.: Webots: Professional Mobile Robot Simulation. IJARS **1**(1) (2004)

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

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

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

[18] Arya, S., Mount, D.M.: Approximate nearest neighbor searching. In: Proc. 4th Annual ACM-SIAM Symposium on Discrete Algorithms. (1993) 271–280

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

[20] Eaton, J.W.: GNU Octave Manual. Network Theory Limited (2002)

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

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

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