IEEE Inter Conf on Robotics and Automation (ICRA), Rome, Italy, 2007, pp. 3711–3716

OOPS for Motion Planning: An Online, Open-source, Programming System

Erion Plaku Kostas E. Bekris Lydia E. Kavraki

***Abstract*— The success of sampling-based motion planners has resulted in a plethora of methods for improving planning components, such as sampling and connection strategies, local planners and collision checking primitives. Although this rapid progress indicates the importance of the motion planning problem and the maturity of the field, it also makes the evaluation of new methods time consuming. We propose that a systems approach is needed for the development and the experimental validation of new motion planners and/or components in existing motion planners. In this paper, we present the Online, Open-source, Programming System for Motion Planning ($\mathrm{OOPS_{MP}}$), a programming infrastructure that provides implementations of various existing algorithms in a modular, object-oriented fashion that is easily extendible. The system is open-source, since a community-based effort better facilitates the development of a common infrastructure and is less prone to errors. We hope that researchers will contribute their optimized implementations of their methods and thus improve the quality of the code available for use. A dynamic web interface and a dynamic linking architecture at the programming level allows users to easily add new planning components, algorithms, benchmarks, and experiment with different parameters. The system allows the direct comparison of new contributions with existing approaches on the same hardware and programming infrastructure.**

I. INTRODUCTION

Motivated by the success of the initial sampling-based motion planning framework [1] in dealing with high-dimensional problems [2], [3], a series of alternative approaches and planning components have been proposed over the years [4]–[15]. While the extensive literature constitutes significant progress toward solving more difficult problems, it also makes the experimental validation of new methods increasingly challenging and time consuming.

The lack of a common infrastructure for comparing motion planning techniques implies that a considerable amount of time and effort must be spent on implementing existing algorithms. This may have the negative effect of significantly reducing the scope of such comparisons. Even when the effort is spent, it is generally difficult to obtain a fair result. The correct and efficient implementation of algorithms solely based on their high-level description is difficult. Disparities in results can also occur due to low-level programming choices and platform-dependant issues. Although the distribution of source code or binaries is a viable solution, it is often hindered by differences in coding and interface standards.

Work on this paper has been supported in part by NSF 0308237 and a Sloan Fellowship to LEK. Experiments reported in this paper have been obtained on equipment supported by NSF CNS 0454333, and NSF CNS 0421109 in partnership with Rice University, AMD, and Cray.

The authors are with the Computer Science Department, Rice University, Houston, TX, 77005, USA. {plakue, bekris, kavraki}@rice.edu

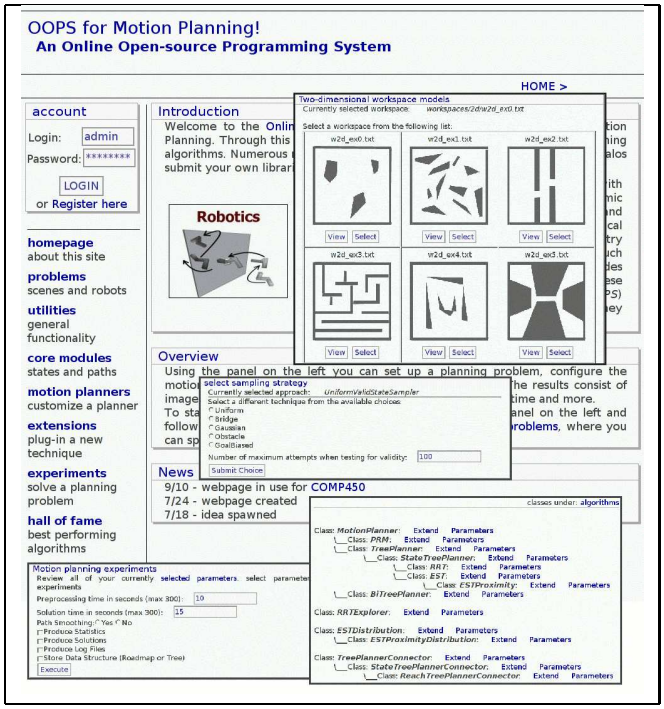


Fig. 1. Various instances of the $\mathrm{OOPS_{MP}}$ webpage on top of the home page. These instances highlight some of its features, such as the set of benchmarks, the capability to set various parameters, to execute online experiments, extend implemented classes, and plug-and-play components.

In addition, components unrelated to the comparisons could affect the result. For example, a comparison between two different sampling strategies can be affected by the choice of collision detectors, proximity algorithms, or even utility routines such as vector and matrix operations.

An additional and important difficulty is the lack of an openly available set of benchmarks that can be easily used by different researchers to experimentally validate the efficiency of their methods. This common set of test cases is important for reproducing experimental results and comparing against alternative approaches.

This paper presents the Online, Open-source, Programming System for Motion Planning ($\mathrm{OOPS_{MP}}$). The objective of $\mathrm{OOPS_{MP}}$ is to aid researchers in the development, experimental validation, and comparison of motion planning components and methods. We undertook a systems approach to develop a programming system for motion planning that is simple to use, easy to extend, robust, and efficient. Some of the functionality provided by $\mathrm{OOPS_{MP}}$ is as follows.

a) Benchmarks: $\mathrm{OOPS_{MP}}$ provides a common set of benchmarks. This set includes two-dimensional and three-

dimensional static workspaces, together with polygonal and polyhedral robots and facilitates planning problems with multiple rigid bodies.

b) Modules: $\mathrm{OOPS_{MP}}$ identifies programming modules and algorithmic components that are common across different motion planning methods. It provides implementations for these components, together with general purpose utilities and data structures.

c) Motion planning methods: On top of these modules, popular motion planning algorithms have been implemented, such as the Probabilistic RoadMap (PRM) [1], Rapidly-exploring Random Tree (RRT) [16], Expansive Spaces Tree (EST) [17], and many others.

d) General functionality for experimental validation: $\mathrm{OOPS_{MP}}$ contains functionality to select different parameters of motion planners and other components, execute experiments, gather statistics, and visualize results.

e) Online plug-and-play functionality: With the aid of an online web interface, $\mathrm{OOPS_{MP}}$ allows researchers to plug-and-play new motion planning components, methods, and benchmarks. The uploaded code can be dynamically linked with the existing infrastructure and experiments can be automatically executed for the uploaded plugins.

f) Open-source: $\mathrm{OOPS_{MP}}$ provides direct access to the source code. We hope that this will allow other researchers to contribute to $\mathrm{OOPS_{MP}}$. Such contributions will significantly improve the system and the quality of current implementations of different motion planning components and methods.

Overview

The rest of this paper presents in more detail the proposed system. The hierarchical and modular code structure of $\mathrm{OOPS_{MP}}$ is described in section II. The online web interface through which $\mathrm{OOPS_{MP}}$ is accessed and extended is discussed in section III. The most important feature of $\mathrm{OOPS_{MP}}$ is the capability to extend its current functionality. A description of the necessary steps required to plug-and-play new components onto $\mathrm{OOPS_{MP}}$ is provided in section IV. The simplicity of extending $\mathrm{OOPS_{MP}}$ is also demonstrated in section IV, where it is shown how to add a recent tree-based motion planning method to $\mathrm{OOPS_{MP}}$. The implementation of the tree-based planner uses many of the available components in $\mathrm{OOPS_{MP}}$. A description of how to run experiments, gather statistics, and visualize results using $\mathrm{OOPS_{MP}}$ and some experimental results using different sampling components and different tree-based planners are given in section IV.

II. $\mathrm{OOPS_{MP}}$ – Code Structure

A systems approach and a modular design were followed to make the code structure of $\mathrm{OOPS_{MP}}$ simple to use and easily extendible. The design process of $\mathrm{OOPS_{MP}}$ focused on identifying general purpose utilities and common components used by many sampling-based motion planning methods. The result of this design process was a hierarchical division of the code into three main modules: (i) utilities, (ii) core, and (iii) motion planners. The core module depends on the utilities module and the motion planners module depends on the utilities and core modules. The next issue considered in the design of $\mathrm{OOPS_{MP}}$ was the programming language to be used for the implementation of these modules. Since the objectives of $\mathrm{OOPS_{MP}}$ are to achieve robustness, efficiency, and extendibility, using a combination of C and C++ was considered to be the most appropriate choice for implementing the modules of $\mathrm{OOPS_{MP}}$. The low-level and frequently used utilities are implemented in C to ensure that the generated code is efficient. The high-level functionality is implemented using C++ to make the code easy to extend by simply providing concrete implementations of abstract classes or further extending existing classes.

Each main module is further divided into smaller modules, as described in the rest of this section. Fig. 2 illustrates the hierarchical structure of the code in $\mathrm{OOPS_{MP}}$.

A. Utilities Module

The utilities module encompasses general purpose functionality that is needed by higher-level modules, such as core and motion planners modules. It is currently divided into several submodules: (1) general, (2) math, (3) data structures, and (4) geometry. Fig. 2(b) provides an illustration.

1) General Submodule: This submodule contains implementations of general purpose functionality such as generation of pseudorandom and quasirandom sequences [18], uniform and Gaussian sampling of numbers, uniform generation of points inside disks or on the surface of high-dimensional spheres or ellipsoids. It also contains functionality to generate random or enumerate all possible permutations and combinations of k elements out of n objects. These low-level functionalities are typically necessary to construct high-level sampling strategies.

Additional functionality contained in this module includes methods for measuring computational time or gathering other statistics. It also includes memory management functionality such as the allocation and freeing of one-dimensional, two-dimensional, or high-dimensional C arrays.

2) Math Submodule: The math submodule, as the name indicates, encompasses mathematical functionality such as low-dimensional vector and matrix operations, including affine and other types of transformation. There are also implementations of topological Lie groups and algebras, such as the special Euclidean groups SE(2) and SE(3) and their associated algebras, including the generation of uniform random samples, multiplication and addition, logarithmic and exponential operations, geodesic interpolations, and distance measures. In addition, there is also functionality to do numerical integration of ordinary differential equations using Euler, Runge-Kutta, or Bulirsch-Stoer methods of different orders of approximation.

We note that the above module provides the most commonly needed functionality used by sampling-based motion planning methods. The problem of obtaining correct and efficient implementations of many of these routines has also been raised in [19]. For this reason, special care is taken to

obtain robust, fast, and optimized implementations of many of these routines.

3) Data Structures Submodule: This submodule contains implementations of different data structures such as roadmaps and trees and functionality to search and manipulate these data structures. The data structures submodule also includes implementations of proximity algorithms, which are often used by sampling-based motion planners to compute nearest neighbors. Currently, it contains proximity methods for computing exact nearest neighbors, such as the brute-force linear approach or the efficient Gnat method [15], or computing approximate nearest neighbors, such as Random and DPES [5]. The underlying implementation of these data structures is in C in order to make them as efficient as possible. However, there is also a C++ interface to wrap up the C implementations and furthermore allow extendibility of these data structures.

4) Geometry Submodule: The geometry submodule contains implementation of 2D and 3D workspaces and collision detectors and distance computations between polygons and polyhedra, respectively. The 2D implementations uses fast and optimized primitives, while 3D implementations link to publicly available collision detectors such as `PQP` [20].

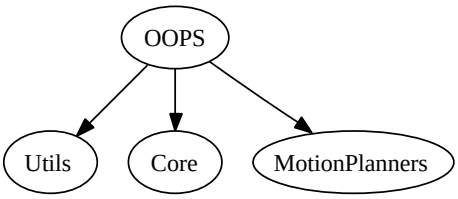
B. Core Module

The core module contains definition and implementations of common components required by sampling-based methods. The current division of the core module includes the (1) state space, (2) state sampling, and (3) path submodules. Fig. 2(c) provides an illustration.

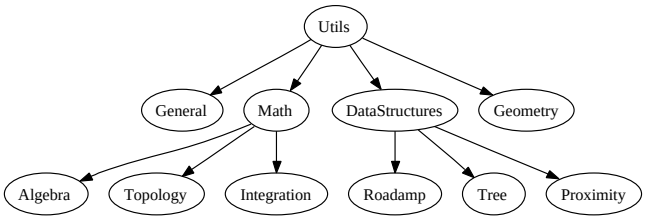
1) State-Space Submodule: The state-space submodule defines the topology of the motion planning problem. As an example, the topology of the sofa problem is SE(2), since in the sofa problem the robot is allowed to only rotate and translate. The state space also includes the definition of distances between two points, i.e., configurations in the case of the sofa problem. This submodule also contains an implementation of the state space for multiple robots as a Cartesian product of the state spaces associated with each robot. In this way, high-level sampling-based motion planners can be used to solve problems involving more than one robot.

2) State-Sampling Submodule: The state-sampling submodule contains different sampling strategies. Current implementations include uniform, Gaussian [14], bridge [21], obstacle-based [13], and goal-biased methods.

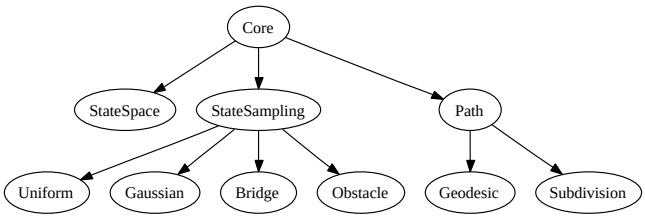
3) Path Submodule: The path submodule corresponds to what is commonly referred to in motion planning as the local planner. Note that the local planner typically consists of two parts: path definition, which indicates how two states are connected to each-other, and path validation, which checks to ensure that the path satisfies all the constraints, e.g., collision avoidance. A path is parameterized by its length. In the case of SE(2) and SE(3) state spaces, paths are defined as either linear or geodesic interpolations. Path validation uses either an incremental or subdivision approach.



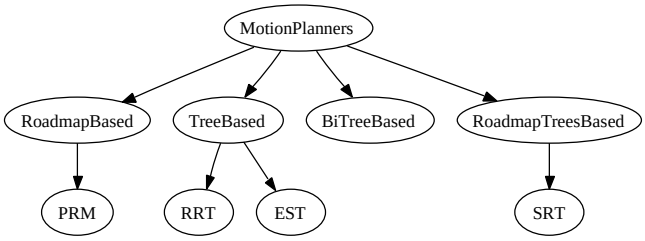
(a) The three main modules of $\mathrm{OOPS_{MP}}$.



(b) Partial structure of utilities module.



(c) Partial structure of core module.



(d) Partial structure of motion planners module.

Fig. 2. Code structure of $\mathrm{OOPS_{MP}}$. The figure illustrates the hierarchical and modular structure of the code by showing many, but not all, available modules and submodules. Researchers could contribute improvements to existing modules, create new modules by extending existing ones, or add completely new modules that are not currently available in $\mathrm{OOPS_{MP}}$. The code in $\mathrm{OOPS_{MP}}$ supports the plug-and-play paradigm.

C. Motion Planners Module

This module contains definitions and implementations of sampling-based motion planning methods. Since it relies upon the utilities and core modules, low-level planning parameters are hidden at this point. This modular approach facilitates the implementation of new methods. The planners can be used to solve motion planning problems with one or more robots using single or multiple queries. Sampling-based motion planners are divided into methods that construct a roadmap, single tree, bi-directional tree, or a roadmap of trees. In order to ensure extendibility, all the sampling-based methods extend or provide implementations of an abstract motion planning class. Fig. 2(d) provides an illustration.

Roadmap methods include `PRM` and its variants. Note however that, for example, `GaussianPRM` is obtained by replacing the uniform sampling component in `PRM` by the

Gaussian sampling component. The connection strategies in PRM are defined by the particular proximity component that is being used. Similarly, the local planner is defined by the selected path-definition and path-validation components. Single and bi-directional tree methods include implementations of RRT and EST. These implementations are also modular, since they rely on different components, such as extend, connect, explore, etc. Furthermore, tree-based methods also use sampling, path, and proximity components. The roadmap of trees methods, such as SRT [4], utilize the roadmap and tree-based modules that are currently in use.

III. $\mathrm{OOPS_{MP}}$ – Online access and interface

One of the primary goals of our system is to allow the direct comparison of motion planners on the same hardware and programming infrastructure. A user of $\mathrm{OOPS_{MP}}$ is able to achieve this by selecting the appropriate modules and parameters to run an experiment and executing the code on a publicly accessible server. We provide a webpage interface to assist this procedure that is illustrated in Fig. 1. In this section we describe how this interface can be used and how it has been implemented.

The webpage has been built using "php", with the objective of being dynamic, parameterizable, simple to extend and integrate with the code. Since the user of the webpage is provided the capability to upload and execute code on a public server, an authorization procedure is used so as to hopefully avoid possible malicious attacks. After registration, a corresponding file space is allocated on the server where the various files specific to a user are stored. These files are related to benchmarks, code, parameter selection files, and outputted results. In this way, each account operates independently and can publish results, such as benchmarks or source code, to other users only upon request.

The basic functionality for the webpage interface is to construct an input parameter file according to HTML forms that are passed as a parameter to the executable of $\mathrm{OOPS_{MP}}$ responsible for running experiments. The generation of the forms is an automated procedure based on templates that define a protocol between the code and the interface. This is an important element of $\mathrm{OOPS_{MP}}$ because it allows the inclusion of potential additional parameters with a small overhead and allows user extensions to the current infrastructure as described in the next section.

The webpage structure mostly follows the architecture of the code and the order with which parameters need to be specified to run an experiment.

A. Benchmarks

After the initial homepage, a user is able to define the problem that will be solved. $\mathrm{OOPS_{MP}}$ currently supports motion planning problems with multiple rigid bodies. The parameters that have to be specified are the workspace, the number of robots and the type of each robot. Through the "problems" link of the webpage, access is provided to the libraries of benchmarks available with the system. Providing easy access to these sets of benchmarks is an important contribution of $\mathrm{OOPS_{MP}}$. Both two dimensional and three dimensional problems are supported, so there are two types of environments and robots available, accessible from corresponding online libraries which visualize how each workspace and robot geometry look like. A user can directly access a benchmark file, corresponding either to a workspace or a robot. The format of benchmarks is also available and users can upload their own files. A visualization of the benchmark is provided if the format of the file is recognized by the system and the file is accessible to be selected for experiments. As it is mentioned earlier, uploaded files are not accessible to other members of $\mathrm{OOPS_{MP}}$ until the user decides to publish the file to the community.

B. Parameter Selection

The links "utilities", "core modules" and "motion planners" provide a way to select parameters for experiments. Access to the related source code is also provided through these links. As mentioned, the only available parameter currently supported for the "utilities" category is the choice of a proximity computation algorithm. Following the code structure, the "core modules" link provides access to choices for state space representation, sampling strategy and definition of a local path. The "motion planners" link provides the interface for the selection of a motion planner. The choices range between the PRM algorithm, tree-based algorithms and bi-directional tree-based algorithms. The current implementations for tree-based algorithms include RRT [16], EST [22], and several of their variations. For algorithms, sampling strategies and local path definitions additional parameters may be specified. For example, for the PRM algorithm, the fraction of total planning time devoted to sampling must be defined. All the selections made by the user are stored and they do not have to be resubmitted each time that an experiment is executed.

C. Experiments and Queries

Through the "experiments" link experiments can be executed over the online interface.

Before the execution of an actual experiment, appropriate queries must be selected to define the motion planning problem. There is functionality for generating random queries given the selected workspaces and robots. Access to a library of queries similar to that of workspaces and robots is also provided. The format of queries is described. A query file, which contains multiple queries, can be uploaded. When a user selects an existing query, a good practice for the user is to validate whether the query file contains valid queries for the given workspace and robot, which is a provided functionality.

After the definition of a query, an experiment can be initiated. An important parameter defines how much time will be devoted in preprocessing during execution (e.g., roadmap construction in PRM) and how much time in query solving. The user has multiple choices on the type of files that will be outputted from an experiment, such as statistics, logs with

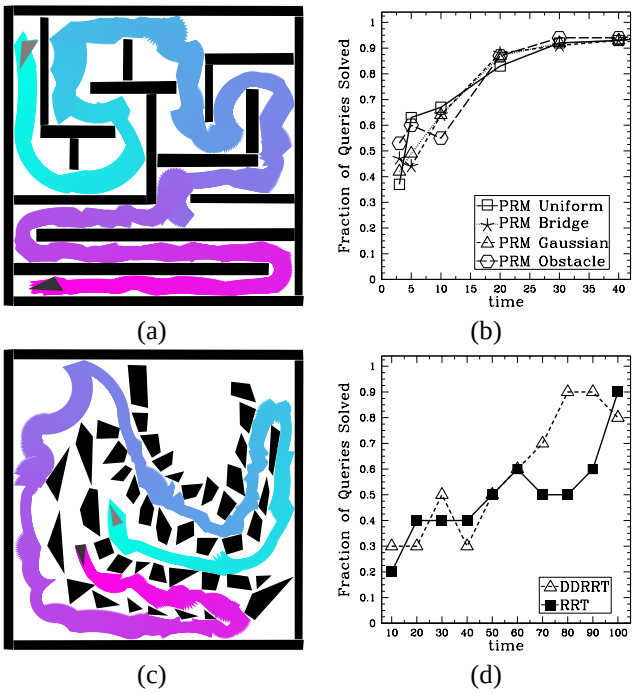


Fig. 3. A postscript file is automatically created at the end of each experiments that contains the figures requested by the user. (a, c) Paths from initial to final configurations for two different environments. Directions of paths are shown as gradients. Obstacles are shown in black. Observe that in several places the robot comes close to the obstacles, almost touching their boundaries. (b) Fraction of queries solved by different PRM variants as a function of total computation time using the workspace and robot in (a). (d) Fraction of queries solved by RRT and DDRRT as a function of total computation time using the workspace and robot in (c).

```
class DDRRTVertex : public RRTVertex
{
public:
 DDRRTVertex(void) : RRTVertex(), m_radius(HUGE_VAL)
 virtual ~DDRRTVertex(void) { }
 virtual double getRadius(void) const { return m_radius; }
 virtual void   setRadius(const double r) { m_radius = r; }
protected:
 double m_radius;
};

class DDRRT : public RRT
{
public:
 static const double DEFAULT_RADIUS;
 DDRRT(void) : RRT(), m_radius(DEFAULT_RADIUS)
 virtual ~DDRRT(void) { }
 virtual double getRadius(void) const { return m_radius; }
 virtual void setRadius(const double r) { m_radius = r; }
 virtual void propagate(const double rtime)
 {
   State_t       rx   = getStateAllocatorStack()->push();
   DDRRTVertex_t vnear= 0x0;
   MyClock_t     clk;
   StartTime(&clk);
   getRRTExplorer()->allowAddingIntermediates(false);
   do
   {
      getValidStateSampler()->sample(rx);
      vnear = dynamic_cast<DDRRTVertex_t>
                (getTree()->getData(getProximity()->neighbor(rx)));
   }
   while(ElapsedTime(&clk) < rtime &&
         getStateSpace()->distance(rx, vnear->getState()) >= vnear->getRadius());
   if(vnear && getRRTExplorer()->explore(vnear, rx,
        rtime - ElapsedTime(&clk))== RRTExplorer::EXPLORER_TRAPPED)
      vnear->setRadius(getRadius());
   getStateAllocatorStack()->pop();
 }
protected:
 double m_radius;
};

OOPSMP_RegisterContribution(DDRRT);
```

Fig. 4. C++ code illustration for implementing DDRRT, a recent tree-based motion planning method. The functionality and infrastructure provided by OOPS_{MP} facilitates the implementation of new ideas. Once the code is submitted through the web interface, DDRRT is immediately available for comparisons with existing methods.

debug messages and information about the data structure constructed by the algorithm.

Fig. 3 gives examples of experiments that can be executed with OOPS_{MP}. Fig. 3(a, b) provides a comparison among various sampling strategies, while Fig. 3(c, d) shows the result of a comparison between RRT and DDRRT [9].

After the execution of a motion planning experiment, statistics about the performance of the planners employed during the experiment can be reported, such as solution time for all the queries and number of queries solved by the planner. Furthermore, profiling information is reported, which informs the user how the total execution time was divided among different motion planning modules. Finally, for experiments where the planner was successful in finding a solution, a postscript file containing the solution path is automatically created and can be accessed online. The roadmap or the tree data-structure constructed during an experiment can also be visualized.

IV. OOPS_{MP} – EXTENDIBILITY

The most important property of OOPS_{MP} is the ability to implement an alternative algorithm for one of the planning modules or a new motion planning algorithm and link it directly with the existing infrastructure, run experiments, and compare against motion planning methods and components already implemented in OOPS_{MP}.

We illustrate the simplicity of using OOPS_{MP} to implement new ideas by showing all the necessary steps that are needed to implement DDRRT [9], a recent tree-based motion planning method.

1) Select module for extension: From the "extensions" link of the online interface a user gets access to a webpage that provides a tree structure of the extendible classes in the existing code. In the case of DDRRT, the user selects to extend RRT, since DDRRT is based on RRT. The webpage creates a template for DDRRT, which simply indicates that DDRRT extends RRT, and shows DDRRT as a branch under RRT in the tree structure of the extendible classes.

2) Define additional parameters required by the new module: The new module inherits all the parameters of the module that it is extending. The online web interface also allows the user to define additional parameters required by the new module. In the case of DDRRT, the user specifies that a parameter of type "real" is required for the "setRadius" function in DDRRT. The current implementation in OOPS_{MP} supports several parameter types, such as "bool," "int," "real," "string," or any combination of these types.

All the parameter types and the keywords associated with these types are added to the template created for DDRRT. The web interface gives the user the ability to add new parameters, remove, or edit existing parameters.

3) Upload code and create module: Using the web interface, the user can upload any number of C/C++ files. Fig. 4 provides an illustration of the C++ code required to

implement DDRRT using the $\mathrm{OOPS_{MP}}$ infrastructure.

Upon uploading the code, the user clicks on a "Create Module" button, which invokes $\mathrm{OOPS_{MP}}$ to compile the code and create a dynamically linked library that is stored only in the user's space. Possible error messages from the compiler are also displayed on the webpage. When all the errors have been fixed, the new module is ready for use.

4) Experimental validation: The user can now use the web interface to select and use the submitted module for experiments. To continue with the example, DDRRT would be presented as a possible choice when the user selects a motion planner. Upon selecting DDRRT as the motion planner, the user has the option of entering values for the parameters such as "setRadius," or use the provided default values. We note that it is only the user who created DDRRT that has access to this module. The other users are not presented with the choice of DDRRT as a possible motion planner, since this module has not been made publicly available yet.

After setting the parameters, the user can run experiments using DDRRT. $\mathrm{OOPS_{MP}}$ uses C/C++ functionality to search and load the DDRRT module from the dynamically linked libraries. At the end of each experiment, the user can gather statistics and visualize the results. The illustrations in Fig. 3 were produced by following the step-by-step procedure described in this section.

V. DISCUSSION

The contribution of this work is $\mathrm{OOPS_{MP}}$, an online, open-source, programming system for motion planning. Researchers can use $\mathrm{OOPS_{MP}}$ to implement new ideas, experimentally evaluate the performance of their methods, or run comparisons with existing methods. By making $\mathrm{OOPS_{MP}}$ open-source, easily accessible and extendible through an online web interface, we hope that other researchers will contribute to it motion planning benchmarks, improve implementations of current modules, and add new motion planning components and methods. A community-based effort is the key ingredient in ensuring that the infrastructure of $\mathrm{OOPS_{MP}}$ continues to be useful to researchers as the research in motion planning progresses and new motion planners are developed. Our objective is to continue improve the infrastructure of $\mathrm{OOPS_{MP}}$, add support for large-scale experiments, and in the near future implement parallel and distributed motion planning algorithms that utilize the functionality provided by $\mathrm{OOPS_{MP}}$.

ACKNOWLEDGMENT

The code infrastructure of $\mathrm{OOPS_{MP}}$ is developed by Erion Plaku and the web interface is developed by Kostas E. Bekris. Since the creation of $\mathrm{OOPS_{MP}}$, a growing number of researchers and students have made significant contributions to $\mathrm{OOPS_{MP}}$. A detailed list of contributors and their contributions can be found at `http://www.cs.rice.edu/CS/Robotics/OOPSMP.html`. The authors thank Drew Bryant for his assistance with the web server.

REFERENCES

[1] L. E. Kavraki, P. Švestka, J.-C. Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE Transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, June 1996.

[2] H. Choset, K. M. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. E. Kavraki, and S. Thrun, *Principles of Robot Motion: Theory, Algorithms, and Implementations*. Cambridge, MA: MIT Press, 2005.

[3] S. M. LaValle, *Planning Algorithms*. Cambridge, MA: Cambridge University Press, 2006.

[4] E. Plaku, K. E. Bekris, B. Y. Chen, A. M. Ladd, and L. E. Kavraki, "Sampling-based roadmap of trees for parallel motion planning," *IEEE Transactions on Robotics*, vol. 21, no. 4, pp. 597–608, 2005.

[5] E. Plaku and L. E. Kavraki, "Quantitative analysis of nearest-neighbors search in high-dimensional sampling-based motion planning," in *Workshop Algorithmic Foundations of Roboticsxs*, New York, NY, 2006.

[6] A. M. Ladd and L. E. Kavraki, "Fast tree-based exploration of state space for robots with dynamics," in *Algorithmic Foundations of Robotics VI*, M. Erdmann, D. Hsu, M. Overmars, and A. F. van der Stappen, Eds. Springer, STAR 17, 2005, pp. 297–312.

[7] D. Hsu, J.-C. Latombe, and H. Kurniawati, "On the probabilistic foundations of probabilistic roadmap planning," *International Journal of Robotics Research*, vol. 25, no. 7, pp. 627–643, 2006.

[8] G. Sánchez and J.-C. Latombe, "On delaying collision checking in PRM planning: Application to multi-robot coordination," *International Journal of Robotics Research*, vol. 21, no. 1, pp. 5–26, 2002.

[9] A. Yershova, L. Jaillet, T. Simeon, and S. M. LaValle, "Dynamic-domain RRTs: Efficient exploration by controlling the sampling domain," in *IEEE International Conference on Robotics and Automation*, 2005, pp. 3856–3861.

[10] M. Morales, S. Rodriguez, and N. Amato, "Improving the connectivity of PRM roadmaps," in *IEEE International Conference on Robotics and Automation*, 2003, pp. 4427–4432.

[11] D. Hsu, R. Kindel, J.-C. Latombe, and S. Rock, "Randomized kinodynamic motion planning with moving obstacles," in *Workshop on Algorithmic Foundations of Robotics*, 2002.

[12] B. Burns and O. Brock, "Sampling-based motion planning using predictive models," in *IEEE International Conference on Robotics and Automation*, 2005, pp. 3120–3125.

[13] N. M. Amato, B. Bayazit, L. Dale, C. Jones, and D. Vallejo, "OBPRM: An obstacle-based PRM for 3d workspaces," in *Robotics: The Algorithmic Perspective*, P. Agarwal, L. E. Kavraki, and M. Mason, Eds. AK Peters, 1998, pp. 156–168.

[14] V. Boor, M. H. Overmars, and A. F. van der Stappen, "The gaussian sampling strategy for probabilistic roadmap planners," in *IEEE International Conference on Robotics and Automation*, 1999, pp. 1018–1023.

[15] S. Brin, "Near neighbor search in large metric spaces," in *International Conference on Very Large Data Bases*, 1995, pp. 574–584.

[16] S. M. LaValle and J. J. Kuffner, "Rapidly-exploring random trees: Progress and prospects," in *New Directions in Algorithmic and Computational Robotics*, B. R. Donald, K. Lynch, and D. Rus, Eds. AK Peters, 2001, pp. 293–308.

[17] D. Hsu, J.-C. Latombe, and R. Motwani, "Path planning in expansive configuration spaces," *Int. J. Comput. Geom. & Appl.*, 1998, to appear.

[18] S. M. LaValle and M. S. Branicky, "On the relationship between classical grid search and probabilistic roadmaps," in *Algorithmic Foundations of Robotics*, J. D. Boissonat, J. Burdick, K. Y. Goldberg, and S. A. Hutchinson, Eds., 2003.

[19] J. Kuffner, "Effective sampling and distance metrics for 3d rigid body path planning," in *IEEE International Conference on Robotics and Automation*, vol. 4, 2004, pp. 3993–3998.

[20] E. Larsen, S. Gottschalk, M. C. Lin, and D. Manocha, "Fast proximity queries with swept sphere volumes," Department of Computer Science, University of N. Carolina, Chapel Hill, TR99-18, 1999.

[21] D. Hsu, T. Jiang, J. Reif, and Z. Sun, "The bridge test for sampling narrow passages with probabilistic roadmap planners," in *IEEE International Conference on Robotics and Automation*, 2003.

[22] D. Hsu, R. Kindel, J.-C. Latombe, and S. Rock, "Randomized kinodynamic motion planning with moving obstacles," *International Journal of Robotics Research*, vol. 21, no. 3, pp. 233–255, 2002.