Journal of Software Engineering for Robotics 1(1), September 2013, 1-10
ISSN: 2035-3928

Reducing the Barrier to Entry of Complex Robotic Motion Planning Software

David COLEMAN[1] Nikolaus CORRELL[1]

[1] Department of Computer Science, University of Colorado at Boulder, 430 UCB, Boulder, Colorado 80309-0430

Abstract—Developing robot-agnostic motion planning software involves synthesizing many disparate fields of software engineering and robotic theory while at the same time accouting for a large variability in hardware designs and control paradigms. As the capabilities and power of robotic software increases, the complexity and learning curve to new users also increases. If the learning curve for applying and using the software on robots is too high, even the most powerful of motion planning frameworks is useless. A growing need exists in robotic software engineering to aid users in getting started with and customizing the provided components as necessary for particular robotic applications. In this paper a case study is presented for some of the best practices found for lowering the barrier of entry of the MoveIt motion planning framework that allows users to 1) quickly get basic motion planning functionality with very little initial setup, 2) automate the configuration and optimization of the framework where possible, 3) easily customize aspects of the toolchain, and 4) benchmark the results of different configurations.

Index Terms—Robotic Motion Planning, Frameworks, Barrier to Entry, Setup, Usability, MoveIt

1 Introduction

Managing the increasing complexity of modern robotic software is a difficult engineering challenge faced by roboticists today. The size of the code bases of common open source robotic software frameworks such as ROS [1] and OROCOS [2] are swelling [3], and the reqired breadth of knowledge for understanding the deep stack of software from control drivers to high level planners is increasing in formidablity. As it is beyond the capabilities for any one researcher to have the necessary domain knowledge for every aspect of a robot's tool chain, it is necessary to assist users in the configuration, customization, and optimization of the various software components of a robotic framework.

1.1 Barriers to Entry

The term *barriers to entry* is used here in the context of robotic software engineering to refer to the time, effort, and knowledge that a new user must invest in the integration of a software component to an arbitrary robot. This can include for example creating a virtual model of the robot's geometry and dynamics, customizing configuration files, choosing the fastest algorithmic approach for the application, and finding the best parameters for various algorithms.

Motion planning, and described in more detail later, requires many varying degrees of customization and optimization for any particular robot to operate properly. Choosing the right parameters for each utilized algorithm and software component typically involves expert human input using domain-specific knowledge. Many new users to a software package, particularly as robotics becomes more mainstream, will not have the breadth of knowledge to customize every aspect of the planning tool chain. When the knowledge of a new user is insuficient for the requirements of the software, the barriers to entry become insurmountable and the software unusable. One of the emerging requirements of a robot agnostic motion planning framework is implementing mechanisms that will automatically setup and tune the pipeline for arbitrary robots.

Another motivation for lowering the barrier to entry of complex robotics software is the *paradox of the active user*. This paradox explains a common observation in many user studies that *users never read manuals* but start attempting to use the software immediatley [4].

Even experts in the required fields

Talk about paradox of active user?

Discuss methods to lower barrier of entry:

Regular paper – *Manuscript received April 19, 2009; revised July 11, 2009.*

- *This work was supported by xxxxxxxx (No.xxxxxxxx) (sponsor and financial support acknowledgment goes here).*

Very easy to run / quick to setup demo (Hello World)
User interface with well laid out user affordances
Automatic generation of configuration files
Reduction of Magic Numbers
Auto-tuning of necessary parameters via algorithms
Customization of specific components of the framework
Benchmarking for quantitative results

1.2 Benefits of Larger User Base

This need to lower the barrier of entry is beneficial to the software itself in that it enables more users to utilize the framework. If the software is being sold for profit, the benefits of larger user base is obvious. If instead the software is a free open-source project, as most successful robotic frameworks have been [2], lowering the barrier to entry is also beneficial in that it creates a larger community of users. As the number of users increases, the speed in which bugs are identified and fixed increases. It is also hoped that development contributions to the code base increases, though this correlation is not as strong [3]. Additionally, one of the key strengths of a larger community for an open source project is increased participation of users assisting with quality assurance, documentation, and support [4].

Another benefit of lowering the barrier of entry is its implications in educational and research applications?

1.3 Motion Planning

Motion planning is a growing field of robotics that enables robots to move within their environment from one task or configuration to another. Its typical use case is the control of robotic arms from one location to another while taking into account the arm's reachability constraints, performing collision checking, and other constraints such as motion dynamics.

The development of a motion planning framework involves combining many different fields of robotics and software engineering. Of particular importance is creating the structures and classes to share common data between the many different components. These basic data components include a model of a robot with collision bodies, a method for maintaining the state of the robot during planning and execution, and a method for maintaining the environment as perceived by the robot's sensors, henceforth referred to as the "planning scene".

In addition to these basic common data structures, a motion planning framework requires many different functional components. Of primary importance is the motion planning module, which includes one or more algorithms suited for the solving the expected motion planning problems a robot might encounter. The field of motion planning algorithms is large and no one-size fits all solution exists yet, so a framework that is robot agnostic should likely include an assortment of algorithms. For more information on the selection of the proper algorithm for any particular planning problem, the reader is referred to [XX].

Other primary functional components include the collision checking module, which should be as fast as possible for the geometric primitives and meshes in the planning scene and robot model. A forward kinematics solver is required to propagate the robot's geometry based on its joint positions and an inverse kinematics solver is required when planning in the configuration of an end effector. Also required is a component for taking into account other potential constraints, such as joint/velocity/torque limits, and stability requirements.

Secondary components must also be integrated into a powerful motion planning framework. Depending on what configuration space a problem was solved in, a generated motion planning solution of position waypoints must be parameterized into a time-variant trajectory to be executed. A controller manager must decide the proper low level controllers for the necessary joints for each trajectory. Finally a perception interface must update the planning scene with recognized objects from a perception pipeline as well as optional raw sensor data.

Higher level applications are then built on top of these motion planning components to coordinate more complex tasks, such as pick and place routines. Other optional components of a motion planning framework can include benchmarking tools, introspection and debug tools, as well as user-facing graphical user interfaces.

2 Related Work

There has been much written and developed to addressed these issues in robotics, but typically the identified design goals for robotic software engineering have emphasized the need for platform independence, scalability, real-time performance, software reuse, and distributed layouts [5], [6], [7]. In [7]'s survey of nine open source robotic development environments, a collection of metrics were used which included documentation and graphical interfaces, but no mention was made of setup time, barrier to entry, or automated configuration.

In this paper we will be using the new MoveIt Motion Planning Framework as our case study for barriers of entry to the usage of robotics software. There already exists a number of motion planning frameworks available today, both open and closed source. A quick survey of these software projects...

OpenRave OOPSMP MPK - Motion Planning Kit Motion Strategy Library MoveIt [1]

3 MoveIt Motion Planning Framework

Details about Moveit
ROS [cite] sdas
asdasdas asd asd asdasdasd asd as

4 Lowering the Barrier of Entry

The following features have been implemented in MoveIt with the motivation to attract as many users as possible. The results

of these features on the size of the user base will be discussed in a later section.

A. Basic Motion Planning Out of the Box

1. MoveIt Setup Assistant

One of the main features of MoveIt that has made it popular is its ratio of power and features to required setup time. A beginner to motion planning software can with very little effort take a kinematic model of an arbitrary articulated rigid body robot and execute motion plans in a virtual environment. With a few extra steps of setting up the correct hardware interfaces, one can then execute the motion plans on their actual robotic hardware.

The main facility that provides out of the box support for beginners is the MoveIt Setup Assistant (SA). The SA is a graphical user interface that steps new users though the initial configuration requirements of using a custom robot with the motion planning framework (Figure 1). It accomplishes the task of automatically generating text-based configuration files necessary for the initial operation of MoveIt. These configurations include a self-collision matrix, planning group definitions, robot poses, end effector semantics, virtual joints, and passive joints. A three dimensional model of the robot being configured is displayed on the right side of the SA GUI and various links on the robot are highlighted during configuration to visually confirm the actions of the user.

FIGURE 1: Screenshot of MoveIt Setup Assistant

The required input for the SA is a robot model file, which currently accepts the Universal Robotic Description Format (URDF) [5] or Collada [6] file formats. These XML schemas describe the physical layout of a robot's link, joints, and other necessary modeling components. They can also combine appropriate 3D CAD meshes, collision models, dynamics, joint limits, sensors and other components. Using a properly formatted robot model file, MoveIt can automatically accomplish many of the required tasks in a motion planning framework including forward and inverse kinematics, collision checking, and joint limit enforcement.

If one truly desired, the steps within the SA could almost entirely be automated themselves, but were kept manual so as to allow edge cases and unusual customizations to be accomplished.

2. MoveIt Rviz Motion Planning Plugin

The details of the automated configuration is left for the next section, but after the steps in the SA are completed the output is a ROS package containing a collection of configuration files and launch scripts (roslaunch files). The launch files include a demo script that will startup a visualization tool (rviz) with the new robot loaded and ready to run state of the art motion planning algorithms in non-physics based simulation. Using simple mouse-based interactive markers and the robot's various planning groups, the user can configure simple motion planning problems and quickly test the framework's capabilities.

An example demo task would be using the mouse to drag a robot end effector from a start position to a goal position around some virtual obstacle. Using another GUI - the MoveIt Rviz Motion Planning Plugin - the user can click the ?Plan? button and watch MoveIt effortlessly plan the arm in a collision free path around the obstacle (Figure 2).

FIGURE 2: Screenshot of MoveIt Rviz Motion Planning Plugin

It is important to emphasize the effect of a quick ?Getting Started? demo on a new user unaccustomed to MoveIt or motion planning frameworks in general. The reinforcement effect of initial success encourages the novice and enables them to start going deeper into the functionality and code base. If the entry barrier is too low, a new user will likely give up and turn to other frameworks or custom solutions rather than continue to blindly fix software that they have no experience in.

3. Hardware Configuration and Execution

Once the user is comfortable with the basic tools and features provided by MoveIt, the next step is to configure their robot's various actuators and control interfaces to accept trajectory commands from MoveIt. This step usually requires some custom coding to account for the specifics of the robot hardware - the communication bus, real-time requirements, and control theory implementations. At the abstract level, all MoveIt requires is that the robot hardware accepts a standard ROS trajectory message containing a discretized set of time-variant waypoints including desired positions, velocities, and accelerations.

B. Automate the configuration and optimization of the framework

The size and complexity of a feature-rich motion planning framework like MoveIt requires many parameters and configurations of the software be automatically setup and tuned. MoveIt accomplishes this both in the setup phase of a new robot - using the Setup Assistant - and sometimes during the runtime of the application.

1. Self Collision Matrix

The second step of the SA is the generation of a self-collision matrix for the robot. This collision matrix encodes pairs of links on a robot that never need to be checked for self-collision due to the kinematic infeasibility of there actually being a collision. Reasons for having collision checking disabled between two links includes 1) links that can never reach each other kinematically, 2) adjoining links that are connected and so are by design in collision, and 3) links that are always in collision for any other reason including inaccuracies in the robot model and precision error. This self-collision matrix is generated by running the robot through tens of thousands of random joint configurations and recording statistics of each link's collision status. The algorithm then automatically populates a list of link pairs that have been determined to never need to be collision checked. This saves future motion planning runs time because it reduces the amount of collision checks that are required.

2. Semantic Robotic Description Format

The other six steps of the SA all provide graphical front ends for the data required to populate the semantic robotic description format (SRDF) file used by MoveIt. The SRDF provides meta data of the robot model useful to motion planning, such as which set of joints constitutes an arm and which set of links is considered part of the end effector. Requiring the user to configure all the required semantic information by hand in a text editor would be far more tedious and difficult than using an interface that populates the available options for each required field.

The last step of the SA is to generate all launch scripts and configuration files. Not only does this step involve outputting to file the collected configurations during the step-by-step user interface, but it all generates a series of default configuration and launch files that are customized for the particular robot using the URDF and SRDF information. These defaults include the velocity and acceleration limits for each joint, the kinematic solvers for each planning group, the available planning algorithms and projection evaluators for planning. Default planning adapters are setup for pre and post-processing of motion plans, such as fixing slightly invalid start states and smoothing generated trajectories. Default benchmarking setups, controller and sensor manager scripts, and empty object databases are all generated.

3. Internal Optimization

I think there are examples of this but Ioan knows more about this than I...

C. Easily customize aspects of the toolchain

Out of the box MoveIt lowers the barrier to entry by not requiring the user to provide their own implementation of any of the components in the motion planning framework. The Open Motion Planning Library (OMPL) [7] is configured as the default set of utilized planning algorithms. The Fast Collision Library (FCL) [8] is pre-configured as the collision checking component and the Orocos Kinematics and Dynamics Library (KDL) [9] is the default kinematics solver for kinematic chains. By default no perception components are configured

All these default choices however are limiting to more advanced users who have their own research or application-specific needs to fulfil. All of the just mentioned components are setup using plugin interfaces that are shared objects loaded at run time. Using common plugin interfaces, MoveIt can easily be customized by user created plugins for any or all of the motion planning components.

MoveIt is a plugin-centric framework that mostly avoids using message-passing inter-component communication, such as ROS messages. This decreases much of the latency delays inherent in message passing techniques and increases. The extensibility of the framework is greatly enhanced by not forcing users to use any particular algorithmic approach. Essentially, MoveIt provides a set of data sharing and synchronization tools, sharing between all components the robot's model and state.

One plugin in particular that MoveIt reduces the barrier to entry for customization is the inverse kinematics plugin. The default KDL plugin uses numerical techniques to convert from an end effector to joint configuration space. A must faster solution can be achieved by configuring OpenRave's IKFast [10] plugin that analytically solves the inverse kinematics problem. A combination of MoveIt scripts and the IKFast Robot Kinematics Compiler automatically generates the C++ code and plugin needed to increase the speed of motion planning solutions by up to 3 orders of magnitude.

D. Benchmark the results of different configurations

Be able to configure and switch out motion planning algorithms and approaches is a powerful feature of MoveIt, but its usefulness is limited without the ability to quantify the results of any changes. Optimization criteria such as path length, planning time, smoothness, distance to nearest obstacle, and energy minimization need benchmarking tools to enable users and developers to find the correct set parameters and components for any given robotic application.

MoveIt again provides a low barrier to entry for benchmarking with easy to create benchmarking configuration files that allow each test to be setup for comparison against other algorithms and values. Multivariable parameter sweeps for finding the optimal value for an algorithm's performance can be accomplished by simply supplying a upper and lower search value as well as an increment amount. Results can be output into generic formats for use in different plotting tools.

5 Results

Quantify MoveIt's popularity so far Released 05/06/2013 Number of debian downloads? ask Tully to get this? 163 users on the MoveIt mailing list Show plot of MoveIt posts over time Show plot of users on mailing list over time Show plots from https://www.ohloh.net/p/moveit

6 Usability Issues with MoveIt

There are still barriers to entry Setting up controllers is difficult Large code base is intimidating Built by one programmer so not the easiest layout

7 Conclusion

Beyond the usual considerations in building a successful motion planning framework for robotics, an open source project that desires to maintain an active user base needs to take into account the barrier of entry to new users. As the algorithms become more complicated and the number of components and code base increases, configuring an arbitrary robot to utilize this framework becomes a daunting task requiring domain-specific expertise in a very large breadth of theory and implementation. To account for this, quick and easy initial configuration, with partially automated optimization,

and easily extensible components for future customization are becoming a greater necessity in motion planning and robotic software engineering in general.

ACKNOWLEDGMENTS

The authors would like to thank...

REFERENCES

[1] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, no. 3.2, 2009. 1

[2] H. Bruyninckx, "Open robot control software: the orocos project," in *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, vol. 3. IEEE, 2001, pp. 2523–2528. 1

[3] A. Makarenko, A. Brooks, and T. Kaupp, "On the benefits of making robotic software frameworks thin." 1

[4] J. M. Carroll, *Interfacing thought: Cognitive aspects of human-computer interaction.* The MIT Press, 1987. 1.1

[5] Y. hsin Kuo and B. MacDonald, "A distributed real-time software framework for robotic applications," in *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*, 2005, pp. 1964–1969. 2

[6] T. H. Collett, B. A. MacDonald, and B. P. Gerkey, "Player 2.0: Toward a practical robot programming framework." 2

[7] J. Kramer and M. Scheutz, "Development environments for autonomous mobile robots: A survey," *Autonomous Robots*, vol. 22, no. 2, pp. 101–132, 2007. 2

First Author received his B. Sc. and M. Sc. degrees in mechanical engineering from the *** University, in 1977 and 1984, respectively, and the Ph. D. degree in computing from *** University, in 1992. In 1994, he was a faculty member at *** University and in 1996 at *** University. Currently, he is a professor in the Department of Information System Engineering at *** University. He has published about 100 refereed journal and conference papers. His research interest covers robotics, software engineering, and distributed systems. Prof. Author received research award from Science Foundation, and the Best Paper Award of the XX International Conference in 2000 and 2006, respectively. He is a member of ACM and IEEE.

Second Author received his B. Sc. and M. Sc. degrees in mechanical engineering from the *** University, in 1977 and 1984, respectively, and the Ph. D. degree in computing from *** University, in 1992. In 1994, he was a faculty member at *** University and in 1996 at *** University. Currently, he is a professor in the Department of Information System Engineering at *** University. He has published about 100 refereed journal and conference papers. His research interest covers robotics, software engineering, and distributed systems. Prof. Author received research award from Science Foundation, and the Best Paper Award of the XX International Conference in 2000 and 2006, respectively. He is a member of ACM and IEEE.