



Google
Summer of Code

Update OMPL and ompl_interface [Moveit 2]

29.03.2023

Name: Sameer Gupta

Country: India

Email: sameergupta4873@gmail.com

GitHub ID: [sameergupta4873](https://github.com/sameergupta4873)

Timezone: IST (GMT + 5:30)

Resume: [Sameer Gupta's Resume](#)

Motivation

By contributing to MoveIt2, I can create a powerful and versatile platform for robot manipulation and motion planning that can be used in a wide range of applications, from manufacturing and logistics to healthcare and entertainment. Moreover, contributing to MoveIt2 can provide a unique opportunity to hone my skills in C++, which is a widely used language in the field of robotics. In addition to the technical benefits, contributing to MoveIt2 can also be a great way to meet like-minded people and build connections in the robotics community. By participating in discussions, attending events, and collaborating on projects, you can expand your network and learn from others who share your passion for robotics.

Studies

I'm in my Second Year at **Veermata Jijabai Technological Institute**, Mumbai, India

Degree: Information Technology

Relevant Coursework: Theory of Computation, Data Structures and Algorithms, Discrete Math, Database Management Systems, Web Technologies

Contributions

Merged PRs for respective issues::

1. [Char is a char. string is a string #871](#)
2. [Duplicate imports in Servo launch files #818](#)
3. [Specify LANGUAGE CXX in cmake where appropriate #1923](#)

Proposed Deliverables

1. Using **clang-tidy** to identify areas of improvement in **OMPL**.
2. Remove **raw pointer** usage in the most commonly used parts and Eliminate ambiguity in the API about the **lifetime of objects**.
3. Ideas for moving **ompl_interface** into **moveit_core**.

Timeline

Date and Time	Event	Proposed Tasks
March 20	Applications Open	<ul style="list-style-type: none"> Register for GSoC 2022
April 4 - 11:30PM IST	GSoC contributor application deadline	<ul style="list-style-type: none"> Submit final proposals on the portal
April 4 - May 4	Pre-Selection Phase	<ul style="list-style-type: none"> Thoroughly examining the <code>ompl_interface</code> plugin. Looking out for the improvement areas. Learn best C++ practices and modern C++ techniques and algorithms.
May 5 - May 28	Accepted GSoC projects announced + Community Bonding Period	<ul style="list-style-type: none"> I have my final semester exams during this period(May 8 – May 31) so it might be tough to dedicate myself completely to this. Talk to and learn from other people and their interests. Learn and Add clang-tidy to the toolbox and familiarize myself with its applications. Diligently study any additional resources provided.
May 29 - June 11 Week 1 & Week 2	Coding Starts	<ul style="list-style-type: none"> Running all the modernized checks of clang-tidy for <code>ompl_interface</code> Marking and fixing the areas of improvements and coming up with the best possible solutions.
June 12 - June 26 Week 3 & Week 4		<ul style="list-style-type: none"> Eliminating the usage of raw pointers throughout the <code>ompl_interface</code> plugin Reviewing the affected areas due

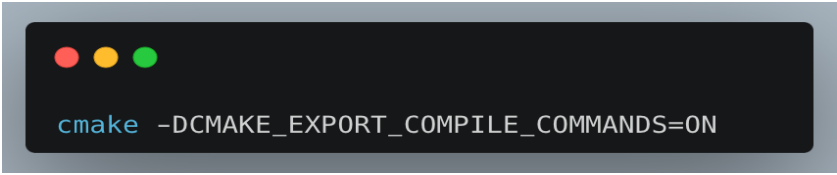
		to changes made.
June 26 - July 10 Week 5 & Week 6		<ul style="list-style-type: none"> • Eliminating the usage of raw pointers throughout the <code>ompl_interface</code> plugin • Reviewing the affected areas due to changes made.
July 11 - July 18 Week 7	Phase 1 Submissions	<ul style="list-style-type: none"> • Performing tests to ensure OMPL runs perfectly. • Submit documentation mentioning checks and generated warnings, errors and solutions .
July 18 - July 31 Week 8 & Week 9		<ul style="list-style-type: none"> • Examining the code again to filter out usage of type-casting and ambiguity of lifetimes of objects in api. • Fixing the affected areas due to changes made.
August 1 - August 14 Week 10 & Week 11		<ul style="list-style-type: none"> • Re-running the checks and assuring that no clang-tidy errors or warning is encountered. • Testing the Changes made in the <code>ompl_interface</code>.
August 15 - August 28 Final Weeks	Final Submissions	<ul style="list-style-type: none"> • Submit the complete changes made in the OMPL and tests performed. • Submit complete documentation of the changes and achieve results.
Post GSoC		<ul style="list-style-type: none"> • Continue some work with generalization of <code>ompl_interface</code> in <code>moveit_core</code>. continue contributing to other areas of Moveit2 :)

In-Depth Breakdown

1. Using clang-tidy to identify areas of improvement in OMPL

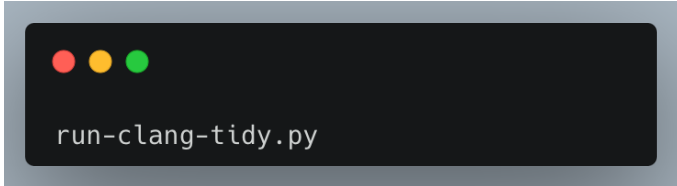
We will use the following steps to identify areas for improvement in OMPL:

1. Installing **clang-tidy** on your system. We can install it by downloading the binary from the official LLVM website.
2. Navigate to the **ompl_interface** directory of our moveit2 package in a terminal.
3. Run the following command to generate a compile_commands.json file in the **build/** directory:



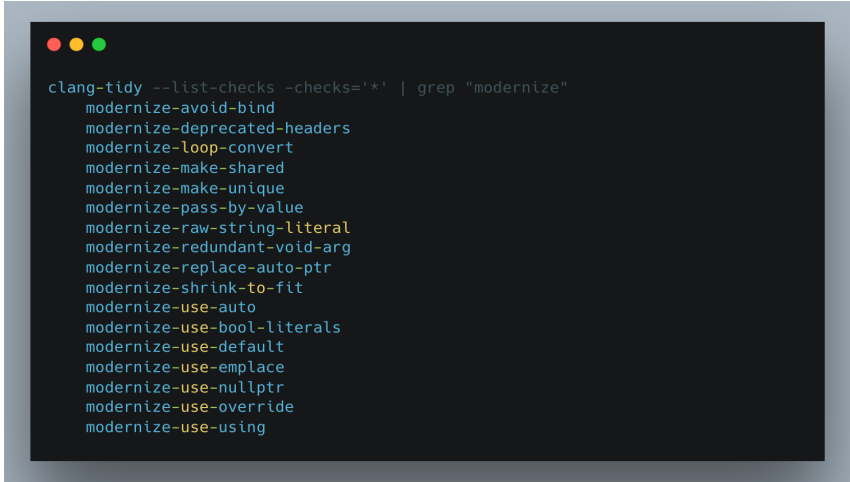
```
cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=ON
```

4. Run the following command to identify areas of improvement in ompl_interface :
 - a. Running the tool with the default checks on each translation unit in the project we simply run:




```
run-clang-tidy.py
```

- b. Let's check what other checkers it has to offer (by passing **-checks='*'** to see them all), and specifically grep for the ones with **modernize** in their names. Those checkers advocate usage of modern language constructs:d,



```
clang-tidy --list-checks -checks='*' | grep "modernize"
modernize-avoid-bind
modernize-deprecated-headers
modernize-loop-convert
modernize-make-shared
modernize-make-unique
modernize-pass-by-value
modernize-raw-string-literal
modernize-redundant-void-arg
modernize-replace-auto-ptr
modernize-shrink-to-fit
modernize-use-auto
modernize-use-bool-literals
modernize-use-default
modernize-use-emplace
modernize-use-nullptr
modernize-use-override
modernize-use-using
```

And according to this list, we will check the areas of improvement in `ompl_interface` by one-by-one running the above checks to modernize the package:



```
run-clang-tidy.py -header-filter='.*' -checks='-* ,modernize-use-override' -fix
```


Here, *clang-tidy* will now be invoked on each translation unit in your project and will add *overrides* where they are missing. The parameter ***-header-filter='.*'*** makes sure ***clang-tidy*** actually refactors code in the headers being consumed in the translation unit. The parameter *checks='-* ,...'* makes sure all default checks are disabled.

5. rerun the tool to ensure that all issues have been addressed.

We know that `clang-tidy` is a static analysis tool, so it can only detect issues that can be identified without actually running our code. It may not catch all instances of ambiguity and raw pointer usage, so it's still important to thoroughly test our code to ensure its correctness.

For example, let us check the modernize checks on a C++ file in `ompl_interface` (i.e., [ompl_planner_manager.cpp](#)).

Run command:



```
clang-tidy ompl_planner_manager.cpp -header-filter=.* -checks=-* ,modernize-*
```

Output:

```

/home/parallels/ws_moveit2/src/moveit2/moveit_planners/ompl/ompl_interface/src/ompl_planner_manager.cpp:85:8
: warning: use a trailing return type for this function [modernize-use-trailing-return-type]
  bool initialize(const moveit::core::RobotModelConstPtr& model, const rclcpp::Node::SharedPtr& node,
    ^
  auto
/home/parallels/ws_moveit2/src/moveit2/moveit_planners/ompl/ompl_interface/src/ompl_planner_manager.cpp:93:8
: warning: use a trailing return type for this function [modernize-use-trailing-return-type]
  bool canServiceRequest(const moveit_msgs::msg::MotionPlanRequest& req) const override
    ^
  auto                                     -> bool
/home/parallels/ws_moveit2/src/moveit2/moveit_planners/ompl/ompl_interface/src/ompl_planner_manager.cpp:98:1
5: warning: use a trailing return type for this function [modernize-use-trailing-return-type]
  std::string getDescription() const override
    ^
/home/parallels/ws_moveit2/src/moveit2/moveit_planners/ompl/ompl_interface/src/ompl_planner_manager.cpp:121:
3: warning: use a trailing return type for this function [modernize-use-trailing-return-type]
  getPlanningContext(const planning_scene::PlanningSceneConstPtr& planning_scene,
    ^

```

So, as the clang modernize checks suggest the changes, performing the changes would be tedious because using **"auto"** as a return type for an overridden function isn't right unless we knew the return type at compile time. So we could ignore the changes suggested, and similarly, for multiple files, the above checks could be performed.

2. Removing raw pointer usage in the most commonly used parts of ompl_interface.

To remove raw pointer usage in the most commonly used parts, we can follow these steps:

1. Identifying the parts of code where raw pointers are most commonly used. This may include functions that accept or return raw pointers, as well as areas where memory is allocated or deallocated manually.
2. Determine if the raw pointers can be replaced with smart pointers, such as `std::unique_ptr` or `std::shared_ptr`. Smart pointers are a safer and more robust way to manage dynamic memory in C++ because they automatically handle memory allocation and deallocation.
3. Updating all code that interacts with the affected areas to use the new smart pointer types instead of raw pointers.
4. If smart pointers cannot be used, consider using container classes such as `std::vector`, `std::array`, or `std::map` that manages memory.
5. Testing our code after making changes to ensure that it still behaves as expected.

If we have already removed the usage of raw pointers in our C++ package, we have taken a significant step towards eliminating ambiguity in the API about the lifetime of objects. However, there are still some additional steps we can take to ensure clarity and consistency in our APIs:

1. Analyzing who is responsible for creating and destroying objects, as well as any other relevant information about the lifetime of the objects.
2. Use smart pointers, such as `std::unique_ptr` or `std::shared_ptr`, to manage the lifetime of objects. This can help make the lifetime requirements of objects more explicit and reduce the risk of memory leaks.
3. Use `const` and reference qualifiers to specify the read/write permissions of function arguments. This can help prevent unintentional modifications to objects that should not be modified and make the lifetime requirements of objects more explicit.

By taking these steps, we can help ensure that your APIs are clear and consistent with regards to the lifetime of objects, even if we have already removed the usage of raw pointers.

Instances in the `ompl_interface` where raw pointers are used and their elimination with `unique` or `shared` pointers:

1. [projection_evaluators.h](#)

Solu

```
class ProjectionEvaluatorLinkPose : public ompl::base::ProjectionEvaluator
{
public:
    ProjectionEvaluatorLinkPose(const ModelBasedPlanningContext* pc, const std::string& link);

    unsigned int getDimension() const override;
    void defaultCellSizes() override;
    void project(const ompl::base::State* state, OMPLProjection projection) const override;

private:
    const ModelBasedPlanningContext* planning_context_; // raw pointers
    const moveit::core::LinkModel* link_; // raw pointers
    TSSStateStorage tss_;
};
```

If `planning_context_` is owned by the current object, we could use a `std::unique_ptr<ModelBasedPlanningContext>` instead:

```
std::unique_ptr<ModelBasedPlanningContext> planning_context_;
```


This would prevent memory leaks or dangling pointers and guarantee that the pointed-to object is automatically removed when the current object is destroyed. Similarly, we may use a reference in place of `link_` if `link_` is certain to exist for the duration of the current object:

```
const moveit::core::LinkModel& link_;
```

2. [projection evaluators.h](#)

```
namespace ompl_interface
{
class TSStateStorage
{
public:
    TSStateStorage(const moveit::core::RobotModelPtr& robot_model);
    TSStateStorage(const moveit::core::RobotState& start_state);
    ~TSStateStorage();

    moveit::core::RobotState* getStateStorage() const;

private:
    moveit::core::RobotState start_state_;
    mutable std::map<std::thread::id, moveit::core::RobotState*> thread_states_; // raw pointer
    mutable std::mutex lock_;
};
} // namespace ompl_interface
```

Solution:

Use `std::unique_ptr` or `std::shared_ptr` instead of raw pointers in the `std::map` to manage the lifetime of the `moveit::core::RobotState` objects. Here's an example using `std::unique_ptr`:

```
std::map<std::thread::id, std::unique_ptr<moveit::core::RobotState>> thread_states_;
```

3. Ideas for moving `ompl_interface` into `moveit_core`

To generalize `ompl_interface`, we can follow these steps:

1. Identify the common functionality that needs to be generalized across different implementations of the interface.
2. Define an abstract base class with pure virtual functions that declare the common interface that needs to be generalized. This class will serve as the blueprint for the different implementations of the interface.

3. Implement the different versions of the interface by deriving classes from the abstract base class and providing implementations for the pure virtual functions. Each derived class can specialize the behavior of the interface in its own way while still conforming to the general interface.
4. Ensure that the derived classes have a consistent naming convention and that the interface functions have the same names and parameters across all implementations. This will help maintain consistency and make it easier for users to understand and use the interface.
5. Use the abstract base class as a pointer or reference type to allow for polymorphism, which means that the interface can be used with any of the derived classes.
6. Finally, document the interface and its usage thoroughly to ensure that users can easily understand how to use the generalized interface with any of its implementations.

These are some of the ideas for moving `ompl_interface` to `moveit_core`, which will be second priority if the first two deliverables are fulfilled.

Programming Background

Programming has been going on for about two years. I only became interested in robots after beginning college and joining the SRA (Society of Robotics and Automation), the on-campus robotics group. I learned C++ as part of my curriculum, which assisted me in understanding the difficult concepts of C++ and OOP. Some of my earlier work includes:

1. [Obstacle Avoidance Race Car](#):

Obstacle Avoidance Racecar is an autonomous robot that was created in **Solidworks** and tested on **ROS, Gazebo, RViz**, and other platforms. Its primary goal is to avoid obstacles by following lines using the **ODG-PF algorithm, OpenCV, and PID**.

3. [Analog Controlled Robot](#):

A package and website with an analogue controller component were developed, talking with the **ESP-32** in C through wifi and allowing simple inputs through a web server by publishing analogue controller parameters like x and y speeds and angle.

Specifications

I am using a MacBook Air M1 with Parallels (VM) to run Ubuntu.

OS: Ubuntu 22.04 LTS 64-bit, MacOS

Processor: M1 Silicon.

GSoC Participation

I have not participated in GSoC before, and as such, I'm quite eager to explore and contribute to the world of open source. I understand the responsibility of GSoC and wish to assure you that I will do my utmost to fulfill it.

References

1. <https://www.kdab.com/clang-tidy-part-1-modernize-source-code-using-c11c14/>
2. https://github.com/ros-planning/moveit2/tree/main/moveit_planners/ompl/ompl_interface
3. <https://clang.llvm.org/extra/clang-tidy/#id1>
4. <https://www.cppstories.com/2018/04/deprecating-pointers/>
5. <https://stackoverflow.com/questions/26268721/why-cant-virtual-functions-use-return-type-deduction>

6. https://hanada31.github.io/pdf/ase19_smartpointer.pdf