Module Guide for 2D-RAPP

Ziyang(Ryan) Fang

April 17, 2025

1 Revision History

Date	Version	Notes
2025 March 19	1.0	Notes

2 Reference Material

This section records information for easy reference.

2.1 Abbreviations and Acronyms

symbol	description
AC	Anticipated Change
DAG	Directed Acyclic Graph
M	Module
MG	Module Guide
OS	Operating System
R	Requirement
SC	Scientific Computing
2D-RAPP	Explanation of program name
UC	Unlikely Change
SRS	Software Requirements Specification

Contents

List of Tables

List of Figures

3 Introduction

Decomposing a system into modules is a commonly accepted approach to developing software. A module is a work assignment for a programmer or programming team (Parnas et al., 1984). We advocate a decomposition based on the principle of information hiding (Parnas, 1972). This principle supports design for change, because the "secrets" that each module hides represent likely future changes. Design for change is valuable in SC, where modifications are frequent, especially during initial development as the solution space is explored.

Our design follows the rules layed out by Parnas et al. (1984), as follows:

- System details that are likely to change independently should be the secrets of separate modules.
- Each data structure is implemented in only one module.
- Any other program that requires information stored in a module's data structures must obtain it by calling access programs belonging to that module.

After completing the first stage of the design, the SRS, the MG is developed (Parnas et al., 1984). The MG specifies the modular structure of the system and is intended to allow both designers and maintainers to easily identify the parts of the software. The potential readers of this document are as follows:

- New project members: This document can be a guide for a new project member to easily understand the overall structure and quickly find the relevant modules they are searching for.
- Maintainers: The hierarchical structure of the module guide improves the maintainers' understanding when they need to make changes to the system. It is important for a maintainer to update the relevant sections of the document after changes have been made.
- Designers: Once the module guide has been written, it can be used to check for consistency, feasibility, and flexibility. Designers can verify the system in various ways, such as consistency among modules, feasibility of the decomposition, and flexibility of the design.

The rest of the document is organized as follows. Section 4 lists the anticipated and unlikely changes of the software requirements. Section 5 summarizes the module decomposition that was constructed according to the likely changes. Section 6 specifies the connections between the software requirements and the modules. Section 7 gives a detailed description of the modules. Section 8 includes two traceability matrices. One checks the completeness of the design against the requirements provided in the SRS. The other shows the relation between anticipated changes and the modules. Section 9 describes the use relation between modules.

4 Anticipated and Unlikely Changes

This section lists possible changes to the system. According to the likeliness of the change, the possible changes are classified into two categories. Anticipated changes are listed in Section 4.1, and unlikely changes are listed in Section 4.2.

4.1 Anticipated Changes

Anticipated changes are the source of the information that is to be hidden inside the modules. Ideally, changing one of the anticipated changes will only require changing the one module that hides the associated decision. The approach adapted here is called design for change.

AC1: The specific hardware on which the software is running.

AC2: The format of the initial input data may change, while still assuming a 2D environment.

AC3: The format of the initial input parameters may change, but the fundamental 2D assumption remains.

AC4: The constraints on the input parameters

AC5: The format of the final output data.

AC6: The constraints on the output results.

AC7: How the geometric constraints for obstacle avoidance are defined using input obstacle parameters.

AC8: How distance-based inverse kinematic equations are formulated based on input robot parameters

AC9: How the overall control flow of the path-planning calculations and iterations is orchestrated.

AC10: The implementation used for storing and managing the joint-space grid or graph data structure.

AC11: The algorithm selected for solving the inverse kinematics (IK) optimization problem.

AC12: The method or library used for visualizing and plotting robot paths and configurations.

4.2 Unlikely Changes

The module design should be as general as possible. However, a general system is more complex. Sometimes this complexity is not necessary. Fixing some design decisions at the system architecture stage can simplify the software design. If these decision should later need to be changed, then many parts of the design will potentially need to be modified. Hence, it is not intended that these decisions will be changed.

UC1: The fundamental requirement of navigating from a start position to a goal position remains unchanged.

UC2: The system assumes a two-dimensional environment, which is not expected to change.

UC3: The fundamental requirement of determining a feasible path from a single start position to a single goal position remains unchanged.

UC4: The system assumes a strictly two-dimensional planar environment.

UC5: The robot arm is assumed to have a serial manipulator structure, which will remain constant.

UC6: Obstacles within the environment are static and their positions and sizes do not change during path planning.

UC7: The robot will always have exactly one start and one goal configuration per planning task; handling multiple simultaneous goal points is out of scope.

5 Module Hierarchy

This section provides an overview of the module design. Modules are summarized in a hierarchy decomposed by secrets in Table 1. The modules listed below, which are leaves in the hierarchy tree, are the modules that will actually be implemented.

M1: Hardware-Hiding Module

M2: Input Parameters Module

M3: Output Format Module

M4: Output Verification Module

M5: Path Planning Module

M6: Collision Detection Module

M7: Inverse Kinematics Solver Module

M8: Plotting Module

M9: Configuration Management Module

M10: Logging and Debugging Module

M11: Control Module

Level 1	**Level 2**
Hardware Hiding	
Behaviour Hiding	Input Parameters Module Output Format Module Output Verification Module Inverse Kinematics Solver Module Configuration Management Module Control Module Path Planning Module Collision Detection Module
Software Decision	Plotting Module

Table 1: Module Hierarchy

6 Connection Between Requirements and Design

The design of the system is intended to satisfy the requirements developed in the SRS. In this stage, the system is decomposed into modules. The connection between requirements and modules is listed in Table 2.

7 Module Decomposition

Modules are decomposed according to the principle of "information hiding" proposed by Parnas et al. (1984). The *Secrets* field in a module decomposition is a brief statement of the design decision hidden by the module. The *Services* field specifies *what* the module will do without documenting *how* to do it. For each module, a suggestion for the implementing software is given under the *Implemented By* title. If the entry is *OS*, this means that the module is provided by the operating system or by standard programming language libraries. *2D-RAPP* means the module will be implemented by the 2D-RAPP software.

Only the leaf modules in the hierarchy have to be implemented. If a dash (–) is shown, this means that the module is not a leaf and will not have to be implemented.

7.1 Hardware Hiding Modules (M1)

Secrets: The data structure and algorithm used to implement the virtual hardware.

Services: Serves as a virtual hardware used by the rest of the system. This module provides the interface between the hardware and the software. So, the system can use it to display outputs or to accept inputs.

Implemented By: OS

7.2 Behaviour-Hiding Module

Secrets: The contents of the required behaviours.

Services: Includes programs that provide externally visible behaviour of the system as specified in the software requirements specification (SRS) documents. This module serves as a communication layer between the hardware-hiding module and the software decision module. The programs in this module will need to change if there are changes in the SRS.

Implemented By: OS or Simulation Framework

7.2.1 Input Parameters Module (M2)

Secrets: The structure, format, and validation rules of the robot parameters, obstacle data, and environment definitions.

Services: Accepts input parameters from the user, verifies the correctness and completeness of the input parameters, and converts them into appropriate internal data structures for further processing.

Implemented By: 2D Robot Arm Path Planner

Type of Module: Abstract Data Type

7.2.2 Output Format Module (M3)

Secrets: The structure and format of the system's output data, including joint positions, angles, and planned paths.

Services: Provides formatted output data after path planning and inverse kinematics computations, enabling further visualization or verification.

Implemented By: 2D Robot Arm Path Planner

Type of Module: Abstract Data Type

7.2.3 Output Verification Module (M4)

Secrets: The criteria and methods used to validate the computed paths and configurations.

Services: Verifies the correctness and feasibility of computed paths, ensuring there are no collisions and all constraints are satisfied. Generates warnings or errors when violations are detected.

Implemented By: 2D Robot Arm Path Planner

Type of Module: Abstract Object

7.2.4 Path Planning Module (M5)

Secrets: The algorithm and logic used to plan collision-free paths from start to goal configuration.

Services: Calculates feasible paths using provided obstacle information, initial and goal configurations. Ensures obstacle avoidance through distance-geometric and graph-based methods.

Implemented By: 2D Robot Arm Path Planner

Type of Module: Abstract Object

7.2.5 Collision Detection Module (M6)

Secrets: The algorithms and conditions for detecting collisions between the robot arm and environment obstacles.

Services: Determines if a specific robot configuration or path results in a collision. Reports detailed collision information to path planning and verification modules.

Implemented By: 2D Robot Arm Path Planner

Type of Module: Abstract Object

7.2.6 Inverse Kinematics Solver Module (M7)

Secrets: The mathematical methods and algorithms used for solving the inverse kinematics problem based on distance-geometric representation.

Services: Computes robot joint angles and positions from a given path. Provides accurate geometric conversions for the planned trajectory.

Implemented By: 2D Robot Arm Path Planner

Type of Module: Abstract Data Type

7.2.7 Configuration Management Module (M9)

Secrets: The management, storage, and retrieval methods for system configuration parameters such as robot dimensions, joint constraints, and obstacle definitions.

Services: Provides centralized management of configuration parameters, allows consistent and efficient access to these parameters by other modules.

Implemented By: 2D Robot Arm Path Planner

Type of Module: Abstract Data Type

7.3 Software Decision Module

Secrets: The design decision based on mathematical theorems, physical facts, or programming considerations. The secrets of this module are *not* described in the SRS.

Services: Includes data structures and algorithms used in the system that do not provide direct interaction with the user.

Implemented By: Python (using standard libraries and third-party packages)

7.3.1 Plotting Module (M8)

Secrets: The choice of visualization libraries and plotting algorithms for displaying robot configurations and paths.

Services: Generates graphical visualizations of robot arm configurations, joint trajectories, and obstacle environments. Facilitates analysis and debugging by providing clear visual representations of results.

Implemented By: Matplotlib (Python)

Type of Module: Library

7.3.2 Logging and Debugging Module (M10)

Secrets: The selection and configuration of logging libraries and debugging tools.

Services: Provides logging of system states, events, exceptions, and debugging information during execution. Enables easier tracing of errors and verification of software correctness and performance.

Implemented By: Python logging module

Type of Module: Library

7.3.3 Control Module (M11)

Secrets: The logic and sequencing of interactions among different modules, including the order of invocation for path planning, collision detection, inverse kinematics solving, and output verification.

Services: Coordinates and manages the workflow of the system by invoking appropriate modules in the correct order. Specifically, it obtains input parameters, triggers path planning (using A*), performs collision checks, computes inverse kinematics, verifies output correctness, and generates final results for plotting and logging.

Implemented By: 2D Robot Arm Path Planner

Type of Module: Abstract Object

8 Traceability Matrix

This section shows two traceability matrices: between the modules and the requirements and between the modules and the anticipated changes.

Requirement	Modules
R1	M2, M9
R2	M2, M3
R3	M5, M6, M7
R4	M4, M6, M7
R5	M3, M8, M10

Table 2: Trace Between Requirements and Modules

AC	Modules
AC1	M1
AC2	M2
AC3	M2, M9
AC4	M2, M9
AC5	M3
AC6	M4
AC7	M6, M5
AC8	M7
AC9	M5, M10
AC10	M5
AC11	M7
AC12	M8

Table 3: Trace Between Anticipated Changes and Modules

9 Use Hierarchy Between Modules

In this section, the uses hierarchy between modules is provided. Parnas (1978) said of two programs A and B that A *uses* B if correct execution of B may be necessary for A to complete the task described in its specification. That is, A *uses* B if there exist situations in which the correct functioning of A depends upon the availability of a correct implementation of B. Figure 1 illustrates the use relation between the modules. It can be seen that the graph is a directed acyclic graph (DAG). Each level of the hierarchy offers a testable and usable subset of the system, and modules in the higher level of the hierarchy are essentially simpler because they use modules from the lower levels.

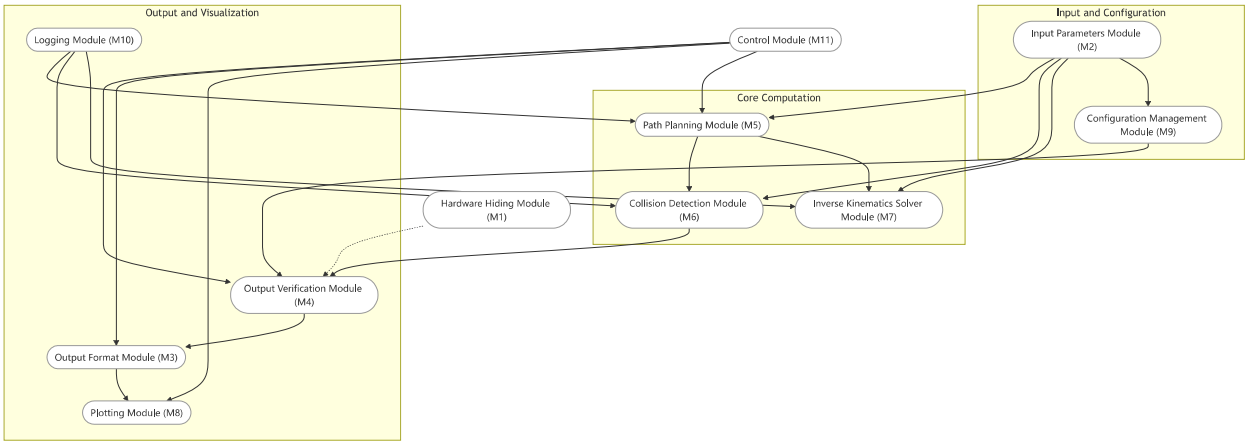

Figure 1: Use hierarchy among modules

10 User Interface

The user interface (UI) for the 2D robot arm path planning software is designed for ease of use, providing a clear workflow from data input to visualization of computed paths. Given the primary users are researchers, engineers, and developers, the interface emphasizes simplicity, clarity, and flexibility.

The main components of the user interface include:

- **Input Configuration Panel:** Allows users to define and modify input parameters such as robot arm lengths, initial and goal positions, obstacle coordinates, and other relevant constraints.
- **Visualization Window:** Displays the robot arm configuration, obstacles, and computed collision-free paths clearly in a graphical manner. Users can visually inspect and verify planned paths interactively.
- **Control and Parameter Adjustment Panel:** Provides interactive controls allowing users to adjust planning parameters, such as resolution of the planning grid, algorithm selection, and tolerances.
- **Logging and Debugging Output:** Offers real-time logs and debugging information to facilitate tracing the planning process and to quickly identify any issues.

References

David L. Parnas. On the criteria to be used in decomposing systems into modules. *Comm. ACM*, 15(2):1053–1058, December 1972.

David L. Parnas. Designing software for ease of extension and contraction. In *ICSE '78: Proceedings of the 3rd international conference on Software engineering*, pages 264–277, Piscataway, NJ, USA, 1978. IEEE Press. ISBN none.

D.L. Parnas, P.C. Clement, and D. M. Weiss. The modular structure of complex systems. In *International Conference on Software Engineering*, pages 408–419, 1984.