The Interpreter

Zeid Kootbally
Intelligent Systems Division
National Institute of Standards and Technology

Contents

List of Figures

1 Introduction

This section describes the process for the interpreter. The interpreter reads the plan file and generates the canonical robot commands. The description of this process and subprocesses are mainly described via flowcharts. The main process for the interpreter is depicted in Figure 1 and described in the following sections.

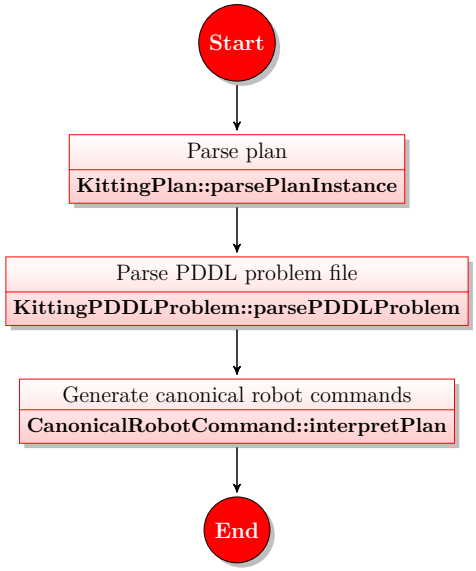

Figure 1: Main Process for the Interpreter.

2 Parse Plan

In this process, the plan file is parsed and each line is read and stored in different lists. The process for parsing the plan file is performed by **KittingPlan::parsePlanInstance** and is depicted in Figure 2. This figure and the following ones in this document display some functions in rectangle which are split in half. The upper part of these rectangles describes the action performed by the functions and the lower part displays the name of the functions in the `C++` source code.

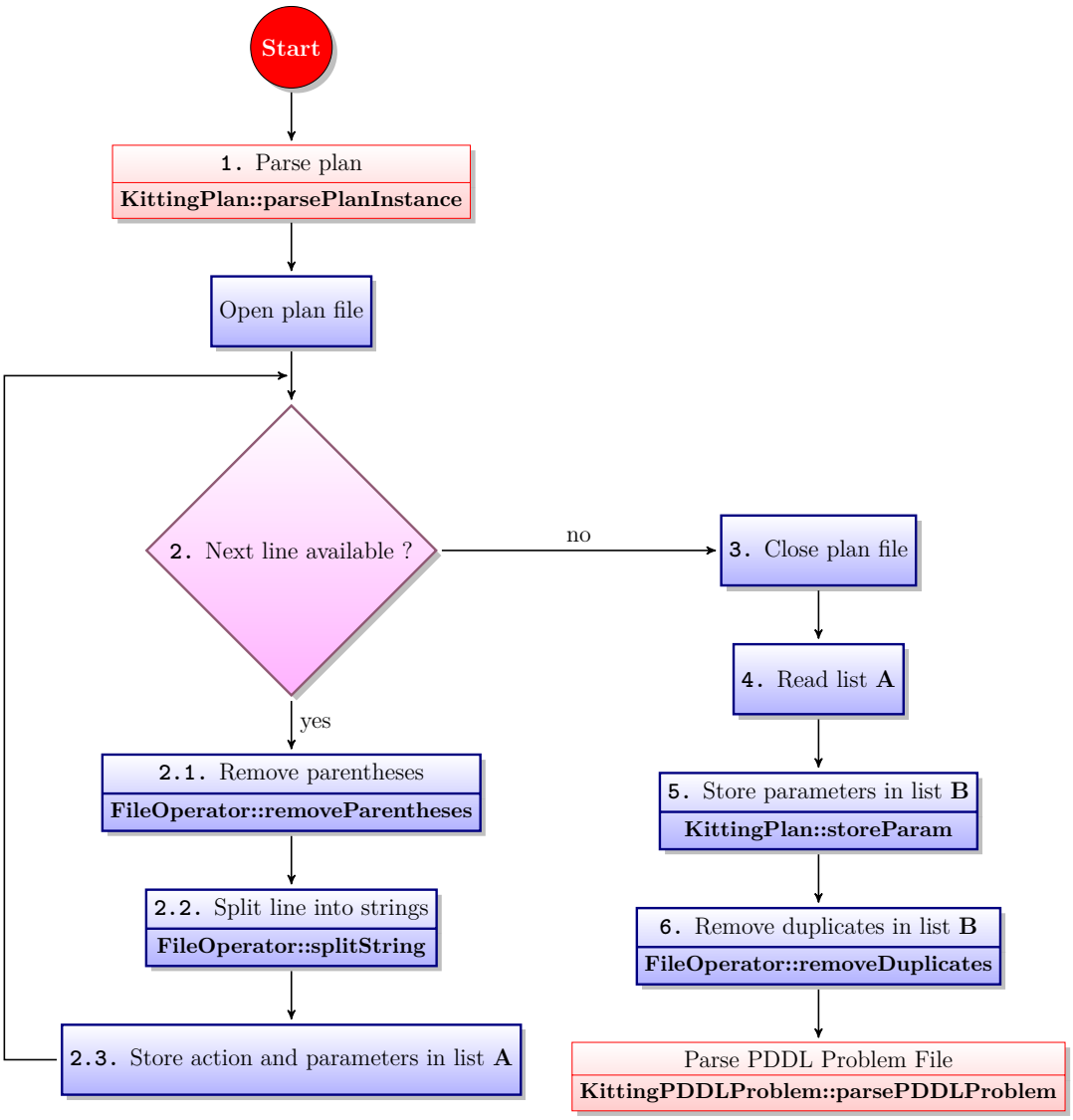

Figure 2: Process for parsing the plan file.

The different steps illustrated in Figure 2 are described below.

- Start: This element indicates the beginning of the execution of the whole program.

1. Open plan file: The location and the name of the plan file can be found in the *Config.h* file. The location of the plan file is given by `#define PLAN_FOLDER` and the name of the plan file is given by `#define PLAN_FILE`.

2. Next line available ?: This while loop reads each line of the plan file and does the following commands:

   ☐ If there is a next line in the plan file

   2.1. Remove parentheses: To describe this function and the following ones, we will use the following examples which describe two lines of the plan file:
   $(\mathsf{action_A}\ \mathsf{param_{P_1}}\ \mathsf{param_{P_2}})$
   $(\mathsf{action_B}\ \mathsf{param_{P_3}}\ \mathsf{param_{P_2}}\ \mathsf{param_{P_4}})$
   The result of this function for each line is:
   $\mathsf{action_A}\ \mathsf{param_{P_1}}\ \mathsf{param_{P_2}}$
   $\mathsf{action_B}\ \mathsf{param_{P_3}}\ \mathsf{param_{P_2}}\ \mathsf{param_{P_4}}$

   2.2. Split line into strings: Each line of the plan file is then split into separate strings.

   2.3. Store action and parameters in list **A**: Each line is stored in list **A** defined by **KittingPlan::m_actionParamList**. Using our example the result of this function is:
   list **A**: $<< \mathsf{action_A}, \mathsf{param_{P_1}}, \mathsf{param_{P_2}} >< \mathsf{action_B}, \mathsf{param_{P_3}}, \mathsf{param_{P_2}}, \mathsf{param_{P_4}} >>$

   ☐ If there is a next line in the plan file

   3. Close plan file

4. Read list **A**

5. Store parameters in list **B**: All parameters present in list **A** are stored in list **B**. List **B** is defined by (**KittingPlan::m_paramList**). The result of this function generates:
   $< \mathsf{param_{P_1}}, \mathsf{param_{P_2}}, \mathsf{param_{P_3}}, \mathsf{param_{P_2}}, \mathsf{param_{P_4}} >$

6. Remove duplicates in list **B**: Duplicates are removed from list **B**. In the example, $\mathsf{param_{P_2}}$ appears twice. The result returns the following list:
   list **B**: $< \mathsf{param_{P_1}}, \mathsf{param_{P_3}}, \mathsf{param_{P_2}}, \mathsf{param_{P_4}} >$

■ Parse PDDL Problem File: This process is used to parse the PDDL problem file in order to retrieve the type of each parameter in list **B**. Section 3 gives a deeper description of this process.

3 Parse the PDDL Problem File

This process parses the PDDL problem file and retrieves the type for each parameter stored in list **B**. This process is performed by **KittingPDDLProblem::parsePDDLProblem** and is depicted in Figure 3.

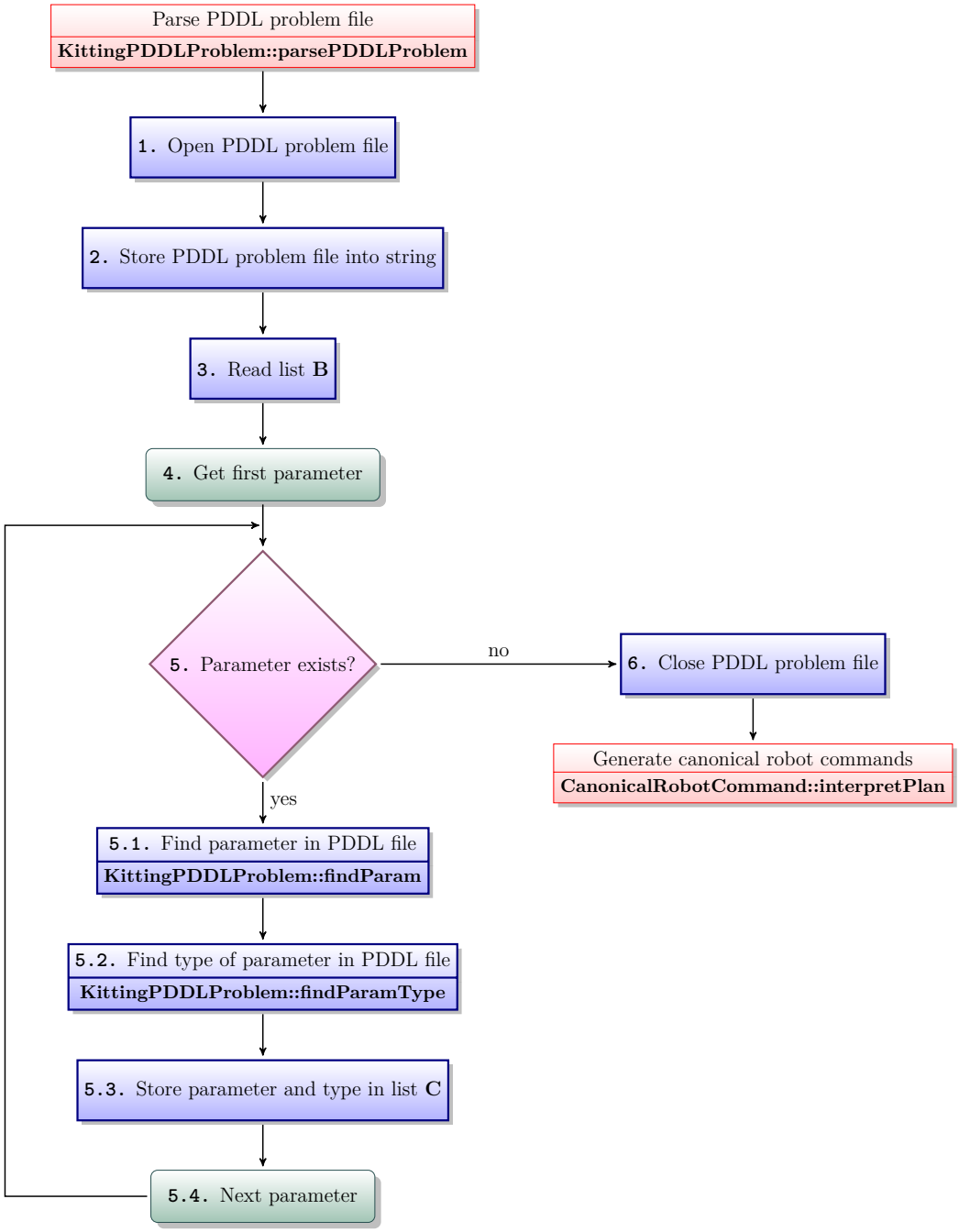

Figure 3: Process for parsing the PDDL problem file.

The different steps illustrated in Figure 3 are described below.

1. Open PDDL Problem file: The location and the name of the PDDL problem file can be found in the *Config.h* file. The location of the PDDL problem file is given by `#define PDDL_FOLDER` and the name of the problem file is given by `#define PDDL_PROBLEM`.

2. Store PDDL problem file into string: The entire PDDL problem file is read and stored in memory (string).

3. Read list **B**.

4. Get the first parameter in list **B**.

5. Parameter exists ?: If the parameter exists, do the following:

   5.1. Find parameter in PDDL problem file: This function retrieves the first occurrence of the parameter in the PDDL problem file. The first occurrence of the parameter appears in the : objects section of the problem file. An excerpt the : objects section for our example is given below:

   1.(: objects
   2. $\text{param}_{P_1} - \text{type}_A$
   3. $\text{param}_{P_2} - \text{type}_B$
   4. $\text{param}_{P_3} - \text{type}_C$
   5. $\text{param}_{P_4} - \text{type}_D$
   6.)

   This function returns a `C++ map<string,int>`, where the first element is the parameter and the second element is the line where the parameter was found in the problem file. According to our example, this function returns the following map:
   $<< \text{param}_{P_1}, 2 >< \text{param}_{P_3}, 4 >< \text{param}_{P_2}, 3 >< \text{param}_{P_4}, 5 >>$

   5.2. Find type of parameter in PDDL problem file: This function takes as input the `map` generated in the previous step and the PDDL problem file. For each line number in `map<string,int>`, the PDDL problem file is read again until the line number is reached. The last element of the line (type of the parameter) in the PDDL file is retrieved.

   5.3. Store parameter in list **C**: The parameter and its type are stored in list **C** which is a `C++ map<string,string>`. The first element of `map<string,string>` is the parameter and the second element is its type. list **C** is defined with **KittingPDDLProblem::m_ParamType**. In our example, the result of this function will be:
   $<< \text{param}_{P_1}, \text{type}_A >< \text{param}_{P_3}, \text{type}_C >< \text{param}_{P_2}, \text{type}_B >< \text{param}_{P_4}, \text{type}_D >>$

6. Close PDDL problem file.

■ Generate canonical robot commands: This process generates the canonical robot commands for each action found in the plan file. More information on this process can be found in section 4.

4 Generate Canonical Robot Commands

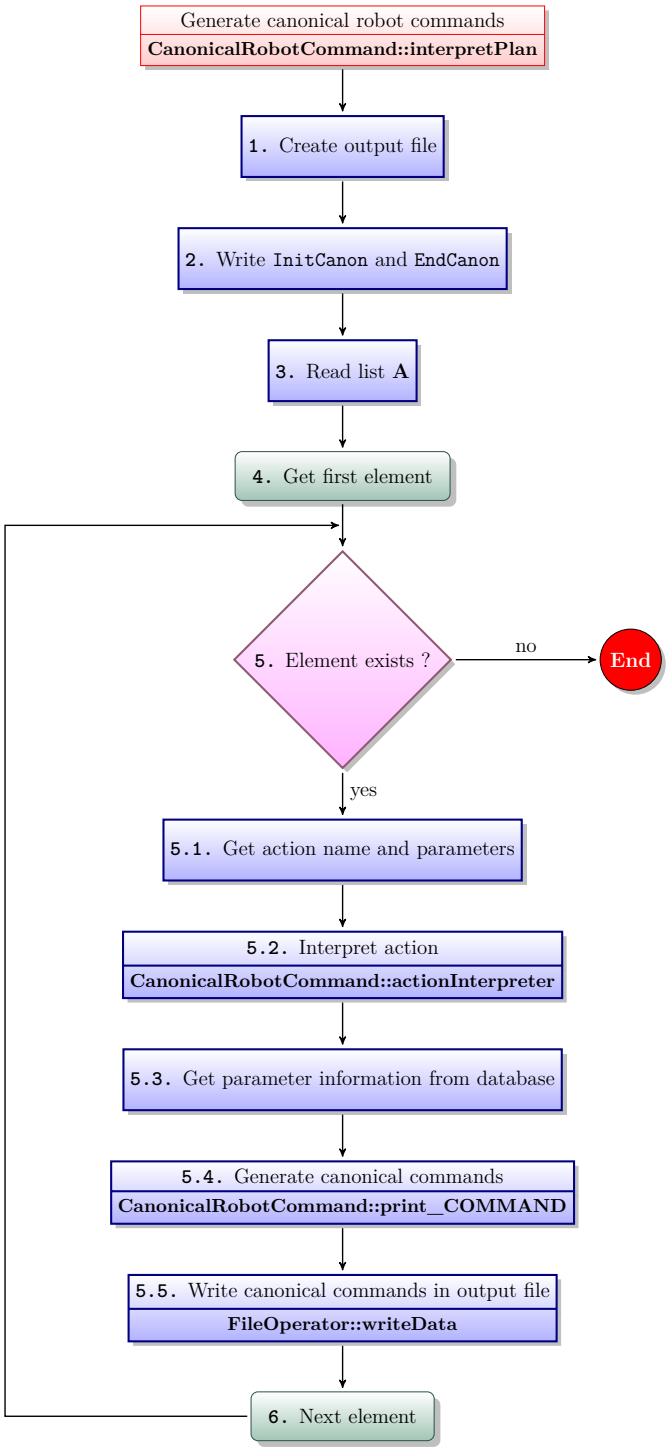

Figure 4: Process for generating canonical robot commands.

The different steps illustrated in Figure 4 are used to generate canonical robot commands

and are described below.

1. Create output file: This step creates the output file that will contain a set of canonical robot commands. This output file will be used by the controller to build kits. The output file creation is performed by **FileOperator::createOutputFile**. First, the name of the output file is retrieved by **FileOperator::getCanonFile**. The location of the output file is given by `#define ROBOT_COMMANDS_FOLDER` and the name of the output file is given by `#define ROBOT_COMMANDS_FILE`, both defined in *Config.h*. If the output file already exists, from a previous execution of the program, the old file is deleted and a new one is created. If the output file does not exist, it will be created.

2. Write `InitCanon` and `EndCanon`: Since all sets of canonical robot commands start with `InitCanon` and end with `EndCanon`, this step write these two robot commands in the output file created in step `1.`

3. Read list **A**: In this step, the function **CanonicalRobotCommand::interpretPlan** reads list **A**, created in the parse plan process (see section 2).

4. Get first element: The first element of list **A** is retrieved. Each element of this list includes the name of the PDDL action and the parameters used by this action.

5. Element exists ?: If the element (first or next) exists, do the following:

   5.1. Get action name and parameters: The name of the action and its related parameters are retrieved.

   5.2. Interpret action: In this step, the name of the action is used to call the corresponding `C++` function. The following table displays available PDDL actions and their corresponding `C++` functions.

PDDL Actions	*C++ Functions*
take-kit-tray	**CanonicalRobotCommand::take_kit_tray**
put-kit-tray	**CanonicalRobotCommand::put_kit_tray**
take-kit	**CanonicalRobotCommand::take_kit**
put-kit	**CanonicalRobotCommand::put_kit**
take-part	**CanonicalRobotCommand::take_part**
put-part	**CanonicalRobotCommand::put_part**
attach-eff	**CanonicalRobotCommand::attach_eff**
remove-eff	**CanonicalRobotCommand::remove_eff**
create-kit	**CanonicalRobotCommand::create_kit**

   The details for each function can be found in the `doxygen-interpreter.pdf` file.

   5.3. Get parameter information from database: Once one of the functions in step `5.2.` is called, a connection is made to the MySQL database to retrieve information on the parameters. The description of the `C++` functions used to access

and query the MySQL database is not in the scope of this paper. The user can have more information about this by looking at the files in the `src/database` directory.

5.4. Generate canonical commands: Canonical robot commands are generated by a set of `C++` **CanonicalRobotCommand::print_COMMAND** functions where **COMMAND** designs the name of the canonical robot command. The following table displays the canonical robot commands and their `C++` counterparts.

Canonical Robot Commands	*C++ Functions*
Dwell	**CanonicalRobotCommand::print_dwell**
InitCanon	**CanonicalRobotCommand::put_initcanon**
EndCanon	**CanonicalRobotCommand::print_endcanon**
CloseGripper	**CanonicalRobotCommand::print_closegripper**
OpenGripper	**CanonicalRobotCommand::print_opengripper**
MoveTo	**CanonicalRobotCommand::put_moveto**

We note that some PDDL actions cannot be executed since the canonical robot commands for these actions have not been implemented in the controller yet. To date, only the PDDL actions take-part and put-part can be interpreted in the canonical robot language. Below is an example of a set of canonical robot commands used for take-part and put-part.

take-part	put-part
`Message ("take part part_b_1")`	`Message ("put part part_b_1")`
`MoveTo(-0.03, 1.62, -0.25, 0, 0, 1, 1, 0, 0)`	`MoveTo(0.269, 0.584, -0.25, 0, 0, 1, 1, 0, 0)`
`Dwell (0.05)`	`Dwell (0.05)`
`MoveTo(-0.03, 1.62, 0.1325, 0, 0, 1, 1, 0, 0)`	`MoveTo(0.269, 0.584, 0.12, 0, 0, 1, 1, 0, 0)`
`CloseGripper ()`	`Dwell (0.05)`
`MoveTo(-0.03, 1.62, -0.25, 0, 0, 1, 1, 0, 0)`	`OpenGripper ()`
`Dwell (0.05)`	`MoveTo(0.269, 0.584, -0.25, 0, 0, 1, 1, 0, 0)`

5.5. Write canonical commands in output file: The **FileOperator::writeData** function opens the output file, write the canonical robot commands, and close the output file.

6. Next element: Go to the next element in list **A**.

■ End: The program ends when all the canonical robot commands have been written in the output file.