The executor: the interpreter and the controller

Zeid Kootbally
Intelligent Systems Division
National Institute of Standards and Technology

Contents

List of Figures

1 The Interpreter

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. Samples of `C++` code are used to capture the attention of the reader on important notes.
The main process for the interpreter is depicted in Figure 1 and described in the following subsections.

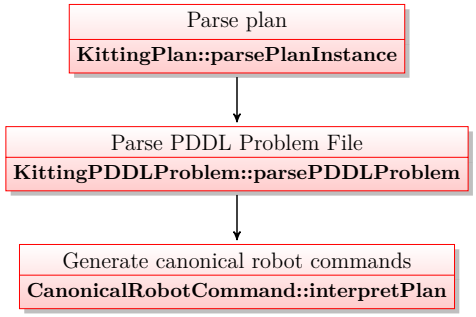

Figure 1: Main Process for the Interpreter.

1.1 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++` code source.

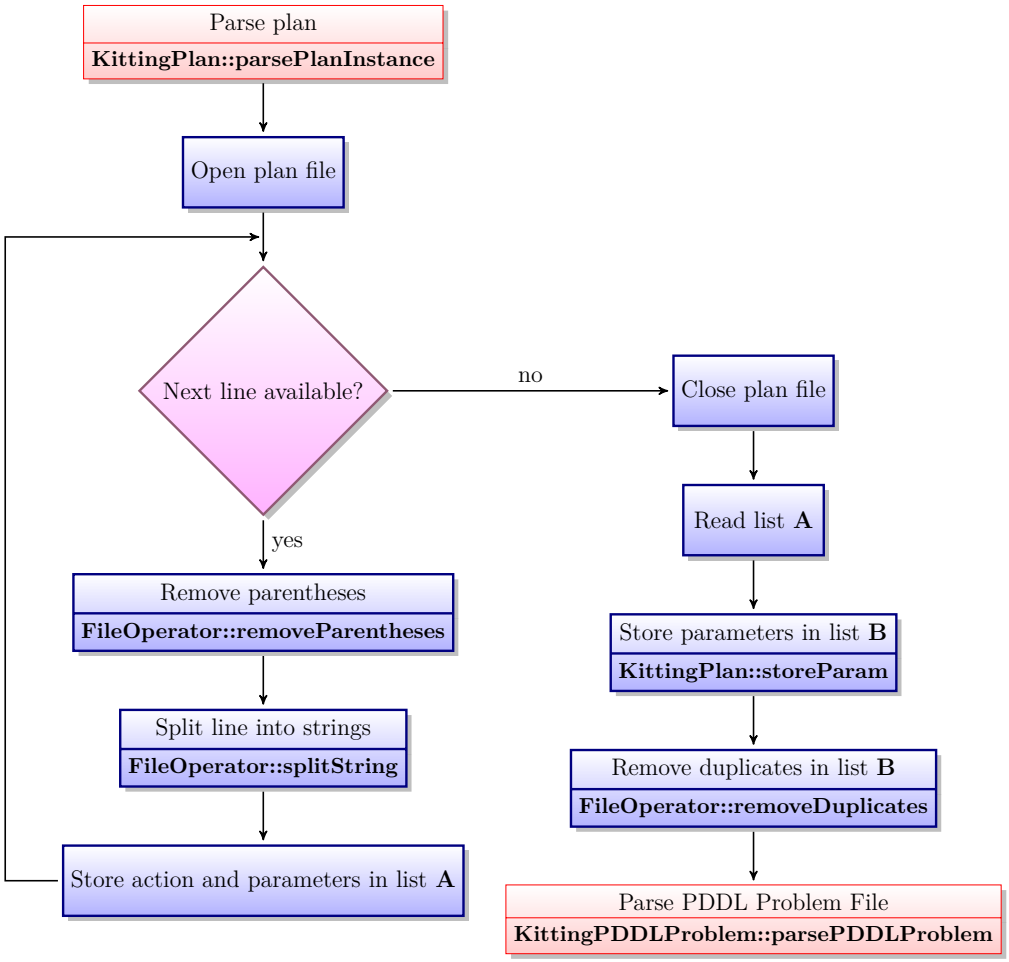

Figure 2: Process for parsing the plan file.

The different steps illustrated in Figure 2 are described below. This process for parsing the plan file is performed by **KittingPlan::parsePlanInstance** and is depicted in Figure 2.

- 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`.
- 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
  - ∗ 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:
    $\mathsf{action_A}$ $\mathsf{param_{P_1}}$ $\mathsf{param_{P_2}}$
    $\mathsf{action_B}$ $\mathsf{param_{P_3}}$ $\mathsf{param_{P_2}}$ $\mathsf{param_{P_4}}$
  - ∗ Split line into strings: Each line of the plan file is then split into separate strings.
  - ∗ Store action and parameters in list **A**(**KittingPlan::m_paramList**): Each line is stored in list **A**. 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
  - ∗ Close plan file

- ■ Read list **A**

- ■ Store parameters in list **B**: All parameters in list **A** are stored in list **B**. The result of this function generates:
  list **B**: <$\mathsf{param_{P_1}}$ $\mathsf{param_{P_2}}$ $\mathsf{param_{P_3}}$ $\mathsf{param_{P_2}}$ $\mathsf{param_{P_4}}$>

- ■ 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**. The next section gives a deeper description of this process.

1.2 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.

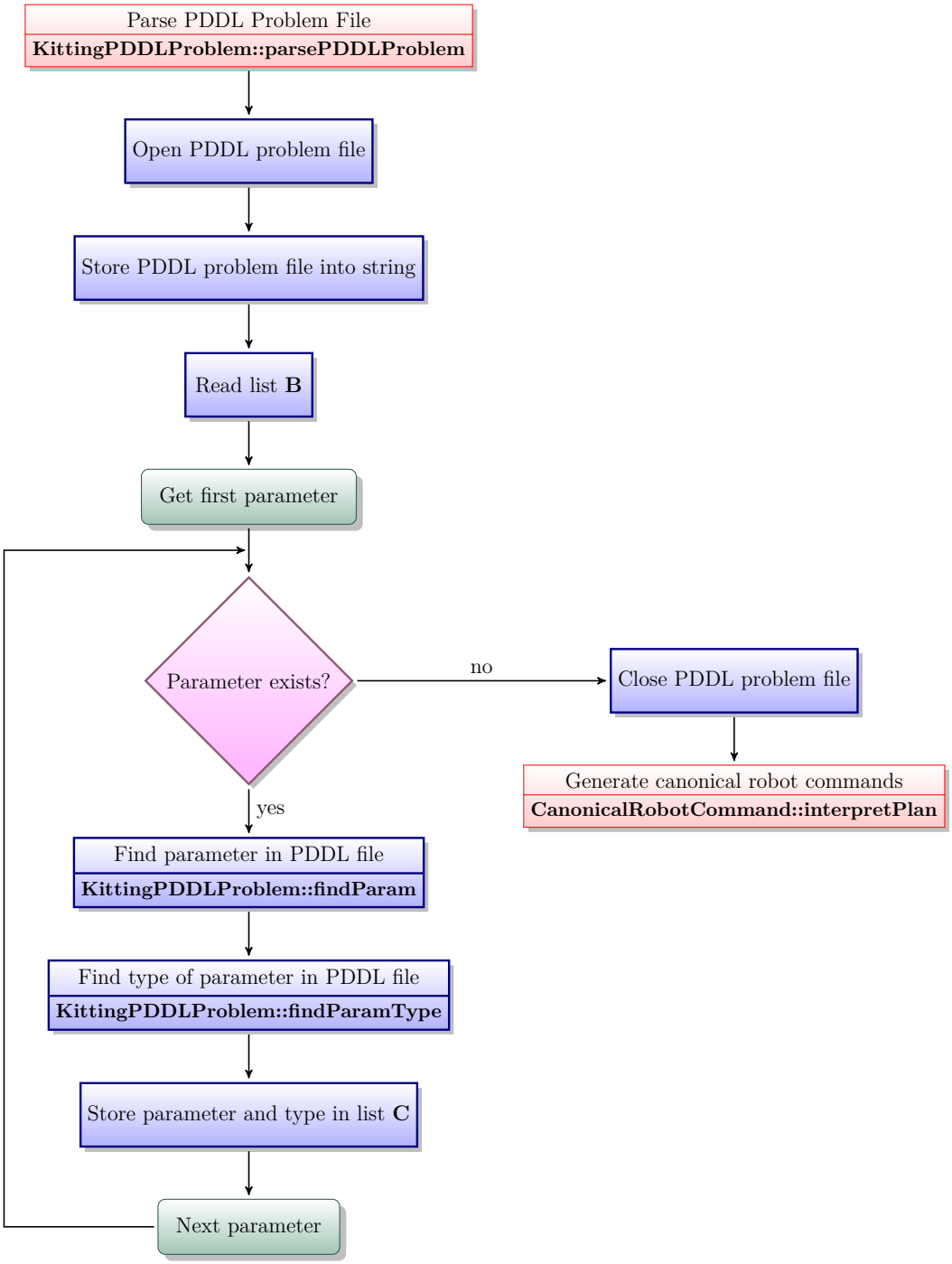

Figure 3: Process for parsing the PDDL problem file.

The different steps illustrated in Figure 3 are described below.

- 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`.
- Store PDDL problem file into string: The entire PDDL problem file is read and stored in memory (string).
- Read list **B**
- For each element in **B**:
  - ☐ Find parameter in PDDL problem file (**KittingPDDLProblem::findParam**): 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 example of the `:objects` section is given below:

```
(define (problem kitting-problem)
        (:domain kitting-domain)
        (:objects
                Object\_1 - Type\_A
                Object\_2 - Type\_B
                Object\_3 - Type\_C
                Object\_4 - Type\_D
        )
```

    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.
  - ☐ Find type of parameter in PDDL problem file:
  - ☐ Store parameter in list **C**:
- Close PDDL problem file.
- Generate canonical robot commands:

1.3 Generate Canonical Robot Commands

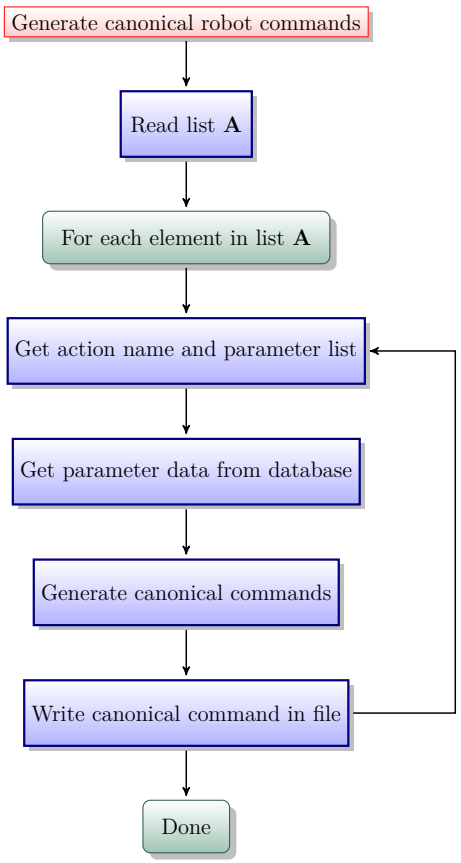

Figure 4: Process for generating canonical robot commands.

2 The Controller

The controller is designed to provide a queue for storing commands during the execution procedure. It provides classes for each of the canonical robot commands, a queuing and dequeuing mechanism, and virtual functions for processing each command. A sample controller is also provided. The files in the distribution are located in the controller directory and consist of:

- `canonicalMsg.hh` - The header file that provides classes for all of the canonical robot commands.
- `controller.cpp` - The base class that should be extended to create the controller.
- `controller.hh` - Include file for the base class.
- `Makefile` - The makefile

- `myController.cpp` - Sample controller that extends the controller class and creates routines to process all of the canonical robot commands.
- `myController.hh` - Include file for above.
- `test.cpp` - Sample threaded controller. The application uses the ulapi routines to create two threads. Thread 1 stuffs commands into the queue. Thread 2 reads them out and processes them.
- `ulapi.cpp` - Machine independent routines for items such as thread control and time.
- `ulapi.hh` - Include file for above.

To run the sample code, compile the application and library (type make), and then issue the command **./test**. You should see print outs that show commands being queued and executed.