The executor: the interpreter and the controller

Zeid Kootbally
Intelligent Systems Division
National Institute of Standards and Technology

Contents

List of Figures

1 Introduction

2 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.

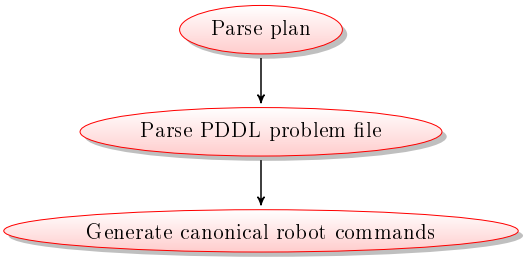

Figure 1: Main Process for the Interpreter.

2.1 Parse Plan

The plan is parsed with *KittingPlan::parsePlanInstance(const char* filename)* where *filename* represents the plan file. The different steps of this function can be described in Figure.

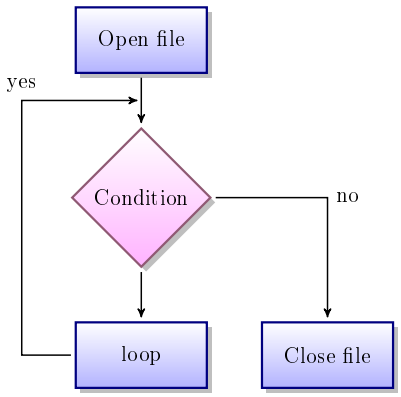

Figure 2: Process for KittingPlan::parsePlanInstance.

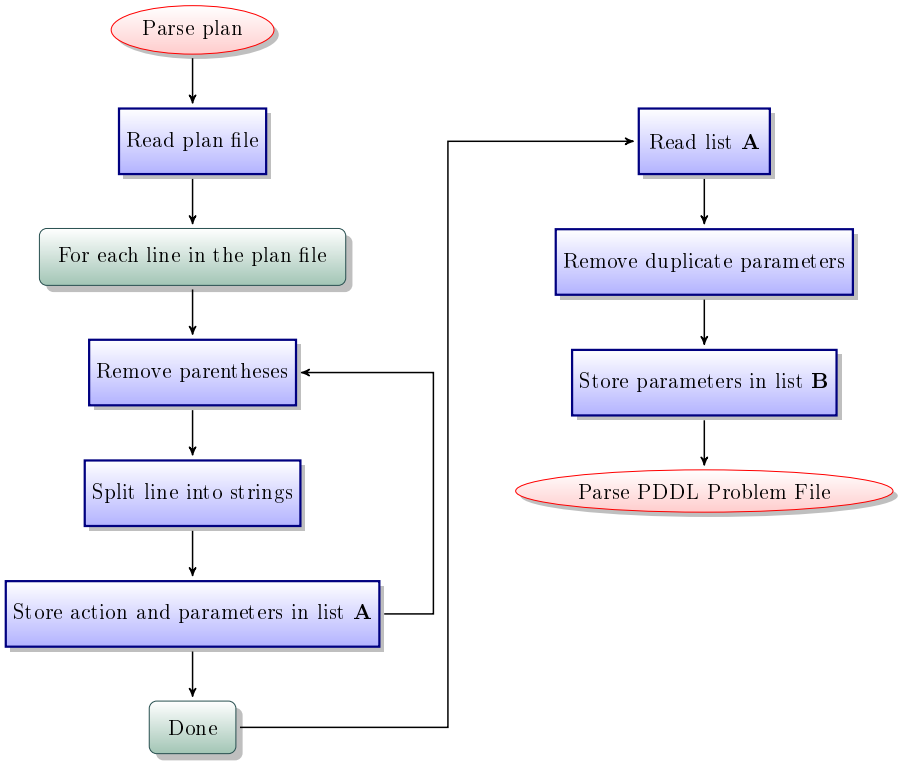

Figure 3: Process for parsing the plan.

2.2 Parse the PDDL Problem File

2.3 Generate Canonical Robot Commands

3 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.

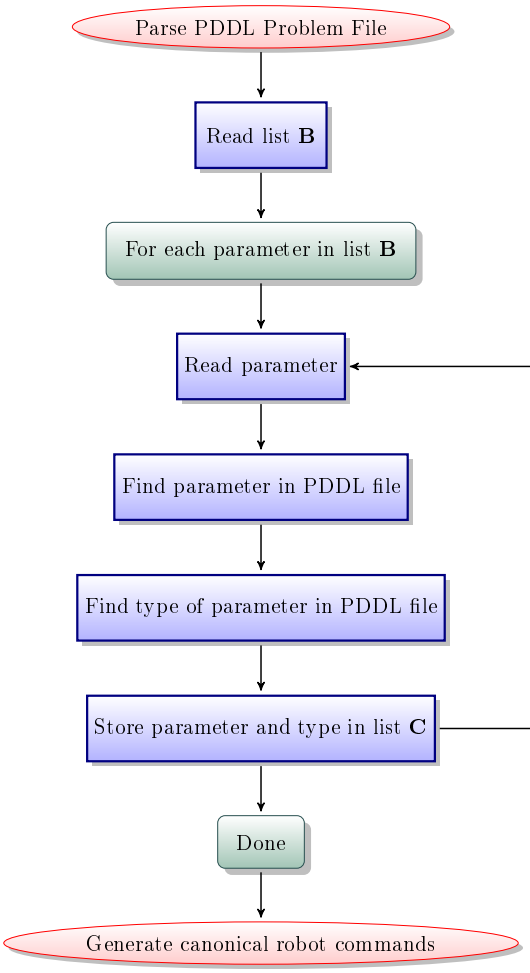

Figure 4: Process for parsing the PDDL problem file.

- `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.

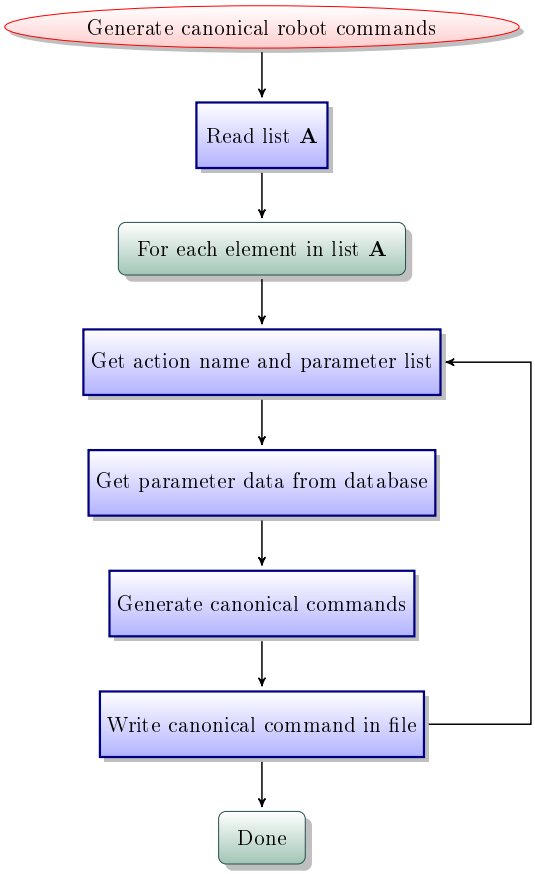

Figure 5: Process for generating canonical robot commands.