NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY

Intelligent Systems Division

Knowledge Driven Planning and Modeling for Part Handling

The Kitting Executor

Craig Schlenoff craig.schlenoff@nist.gov
Stephen Balakirsky stephen.balakirsky@nist.gov
Zeid Kootbally zeid.kootbally@nist.gov
Anthony Pietromartire pietromartire.anthony@nist.gov
Thomas Kramer thomas.kramer@nist.gov

Contents

List of Figures

1 The Executor

The `executor` has three main modules, the interpreter, the database, and the controller. Each module is described in this section

1.1 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. The main process for the interpreter is depicted in Figure 1 and described in the following sections.

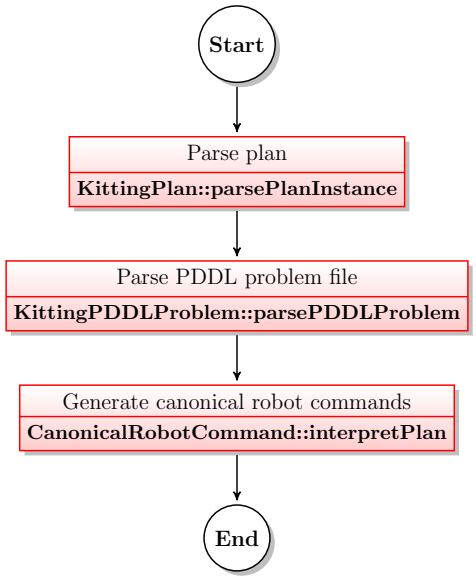

Figure 1: Main Process for the Interpreter.

1.1.1 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++` source code.

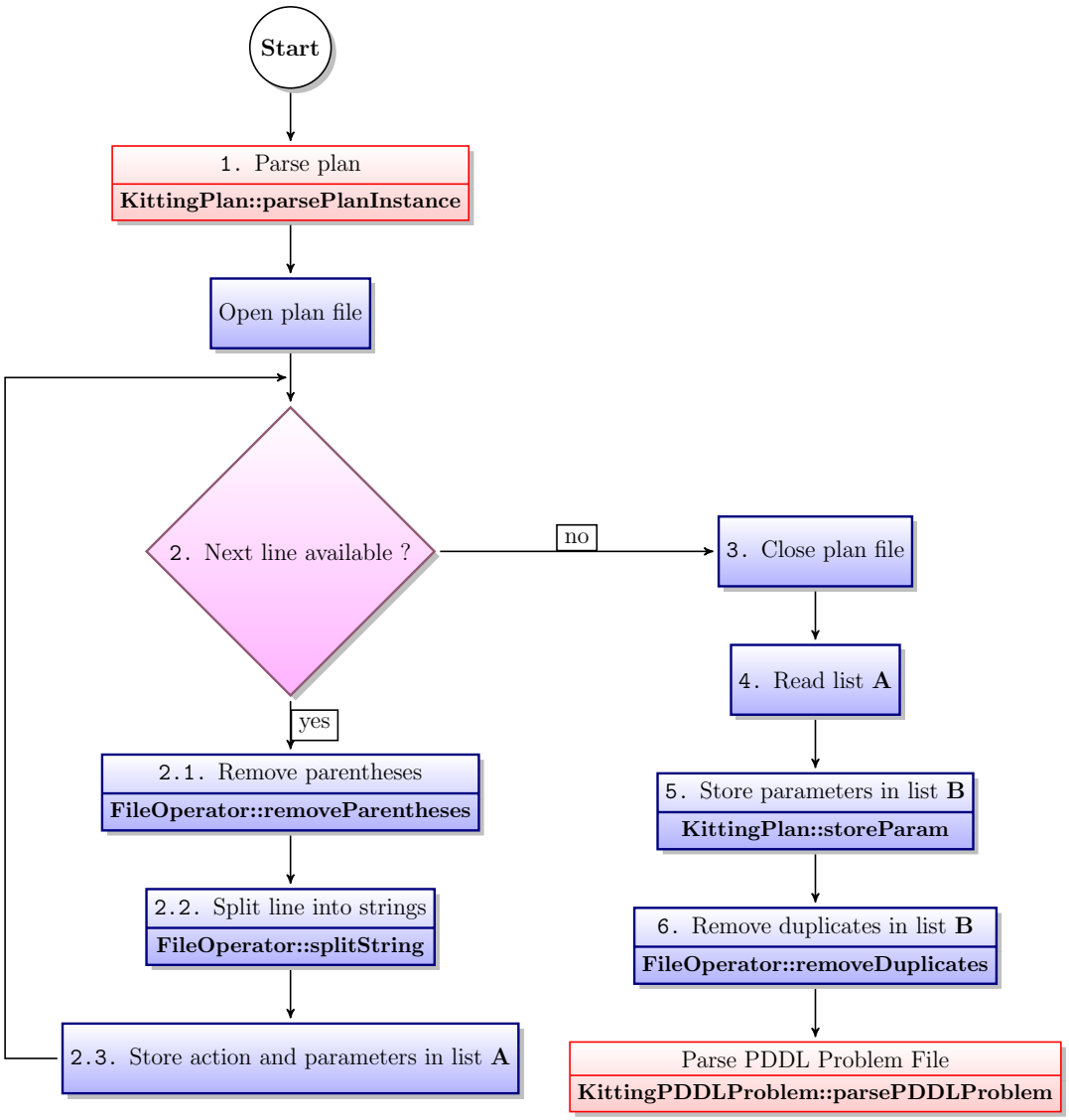

Figure 2: Process for parsing the plan file.

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 1.1.2 gives a deeper description of this process.

1.1.2 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.

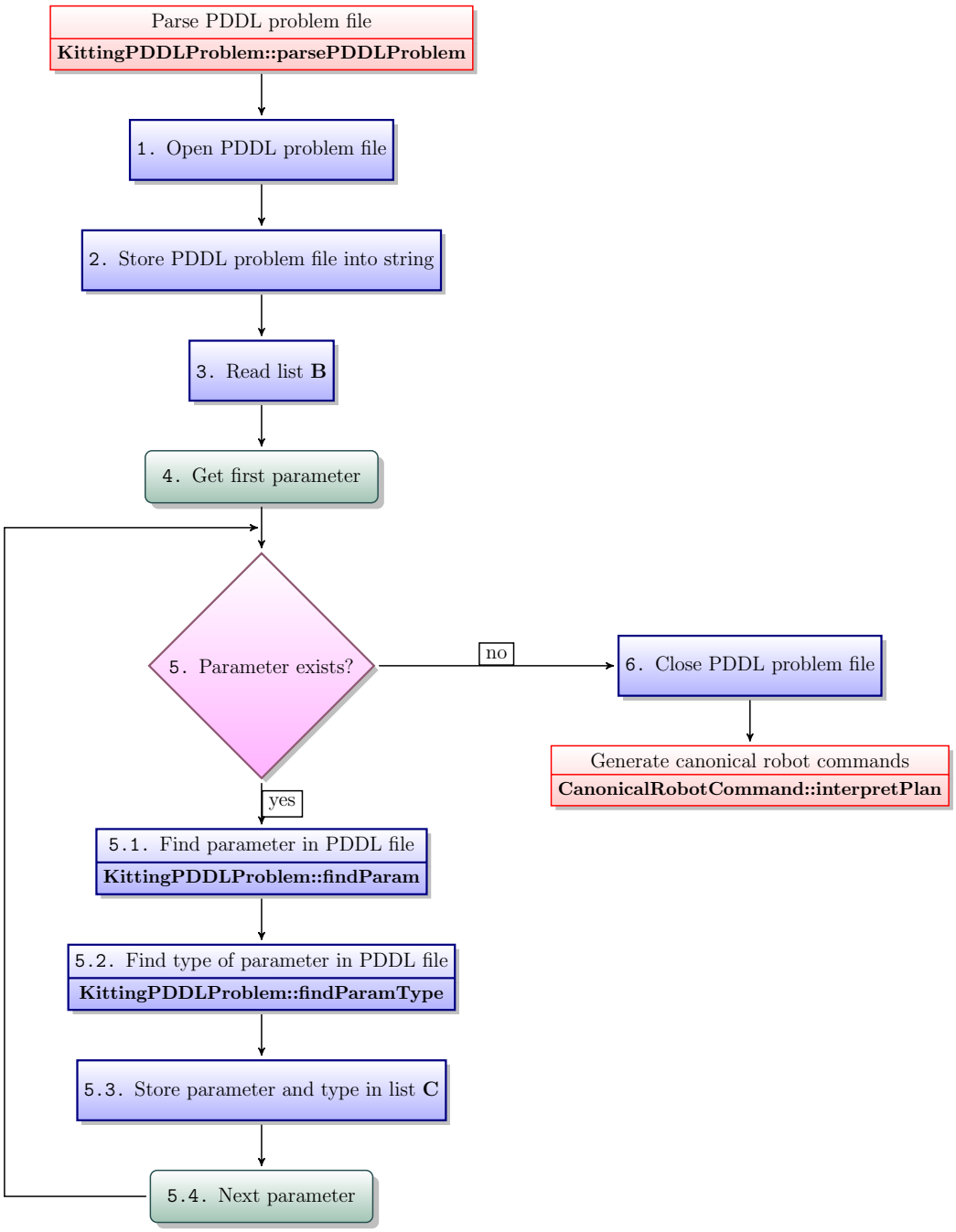

Figure 3: Process for parsing the PDDL problem file.

The different steps illustrated in Figure 3 are described below.

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. $\mathsf{param}_{\mathsf{P}_1} - \mathsf{type}_{\mathsf{A}}$
   3. $\mathsf{param}_{\mathsf{P}_2} - \mathsf{type}_{\mathsf{B}}$
   4. $\mathsf{param}_{\mathsf{P}_3} - \mathsf{type}_{\mathsf{C}}$
   5. $\mathsf{param}_{\mathsf{P}_4} - \mathsf{type}_{\mathsf{D}}$
   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:
   $<< \mathsf{param}_{\mathsf{P}_1}, 2 >< \mathsf{param}_{\mathsf{P}_3}, 4 >< \mathsf{param}_{\mathsf{P}_2}, 3 >< \mathsf{param}_{\mathsf{P}_4}, 5 >>$

   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:
   $<< \mathsf{param}_{\mathsf{P}_1}, \mathsf{type}_{\mathsf{A}} >< \mathsf{param}_{\mathsf{P}_3}, \mathsf{type}_{\mathsf{C}} >< \mathsf{param}_{\mathsf{P}_2}, \mathsf{type}_{\mathsf{B}} >< \mathsf{param}_{\mathsf{P}_4}, \mathsf{type}_{\mathsf{D}} >>$

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 1.1.3.

1.1.3 Generate Canonical Robot Commands

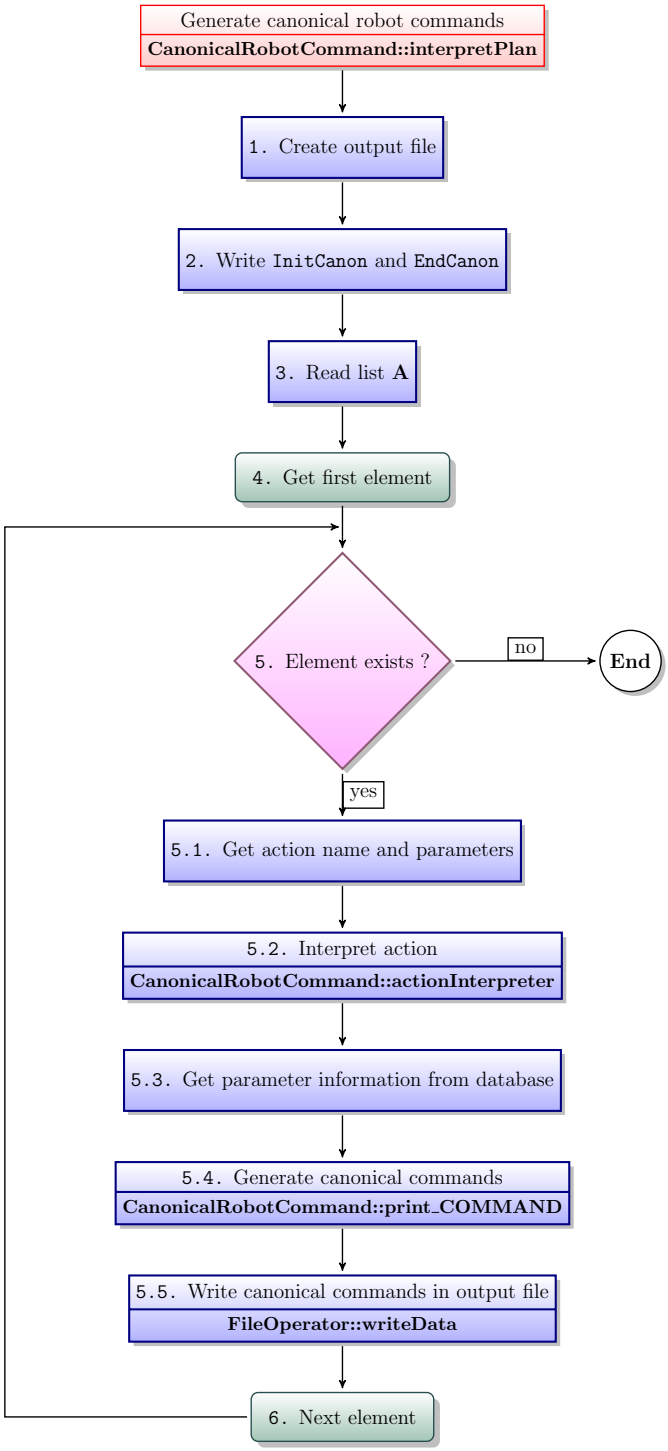

Figure 4: Process for generating canonical robot commands.

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 1.1.1).

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.

2 Build the Executor Project Structure with Autotools

This section provides a tutorial on how the user can build the structure of the executor project for distribution.

2.1 Generate the Default Structure

The execution of the script `autotools.sh` generates the structure for a simple "Hello World" project. To create the executor project using autotools files, the following instructions can be followed:

1. Create a directory and name it **executor**
2. Move the script `autotools.sh` in the directory **executor**
3. Run `autotools.sh` with `./ autotools.sh`

The structure of the new directory is depicted in Figure 5.

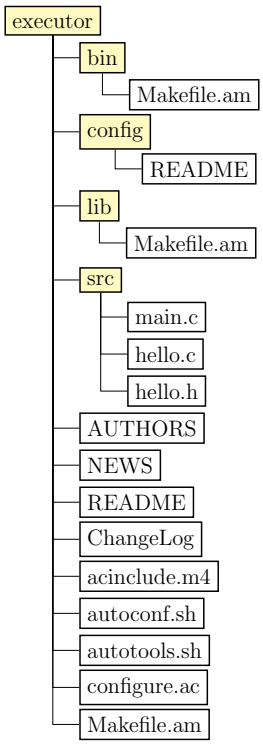

Figure 5: Default structure of the project executor.

2.2 Modify the Default Structure for the New Project

The default structure presented in section 2.1 needs to be modified as follows in order to include and use the source files for the executor.

- Delete the **lib** directory: We don't need this directory since the generated libraries will be stored in the same directory as the source files.
- Delete `main.c`, `hello.c`, and `hello.h` in the **src** directory: These files are required for a simple "Hello World" project and are not needed for our project.
- Add `Makefile.am` in the **bin** directory: This file will be used to generate the executable.
- In the `src` directory:
  - ☐ Add `Makefile.am`.
  - ☐ Add the **controller** directory with the source files.
  - ☐ Add the **database** directory with the source files.
    - ∗ Create `Makefile.am` in this directory.
  - ☐ Add the **interpreter** directory with the source files.
    - ∗ Create `Makefile.am` in this directory.
- Add the **etc** directory along with the files in it.
- Edit `configure.ac`
- Create and edit the script `bootstrap.sh`.

The previous steps generates the new structure of the executor project as depicted in Figure 6.

The rest of this section describes how to configure `configure.ac` and the different `Makefile.am` files.

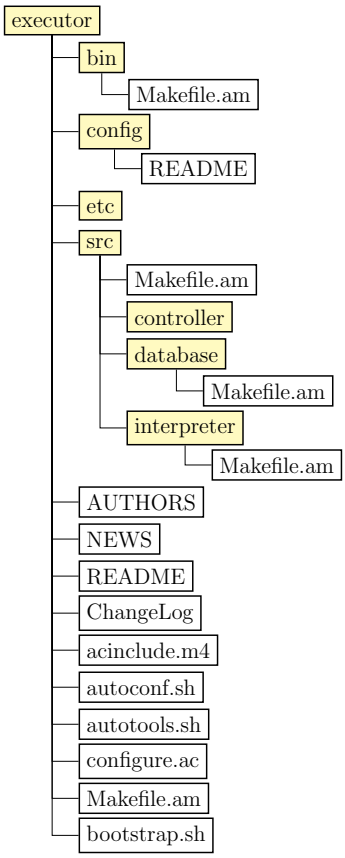

Figure 6: New structure of the project executor.

2.2.1 The `configure.ac` File

```
#Initial information about the project
#Our program will be called ‘‘executor’’, the version number is 1.0 and bug-reports should go to zeid.kootbally@nist.gov.
AC_INIT([executor], [1.0], [zeid.kootbally@nist.gov])

#We have to add a line AM_INIT_AUTOMAKE directly below AC_INIT.
#This initializes the automake process.
#AM_INIT_AUTOMAKE(executor, 1.0)

#This macro simply checks for the existence of a file in the source directory.
#This is only a safety check but should also be used every time.
AC_CONFIG_SRCDIR([src/main.cc])

#This macro call checks if a compatible C++ compiler is available.
AC_PROG_CXX
#It is also very helpful to check for an install program which will install the resulting binary, documentation or datafiles
#into its corresponding directories.
AC_PROG_INSTALL

#This is required if any libraries are built in the package.
AC_PROG_RANLIB

#Use the C++ compiler for the following checks
AC_LANG([C++])

#Distribute additional compiler and linker flags among Makefiles
# --> set and change these variables instead of CXXFLAGS or LDFLAGS (for user only)

#AC_OUTPUT should substitute every occurrence of @AM_CXXFLAGS@ in input files with the values of $AM_CXXFLAGS
AC_SUBST([AM_CXXFLAGS])
#AC_OUTPUT should substitute every occurrence of @AM_CPPFLAGS@ in input files with the values of $AM_CPPFLAGS
AC_SUBST([AM_CPPFLAGS])
#AC_OUTPUT should substitute every occurrence of @AM_LDFLAGS@ in input files with the values of $AM_LDFLAGS
AC_SUBST([AM_LDFLAGS])
#AC_OUTPUT should substitute every occurrence of @LIBS@ in input files with the values of $LIBS
AC_SUBST([LIBS])

#Files to generate via autotools (prepare .am or .in source files)
AC_CONFIG_FILES([Makefile])
AC_CONFIG_FILES([src/Makefile])
AC_CONFIG_FILES([src/interpreter/Makefile])
AC_CONFIG_FILES([src/database/Makefile])
AC_CONFIG_FILES([bin/Makefile])

#Automake will look for various helper scripts, such as install-sh, in the directory named in this macro invocation.
#If AC_CONFIG_AUX_DIR is not given, the scripts are looked in their standard locations.
AC_CONFIG_AUX_DIR([config])

#Automake initialization (mandatory) including a check for automake API version >= 1.9
AM_INIT_AUTOMAKE([1.11])

#Checks for header files.
AC_HEADER_STDC
AC_CHECK_HEADERS([string])
AC_CHECK_HEADERS([iostream])

#Automake will generate rules to rebuild these headers
AC_CONFIG_HEADERS([config/config.h])

#Checks for typedefs, structures, and compiler characteristics.
AC_TYPE_SIZE_T

#Checks the platform.
ACX_HOST

#Generates files that are required for building the packages.
AC_OUTPUT
```

2.2.2 executor: Makefile.am

```
#The top level Makefile.am must tell Automake which subdirectories are to be built
SUBDIRS = src \
            bin

# Directories needed to be distributed even though not covered in the automatic rules
EXTRA_DIST = etc
```

2.2.3 src: Makefile.am

```
#The current Makefile.am must tell Automake which subdirectories are to be built
SUBDIRS = database \
interpreter
```

2.2.4 database: Makefile.am

```
#Additional include pathes necessary to compile the C++ library
AM_CXXFLAGS = -I$(top_srcdir)/src @AM_CXXFLAGS@
AM_CPPFLAGS = -I$(top_srcdir)/src @AM_CPPFLAGS@

#Building a non-installing library
noinst_LIBRARIES = libdatabase.a

#Where to install the headers on the system
libdatabase_adir = $(top_srcdir)/src/database

libdatabase_a_HEADERS = BoxVolume.h BoxyObject.h Connection.h DAO.h \
DataThing.h EndEffector.h EndEffectorChangingStation.h EndEffectorHolder.h \
GripperEffector.h Kit.h KitDesign.h KitTray.h \
KittingWorkstation.h LargeBoxWithEmptyKitTrays.h LargeBoxWithKits.h LargeContainer.h \
Part.h PartRefAndPose.h PartsBin.h PartsTray.h \
PartsTrayWithParts.h PhysicalLocation.h Point.h PoseLocation.h \
PoseLocationIn.h PoseLocationOn.h PoseOnlyLocation.h RelativeLocation.h \
RelativeLocationIn.h RelativeLocationOn.h Robot.h ShapeDesign.h \
SolidObject.h StockKeepingUnit.h VacuumEffector.h VacuumEffectorMultiCup.h \
VacuumEffectorSingleCup.h Vector.h WorkTable.h

libdatabase_a_SOURCES = $(libdatabase_a_HEADERS)  BoxVolume.cpp BoxyObject.cpp Connection.cpp DAO.cpp \
DataThing.cpp EndEffector.cpp EndEffectorChangingStation.cpp EndEffectorHolder.cpp \
GripperEffector.cpp Kit.cpp KitDesign.cpp KitTray.cpp \
KittingWorkstation.cpp LargeBoxWithEmptyKitTrays.cpp LargeBoxWithKits.cpp LargeContainer.cpp \
Part.cpp PartRefAndPose.cpp PartsBin.cpp PartsTray.cpp \
PartsTrayWithParts.cpp PhysicalLocation.cpp Point.cpp PoseLocation.cpp \
PoseLocationIn.cpp PoseLocationOn.cpp PoseOnlyLocation.cpp RelativeLocation.cpp \
RelativeLocationIn.cpp RelativeLocationOn.cpp Robot.cpp ShapeDesign.cpp \
SolidObject.cpp StockKeepingUnit.cpp VacuumEffector.cpp VacuumEffectorMultiCup.cpp \
VacuumEffectorSingleCup.cpp Vector.cpp WorkTable.cpp
```

2.2.5 interpreter: `Makefile.am`

```
#Definition of flags during compilation
#New flags need to be prepend to the other flags, not to append
#-I refers to extra directories needed to build the library
AM_CXXFLAGS = -I$(top_srcdir)/src @AM_CXXFLAGS@

#Building a non-installing library
noinst_LIBRARIES = libinterpreter.a

#Where to install the headers on the system
libinterpreter_adir = $(top_srcdir)/src/interpreter

#The list of header files that belong to the library
libinterpreter_a_HEADERS = IniFile.h CanonicalRobotCommand.h KittingPDDLProblem.h KittingPlan.h Operator.h

#The list of source files that belong to the library
libinterpreter_a_SOURCES = $(libinterpreter_a_HEADERS) \
        IniFile.cc CanonicalRobotCommand.cc KittingPDDLProblem.cc KittingPlan.cc Operator.c
```

2.2.6 bin: `Makefile.am`

```
#Definition of flags during compilation
#New flags need to be prepend to the other flags, not to append
#-I refers to directories needed to build the executor
AM_CXXFLAGS = -O3  -I$(top_srcdir)/src/ -I$(top_srcdir)/src/database -I$(top_srcdir)/src/interpreter @AM_CXXFLAGS@

#The executable file
bin_PROGRAMS = executor

#Main file needed to build the executable file
executor_SOURCES = $(top_srcdir)/src/main.cc

#Libraries needed to build the executable
#The most specific libraries are called before the most common ones
executor_LDADD = -L$(top_srcdir)/src/database -L$(top_srcdir)/src/interpreter  -linterpreter -ldatabase -lmysqlcppconn
```

2.2.7 The `bootstrap.sh` Script

The `bootstrap.sh` file contains all the tools needed to generate extra files and to build the project. More information on the roles of each tool is given in section 3.

```
#!/bin/sh -x

aclocal && \
autoheader && \
automake --gnu --add-missing --copy && \
autoconf && exit 0

exit 1
```

3 Components for Autotools

This section describes all the components required by autotools to build and distribute a project.

3.1 *aclocal*

The *aclocal* program creates the file `aclocal.m4` by combining stock installed macros, user defined macros and the contents of `acinclude.m4` to define all of the macros required by `configure.ac` in a single file. *aclocal* was created as a fix for some missing functionality in *Autoconf*, and as such we consider it a wart. In due course *aclocal* itself will disappear, and *Autoconf* will perform the same function unaided.

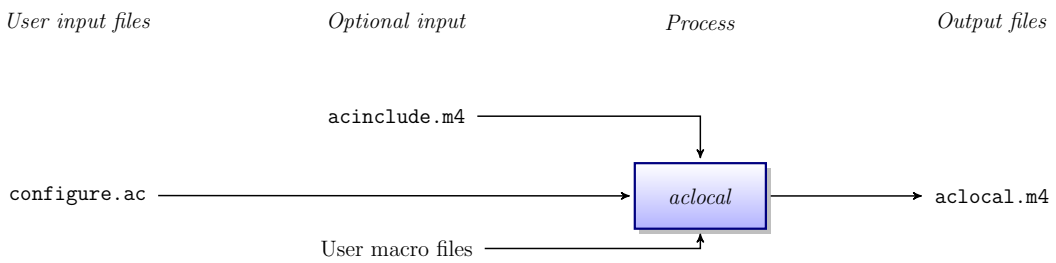

Figure 7: *aclocal* process.

3.2 *autoheader*

autoheader runs m4 over `configure.ac`, but with key macros defined differently than when *Autoconf* is executed, such that suitable cpp and cc definitions are output to `config.h.in`.

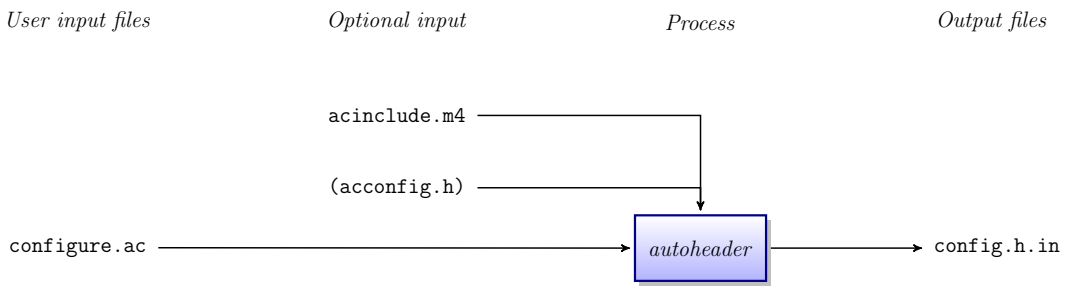

Figure 8: *autoheader* process.

3.3 *automake* and *libtoolize*

automake will call *libtoolize* to generate some extra files if the macro `AC_PROG_LIBTOOL` is used in `configure.ac`. If it is not present then *automake* will install `config.guess` and `config.sub` by itself.

libtoolize can also be run manually if desired; *automake* will only run *libtoolize* automatically if `ltmain.sh` and `ltconfig` are missing.

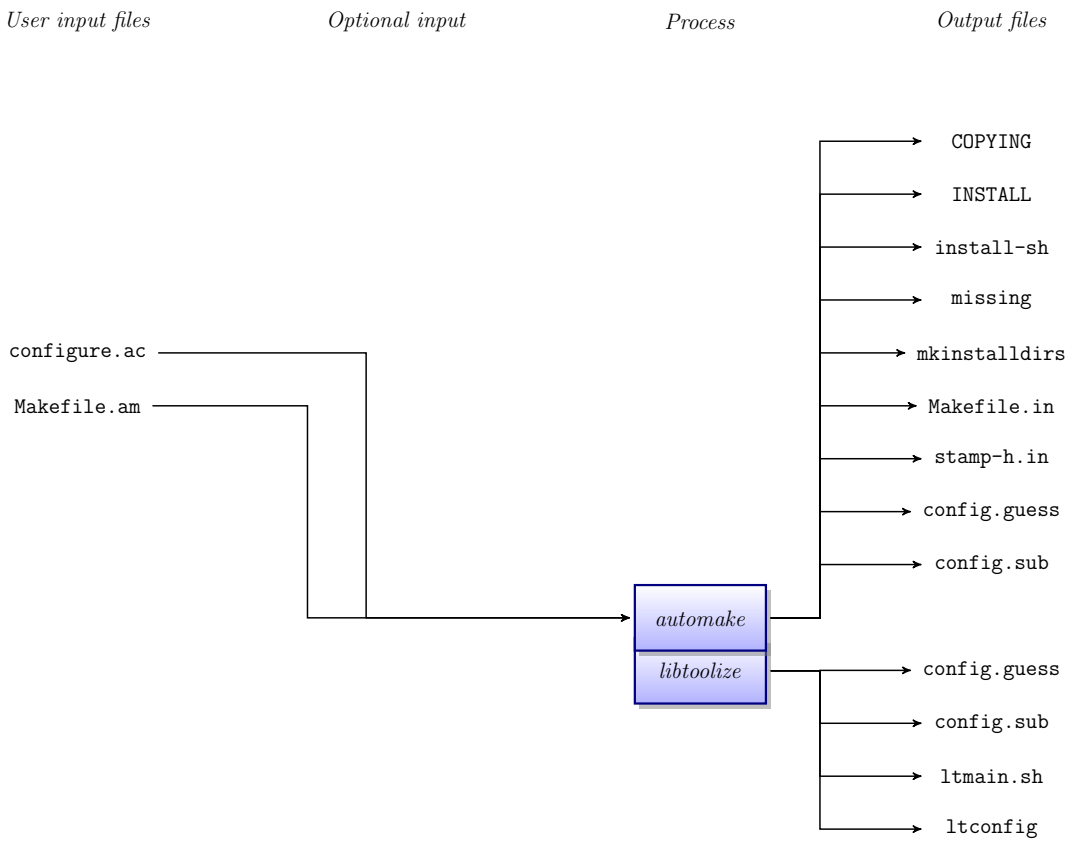

Figure 9: *automake* and *libtoolize* processes.

The versions of `config.guess` and `config.sub` installed differ between releases of *Automake* and *Libtool*, and might be different depending on whether *libtoolize* is used to install them or not. Before releasing your own package you should get the latest versions of these files from `ftp://ftp.gnu.org/gnu/config`, in case there have been changes since releases of the GNU *Autotools*.

3.4 *autoconf*

autoconf expands the m4 macros in `configure.ac`, perhaps using macro definitions from `aclocal.m4`, to generate the configure script.

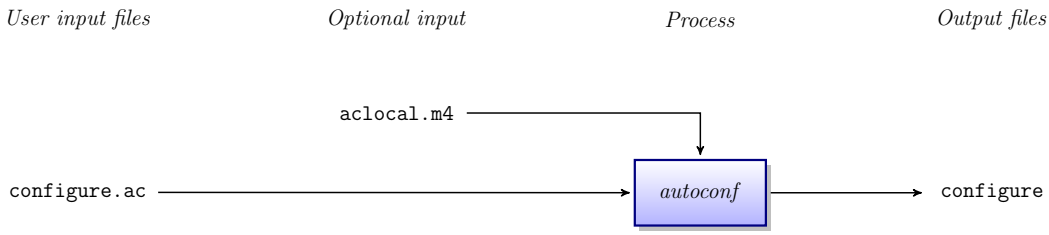

Figure 10: *autoconf* process.

3.5 *configure*

The purpose of the preceding processes was to create the input files necessary for configure to run correctly. You would ship your project with the generated script and the files in columns, other input and processes (except `config.cache`), but configure is designed to be run by the person installing your package. Naturally, you will run it too while you develop your project, but the files it produces are specific to your development machine, and are not shipped with your package – the person installing it later will run configure and generate output files specific to their own machine.

Running the configure script on the build host executes the various tests originally specified by the `configure.ac` file, and then creates another script, `config.status`. This new script generates the `config.h` header file from `config.h.in`, and `Makefile` from the named `Makefile.in`. Once `config.status` has been created, it can be executed by itself to regenerate files without rerunning all the tests. Additionally, if `AC_PROG_LIBTOOL` was used, then *ltconfig* is used to generate a `libtool` script.

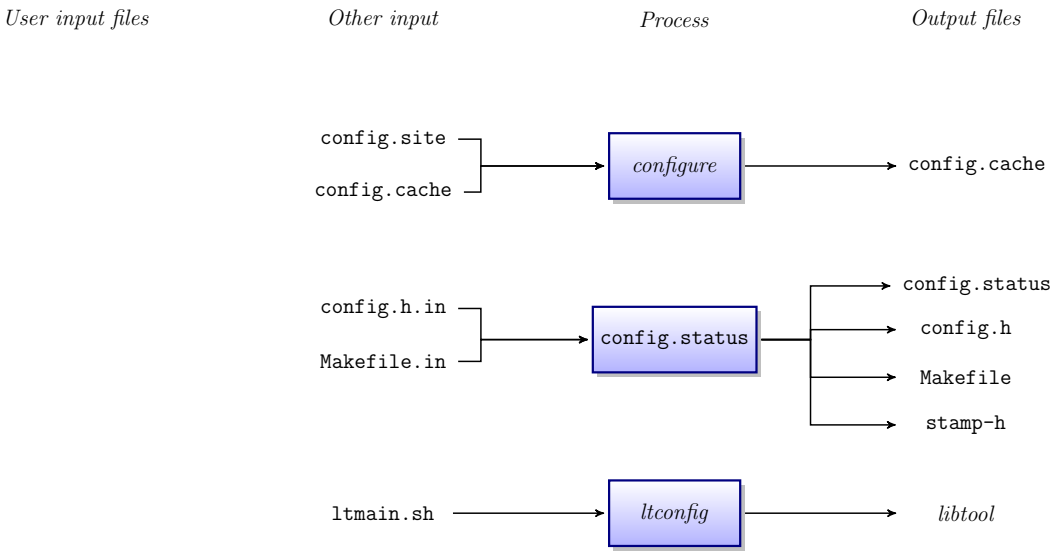

Figure 11: *configure* process.

4 Compile and Run the Executor

Once the executor project is built using the tutorial in section 2 one can build the project as follows.

- Using a Linux shell, place yourself in the `executor` directory.
- `./configure`
- `make`

The executable file `executor` is generated in the `bin` directory. To launch the executor:

- Using a Linux shell, place yourself in the `bin` directory.
- `./executor`

4.1 Configuration File

The input and output files for the executor are currently hard-coded. These filenames are set in the file `Config.h` and may be changed by the user (with a recompile required) to change the input and output behavior of the system.