User Interfaces Development in openDIEL

Argens Ng
August 5, 2016

Abstract
This paper serves the purpose of reporting the progress of user interfaces enhancement in the work flow engine openDIEL. While openDIEL has the potential to become a powerful work flow engine, its interface towards users has been staggering in development which magnifies the urge of this project. In this paper, I will describe the progress of a module making python script known as modMaker, as well as the development tool timer.

1 Introduction
OpenDIEL stands for open Distributive Interoperable Executive Library. It is a lightweight software framework which aims at combining different interoperable computational components to simulate system-wide scientific application. It uses Message Passing Interface (MPI) to facilitate the cooperation between loosely coupled modules and outputs a single executable.

To use openDIEL, user needs to provide:
1. Modules
2. Configuration File (using libconfig)
3. Driver (driver.c)

![Figure 1: The structure of openDIEL.](image)

Our ultimate goal is of course to automate the generation of all of the above three files, or sets of files. In this project, we will focus on the generation of the module first. This will be done by a python script called modMaker.
2 ModMaker

2.1 What is modMaker?
At the current stage, modMaker is a package of 2 python scripts – modMaker.py and worker.py. Together they can transform a C-file or a directory of C-files (and other files from C family) into a module (or modules). This is done by a series of pattern matching of strings as well as the addition of static supporting files.

2.2 What is a module?
For user defined code or simulation models to run in openDIEL, it has to be in the format of an openDIEL module. There are a few requirements. For example, it has to be rid of MPI_COMM_WORLD, MPI_Init and MPI_Finalize to facilitate the cooperation between different modules. This complicated formatting would be done by modMaker in the following manner.

![Diagram of modules](image)

Figure 2: Illustration of modules.

2.3 How to transform a module?
To transform a module, we have to first understand the syntactic structure of the language. As the C-family is our first target, we would naturally focus on it first.

I started off by focusing on the target "int main". As "int main" is now located in driver.c, we would need to replace the main program of individual modules with a function header. To do that, we have to accurately identify "int main" and modify it. (To be more precise, we would also need to consider "void main" but we would focus on "int main" only for the purpose of this paper)

Thinking as a human, I quickly realize that as long as "int" and "main" are two separate and individual strings, it would be unique and correct as the target we would like to change. This actually holds true for all other targets that we wish to change and hence the problem becomes "identifying individual strings correctly separated". From this insight, the following flow diagram is constructed.
2.3.1 Locating identifying feature

Two approaches were considered for locating identifying feature. The first one is "matching by character" and the other is "matching by string". To illustrate the difference, take a look at Fig.4 below.

![Matching-By-Character vs Matching-By-String](image)

The upper diagram illustrates the process of matching-by-character. By reading in and comparing at each character, we can determine if it is a match to any of the patterns that we are trying to match. So since the first character is a ‘/’, which is neither a match to pattern 1 nor pattern 2, we will continue to match the next character with the heads of both string (we will not proceed within the patterns).

The bottom diagram illustrates the process of matching-by-string. By reading in a line at a time, we can use python function string.find(string) to see if the line contains the target patterns, looping by pattern. Again, we found out that there is no match for both patterns at both position 1 and 2 in 4 comparisons.

While the benefit is not significant in this case. Matching-by-character can lead to a performance boost if the patterns have similar sub-string head to start with. For example, when matching pattern "MPI_Init" and "MPI_Finalize", we can determine that both patterns have first 4 characters matched or not with 4 comparisons instead of 8, after understanding that they starts with the same 4
However, it was soon realized that this require extra effort in identifying common sub-string start. As long as the built-in string.find(string) can terminate comparisons prematurely upon finding unmatched characters, the performance boost would be insignificant, especially so when compared with the time waiting user input. Hence matching-by-string was used.

2.3.2 Finding Replacement Candidate

First we need to understand the difference between "locating identifying feature" and "finding replacement candidate". Identifying feature refers to pattern such as "int[space]main" and "int[tab]main" (Notice that they are no different from the compiler’s perspective). However, we would like to replace much more than identifying feature itself. For example, we might want to replace "int main(int argc, char** argv)" or simply "int main ()". This is when syntactic freedom of C proves to become a barrier.

Luckily, C is a language depending heavily on separators, in contrast to Python, which is a indentation based language, and Fortran, which has its own set of strict formatting rules, C can have the whole code in one line and minimal separation. Its extensive use of special characters is both a threat and opportunity for our module transformation. In this case, it is the solution to the above problem.

Figure 5: Finding replacement candidate

It soon became apparent that the closing parenthesis ")" marks the end of our replacement candidate. To tackle this problem more systematically, we divide the statements that we need to convert in C into 3 categories.

1. Function Title
2. Statement
3. Variable

Function title refers to cases like "int main". They are likely ended with a closing parenthesis and then followed by a open bracket ". Statement such as "MPI_Init()" can be function call or assignment of variables. Luckily in C, they are usually ended with a semi-colon ";". Lastly, variables are usually not enclosed in separators and they are identifiable by themselves alone. Examples are "MPI_COMM_WORLD".

By locating the separator in front of an "Identifying Feature" and the one after, we can now locate with high accuracy the "Replacement Candidate" for our modules. After checking the spaces within and making sure each token is in fact a word by itself, we can pass on the results for the user to verify.
2.3.3 User Participation

Notice that we have been using the word ”Replacement Candidate”. This is because we believe that there might be missing factors even after serious investigation, and we do not wish to alter the program without user consent. This can cause serious problem that is hard to debug.

Hence all the replacement that we would like to change would appear on screen in a highlighted manner, with replacement suggestion listed out to see if the user see fit.

This soon raises another problem, which is the huge number of prompts generated. While the modMaker script was created with test cases of 2 files and around 40 lines in total, it was soon discovered that hundreds of files and tens of thousands of codes are common in real life scenario. Hence 3 tactics were employed to combat this problem.

The first one is the combination of similar prompts. In our case, MPI_COMM_WORLD is the most common token and hence generates the most prompts. Hence we decided to count, combine and ask for confirmation all together as illustrated in Fig.8.

The second tactic is to use extension matching to prevent going into un-
wanted files. This takes in multiple inputs and hence for example, ".c" and ".cpp" could both be accepted in one transformation.

This last tactic was to screen out files without any replacement candidate. While this seems obvious in later stages, it was not thought of in earlier stages and a lot of times was wasted for confirming file searches with user input.

As seen in the last figure, a large proportion of files did not require the modMaker to go through nor user to confirm. They are hence simply skipped. A large sum of time is saved after these 3 tactics were implemented.

2.3.4 Testing

Testing is a huge part in modMaker, simply because changing a program at source code level is dangerous and risky. Hence we provide sufficient testing tools for the user to make sure their module runs normally.

After changing a program into a module. It compiles into a library and acts like a function. Hence, a "tester.c" is provided for a main program entry point for the new module. The user can then compile it as a program again and see if it runs correctly.

An artificial "IEL.h" was also provided for the user to test the module out of openDIEL scenario. In the development stage, we can separate the issues of module transformation from any problem in openDLEL. Even in later stages when openDIEL becomes stable, it is still always better to keep the testing environment as clean and simple as possible.
2.4 Timer

As a tool targeting at high-performance computing, openDIEL is supposed to save the user time. It is supposed to automate process combination with minimal overhead and thus by using timer, we can monitor this process and mark any progress.

We will now divide the use of timer from two perspective – that of openDIEL user and that of developer.

2.4.1 For OpenDIEL User

As said in the very beginning of this paper, users have to provide a configuration file to utilize openDIEL. This file consists the dependency of the modules as well as groupings. This file greatly determines the order of execution of modules as well as idle times.

While automated optimization would be ideal in the long run, at the current stage, changing the configuration file would be more than sufficient for optimization at the user end.

At the end of every run, there would be a conclusive timing information as follow.

Seeing that process 11 takes the most time, we can then go into a folder with more detailed information.

User can then tell that Function 5 waits for a lot of its dependencies and this results in a hefty 84% time wasted in process 11. The user can then think of how the configuration file can be rewritten to better optimize openDIEL and the CPU usage.
2.4.2 For OpenDIEL Developer

For openDIEL developer, timer can actually help reduce overhead in coordinating different modules. This is done by using levels defined in "timestamp" function.

Level basically refers to the hierarchy of function call. The top level (or level 1) is always driver.c. This has to be placed inside driver.c by the user after MPI_Init and before MPI_Finalize. Each call then add the level by 1 while return statements generally should be accompanied by a negative level of 1. This goes deeper and deeper and returns similar to call stack.

Levels are not only useful for presentation. It is also useful for extracting information for timing data without intervention of user.

Next to each time, a square bracket would contain a number representing time difference. If the level difference is negative, openDIEL has just returned from a function call or finished a process. The timer would then find the corresponding starting time and the total time used in this process can then be
found. This is shown as a positive number in the square bracket.

If level difference is positive of zero, however, this means that the time
difference would be the time used between neighboring process of time used
to prepare for this particular process. This time is most likely overhead that
could be reduced by openDIEL developer. Hence the time difference is shown
as negative.

2.5 Possible Development

While we have gone over several test cases and debugged several times, there
is undeniably potential threats in our approach – that is the approach of string
matching – to module making.

The first one is the dependency on human support. As a developer, it
is quite frankly easy to develop white box test cases that can render mod-
Maker useless. For example, we can create variables that contains the substring
MPI\texttt{COMM\_WORLD}, or we can create wrap MPI\texttt{Init()} in another function
located in another file that is not gone under the radar of modMaker.

It would be even easier if we do not wish to defeat the core purpose of module
making. For example, we can put the keyword ”main” in every program as a
comment, then the pre-detection stage of module making would become useless.
We can also eliminate line break characters from the file. While this would not
affect module making, users can no longer identify the key feature that require
our attention because the limit of screen width would render highlighting useless.

The second one is the unpredictability of program structure. Simply put,
we just cannot be sure we can handle all C-programs flawlessly using mod-
Maker. For example, we only learned that ”argc”, ”argv” replacement should
be available in future versions of modMaker. However a quick implementation
was proven ineffective. This is because we have been treating the ”main file”, or
the file containing main program, indifferently from other files. A effective and
efficient way to do that in the future would be to either identify the ”main file”
or asking the user to provide it, as the ”main file” contains a lot of features that
we would like to work on and is the place we need to add a lot of signatures.

To stress on the seriousness, imagine a program that used preprocessor di-
rectives. They are in effect another language building on top of C-files. These
have even more unpredictable behavior and syntax. Imagine if the user has
always been using it to replace keywords at compile time, thus using the direc-
tives together with compiler to correctly generate executable, the chance of a
successful module making would then be even dimmer.

Problems like this would, in my prediction, come up for every large scale
programs that we convert in the future. While we predicted at the beginning
that human intervention was necessary, the amount needed at the current stage
is far from satisfactory. The hope for rapidly losing this need is also dim. The
creation of a fully automated modMaker would be as complex as creating a
compiler or interpreter. This would not be an ideal solo or short project in the
near future.

2.6 Future Work

Besides continual improvement of modMaker and timer, I believe that future
development of the User Interface of openDIEL can aim at module making of
FORTRAN codes and the issue of multiple file I/O in running multiple copies of parallel modules.

2.7 Reference

1. openDIEL (http://cfdllab.utk.edu/openDIEL/opendiel.php)

2. Stack Overflow (http://stackoverflow.com)

2.8 Acknowledgment

This project is made possible only with the support of my mentor Dr. Kwai Wong, the NSF, the University of Tennessee and Oak Ridge National Laboratory. I would like to express my deepest gratitude towards all of the people, in particular Dr. Wong, who have supported and guided me throughout the project. I would not have achieved close that what I have today without them.

2.9 Appendix

Figure 15: The flowchart of transforming a file
Figure 16: The flowchart of modMaker as a whole.
Figure 17: The flowchart of baseCheck, one of the fundamental checking programs
import worker
import sys
import os

tokensOfInterest = ["MPI_COMM_WORLD", "MPI_Finalize", "MPI_Init", "main" ]
#include "IEL.h"
header = "#include "MODULE_GLOBAL.h"

# Description: Transform a file under the copy mode
# Usage: copyMode (inputFileName, outputFileName, moduleName)
# Variables: [string] inputFileName: the file to be changed
# [string] outputFileName: the filename of the file to be output
# [string] moduleName: the name of module to be created

def copyMode (inputFileName, outputFileName, moduleName):
    global header, tokensOfInterest
    hasInterest = worker.init (inputFileName, tokensOfInterest)
    if not hasInterest:
        return

    (consents, cont) = worker.receiveConsent (inputFileName, ["MPI_COMM_WORLD"])    
    if not cont:
        worker.showProgress ("%s has been left untouched." % (inputFileName))
        return

    try:
        inputFile = open (inputFileName, 'r')
        outputFile = open (outputFileName, 'w')
        outputFile.write (header)

        for line in inputFile:
            (foundMain, changedMain, headerCreated) = worker.getMain ()

            line = worker.checkFunctionTitle (line, moduleName)
            line = worker.checkStatement (line)
            line = worker.checkVariable (line, consents)

            if worker.foundMain and not worker.headerCreated:
                worker.createMainHeader (moduleName, outputFileName)

            if worker.foundMain and not worker.changedMain:
                line = worker.addGlobal (line)

            outputFile.write (line)
            worker.lineCounter += 1

        inputFile.close ()
        outputFile.close ()

    except:
        print "Unexpected error:", sys.exc_info()[0]
        if os.path.exists (outputFileName):
            os.remove (outputFileName)
        extension = len (inputFileName)
        print "
" + "-" * (extent + 1)
print "Module building in %s have been reverted while prior changes retained." % (inputFileName)
print "
" + "-" * (extent + 1)
raise

# Description: Transform a file under the replace mode
# Usage: replaceMode (inputFileName, outputFileName, moduleName)
# Variables: [string] inputFileName: the file to be changed
# [string] moduleName: the name of module to be created

def replaceMode (filename, moduleName):
global header, tokensOfInterest
hasInterest = worker.init (filename, tokensOfInterest)
if not hasInterest:
    return

(consents, cont) = worker.receiveConsent (filename, ["MPI_COMM_WORLD"])  
if not cont:
    worker.showProgress (""%s has been left untouched." % (filename))
    return

inputFile = open (filename, 'r')
lines = inputFile.readlines ()
inputFile.close ()

try:
    outputFile = open (filename, 'w')
    outputFile.write (header)
    for line in lines:
        (foundMain, changedMain, headerCreated) = worker.getMain ()
        line = worker.checkFunctionTitle (line, moduleName)
        line = worker.checkStatement (line)
        line = worker.checkVariable (line, consents)

        if worker.foundMain and not worker.headerCreated:
            worker.createMainHeader (moduleName, filename)
        if worker.foundMain and not worker.changedMain:
            line = worker.addGlobal (line)

        outputFile.write (line)
        worker.lineCounter += 1
    outputFile.close ()
except:
    print "Unexpected error:", sys.exc_info ()[0]
    outputFile = open (filename, 'w')
    for line in lines:
        outputFile.write (line)
    outputFile.close ()
extension = len (filename)
print "
"— " (extension + 1)
print "Changes in %s have been reverted while prior changes retained." % (filename)
print "
"— " (extension + 1)

# Description: Transform a file under the archive mode
# Usage: archiveMode (inputFileName, outputFileName, moduleName)
# Variables: [string] inputFileName: the file to be changed
# [string] outputFileName: the filename of the archived file
# [string] moduleName: the name of module to be created

def archiveMode (inputFileName, outputFileName, moduleName):
    global header, tokensOfInterest
    hasInterest = worker.init (inputFileName, tokensOfInterest)
    if not hasInterest:
        return
    (consents, cont) = worker.receiveConsent (inputFileName, ["MPI.COMM_WORLD"])  
    if not cont:
        worker.showProgress ("%s has been left untouched." % (inputFileName))
        return

    inputFile = open (inputFileName, 'r')
    lines = inputFile.readlines ()
    inputFile.close ()
    outputFileName = worker.sequenced (outputFileName)
    try:
        outputFile = open (inputFileName, 'w')
        archiveFile = open (outputFileName, 'w')
        outputFile.write (header)
    for line in lines:
        archiveFile.write (line)

        (foundMain, changedMain, headerCreated) = worker.getMain ()
        line = worker.checkFunctionTitle (line, moduleName)
        line = worker.checkStatement (line)
        line = worker.checkVariable (line, consents)

        if worker.foundMain and not worker.headerCreated:
            worker.createMainHeader (moduleName, inputFileName)

        if worker.foundMain and not worker.changedMain:
            line = worker.addGlobal (line)

        outputFile.write (line)
        worker.lineCounter += 1

    archiveFile.close ()
    outputFile.close ()
outputFile.close()
archiveFile.close()

except:
    print "Unexpected error:", sys.exc_info()[0]
outputFile = open (inputFileName, 'w')
for line in lines:
    outputFile.write(line)
outputFile.close()
if os.path.exists(outputFileName):
    os.remove(outputFileName)
extension = len (inputFileName)
print "
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""

print "Changes in %s have been reverted while prior changes retained." % (inputFileName)
print "
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
raise

status = ""
(inputFileName, outputFileName, mode, directory, moduleName, fileExtension) = worker.receiveArguments(sys.argv)
option = "!"
while option != "" and option != "exit":
    option = raw_input("\nPress Enter to Continue or type \"exit\" to quit: ")
if option == "exit":
sys.exit(0)
if directory == "F":
    if mode == "C": #-----------------------------------------
        copyMode (inputFileName, outputFileName, moduleName)
    elif mode == "R": #-----------------------------------------
        replaceMode (inputFileName, moduleName)
    elif mode == "A": #-----------------------------------------
    if not os.path.exists(outputFileName):
        try:
            os.makedirs(outputFileName)
        except OSError:
            worker.displayMsg ("We do not have the permission to create a directory OR disk space is full OR directory already exists. Please either:
            1) Create a directory;
            2) Remove some files;
            OR
            3) Use another mode\n")
sys.exit (1)
outputFileName = os.path.join(outputFileName, inputFileName)
archiveMode (inputFileName, outputFileName, moduleName)

elif directory == "D":
    if mode == "C":
        if not os.path.exists(outputFileName):
            try:
            
"
os.makedirs(outputFileName)
except OSError:
    worker.displayMsg("We do not have the permission to create a directory OR disk space is full OR directory already exists. Please either:
    
1) Create a directory;
    
2) Remove some files; OR
    
3) Use another mode")
sys.exit(1)

worker.createGlobalHeader(outputFileName)

for dirname, dirnames, filenames in os.walk(inputFileName):
    for filename in filenames:
        if not filename.startswith('. '):
            iName = os.path.join(dirname, filename)
            ext = os.path.splitext(filename)[1].lower()
            if ext in fileExtension:
                oName = os.path.join(outputFileName, filename)
                oName = worker.sequenced(oName)
                worker.showProgress("\n= = = = = In file %s ===== \
" % (iName))
                copyMode(iName, oName, moduleName)

elif mode == "R":
    worker.createGlobalHeader(inputFileName)

for dirname, dirnames, filenames in os.walk(inputFileName):
    for filename in filenames:
        if not filename.startswith('. '):
            iName = os.path.join(dirname, filename)
            ext = os.path.splitext(filename)[1].lower()
            if ext in fileExtension:
                worker.showProgress("\n= = = = = In file %s ===== \
" % (iName))
                replaceMode(iName, moduleName)

elif mode == "A":
    worker.createGlobalHeader(inputFileName)

if not os.path.exists(outputFileName):
    try:
        os.makedirs(outputFileName)
    except OSError:
        worker.displayMsg("We do not have the permission to create a directory OR disk space is full OR directory already exists. Please either:
        
1) Create a directory;
        
2) Remove some files; OR
        
3) Use another mode")
sys.exit(1)

for dirname, dirnames, filenames in os.walk(inputFileName):
    for filename in filenames:
        if not filename.startswith('. '):
            iName = os.path.join(dirname, filename)
            ext = os.path.splitext(filename)[1].lower()
            if ext in fileExtension:
                worker.showProgress("\n= = = = = In file %s ===== \
" % (iName))
                copyMode(iName, oName, moduleName)
worker.showProgress ("
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"}

# This is the worker program that uses python to do work
# It can read in a c file and return a "package" used under the IEL model

import sys
from array import *
import os

#Global Flags

#initRemoved = False
#finRemoved = False
#mainRemoved = False

#Global Buffer

status = ""
lineCounter = 1
lines = []
foundMain = False
changedMain = False
headerCreated = False
changed = False

# Display Functions

# Clears the screen for displaying messages
def clearScreen ():
    print("\033c")

# Shows debug messages without being cleared (not exactly by not clearing)
def debug (s):
    showProgress (s)

# Shows the archived "more-important" messages that would "not be cleared"
def displayArchive ()
    clearScreen ()
    print status

# Display a one-time message
def displayMsg (s):


displayArchive()
print s

# Shows progress and archive the progress
def showProgress(s):
globalsstatus
status = status + s + "\n"
displayArchive()

# Displays a line with its highlight beneath certain region
def displayWithHighlight(lineNumber, line, start, end):
    count = line.count("\t")
    tempString = "\t" * (count + 1)
    tempString = tempString + " " * (start - count)
    tempString = tempString + "-" * (end - start + 1)
    tempString = tempString + "\n"
print tempString

# # # # # # # # # # # # # # # # # Worker Functions # # # # # # # # # # # # # # # # #

# Description: Validate user-input path
# Usage: pathName = validatePath(pathName, True)
# Variables: [str] pathName: user-input pathname
# [bool] existing: whether an existing or non-existing
# pathName is needed
# [bool] directory: whether a directory or file pathName
# is needed, true refers to directory
# Returns [str] a valid pathName as-per required

def validatePath(pathName, existing, directory):
    while existing and pathName == ":
        if directory:
            pathName = raw_input("Please specify a directory for transformation:").strip()
        else:
            pathName = raw_input("Please specify a file for transformation:").strip()
    if existing and directory:
        while not os.path.exists(pathName):
            print "\%s\ doesn’t exist" % (pathName)
            pathName = raw_input("Please enter an existing dirname: ").strip()
        while not os.path.isdir(pathName):
            print "\%s\ is not a directory" % (pathName)
            pathName = raw_input("Please enter an existing dirname: ").strip()
    if existing and not directory:
        while not os.path.exists(pathName):
            print "\%s\ doesn’t exist" % (pathName)
            pathName = raw_input("Please enter an existing filename: ").strip()
        while not os.path.isfile(pathName):
            print "\%s\ is not a file" % (pathName)
            pathName = raw_input("Please enter an existing filename: ").strip()
strip()

if not existing and directory:
    while os.path.exists(pathName):
        print """\"%s\" exists already" % (pathName)
        pathName = raw_input("Please enter a non-existing dirname or leave blank for auto-generated directory: ") . strip()

if not existing and not directory:
    while os.path.exists(pathName):
        print """\"%s\" exists already" % (pathName)
        pathName = raw_input("Please enter a non-existing filename or leave blank for auto-generated filename: ") . strip()

return pathName

# Description: Find the previous occurrence of string s in line with respect of startLocation
# Usage: void function (); int main();
# This function can locate the first ";";
# Variables: [str] line: the line in question
# [int] startLocation: the token must be before this position
# [str] s: token in question
# Returns [int] location of token. -1 if not found.
def findLastSeparator(line, startLocation, s):
    location = line . find (s)
    prev = location
    if location == -1 or location > startLocation:
        return -1
    while (location < startLocation):
        prev = location
        location = line . find (s, location + 1)
        if location == -1:
            return prev
    return prev

# Description: Determines if a given character is alpha–numeric.
# (10 numbers, 26 characters of upper- and lowercases and underscore)
# Usage: if isAlphanumeric (c): ...
# Variables: [char] c: character in question
# Returns [bool] True if c is alphanumeric. False if c is not
def isAlphanumeric(c):
    valid = False
    if (ord (c) >= 48 and ord (c) <= 57):
        valid = True
    if (ord (c) >= 65 and ord (c) <= 90):
        valid = True
    return valid
if (ord(c) >= 97 and ord(c) <= 122):
    valid = True
if (ord(c) == 95):
    valid = True
return valid

# Description: Checks if a sub-line contains only spaces (or space-equivalent, i.e. tab)
# Usage: if spaceOnly (line, startLocation, endLocation): {do sth}
# Variables: [str] line: the line in question
#            [int] startLocation: start of substring
#            [int] endLocation: end of substring
# Example: 0 1 2 3 4 5 6 7 8 9
#          word word
#          Pass in (line, 4, 6)
#          Returns [bool] True if there's only spaces. False otherwise.
def spaceOnly (line, startLocation, endLocation):
    if startLocation >= endLocation:
        return False
    valid = True
    for counter in xrange(endLocation - startLocation):
        if line[startLocation + counter] != ' ' and line[startLocation + counter] != '\t':
            valid = False
    return valid

# Description: To prevent overwriting existing files by adding numbers at the back
# Usage: outputFileName = sequenced (outputFileName)
# Variables: [str] filename: the original filename
# Returns [str] filename: the "safe", "updated" filename
def sequenced (filename):
    temp = filename
counter = 1
    while os.path.exists(temp):
        temp = filename + "_{:0{}d}".format(counter)
counter += 1
    return temp

# User Communication
# Description: Let the user decide whether or not to make a change
# Usage: if promptChange (line, ln, start, end, reason): {do sth}
def promptChange(line, lineNumber, start, end, reason):
    if start == end:
        return line
    print "We have found something that may need to be changed in your code for the following reason:\n"
    print "\t%s\n" % (reason)
    end = end - 1
    while (True):
        displayWithHighlight(lineNumber, line, start, end)
        option = raw_input("If you agree on this change press [Enter] or key in [y/Y]. If it is incorrect, key in [n/N].")
        if option == "" or option == "y" or option == "Y":
            return True
        elif option == "n" or option == "N":
            return False
        else:
            print "Unrecognized input. Please try again."

def receiveArguments(argv):
    clearScreen()
    mode = "C"
directory = "F"
modeSpec = False
dirSpec = False
li = sorted(sys.argv[1:])
counter = 0
while (counter < len(li) and li[counter][0] == "-"):
if len(li[counter]) == 1:
    print "Please check on input format and spacing. Unrecognized Input!"
    sys.exit(1)

elif li[counter][1] == "r" or li[counter][1] == "R":
    if not modeSpec:
        mode = "R"
        modeSpec = True
    else:
        print "Overlapped Flags!! %s, %s" % ("-" + mode, "-R")
        sys.exit(1)

elif li[counter][1] == "c" or li[counter][1] == "C":
    if not modeSpec:
        mode = "C"
        modeSpec = True
    else:
        print "Overlapped Flags!! %s, %s" % ("-" + mode, "-C")
        sys.exit(1)

elif li[counter][1] == "a" or li[counter][1] == "A":
    if not modeSpec:
        mode = "A"
        modeSpec = True
    else:
        print "Overlapped Flags!! %s, %s" % ("-" + mode, "-A")
        sys.exit(1)

elif li[counter][1] == "d" or li[counter][1] == "D":
    if not dirSpec:
        directory = "D"
        dirSpec = True
    else:
        print "Overlapped Flags!! %s, %s" % ("-" + directory, "-D")
        sys.exit(1)

elif li[counter][1] == "f" or li[counter][1] == "F":
    if not dirSpec:
        directory = "F"
        dirSpec = True
    else:
        print "Overlapped Flags!! %s, %s" % ("-" + directory, "-F")
        sys.exit(1)

elif li[counter][1] != "o" and li[counter][1] != "O":
    print "Unrecognized Flag %s" % (li[counter])

counter += 1

inputFileName = ""
outputFileName = ""
moduleFlag = False
moduleName = ""

while (counter < len(sys.argv)):
    if moduleFlag and moduleName == "":
        moduleName = sys.argv[counter]

    if sys.argv[counter][0] != "-":
        inputFileName = sys.argv[counter]
if inputFileName == ":
    inputFileName = sys.argv[counter]
elif outputFileName == ":
    outputFileName = sys.argv[counter]

if sys.argv[counter] == "-o" or sys.argv[counter] == "-O":
    if not moduleFlag:
        moduleFlag = True
    else:
        print("Overlapped Flags!! %s, %s" % (sys.argv[counter], "-O")
        sys.exit(1)

    counter += 1

if mode == "C": #
    if directory == "F":
        inputFileName = validatePath(inputFileName, True, False)
        outputFileName = validatePath(outputFileName, False, False)
    else:
        inputFileName = validatePath(inputFileName, True, True)
        outputFileName = validatePath(outputFileName, False, True)

elif mode == "R": #
    if directory == "F":
        inputFileName = validatePath(inputFileName, True, False)
    else:
        inputFileName = validatePath(inputFileName, True, True)

    if outputFileName != "":
        displayMsg("You have chosen to use the replace option.
        Changes will be made directly to the original file and %s doesn't matter\n" % (outputFileName))
        warning = raw_input("Type \"exit\" to end program or [Enter]
to continue: ")
        while warning != "" and warning != "exit":
            warning = raw_input("Type \"exit\" to end program or [Enter] to continue: ")
        if warning == "exit":
            sys.exit(1)

elif mode == "A": #
    if directory == "F":
        inputFileName = validatePath(inputFileName, True, False)
    else:
        inputFileName = validatePath(inputFileName, True, True)

    if directory == "D":
        dirD = "directory"
    else:
        dirD = "file"

    clearScreen()
    if moduleName == "":
        moduleName = "moduleMain"
    if mode == "A":
        if outputFileName == "":
            outputFileName = "Archive_modMaker"
(head, tail) = os.path.split(inputFileName)

if head:
    outputFileName = os.path.join(head, outputFileName)

showProgress("Transforming %s \"%s\" into module \"%s\"
               into module \"%s\" % (dirD, inputFileName, moduleName))

showProgress("Old files will be put into directory %s" %
               outputFileName)

elif mode == "R":
    showProgress("Transforming %s \"%s\" into module \"%s\"
               which will replace the original %s"
               % (dirD, inputFileName, moduleName, dirD))
    outputFileName = ""

elif mode == "C":
    if outputFileName == "" and directory == "F":
        (head, tail) = os.path.split(inputFileName)
        outputFileName = "module_" + tail

    if head:
        outputFileName = os.path.join(head, outputFileName)

    outputFileName = sequenced(outputFileName)

    if outputFileName == "" and directory == "D":

        (head, tail) = os.path.split(inputFileName)
        outputFileName = "Module_" + tail

        if head:
            outputFileName = os.path.join(head, outputFileName)

        outputFileName = sequenced(outputFileName)

        showProgress("Duplicating %s \"%s\" into module under
                      the name \"%s\" and filename \"%s\"
                      % (dirD, inputFileName, moduleName, outputFileName))

        showProgress("Original files will not be lost")

    if directory == "D":
        displayMsg("What file types should we focus on? Please
                    optionally enter extensions one by one and end
                    with a blank line.\nInclude \".\"\n                    and use small letters. Example: \".c\"\n                    ");
        extension = list()
        ext = raw_input("Extension: ")
        while ext != "":
            extension.append(ext)
            ext = raw_input("Extension: ")

    return (inputFileName, outputFileName, mode, directory,
            moduleName, extension)

# Description: Receive consent to speed up replacement process
# Usage:     consents = (inputFileName, strings)
# Variables: [str] inputFileName: the file to undergo the process
# list[st]r stringList: the strings to look for
# Returns list[bool] consents: the consents [True, True, False, ...] compiled
# [bool] continue: whether or not the file should continue transformation

def receiveConsent(inputFileName, stringList):
    if len(stringList) == 0:
        return None

    if not os.path.exists(inputFileName):
        debug("File doesn't exists")
        sys.exit(1)

    consent = []
    option = "s"
    while (option != "" and option != "N" and option != "n"):
        option = raw_input("We are about to conduct transformation on %s. "
        \nPress [Enter] to begin or key-in "n/N" to skip this file.
        " % (inputFileName))

    if option == "":
        inputFile = open(inputFileName, 'r')
        lines = inputFile.readlines()
        inputFile.close()

        for string in stringList:
            displayArchive()
            lineNumber = 1
            counter = 0

            for line in lines:
                start = line.find(string)
                if start != -1:
                    displayWithHighlight(lineNumber, line, start, start +
                    len(string))
                    counter += 1
                    lineNumber += 1

                if counter != 0:
                    option = "s"
                    while (option != "" and option != "N" and option != "n"):
                        option = raw_input("We have found %d occurrences of %s.
                        Press [Enter] to authorize every change or key-in "n/N" to
                        authorize one-by-one." % (counter, string))

                if option == "":
                    consent.append(True)

                if counter == 1:
                    showProgress("%d %s has been replaced." % (counter, string))
                else:
                    showProgress("%d %s have been replaced." % (counter, string))
                else:
consent.append(False)

if counter == 1:
    showProgress("%s remains unchanged." % (string))
else:
    showProgress("%d suspected %s remain unchanged." % (counter, string))
else:
    consent.append(False)
return(consent, True)
else:
    for string in stringList:
        consent.append(False)
return(consent, False)
def init(filename, tokenList):
    #global changed
    #changed = False
    file = open(filename, 'r')
    lines = file.readlines()
    file.close()
    lineCounter = 1
    for line in lines:
        for token in tokenList:
            if line.find(token) != -1:
                debug((token)
    return True

return False
def createGlobalHeader(path):
    if os.path.isfile(path):
        debug("Directory path required to create global header.")
    return
    filename = os.path.join(path, "MODULE\GLOBAL.h")
    file = open(filename, 'w')
    file.write("#include "IEL.h"

extern IEL_exec_info_t * exec_info;
"
    filename = os.path.join(path, "IEL.h")
    file = open(filename, 'w')
    file.write("#include "mpi.h"

typedef struct module_depend_t {
int argc;
tchar ** argv;
} module_depend_t;
"
    file_write("typedef struct IEL_exec_info_t {
int MPI_Comm_copy_comm;
tstruct module_depend_t* modules;
IEL_exec_info_t;" endif"
    file.close()
    return

def createMainHeader(moduleName, path):
global header

headerCreated = True

if not os.path.isfile (path):
    debug ("Main header file creation error. Require filepath to main.")
    return

# --- Creating Module Header --- #

name = os.path.splitext (path) [0]
headername = name + ".h"

if os.path.exists (headername):
    debug ("Main header already exists. Please manual declare module in header.")
else:
    file = open (headername, 'w')
    file.write ("#include "IEL.h"\n")
    file.write ("int " + moduleName + " (IEL_exec_info_t *
      iel_exec_info);\n")
    showProgress ("Header file " + headername + " created successfully.")
    file.close ()

# --- Creating Test Main.c --- #

name = os.path.dirname (path)
filename = os.path.join (name, "IEL_main.c")
filename = sequenced (filename)
debug (filename)

file = open (filename, 'w')

file.write ("#include "MODULEGLOBAL.h"\n")
file.write ("#include "\n")
file.write (os.path.basename (headername))
file.write ("\n")
file.write ("#include <stdlib.h>\n")
file.write ("#include "\n")
file.write ("int main(int argc, char* argv[]){\n")
file.write ("tMPI_Init (&argc, &argv);\n")
file.write ("tIEL_exec_info_t *temp = (IEL_exec_info_t *) malloc (\n      sizeof (IEL_exec_info_t));\n")
file.write ("ttemp -> modules = (module_depend_t *) malloc (\n      sizeof (module_depend_t));\n")
file.write ("ttemp -> modules -> mod_argc = argc;\n")
file.write ("ttemp -> modules -> mod_argv = argv;\n")
file.write ("tmoduleMain (temp);\n")
file.write ("tMPI_Finalize();\n")
file.write ("treturn 0;\n")
file.write ("}\n")

file.close ()

showProgress ("Test main created successfully.")

return
def setGlobal ( line ) :
    global changedMain

    location1 = line . find ( "iel . exec_info" )
    location2 = line . find ( "}", location1 )
    if location2 != -1:
        debug ( "Main is changed here" )
        changedMain = True
        return ( line [: location2 + 1] + "\n\ntextexec_info = iel . exec_info \\
    ;\n    \n+ line [ location2 + 1 : ] )
    else:
        changedMain = False
        return line

def addGlobal ( line ) :
    global changedMain

    location2 = line . find ( "{" )
    if location2 == -1:
        return line
    else:
        changedMain = True
        debug ( "Main is changed here" )
        return ( line [: location2 + 1] + "\n\ntextexec_info = iel . exec_info \\
    ;\n    \n+ line [ location2 + 1 : ] )

def getMain () :
    return ( foundMain , changedMain , headerCreated )

# # # # # # # # # # # # # # # # # Checking Functions # # # # # # # # # # # # # # # # #

# Wrapper function for checking all function titles
def checkFunctionTitle ( line , moduleName ) :
    line = checkMain ( line , moduleName )
    return line

# Wrapper function for checking all in-line statements
def checkStatement ( line ) :
    line = checkInit ( line )
    line = checkFinalize ( line )
    return line

# Wrapper function for checking all in-line variables
def checkVariable ( line , consents ) :
    if consents [ 0 ] :
        line . replace ( "MPI_COMM_WORLD" , "exec_info->
        module_copy_comm" )
    else:
        line = checkMPICommWorld ( line )

    # if foundMain:
    #    line = checkArgc ( line )
    #    line = checkArgv ( line )

    return line

# Defines parameters to check the main
def checkMain ( line , moduleName ) :
    msgB = [ ]
msgB.append("Only one Main can exist and it belongs in the driver.c")
msgB.append("Main has been changed to " + moduleName)
msgB.append("Suspected Main remained unchanged")
replacement = "IEL_exec_info_t * exec_info;\n" + "int " + moduleName + " (IEL_exec_info_t *iel_exec_info)"
return baseCheck(line, ["int", "main"], ",", "")

# Defines parameters to check MPI_Init
def checkInit (line):
    msgB = []
    msgB.append("Only one Init can exist and it belongs in the driver.c")
    msgB.append("MPI_Init has been removed")
    msgB.append("Suspected MPI_Init remained unchanged")
    return baseCheck(line, ["MPI_Init"], ";", ";", msgB, ",", ",")

# Defines parameters to check MPI_Finalize
def checkFinalize (line):
    msgB = []
    msgB.append("Only one Finalize can exist and it belongs in the driver.c")
    msgB.append("MPI_Finalize has been removed")
    msgB.append("Suspected MPI_Finalize remained unchanged")
    return baseCheck(line, ["MPI_Finalize"], ";;", ";;", msgB, ",", ",")

# Defines parameters to check MPI_COMM_WORLD
def checkMPICommWorld (line):
    msgB = []
    msgB.append("MPI_COMM_WORLD has been divided into subcommunicators")
    msgB.append("MPI_COMM_WORLD has been replaced")
    msgB.append("Suspected MPI_COMM_WORLD remained unchanged")
    return baseCheck(line, ["MPI_COMM_WORLD"], "", ";", "", msgB, ",")

# Defines parameters to check argc
def checkArgc (line):
    # msgB = []
    # msgB.append("argc is not directly visible at this level")
    # msgB.append("argc has been replaced")
    # msgB.append("Suspected argc remained unchanged")
    # return baseCheck(line, ["argc"], "", ";", msgB, ",")

# Defines parameters to check argv
def checkArgv (line):
    # msgB = []
    # msgB.append("argv is not directly visible at this level")
    # msgB.append("argv has been replaced")
    # msgB.append("Suspected argv remained unchanged")
    # return baseCheck(line, ["argv"], ";;", msgB, ",")

# Description: Base checking function which is versatile and powerful
# Usage: line = baseCheck (...) 
# Variables: [str] line: the line in question
# [list] tokenList: list of tokens to match to
# [str] prevSep: Separator that is likely to lead the
# segment in question. Replacement starts right afterwards.
# Starts from start of token match if omitted.
# [str] nextSep: Separator that is likely to follow the
# segment. Replacements continues to this point. Ends at end of
# token match if omitted.
# [list] msgBundle: 0: contains "reasons", why the
# segment needs to be replaced / removed
# 1: contains "successful msg", showing that
# the segment was properly dealt with
# 2: contains "unchanged msg", showing that the
# segment was not affected
# [str] replacement: if omitted, substring in concern is
# commented out. Otherwise, provides replacement for substring.
# [str] returnsValue: if set, the replacement concerns a
# function which returns value type specified. This only affects
# "commenting out"
# Returns: line that can be altered or unchanged

def baseCheck (line, tokenList, prevSep, nextSep, msgBundle, replacement, returnsValue):

global foundMain
firstPos = []

for token in tokenList:
    pos = line.find (token)
    if pos == -1:
        return line
    else:
        firstPos.append (pos)

candidates = findCandidates (line, tokenList, firstPos)

if replacement == "":
    replace = False
else:
    replace = True

for (begin, end) in reversed (candidates):
    valid = True
    if begin != 0:
        if isAlphanumeric (line [begin - 1]):
            valid = False
    if isAlphanumeric (line [end]):
        valid = False
    if valid:
        typePos = -1
        if nextSep != "":
            end = line.find (nextSep, end + 1) + 1
        if returnsValue != "":
            typePos = findLastSeparator (line, begin, returnsValue)
        if prevSep != "":
            temp = findLastSeparator (line, begin, prevSep)
        if temp != -1:
begin = temp + 1

else:
    begin = 0

if promptChange (line, lineCounter, begin, end, msgBundle [0]):
    if replace:
        output = ""
        output = output + line [:begin]
        output = output + replacement
        output = output + line [end:]
        showProgress (msgBundle [1])
        line = output
    else:
        output = ""
        if typePos != -1 and typePos >= begin and typePos < end:
            equals = line.find ("=", typePos)
            if equals != -1 and equals > typePos:
                if line[equals - 1] == "":
                    output = output + line [:equals - 1]
                else:
                    output = output + line [:equals]
        elif begin != 0:
            output = output + "\n"
            output = output + line [:begin]
            output = output + "\n"
        output = output + "//"
        output = output + line [begin:end]
        if line[end] != "$n"
            output = output + "$n"
        output = output + line [end:]
        showProgress (msgBundle [1])
        line = output

# changed = True
if len (tokenList) > 1 and tokenList [1] == "main":
    foundMain = True
    line = setGlobal (line)
else:
    showProgress (msgBundle [2])

return line

# Description: Verstaille token matching function
# Usage: candidates = findCandidates (line, tokenList, firstPos)
# Variables: [str] line: the line in question
# [list] tokenList: list of tokens to match to
# [str] prevSep: Separator that is likely to lead the
# segment in question. Replacement starts right afterwards.
# Starts from start of token match if omitted.
# [str] nextSep: Separator that is likely to follow the
# segment. Replacements continues to this point. Ends at end of
# token match if omitted.
msgBundle: 0: contains "reasons", why the segment needs to be replaced / removed
# 1: contains "successful msg", showing that the segment was properly dealt with
# 2: contains "unchanged msg", showing that the segment was not affected

[str] replacement: if omitted, substring in concern is commented out. Otherwise, provides replacement for substring.

```python
def findCandidates(line, tokenList, firstPos):
    locationDictionary = []
    result = []
    counter = 0
    arraySize = len(tokenList)
    for token in tokenList:
        locationArray = []
        while (firstPos[counter] != -1):
            locationArray.append(firstPos[counter])
            firstPos[counter] = line.find(tokenList[counter], firstPos[counter] + 1)
            counter += 1
        locationDictionary.append(locationArray)
        accessArray = []
        for i in xrange(arraySize):
            accessArray.append(0)
        while (True):
            valid = True
            for i in xrange(arraySize - 1):
                j = accessArray[i]
                k = accessArray[i + 1]
                if locationDictionary[i][j] > locationDictionary[i + 1][k]:
                    valid = False
                    accessArray[i + 1] += 1
                if accessArray[i + 1] >= len(locationDictionary[i + 1]):
                    return result
            if not spaceOnly(line, locationDictionary[i][j] + len(tokenList[i]), locationDictionary[i + 1][k]):
                valid = False
                for c in xrange(i + 1):
                    if accessArray[c] >= len(locationDictionary[c]):
                        return result
                if valid:
                    begin = locationDictionary[0][accessArray[0]]
                    end = locationDictionary[arraySize - 1][accessArray[arraySize - 1] + len(tokenList[arraySize - 1])]
                    result.append((begin, end))
                    accessArray[i] += 1
                    if accessArray[i] >= len(locationDictionary[i]):
                        return result
```