



The DeBasher Software Package

Daniel Ortiz Martínez

Mathematics and Computer Science Department
University of Barcelona

Table of Contents

1. Introduction
2. Getting Started
3. Executing DeBasher Programs
4. Additional Information

Introduction

- DeBasher is a flow-based programming extension for Bash
- **Flow-based programming** combines data flow programming with component-based software engineering
 - facilitates the implementation of highly modular programs
 - exploits the parallelism implicitly determined by data dependencies between components

Getting Started

Installation

- Obtain the package using git:

```
git clone https://github.com/daormar/debasher.git
```

- Change to the directory with the package's source code and type:

```
./reconf  
./configure  
make  
make install
```

NOTE: use `--prefix` option of `configure` to install the package in a custom directory

Third Party Software

- Bash
- Python
- Slurm Workload Manager (optional)
- Conda (optional)
- Docker (optional)

Quickstart Example

- How to implement the “Hello World!” program?
- If we call our program `debasher_hello_world`, we should create a file with the same name and Bash extension, `debasher_hello_world.sh`:

Quickstart Example: Code

```
hello_world_explain_cmdline_opts()
{
    # -s option
    local description="String to be displayed ('Hello World!' by default)"
    explain_cmdline_opt "-s" "<string>" "$description"
}

hello_world_define_opts()
{
    # Initialize variables
    local cmdline=$1
    local optlist=""

    # Obtain value of -s option
    local str=$(get_cmdline_opt "${cmdline}" "-s")

    # -s option
    if [ "${str}" = "${OPT_NOT_FOUND}" ]; then
        define_opt "-s" "Hello World!" optlist || return 1
    else
        define_opt "-s" "${str}" optlist || return 1
    fi

    # Save option list
    save_opt_list optlist
}

hello_world()
{
    # Initialize variables
    local str=$(read_opt_value_from_func_args "-s" "$0")

    # Show message
    echo "${str}"
}

debasher_hello_world_program()
{
    add_debasher_process "hello_world" "cpus=1 mem=32 time=00:01:00"
}
```

Quickstart Example: Main Elements

- DeBasher uses three entities: processes, programs and modules
- A program is composed of a set of processes
- A module is a file storing multiple processes and one program
- Processes and modules are identified by a particular name, and their behavior is defined by a set of functions
- The program associated with a module is also defined by a function

Quickstart Example: Approach

- DeBasher adopts an object-oriented programming approach
 - Each function implements a specific method
 - Function names have two parts: the name of the program or module, and a suffix identifying the method
- In the “Hello World!” example we have a module named `debasher_hello_world` that is stored in the `debasher_hello_world.sh` file
- The module defines a program that executes the process `hello_world`

Quickstart Example: Functions Involved

- `hello_world_explain_cmdline_opts`: documents the command line options that can be provided to `hello_world`. In this case, the `-s` option, which allows to specify the string to be shown. For this purpose, the `explain_cmdline_opt` API function is used
- `hello_world_explain_define_opts`: the `define_opts` method specifies the options that will be provided to the `hello_world` process. Using the `get_cmdline_opt` API function, it retrieves the value of the `-s` command-line option, (or `-s "Hello World!"` if not provided). The code uses the `define_opt` API function to register options and the `save_opt_list` function to save them

Quickstart Example: Functions Involved

- `hello_world`: implements the process itself (in this case the function name does not incorporate any suffix). `hello_world` reads its options using the `read_opt_value_from_func_args` API function
- `debasher_hello_world_program`: the `program` method allows to define the processes involved in the execution. In this case, only one process is involved, `hello_world`, which is added to the by means of the `add_debasher_process` function

Quickstart Example: Program Execution

- In order to execute the program, DeBasher incorporates the `debasher_exec` tool
- Provided that the `debasher_hello_world.sh` module file is in the current directory and that `debasher_exec` is included in the `PATH` variable, we can execute the following:

```
$ debasher_exec --pfile debasher_hello_world.sh --outdir out
```

where `out` is the output directory

Quickstart Example: Program Execution

- Since the output of the program is just a string printed to the standard output, we can use the `debasher_get_stdout` command to visualize such a string
- For this purpose, we should provide the name of the output directory and the name of the process whose standard output we want to visualize:

```
$ debasher_get_stdout -d out -p hello_world
```

- The output of the previous command is:

```
Hello World!
```

Quickstart Example: Program Execution

- It is also possible to inspect the scheduler output, which includes:
 - The standard and error output of a particular process
 - some scheduling-related information
- The scheduler output is useful for debugging
- To visualize the scheduler output we can use:

```
$ debasher_get_sched_out -d out -p hello_world
```

- The output of the previous command is:

```
Process started at 07/30/24 18:17:06
* Resetting output directory for process...
Hello World!
Function hello_world successfully executed
Process finished at 07/30/24 18:17:06
```


Executing DeBasher Programs

Program Execution

- DeBasher incorporates a specific tool to execute programs called `debasher_exec`
- To get help for the tool we can execute:

```
$ debasher_exec --help
```

- `debasher_exec` only has two mandatory input options:
 - `--pfile <string>`: allows to define the DeBasher file specifying the program to be executed
 - `--outdir <string>`: specifies the name of the output directory

Program Execution: Visualizing Command Line Options

- The command line options for the DeBasher program should be provided to `debasher_exec`.
- To visualize them, we can use the `--show-cmdline-opts` option. Assuming we work with the “Hello World!” program:

```
$ debasher_exec --pfile debasher_hello_world.sh --outdir out --show-cmdline-opts
```

Program Execution: Visualizing Command Line Options

- The output of the previous command is:

```
# Command line options for the program...  
CATEGORY: GENERAL  
-s <string> String to be displayed ('Hello World!' by default) [hello_world]
```

- Command line options can be divided into different categories, being the “GENERAL” category the default one
- In this case, the “GENERAL” category contains the `-s` option, used to define the string to be displayed

Program Execution: Troubleshooting Options

- When working with complex programs, it is useful to ensure that all the options are passed correctly
- `debasher_exec` incorporates two options for that purpose:
 - `--check-proc-opts`: when providing this option, `debasher_exec` checks if all the options are available for the processes that compose a program. If there are options missing, an error message is shown. Otherwise, the options are shown and the tool finishes its execution
 - `--debug`: this option carries out all the necessary steps to execute a DeBasher program, with the exception of the execution itself

Program Execution: Using the Built-In Scheduler

- `debasher_exec` incorporates the `--sched` option to specify which scheduler is used to orchestrate program execution
- DeBasher incorporates a built-in scheduler that can be activated for the “Hello Word!” program as follows:

```
$ debasher_exec --pfile debasher_hello_world.sh --outdir out --sched BUILTIN
```

- It is possible to specify the number of CPUs and the amount of RAM that is available using the following options:
 - `--builtinsched-cpus <int>`: number CPUs available for the built-in scheduler. A value of zero means unlimited CPUs
 - `--builtinsched-mem <int>`: RAM in MB that can be used by the built-in scheduler. A value of zero means unlimited memory

Program Execution: Using External Schedulers

- It is also possible to combine `debasher_exec` with external schedulers
- Currently, DeBasher provides support for the well known Slurm scheduler:
 - To make Slurm work in combination with DeBasher, it is necessary to install and properly configure the scheduler in the machine where DeBasher will be executed
 - Once Slurm is installed, we can proceed with the installation of DeBasher, which will automatically detect Slurm's availability.
 - If Slurm and DeBasher are correctly configured, then we can combine them by means of the following command:

```
$ debasher_exec --pfile debasher_hello_world.sh --outdir out --sched SLURM
```

Structure of the Output Directory

- `debasher_exec` stores its results in an output directory
- Assuming that we executed program with two processes `a` and `b`, and that the output directory was `out`, its content would be:

```
out
|-- a
|   |-- file1
|   |-- file2
|   |-- ...
|-- b
|   |-- file1
|   |-- file2
|   |-- ...
|-- __exec__
|   |-- a
|   |   |-- a
|   |   |-- a.finished      (depends on execution status)
|   |   |-- a.id            (depends on execution status)
|   |   |-- a.sched_out     (depends on execution status)
|   |   |-- a.stdout        (depends on execution status)
|   |-- b
|   |   |-- b
|   |   |-- b.finished      (depends on execution status)
|   |   |-- b.id            (depends on execution status)
|   |   |-- b.sched_out     (depends on execution status)
|   |   |-- b.stdout        (depends on execution status)
|-- __graphs__
|   |-- dependency_graph.dot
|   |-- dependency_graph.pdf
|   |-- process_graph.pdf   (optional)
|   |-- process_graph.pdf   (optional)
|-- command_line.sh
|-- program.fifos
|-- program.opts
|-- program.procspec
```


Structure of the Output Directory: Directories

- `a`: stores the output files (if any) of process `a`
- `b`: it is the output directory of process `b`
- `__exec__`: stores the execution information for each process. The following files are created for any given process:
 - `<process_name>`: contains the code to be executed
 - `<process_name>.finished`: this file is created to signal that the execution of the process has finished
 - `<process_name>.id`: contains an identifier of the process created by the scheduler being used
 - `<process_name>.sched_out`: contains the process standard and error outputs and also some scheduler information
 - `<process_name>.stdout`: contains the process' standard output

Structure of the Output Directory: Directories

- `__graphs__`: contains graphical representations of the program:
 - `dependency_graph.pdf`: graph showing process dependencies
 - `process_graph.pdf`: shows relations between process options, generated only if `--gen-proc-graph` is provided to `debasher_exec`

Structure of the Output Directory: Files

- `comand_line.sh`: stores the command line used to execute the program
- `program.fifos`: contains information about the FIFOs used by the program
- `program.opts`: contains an exhaustive list of all the options provided to the program processes
- `program.procspec`: contains a specification for each process executed within a given program

Process Status Visualization

- The `debasher_status` tool obtains information about the execution status of each process involved in a program
- Example for the "Hello World!" program previously executed:

```
$ debasher_status -d out
```

- The output of the tool is:

```
PROCESS: hello_world ; STATUS: FINISHED  
* SUMMARY: num_processes= 1 ; finished= 1 ; inprogress= 0 ; unfinished= 0 ; unfinished_but_runnable= 0 ; todo= 0
```

Process Status Visualization

- DeBasher Statuses:
 - `finished`: the process successfully completed execution
 - `inprogress`: the process is currently being executed
 - `unfinished`: the process did not successfully complete execution
 - `unfinished_but_runnable`: the process has not completed execution yet, and is not being executed. However, it can resume its execution
 - `todo`: the process has not yet started execution
- `debasher_status` can also show the status of an individual process using the `-p` option:

```
$ debasher_status -d out -p hello_world
```

Program Stop

- When a DeBasher program is being executed, it is possible to stop a process or the whole program using the `debasher_stop` tool
- The tool requires the name of the output directory and, optionally, the name of a specific process should be provided
- The following command stops the `hello_word` process that is executed within the “Hello World!” program:

```
$ debasher_stop -d out -p hello_world
```

- To stop the whole program, the “-p” option is omitted:

```
$ debasher_stop -d out
```

Program Statistics Generation

- The `debasher_stats` tool allows to generate statistics for a DeBasher program
- The program report process statuses and the elapsed time in seconds until completion
- Input parameters include the program's output directory and, optionally, the process name whose statistics should be obtained

Program Statistics Generation

- Example for the `hello_world` process that belongs to the “Hello World!” program:

```
$ debasher_stats -d out -p hello_world
```

- To get statistics for all processes, we can simply type:

```
$ debasher_stats -d out
```


Additional Information

<https://debasher.readthedocs.io/en/latest/index.html>