Distributed Version Control System using Functional Programming

Deep Patel IMT2021011
Krutik Patel IMT2021024
Suyash Chavan IMT2021048

May 13, 2024

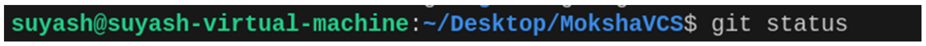
1 Problem Description

- Objective: Develop a general distributed version control system using OCaml and functional programming concepts.
- Approach:
  - Define a command language for version control commands.
  - Implement a parser to parse commands written in the command language.
  - Map parsed commands to corresponding Version control functions for execution.
- Command Language:
  - Syntax or structure defined for basic version control Commands.
  - Commands include initialization, file addition, commit creation, pushing, and pulling, etc.
- Parser:
  - Converts commands into executable actions.
- Version Control Functions:
  - Perform corresponding operations such as adding files, creating commits, and updating configurations for given command.
- Testing:
  - Ensure correctness and reliability of the version control system.
  - Validate behavior under various scenarios and edge cases.
- Outcome: Provide developers a generalized version control solution that integrates seamlessly with their development workflow.

2 Solution Outline

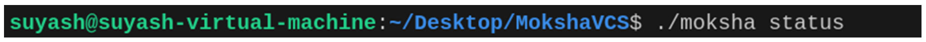
2.1 MokshaVCS: A Distributed Version Control System

- MokshaVCS is a distributed version control system (DVCS) built using OCaml, a powerful and functional programming language.
- It aims to provide developers with a secure, efficient, and user-friendly platform for managing their codebase throughout the development lifecycle.
- MokshaVCS offers a Command Line Interface (CLI) similar to Git, with a range of commands facilitating version control operations.



```
suyash@suyash-virtual-machine:~/Desktop/MokshaVCS$ git status
```

Figure 1: Git command



```
suyash@suyash-virtual-machine:~/Desktop/MokshaVCS$ ./moksha status
```

Figure 2: MokshaVCS command

2.2 Concepts Used from Functional Programming

- **Immutability**: Side effects are avoided, promoting a functional programming paradigm.
- **Higher Order Functions and Pure Functions**: Functions are treated as first-class citizens, enabling the use of higher-order functions and promoting purity.
- **Pattern Matching**: Allows for concise and powerful conditional branching based on the structure of data.
- **Recursive Paradigm and Tail-Recursions**: Embraces recursion as a fundamental concept, optimizing tail-recursions for efficiency.
- **Immutable Data Types**: Utilizes immutable data structures like lists, promoting stability and predictability.
- **Types and Modules**: Encourages strong typing and modular organization for better code structure and maintainability.
- **Error Handling**: Embraces functional techniques for handling errors, often through mechanisms like failwith or Raise types.

- **Makefile to Build the Project**: Implements a Makefile for building the project, promoting automation and reproducibility.
- **Module Interface to Handle Mutable Data Types**: Utilizes module interfaces to encapsulate and control access to mutable data, preserving purity where possible.

2.3 Deliverables

1. **Application Design and Flow Chart**:
    - Document system architecture and interactions.
    - Provide flow chart illustrating command execution flow.
2. **Command Language Specification and Parser Implementation**:
    - Define syntax and semantics of version control commands.
    - Implement parser in OCaml.
3. **Version Control System Functions**:
    - Implement core version control functionalities (e.g., init, commit, add, push, etc) in OCaml.
4. **Unit Testing**:
    - Develop and execute comprehensive unit tests to ensure the reliability and correctness of the system.
5. **Automatic Build using Makefile**:
    - Create a Makefile to automate the build process, ensuring consistency and ease of deployment.

2.4 Command Language Description

- **init <username> <password>**: Initializes a new version control repository in the current directory and adds config file corresponding to given authentication pair of user and password.
- **add <filename>**: Adds a file to the staging area, ready to be included in the next commit.
- **commit -m "<message>"**: Creates a new commit with the changes currently staged in the staging area. The -m flag allows the user to provide a commit message describing the changes.
- **status**: Displays the current status of the repository, including changes not yet staged, changes staged for commit, and untracked files.
- **log**: Displays a log of commits, showing information such as commit ID, directory of committed version, and commit message.
- **force-revert <Hash_of_version>**: Upon providing the hash value corresponding to the target version, this command will initiate a rollback procedure which entails reverting the system to the specified version and subsequently removing all versions created thereafter from the local environment.
- **push <Global_Dir_Path>** : command facilitates the transmission of local commits of the latest version to a remote global repository. Additionally, it makes modifications to the global configuration file.
- **clone <Global_Dir_Path>**: command replicates a remote global repository identified by the provided Path, generating a local duplicate inclusive of all commits. Furthermore, it imports the latest version from the global repository to the local environment and executes necessary adjustments to the local configuration file.

2.5 Application Design

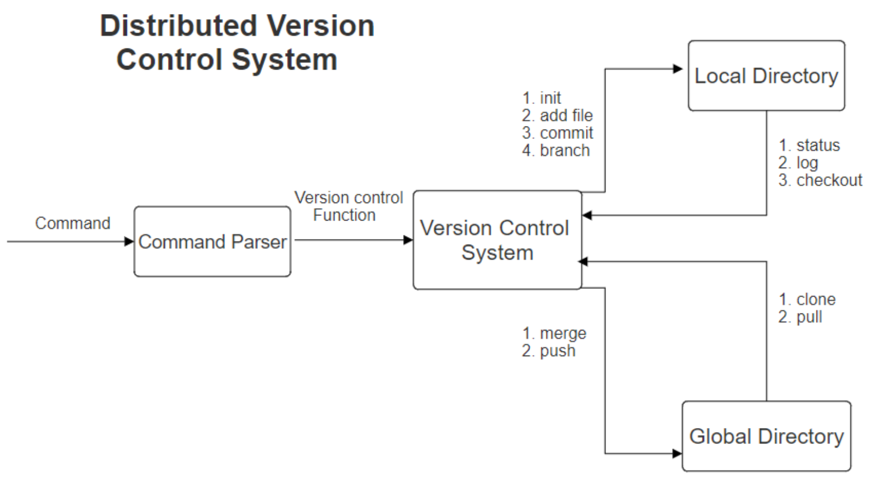


Figure 3: Flow Chart of System

- User Input: The user enters a version control command (e.g., "commit").
- Command Parsing: The parser interprets the command and translates it into a format the system understands.
- Version Control Actions: The parsed command triggers specific functionalities (adding files, creating commits) to perform the desired operation on the local repository.
- Local vs. Global Actions: Depending on the command (e.g., "commit" or "pull"), data may flow between the local repository and a global repository.
- Data Persistence: After executing the command, the system updates the local repository with the changes (e.g., new commit).
- Log Maintenance: Version control systems maintain a log file (e.g., Git uses '.git/logs/HEAD') that chronologically records all commits, including the commit message, author information, and snapshot of the repository state for each commit. This log allows users to track changes, revert to previous versions, and understand the project's history.

3 Implementation Details

3.1 Command Parser

The command parser module undertakes the following steps:

1. **Parsing Command Line Arguments**: The module parses the command line arguments and identifies the corresponding command.
2. **Argument Parsing and Validation**: Arguments associated with the identified command are parsed and checked for validity.
3. **Execution of Command Actions**: Finally, actions corresponding to the identified command are invoked.

3.2 Data structures for System

3.2.1 Config Data structure

- We need something to represent the state of the repository, this is where config data structure comes in

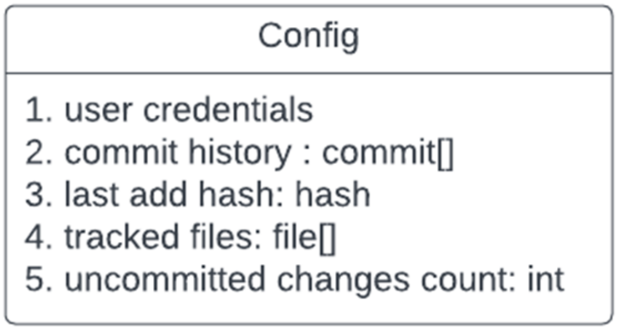


Figure 4: Config data structure

3.2.2 Commit Data structure

- We need something to represent a commit, Each repository can have multiple commits. This is where commit data structure comes in. The hash will be used to check if we have something to commit or not.

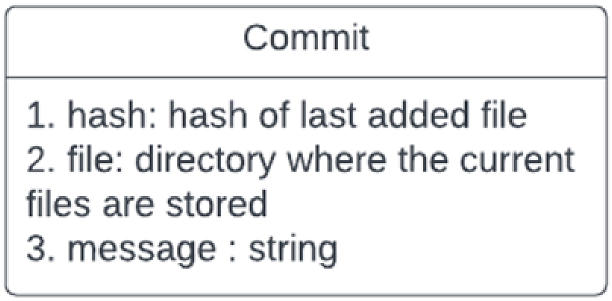
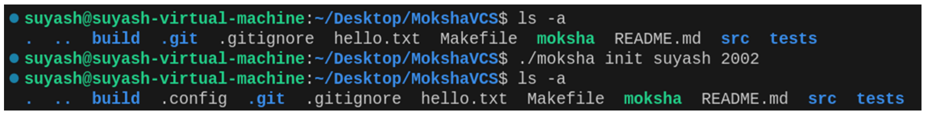

Commit
1. hash: hash of last added file
2. file: directory where the current files are stored
3. message : string

Figure 5: Commit data structure

3.3 Implementation of Commands

3.3.1 INIT : init <username> <password>

- Initializing a repository means creating necessary directory and config data structures, with the fields set to zero or an empty list depending on the data type.

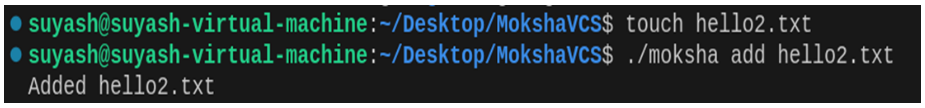


```
● suyash@suyash-virtual-machine:~/Desktop/MokshaVCS$ ls -a
  .  ..  build  .git  .gitignore  hello.txt  Makefile  moksha  README.md  src  tests
● suyash@suyash-virtual-machine:~/Desktop/MokshaVCS$ ./moksha init suyash 2002
● suyash@suyash-virtual-machine:~/Desktop/MokshaVCS$ ls -a
  .  ..  build  .config  .git  .gitignore  hello.txt  Makefile  moksha  README.md  src  tests
```

Figure 6: Init Example

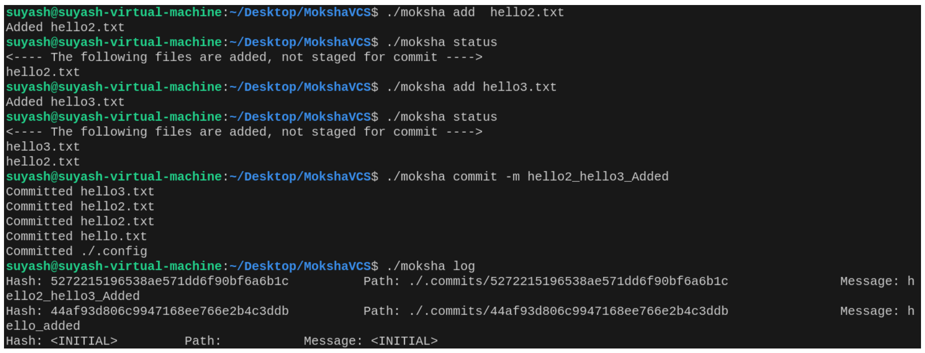
3.3.2 ADD : add <filename>

- This is a mechanism to add new files to the current state.
- We consider adding only one file at a time.
- We add the file path to the tracked files in the config data structure.
- We increment the count of uncommitted files by one and update the last add hash in the config data structure.



```
suyash@suyash-virtual-machine:~/Desktop/MokshaVCS$ touch hello2.txt
suyash@suyash-virtual-machine:~/Desktop/MokshaVCS$ ./moksha add hello2.txt
Added hello2.txt
```

Figure 7: Single Add Example

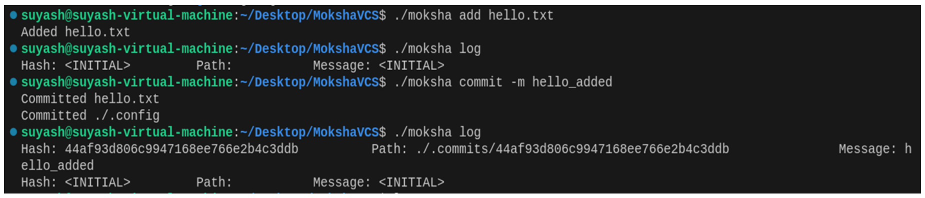


```
suyash@suyash-virtual-machine:~/Desktop/MokshaVCS$ ./moksha add  hello2.txt
Added hello2.txt
suyash@suyash-virtual-machine:~/Desktop/MokshaVCS$ ./moksha status
<---- The following files are added, not staged for commit ---->
hello2.txt
suyash@suyash-virtual-machine:~/Desktop/MokshaVCS$ ./moksha add hello3.txt
Added hello3.txt
suyash@suyash-virtual-machine:~/Desktop/MokshaVCS$ ./moksha status
<---- The following files are added, not staged for commit ---->
hello3.txt
hello2.txt
suyash@suyash-virtual-machine:~/Desktop/MokshaVCS$ ./moksha commit -m hello2_hello3_Added
Committed hello3.txt
Committed hello2.txt
Committed hello2.txt
Committed hello.txt
Committed ./.config
suyash@suyash-virtual-machine:~/Desktop/MokshaVCS$ ./moksha log
Hash: 5272215196538ae571dd6f90bf6a6b1c          Path: ./.commits/5272215196538ae571dd6f90bf6a6b1c               Message: h
ello2_hello3_Added
Hash: 44af93d806c9947168ee766e2b4c3ddb          Path: ./.commits/44af93d806c9947168ee766e2b4c3ddb               Message: h
ello_added
Hash: <INITIAL>         Path:           Message: <INITIAL>
```

Figure 8: Multiple Add Example

3.3.3 COMMIT : commit -m "<message>"

- As a programmer, we need a checkpoint for our progress every now and then. Commit lets you save all the progress you have made.
- We have a message associated with every commit.
- While committing, we check if the "last added file hash in the last commit of commit history" matches the "last add hash" in the config file. If it matches, it means we have nothing more to commit and notify the user. Otherwise, we proceed to the next step.
- We create a new directory for the commit and copy the files in the tracked files in the config data structure one by one into the newly created directory.
- The count of uncommitted files is reset to zero.

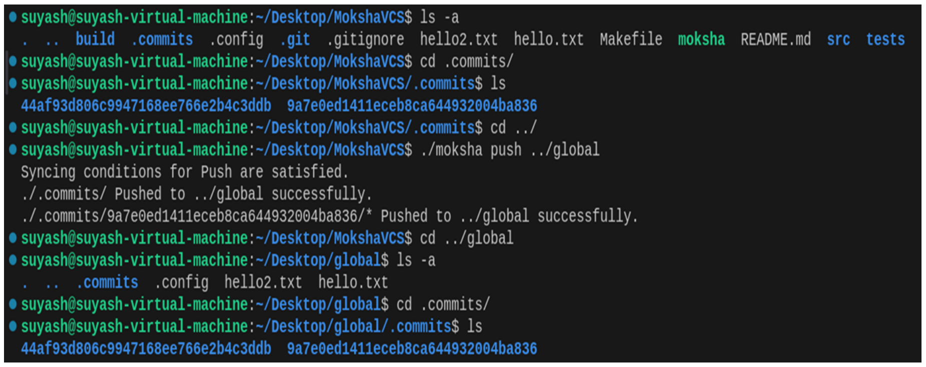


```
suyash@suyash-virtual-machine:~/Desktop/MokshaVCS$ ./moksha add hello.txt
Added hello.txt
suyash@suyash-virtual-machine:~/Desktop/MokshaVCS$ ./moksha log
Hash: <INITIAL>         Path:           Message: <INITIAL>
suyash@suyash-virtual-machine:~/Desktop/MokshaVCS$ ./moksha commit -m hello_added
Committed hello.txt
Committed ./.config
suyash@suyash-virtual-machine:~/Desktop/MokshaVCS$ ./moksha log
Hash: 44af93d806c9947168ee766e2b4c3ddb          Path: ./.commits/44af93d806c9947168ee766e2b4c3ddb               Message: h
ello_added
Hash: <INITIAL>         Path:           Message: <INITIAL>
```

Figure 9: Commit Example

3.3.4 PUSH : push <Global_Dir_Path>

- It is common for multiple people to work on a single repository, so we keep a designated global directory as the push directory which contains the most important version.
- We have a directory associated with the push request - global directory.
- We find the last common commit in the current directory and global directory (Sync Check!) and then copy the commits which were not acknowledged in the global directory.
- We also update the config file in the global directory.



```
suyash@suyash-virtual-machine:~/Desktop/MokshaVCS$ ls -a
.  ..  build  .commits  .config  .git  .gitignore  hello2.txt  hello.txt  Makefile  moksha  README.md  src  tests
suyash@suyash-virtual-machine:~/Desktop/MokshaVCS$ cd .commits/
suyash@suyash-virtual-machine:~/Desktop/MokshaVCS/.commits$ ls
44af93d806c9947168ee766e2b4c3ddb  9a7e0ed1411eceb8ca644932004ba836
suyash@suyash-virtual-machine:~/Desktop/MokshaVCS/.commits$ cd ../
suyash@suyash-virtual-machine:~/Desktop/MokshaVCS$ ./moksha push ../global
Syncing conditions for Push are satisfied.
./.commits/ Pushed to ../global successfully.
./.commits/9a7e0ed1411eceb8ca644932004ba836/* Pushed to ../global successfully.
suyash@suyash-virtual-machine:~/Desktop/MokshaVCS$ cd ../global
suyash@suyash-virtual-machine:~/Desktop/global$ ls -a
.  ..  .commits  .config  hello2.txt  hello.txt
suyash@suyash-virtual-machine:~/Desktop/global$ cd .commits/
suyash@suyash-virtual-machine:~/Desktop/global/.commits$ ls
44af93d806c9947168ee766e2b4c3ddb  9a7e0ed1411eceb8ca644932004ba836
```

Figure 10: Push Example

- **Edge Case :** Suppose, Programmer-1 does two commits, then Programmer-2 does the third commit. Next when Programmer-1 tries to do the fourth commit there can be conflicts in the code (Exception is given in this case).

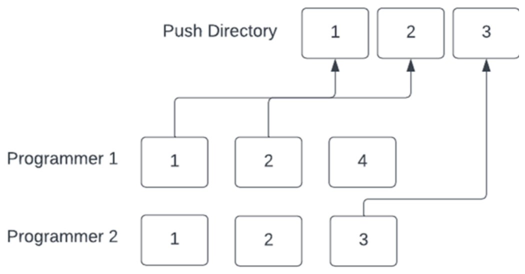
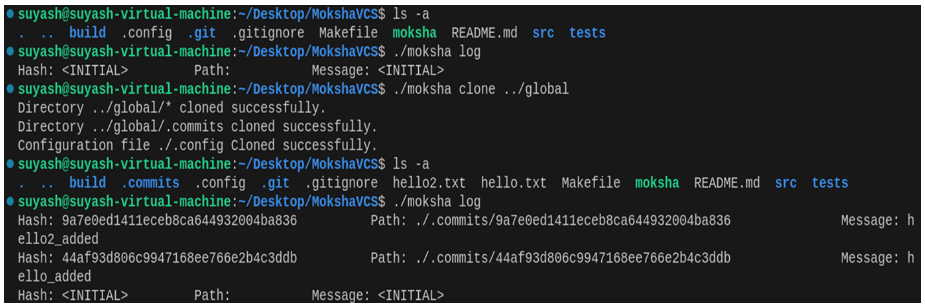


Figure 11: Edge Case for Push

3.3.5 CLONE : clone <Global_Dir_Path>

- Sometimes, people need to start from where others left the repository, so cloning is used to get to the point where others left it.
- We have a source directory associated with the clone command.
- We simply copy all the files available in the source directory to the current working directory.
- The latest version, the ".config" file, and the ".commits/" directory are copied.

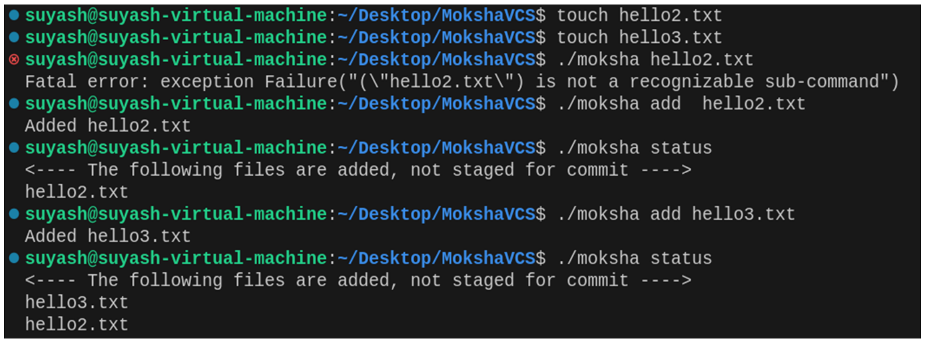


```
suyash@suyash-virtual-machine:~/Desktop/MokshaVCS$ ls -a
.  ..  build  .config  .git  .gitignore  Makefile  moksha  README.md  src  tests
suyash@suyash-virtual-machine:~/Desktop/MokshaVCS$ ./moksha log
Hash: <INITIAL>        Path:          Message: <INITIAL>
suyash@suyash-virtual-machine:~/Desktop/MokshaVCS$ ./moksha clone ../global
Directory ../global/* cloned successfully.
Directory ../global/.commits cloned successfully.
Configuration file ./.config Cloned successfully.
suyash@suyash-virtual-machine:~/Desktop/MokshaVCS$ ls -a
.  ..  build  .commits  .config  .git  .gitignore  hello2.txt  hello.txt  Makefile  moksha  README.md  src  tests
suyash@suyash-virtual-machine:~/Desktop/MokshaVCS$ ./moksha log
Hash: 9a7e0ed1411eceb8ca644932004ba836         Path: ./.commits/9a7e0ed1411eceb8ca644932004ba836              Message: h
ello2_added
Hash: 44af93d806c9947168ee766e2b4c3ddb         Path: ./.commits/44af93d806c9947168ee766e2b4c3ddb              Message: h
ello_added
Hash: <INITIAL>        Path:          Message: <INITIAL>
```

Figure 12: Clone Example
clone

3.3.6 STATUS : status

- We need a tool to check on what new files have been added to the current repository state.
- Status provides a list of tracked files (added files) which are not yet committed.
- We iterate over the tracked files in the config data structure and print them on the console if they are not committed.

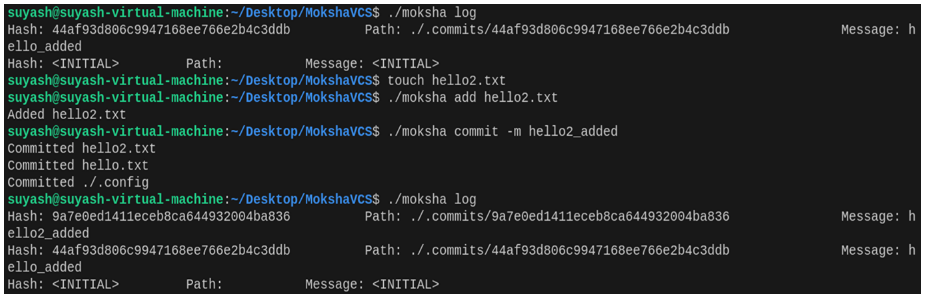


```
● suyash@suyash-virtual-machine:~/Desktop/MokshaVCS$ touch hello2.txt
● suyash@suyash-virtual-machine:~/Desktop/MokshaVCS$ touch hello3.txt
⊗ suyash@suyash-virtual-machine:~/Desktop/MokshaVCS$ ./moksha hello2.txt
  Fatal error: exception Failure("(\"hello2.txt\") is not a recognizable sub-command")
● suyash@suyash-virtual-machine:~/Desktop/MokshaVCS$ ./moksha add  hello2.txt
  Added hello2.txt
● suyash@suyash-virtual-machine:~/Desktop/MokshaVCS$ ./moksha status
  <---- The following files are added, not staged for commit ---->
  hello2.txt
● suyash@suyash-virtual-machine:~/Desktop/MokshaVCS$ ./moksha add hello3.txt
  Added hello3.txt
● suyash@suyash-virtual-machine:~/Desktop/MokshaVCS$ ./moksha status
  <---- The following files are added, not staged for commit ---->
  hello3.txt
  hello2.txt
```

Figure 13: Status Example

3.3.7 LOG : log

- This is a tool to look over the past commits. This shows the progress of the repository.
- We iterate over the commit history and print the relevant information such as commit messages and commit hash for every commit in the past.

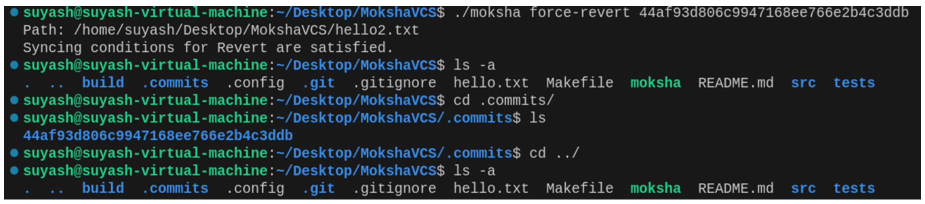


```
suyash@suyash-virtual-machine:~/Desktop/MokshaVCS$ ./moksha log
Hash: 44af93d806c9947168ee766e2b4c3ddb         Path: ./.commits/44af93d806c9947168ee766e2b4c3ddb              Message: h
ello_added
Hash: <INITIAL>        Path:          Message: <INITIAL>
suyash@suyash-virtual-machine:~/Desktop/MokshaVCS$ touch hello2.txt
suyash@suyash-virtual-machine:~/Desktop/MokshaVCS$ ./moksha add hello2.txt
Added hello2.txt
suyash@suyash-virtual-machine:~/Desktop/MokshaVCS$ ./moksha commit -m hello2_added
Committed hello2.txt
Committed hello.txt
Committed ./.config
suyash@suyash-virtual-machine:~/Desktop/MokshaVCS$ ./moksha log
Hash: 9a7e0ed1411eceb8ca644932004ba836         Path: ./.commits/9a7e0ed1411eceb8ca644932004ba836              Message: h
ello2_added
Hash: 44af93d806c9947168ee766e2b4c3ddb         Path: ./.commits/44af93d806c9947168ee766e2b4c3ddb              Message: h
ello_added
Hash: <INITIAL>        Path:          Message: <INITIAL>
```

Figure 14: Log Example

3.3.8 FORCE_REVERT : force-revert <Hash_of_version>

- Everyone makes mistakes, so this is a tool used to go back to a given commit stage.
- We delete all the commits after the given commit.
- We also delete all the files in the current working directory which are not present in the saved files corresponding to the commit given to the revert command.
- Caution: The deleted commits cannot be retrieved again.



```
● suyash@suyash-virtual-machine:~/Desktop/MokshaVCS$ ./moksha force-revert 44af93d806c9947168ee766e2b4c3ddb
  Path: /home/suyash/Desktop/MokshaVCS/hello2.txt
  Syncing conditions for Revert are satisfied.
● suyash@suyash-virtual-machine:~/Desktop/MokshaVCS$ ls -a
  .  ..  build  .commits  .config  .git  .gitignore  hello.txt  Makefile  moksha  README.md  src  tests
● suyash@suyash-virtual-machine:~/Desktop/MokshaVCS$ cd .commits/
● suyash@suyash-virtual-machine:~/Desktop/MokshaVCS/.commits$ ls
  44af93d806c9947168ee766e2b4c3ddb
● suyash@suyash-virtual-machine:~/Desktop/MokshaVCS/.commits$ cd ../
● suyash@suyash-virtual-machine:~/Desktop/MokshaVCS$ ls -a
  .  ..  build  .commits  .config  .git  .gitignore  hello.txt  Makefile  moksha  README.md  src  tests
```

Figure 15: Force-revert Example

3.4 Building Project

- **Compilation Rules**: Defines rules for compiling OCaml source files into object files and linking them to create the executable `moksha`.
- **Variable Definitions**: Defines variables such as `OCAMLC`, `OCAMLFLAGS`, `BUILDDIR`, and `SRCDIR` for compiler commands and directory paths.
- **Automatic Dependency Generation**: Automatically generates dependencies based on file structure and compiles necessary modules.
- **Cleaning Targets**: Provides `clean` and `cleanup` targets to remove build artifacts and directories.

3.5 Unit Testing of system

- Unit testing is used to test some of the important commands.
- Separate unit tests are written for important modules (commands) to check its working.

4 Main References

- https://v2.ocaml.org/docs/
  - The official OCaml documentation will be crucial for explaining how functionalities can be implemented using the OCaml language.
- Devineni, Siva Karthik. (2020). Version Control Systems (VCS) the Pillars of Modern Software Development: Analyzing the Past, Present, and Anticipating Future Trends. International Journal of Science and Research (IJSR). 9. 1816-1829. 10.21275/SR24127210817.
  - This paper mentions an in-depth look at best practices in VCS, including branching strategies, collaborative workflows. This directly aligns with the design and implementation aspects of our DVCS. Covering popular workflows like GitFlow and GitHub Flow can be helpful for the implementation of our VCS.

5 Work Distribution

1. **Deepkumar Patel (IMT2021011)**:
   - Command Language generation and its Parsing
2. **Krutik Patel (IMT2021024)** and **Suyash Chavan (IMT2021048)**:
   - Version Control System Functions
   - Log Maintenance

6 GitHub Repository

Link to project's GitHub repository:
`https://github.com/Krutik-Patel/MokshaVCS`